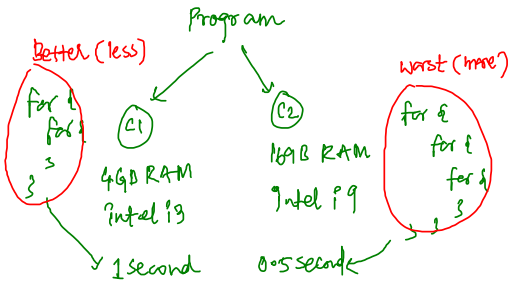
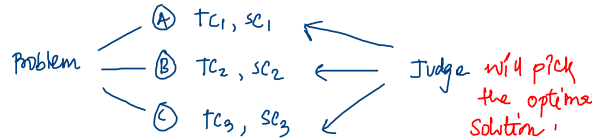
* Time complexity :



→ worst solutions can run faster with high computing resources, where as best solution might be slower with less computing resources

← practical Analysis →
(here TC is depending machine)



→ which solution is best, $\min(TC_1, TC_2, TC_3)$ irrespective of space

→ what if $TC_1 = TC_2 = TC_3$, the pick the solution with $\min(SC_1, SC_2, SC_3)$.

a. $TC_1 = 5min$, $SC_1 = 1MB$
b. $TC_2 = 2min$, $SC_2 = 5MB$
c. $TC_3 = 3min$, $SC_3 = 2MB$

$\min TC = 2min$

* TC/SC are machine Independent (Theoritical/Mathematical Analysis)

①

```
1  function solve1(m, n) {
2    let a = 0;
3    let b = 0;
4    for (let i = 0; i < n; i++) {
5      a = a + i;
6    }
7    for (let j = 0; j < m; j++) {
8      b = b + j;
9    }
10 }
```

1. Look for any loops, function calls, that is where your program takes most of the time.

2. TC = no. of instructions
= no. of Iterations

no. of iterations = n

no. of iterations = m

TC = Total no. of Iterations

= n + m

3. = O(n+m)

* TC will be in terms of user input size.

Big-oh; Mathematical/Asymptotic notation (upper Bound/limit)

meaning: Your program will not take more than n+m time complexity.

~~TC = O(a+i+n+m)~~

a, i are not user defined inputs

→ m = 5, n = 3
ptr = 5+3 = 8

→ m = 10, n = 4
ptr = 10+4 = 14

(2)

```
12  function solve2(n,m) {
13    let a = 0;
14    for (let i = 0; i < n; i++) {
15      for (let j = 0; j < m; j++) {
16        a = a + j;
17      }
18    }
```

Total Iterations = 3 + 3 + 3 + 3
= 4 * 3 = 12

generalize, TC = O(n * m)

Let say n = 4 and m = 3 (take an example to get a better idea on num Iterations)

(1) i = 0
→ j = 0
→ j = 1
→ j = 2
→ j = 3 X (3 < 3) X
} 3

(2) i = 1
→ j = 0
→ j = 1
→ j = 2
→ j = 3 X (3 < 3) X
} 3

(3) i = 2
→ j = 0
→ j = 1
→ j = 2
→ j = 3 X (3 < 3) X
} 3

(4) i = 3
→ j = 0
→ j = 1
→ j = 2
→ j = 3 X (3 < 3) X
} 3

(5) i = 4 X (4 < 4)

(3)

```
26  function solve3(m, n) {
27    let a = 0;
28    for (let i = 0; i < n; i++) {
29      for (let j = n; j > i; j--) {
30        a = a + j;
31      }
32    }
33  }
```

Total iterations = 4 + 3 + 2 + 1 = 10

for any input (n),

TC = no. of iterations = sum of 1st n natural

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

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

* $TC = O(n^2)$
→ pick highest degree term.
→ ignore any constants

Let say n = 4

① i = 0
→ j = 4
→ j = 3
→ j = 2
→ j = 1
→ j = 0 ✗ (0 > 0)
} 4

② i = 1
→ j = 4
→ j = 3
→ j = 2
→ j = 1 ✗ (1 > 1)
} 3

③ i = 2
→ j = 4
→ j = 3
} 2
→ j = 2 (2 > 2) ✗

④ i = 3
→ j = 4
} 1
→ j = 3 (3 > 3) ✗

⑤ i = 4 (4 < 4) ✗

* Why to ignore constants and only pick highest degree term?

$$\frac{n^2+n}{2}$$

for Eg: $n = 10^8$

$$\Rightarrow \frac{(10^8)^2 + 10^8}{2}$$

$$\Rightarrow \frac{10^{16} + 10^8}{2} \simeq 10^6$$

10^8 → negligible

2 → negligible

* You have 1 crore rupees,

10,000 / 1 lakh → negligible in front of 1 crore

④

```
35  function solve4(n) {
36    let i = 1;
37    while (i <= n) {
38      i = i + 2;
39    }
40  }
```

eg: n = 10 → 10/2 = 5

① i=1, 1<=10
② i=3, 3<=10
③ i=5, 5<=10
④ i=7, 7<=10
⑤ i=9, 9<=10
⑥ i=11, 11<=10 ✗

⑤

eg: n = 20 → 20/2 = 10

⇒ numIterations = n/2

TC = O(n/2)

= O(n)

n/k → user

```
while ( i <= n ) {
    i = i + k
}
```

→ i = 1 (+k) jump → n

→ O(n/k)

* You need 50/- and you have unlimited supply of 2/- coins, how many coins are required to make 50/- ?

⇒ 25 coins * 2/- = 50/-

⇒ If you want to make 200/- ⇒ 100 coins

⇒ TotalReqAmt / coinValue

⇒ 200/2 = 100 coins

(5)

```
42  function solve5(n) {
43    while (n > 1) {
44      n = n + 20;
45      n = n - 10;
46      n = n - 15;
47    }
48  }
```

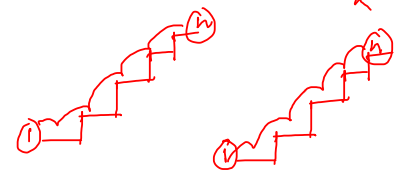
$n = n+20 \quad n+20-10-15$

$n = n-10 \quad \Rightarrow n-5$

$n = n-15$

$n = n-5$

* $1 \xrightarrow{+K} n \Rightarrow n/K$
* $n \xrightarrow{-K} 1 \Rightarrow n/K$



tq : $n = 30$

① $30 > 1$

$\Rightarrow n = 30+20 = 50$

$\Rightarrow n = 50-10 = 40$

$\Rightarrow n = 40-15 = 25$

② $25 > 1$

$\Rightarrow n = 25+20 = 45$

$\Rightarrow n = 45-10 = 35$

$\Rightarrow n = 35-15 = 20$

③ $20 > 1$

$\Rightarrow n = 20+20 = 40$

$\Rightarrow n = 40-10 = 30$

$\Rightarrow n = 30-15 = 15$

$30 \xrightarrow{-5} 25 \xrightarrow{-5} 20 \xrightarrow{-5} 15 \xrightarrow{-5} 10 \xrightarrow{-5} 5 \xrightarrow{-5} 0$

$\Rightarrow$ 6 Iterations $(30/5)$

$\Rightarrow n/5$ Iterations

* $TC = O(n/5) = O(n)$

(6)

```
function solve6(n) {
  let i = 1;
  while (i < n) {
    i = i * 2;
  }
}
```

* $i \xrightarrow{*2} n$ * $i = 1 \xrightarrow{+K} n \Rightarrow TC = O(\log_k^n)$

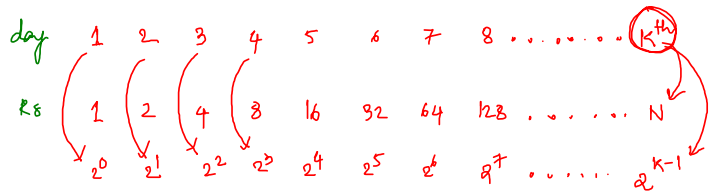
you have a magical 1 Re, coin which doubles itself every day. How many days will it take to become NRe. Coin.

Let say $N = 128 \Rightarrow$ 8 days to reach 128

eg: $n = 128$ $(128 < 128)$

$i = 1, 2, 4, 8, 16, 32, 64, \cancel{128}$

no of iterations = 7



* It will take $\log_2^N + 1$ days to reach N Rs.

* $TC = O(\log_2^N)$

$\Rightarrow 2^{K-1} = N$ (apply $\log_2$ on both sides)

$\Rightarrow \log_2 2^{K-1} = \log_2 N \Rightarrow K-1 \log_2^2 = \log_2^N$

$\Rightarrow K - 1 = \log_2^N$

$\Rightarrow K = \log_2^N + 1$

Basic log properties :

$2^x = 32$

what is x ? $x = 5$

$2^x = 4294967296$

what is x ? we use log

$x = \log_2 4294967296$

↓

use log table to find the value

① $\log_{b^n} a^m = \frac{m}{n} \log_b a$

eg: $\log_{2^1} 2^{k-1} = \frac{k-1}{1} \log_2 2$

② $\log_a a = 1$

eg: $\log_2 2 = 1$

③ $\log_{\textcircled{b}} a + \log_{\textcircled{b}} c = \log_b ac$

bases are same

(7)

```
57  function solve7(n) {
58    let i = n;
59    while (i > 0) {
60      i = i / 2;
61    }
62  }
```

* $i = n \xrightarrow{/2} 0 / 1$

* $TC = O(\log_2 n)$

(1) $i = n = n/2^0$

(2) $i = n/2 = n/2^1$

(3) $i = n/4 = n/2^2$

(4) $i = n/8 = n/2^3$

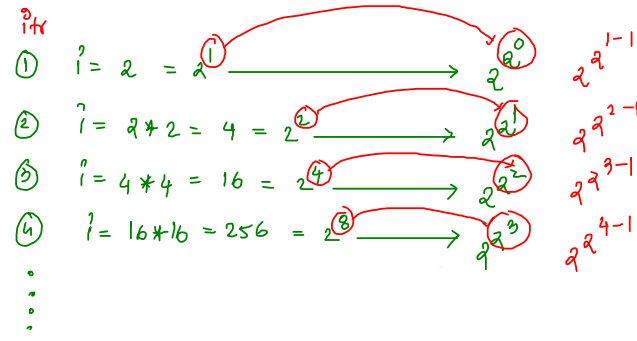
⋮

(k) $i = 1 = n/2^{k-1} \Rightarrow 2^{k-1} = n \Rightarrow k = \log_2 n + 1$

$i = 1 \xrightarrow{+k} n$ $i = n \xrightarrow{-k} n$ Both are same n/k iterations $\simeq O(n/k)$	$i = 1 \xrightarrow{*k} n$ $i = n \xrightarrow{/k} 1$ Both are same $\log_k n + 1$ iterations $\simeq O(\log_k n)$

(8)

```
64  function solve8(n) {
65    let i = 2;
66    while (i < n) {
67      i = i * i;
68    }
69  }
```



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

$$\log_2 2^{2^{k-1}} = \log_2 n$$

$$2^{k-1} \log_2 2 = \log_2 n$$

$$2^{k-1} = \log_2 n$$

$$\Rightarrow \quad k = \log_2 \log_2 n + 1$$

$$k-1 \log_2 2 = \log_2 \log n$$

$$*+c = O(\log_2 \log_2 n)$$

(9)

```
function solve9(n) {
  let a = 0;
  for (let i = 1; i <= n; i++) {
    for (let j = 1; j <= i; j = i + j) {
      a = a + i + j;
    }
  }
}
```

Total = 1 + 1 + 1 + 1

= 4 = n

* TC = O(N)

* Do not fall in trap by looking 2 nested loops → n^2.

* Inner loop is running only once for every i.

Let n = 4

① i = 1

→ j = 1, 1 <= 1 } ①

→ j = i + j = 1 + 1 = 2, 2 <= 1 X

② i = 2

→ j = 1, 1 <= 2 } ①

→ j = i + j = 2 + 1 = 3, 3 <= 2 X

③ i = 3

→ j = 1, 1 <= 3 } ①

→ j = i + j = 3 + 1 = 4, 4 <= 3 X

④ i = 4

→ j = 1, 1 <= 4 } ①

→ j = i + j = 4 + 1 = 5, 5 <= 4 X

⑤ i = 5 (5 <= 4) X

(10)

(n, K)

```
function solve10(n) {
  let a = 0;
  for (let i = 1; i <= n; i++) {
    let p = i ** k;
    for (let j = 1; j <= p; j++) {
      a = a + i + j;
    }
  }
}
```

$TC = 1^K + 2^K + 3^K + \dots + n^K$

$\simeq n^K + n^K + n^K + \dots + n^K$ $\quad * TC = O(n^{K+1})$

$\simeq n \cdot n^K \simeq n^{K+1}$

$= O(n^K)$

let $n = 8$, $k = 2$

① $i = 1$
→ $p = 1^K = 1^2 = 1$
→ $j = 1 \to p$ (1 time)

② $i = 2$
→ $p = 2^K = 2^2 = 4$
→ $j = 1 \to 4$ (4 times)

③ $i = 3$
→ $p = 3^2 = 9$
→ $j = 1 \to 9$ (9 times)

④ $i = 4$ ✗ $(4 \le 3)$

Total Iterations = $1 + 4 + 9$ (when $k = 2$)

$= 1^2 + 2^2 + 3^2 + 4^2 + \dots + n^2$ $\quad * TC = O(n^3)$

= sum of squares of 1st N natural numbers = $\frac{n(n+1)(2n+1)}{6}$

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

```
for(let i = 1; i<=n; i++) {
    let p = i**2;
    for(let j = 1; j<=p; j++) {
        cl("Hello");
    }
}
```

⇒ ~~$TC = O(n * p)$~~ because p is a variable not an input

~~$TC = O(n^4)$~~ Two nested loops → n^2 (trap)

$TC = O(n^3)$

(1) Linear O(N) [d]

(2) Logarithmic $O(\log_2 N)$ [f]

(3) Exponential (2^n, 3^n) [b]

(4) polynomial (n^3, n^4, n^5) [a]

(5) Log Linear ($n \log n$) [c]

(6) Quadratic (n^2) [e]

a) $N^{K+9} \Rightarrow N^{\text{constant}}$

b) $5^{N*2} \Rightarrow 5^N \Rightarrow (\text{Constant})^N$

c) $(N/4)\log_2(N/4000) \Rightarrow N \log_2 N$ (ignore constants)

d) $3^{20}N + 10^5 \Rightarrow N$ (ignore constants)

e) $10N + 9(N/100) + 340N^2 \Rightarrow N + N + N^2 \Rightarrow N^2$

f) $10^3 \log_2(N+3N) \Rightarrow \log_2 4N \Rightarrow \log_2 N$ (ignore constants)

★ Constant $\Rightarrow O(1)$

(no loops, no complex functions)

horrible / need to optimise — Bad

$O(n!) > O(2^n) > O(n^2) > O(n \log n) > O(n) > O(\log n) > O(1)$

$O(n)$: fair; $O(\log n)$: good; $O(1)$: best

Space Complexity :

* If you are using any datastructures | apart from given user input | in your solution
  ⇒ O(size(ds))

* Mostly you will encounter arrays of size 'N'
  ⇒ SC : O(N)
  otherwise, ⇒ SC : O(1)

Reverse array

Copy all elements by traversing k → 1

arr: | 1 | 2 | 3 | 4 | SC = O(N) ↑ TC = O(N) ~

new_arr: | 4 | 3 | 2 | 1 |

Two pointers In place solution

SC : O(1) ↓
TC : O(N) ~

```
function hw1(n) {
  for (let i = 0; i < n; i++) {
    for (let j = 0; j < i; j++) {
      console.log("*");
      break;
    }
  }
}

function hw2(n) {
  let i = 1;
  while (i ** 2 <= n) {
    i = i + 1;
  }
}

function hw3(m, n) {
  while (m != n) {
    if (m > n) {
      m = m - n;
    } else {
      n = n - m;
    }
  }
}
```

```
function hw4(n) {
  let i = 1;
  while (i < n) {
    let j = n;
    while (j > 0) {
      j = parseInt(j / 2);
    }
    i = i * 2;
  }
}

function hw5(n) {
  for (let i = 0; i < parseInt(n / 2); i++) {
    for (let j = 1; j < n - parseInt(n / 2); j++) {
      let m = 1;
      while (m <= n) {
        m = m * 2;
      }
    }
  }
}
```