Reference: Introduction to Algorithm By Coreman

Syllabus: (1) Analysis
(2) Divide and conquer.
(3) Greedy Technique
(4) Dynamic programming.
(5) Hashing & Tree and graph Traversal.

Definition: It is a combination of sequence of finite steps to solve a problem.

Example: Multiplication of Two Numbers

```
MTN() { 1. TAKE 2no's (a,b).
        2. Multiply a and b and store result in c.
        3. return c
        }.
```

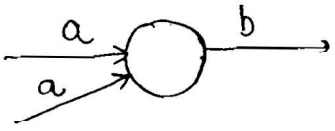
from which function we have come, we have to return there.

- finite steps - finite Time should be there (But it doesn't mean finite steps always leads to finite time)
- infine steps - Infinite Time
- All steps are compulsory, so combination is required, so finally it can solve the problem.

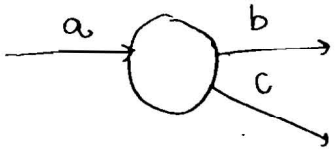
printf → c
cout → c++. } syntax

Properties of Algorithm

1. It should Terminate after finite Time.
2. It should produce "atleast" one output (Min^m 1 output).
3. It should take "0 or More input"
4. It should be "deterministic"
   (different behaviour - Non-deterministic)
   deterministic - always same answer.



deterministic (finite steps) also there.



2 O/Ps. Non deterministic.

No dependency → so we can swap the steps of Algo.
Non deterministic → special case.

Steps Required to Design Algorithm:

1. Problem definition (Knowing problem clearly).
2. Design Algorithm — Divide and conquer
   - Greedy technique
   - Dynamic Prog.
   - Backtracking
   - Branch & Bough (BB).

$\begin{bmatrix} a-b \\ c-d \\ e-f \\ g-3 \end{bmatrix}$

Algorithm Design: After knowing the problem, Map the problem to the existing Algorithm.

3. Draw flow chart (Diagramatic Algorithm).
4. Testing and verification. (The Report we made (Test cases), our Prog should Run for those i/Ps)
5. Coding or Implementation.
6. Analysis the Algorithm.

Run – MM (go to Run)
Save – Hard disk
Running Time → MM (space complexity).
time complexity.

} operating system process state diagram.

Design and Analysis of Algorithm.

C

Analysis : chapter 1

If your problem having More than 1 solution, Best one will be decided by analysis based on 2 factors.

1. Time complexity (CPU Time)
2. Main Memory (Space complexity).

If your Problem having only 1 soln, go with that soln no no need of Analysis.

Time complexity:

Time required for Prog $T(P) = C(P) + R(P)$

- $C(P)$: compile Time of Prog.
  - ⇓ Based on compiler
  - ⇓ S/w
  - ⇓ Based on language of compiler
- $R(P)$: Running Time of Prog.
  - Based on processor
  - ⇓ H/w
  - ⇓ Type of Hardware.

Based on lang. of Program written compiler, is prog.

Types of Analysis

1. Apostiary Analysis.
2. Apriori Analysis

(postponing the things. (By asking a question, instead of giving answer, asking question to us).

Apostiary.

① It is based on (dependend) on language of compile & Type of H/w.

Adv ② Exact Answer (It will give exact answer bcoz we are considering Real things).

Dis ③ system to system different Answer (diff^nt Time) (constants differ sys to sys)

"it is Relative Analysis".

Here processor & compiler lang. is imp.

Apriori

① It is independent on lang – c. & type of H/w.

② Approximate Answer

Advantage ③ system to system same Answer (same ans with diff^nt sys.).

"Absolute Analysis".

if prog is Running faster prog. written in great logic.

NOTE: Everyone cannot buy supercomputer but everyone can write super ~~comput~~ algo. because lord is given same brain to all. But some people will use it some people will not use it.

software company uses – Apriory Analysis.

APriori Analysis:
we are finding strength of logic.

"It is a determination of order of Magnitude of a statement."
↓
while Running statement is Running how many times.

ex①

```
Main()
{
1. a = y+z;  ⟹ 1 (order of Magnitude).
}
```

(put Big Oh (O) before order of M.

O(1).

ex②

```
main()
{  x = y+3;  ——— 1
   for (i=1; i≤n; i++).
   {
      x=y+3;  ——— n.
   }
}.
```

initialization = 1
condition = n+1
statement = n.
i++ = n.

n+1 = O(n)

Exp③

```
main()
{
   x = y+3;  ——— ①
for (i=1; i≤n; i++)
      x = y+3;  ——— n
for (i=1; i≤n; i++)  ——— n.
{
 for (j=1; j ≤ n; j++)  ——— n
 { x = y+3;  ——— n.n
 }
}
}
```

when bracket is in 1 statement is their No bracket.

1 + n(n).
1 + n² = O(n²).

outer loop — add
inner loop = multiply.

Time complexity is finding Bigger loops.
(Where CPU spending More Time).

give this part to cache Memory, so CPU got to know that it is spending More Time, then. program is fast.

Locality of Reference — cache Memory; (which is More imp)

Example (4)

```
Main ()
{ while (i≤n)
  {
    i=i+1
    i=i+4
    i=i+5
  }
}
```

increamentation.

$i = i+10 \Rightarrow \frac{n}{10} \Rightarrow \frac{1}{10} . n.$

$\Rightarrow O(n)$.

How many times loop is executing $n/10$.

```
main ()
{ i=n
  while (i≥1).
  {
    i=i-1
    i=i-9
  }
}
```

decrementation.

$i = i-10 \Rightarrow \frac{n}{10} \Rightarrow O(n)$.

*

```
i=i-1  > -10
i=i-9  /
i=i+7  > 10
i=i+3.
```

$i = -10+10$

$i = 0$. — not incrementing NO decrementing infinite loop.

example:5

```
main()
{
 i=1;
while (i<=n).
  {
   i= 2*i
  }
}.
```

$1<64$ ✓
$2<64$ ✓
$4<64$ ✓
$8<64$ ✓
$16<64$ ✓
$32<64$ ✓
$64<64$ ✗

64 — 6 steps
32 — 5 steps
16 — 4 steps
$n = \log_2 n$.

Proof

1	1
2	3
2^2	3^2
2^4	:
:	$3^k = n$
:	$\log_3 3^k = \log_3 n$
	$k = \log_3 n$
$2^k = n$.	
$\log_2 2^k = \log_2 n$	
$k = \log_2 n$	

$i = 2*i$
$i = 3*i$

$i = 2*i*3$
$i = 6i$
$k = \log_6 n$

$i = 2*i$
$i = 3*i$
$i = 5*i$ $\Rightarrow$ $i = 30i$
$O(\log_{30} n)$.

II
```
main()
{while (i=n
        i>=1)
  {
   i= i/2 ⇒O(log2n).
  }
}
```

n
$n/2$
$n/2^2$
$n/2^3$
:
$n/2^k = 1$
$n = 2^k$
$\log_2 n = k$

$i = i/2$
$i = i/3$
$i = i/4$ $\Rightarrow i/24$.
$\log_{24} n$.

```
main().
i = 1;
while (i ≤ n)
{
    i = 2 * i;
    i = 3 * i;
    i = i + 3;
}
```

if i = 10.

60 → 63 → 613

same

Neglect addition

Example: 6 proof

```
main()
{
    i = 2
    while (i ≤ n)
    {
        i = i^2
    }
}
```

```
main()
{ i = 2
while (i ≤ n)
{ i = i^12
}
}
```

2, 2^{12}, 2^{24}

2^{12^1}, 2^{12^2}, 2^{12^3}, 2^{12^4}, ..., 2^{12^K}.

$\log_2 2^{12K} = \log_2 n$

$12^K \log_{12} 12 = \log_{12} \log_2 n$

$\log_{12} 12^K = \log_{12} \log_2 n$.

$2 < 1000$.

$4 < 1000$

$16 < 1000$

$256 < 1000$

$(256)^2 < 1000$ ×

$2 = 2^{2^1}$

$2^2 = 2^{2^2}$

$(2^2)^2 = 2^{2^3}$

$(2^4)^2 = 2^{2^4}$.

$(28)^2 = 2^{16} \Rightarrow 2^{24}$.

K

$2^{2^k} = n$.

$2K \log_2 2 = \log_2 n$.

$K \log_2 2 \log_2 2 = \log_2 (\log_2 n)$

$\boxed{k = (\log_2 (\log_2 n))}$.

$\log_2 \log_2 1000$.

$2^{①} = 2^1 = 2$

$2^{②} = (2^1)^2 = 2^{2^1}$

$2^{④} = (2^{2^1})^2 = (2^{2^2})^2$

$2^{⑧} = (2^{2^2})^3 = (2^{2^3})^3$

$2^{2K} = n$.

$2^{12 \cdot K} = n$.

$12^K \log_2 2 = \log_2 n$.

$12^K = \log_2 n$

$K \log_{12} 12 = \log_{12} \log_2 n$

$i = i^{29}$

$i = i^2$

$i = i^{31}$

$\log_{12} 12^K = \log_{12} \log_2 n$.

ex i = 25. → inner case.

25^{29^K}

Ex

i = 25

$i = i^{29}$ ⇒ $O(\log_{29} \log_{25} n)$

// outer Base

$i = i^{29}$

$i = i^2$

$i = i^7$

$(i^{29})^2 = (i^{58})^7 = i^{406}$

$O(\log_{406} \log_{25} n)$

Example:7 for square Reverse is "Root" ⇒ Here decreasing.

```
main
{ i=n.                    termination
while (i>2).
  { i = i^1/2
  }
}
```

$n \longrightarrow n$

$n^{1/2} \longrightarrow$

$(n^{1/2})^{1/2}$

$n^{1/2^2}$

$n^{1/2^3}$

$n^{1/2^4}$

$n^{1/2^K} = 2$ ← termination

$\frac{1}{2^K} \log_2 n = \log_2 2$

$\log_2 n = 1 \times 2^K$

$\log_2 \log_2 n = K \log_2 2$.

$\boxed{\log_2 \log_2 n = K}$

36
3
108

$i = i^{1/36}$ → $O(\log_{36} \log_{29} n)$

$i = i^{1/3}$ → $O(\log_{108} \log_{29} n)$ ← termination

$i = i^2$

$O(\log_{.54} \log_{29} n)$