1.3 How to write an Algorithm

```
Algorithm swap (a, b)
{
    temp = a;
    a = b;
    b = temp;
}
```

How to Analyze an Algorithm

1. Time (time function)
2. Space (how much memory it will take)
3. Network Consumption
4. Power Consumption (how much power it is using)
5. CPU registers (how many registers it is using)

1st Time Anaylsis

(simple statement)

every statement takes one unit of time, unless it is calling another algorithm or using another procedure.

```
Algorithm swap (a,b) {
    temp = a;    -1
    a = b;       -1
    b = temp;    -1
}                 3

f(n) = 3
```

ex:

$x = 5 * a + 6 * b - 1$

(its still only 1 unit of time)

Now this is just when analyzing the ~~code~~ Algorithm with pusecode, when usivs an actual language it will be 4 units of time (1 for declaring x, 1 for 5*a, 1 for the +, 1 for 6 * B)

2nd Space analysis

So a is one parameter
B is also a paremeter
and "temp" is a local variable used

} 3 variables

$S(N) = 3$ [constant]

So when it is constant to show with time complexity we just say it as O(1)

1.4 Frequencey count method

This algorith is for funding the sum of all the elements an larrray

```
Algorithm Sum (A, n)
{
    s = 0;                                  ——— 1  (Since it simple statement)
    for (i = 0; i < n; i++)                 ——— n+1 (Since its biggest one)
    {                  1   n+1   n
        s = s + A[i];                       ——— n (Since the statement for loop will
                                                   be runnins "n" amount of times)
    }
    return s;                               ——— 1
}
```

Assume Array:

A | 8 | 3 | 9 | 7 | 2 | $n = 5$ (5 elements)

0 1 2 3 4

$i = 0$

$i = 1$

$i = 2$

$i = 3$

$i = 4$

$i = 5$ X stop since $i = 5$ is Not less than N

Now Adding it all up

1
n+1
n
1

$f(n) = 2n + 3$

$O(N)$

$f(n) = 2n+3$ is $O(N)$ due to the fact that:

$2n$ - represents the algo growing linearly

3 - represents the constant

So we we look at the biggest term since it linearly grows its $O(N)$!

Space complexity using Frency count method

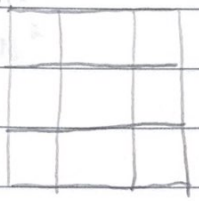
```
Algorithm Sum (A, n)
{
  S = 0;
  for (i = 0; i < n; i++)
  {
    S = S + A[i];
  }
  return S;
}
```

Space
A – n
S – 1
i – 1
n – 1

N – linearly grows
3 – Constant growth
$S(N) = n + 3 \rightarrow \boxed{O(N)}$
Since N is bigger its

Sum of two Matrices

```
Algorithm Add (A, B, n)
{
  for (i = 0; i < n; i++)  – N+1
        1      N+1    N
  {
    for (j = 0; j < n; j++)  – N+1 (N)
          1      N+1    N
    {
      C[i, j] = A[i, j] + B[i, j];  – (N) (N)
    }
  }
}
```

NXN	time complexity
ex: 3x3 	N+1
	N+1 (N)
	(N)(N)
	$f(N) = 2n^2 + 2n + 1$

Whatever is inside of a for loop will also be multipled by N, and it will execute for N times. So for nested for loops it will execute for (N)(N). 2 times!

$2n^2$ - quadratic growth

2n - linear growth

1 - Constant growth

since $2n^2$ is biggest it will be time complexity of $O(N^2)$

Space complexity

A - n2 (Since its 2 demensional arrays)

B - n^2 (Since its 2 demensional arrays)

N - 1

i - 1

j - 1

C - n^2 (Since its 2 demesional arrays)

$3n^2 + 3 = S(N)$

$O(N^2)$ - since its biggest term is n^2

Multiplication of two matrixes

```
Algorithm Multiply (A, B, n)
{
  for ( i=0; i<n; i++)  — n+1
  {
    for ( j=0; j<n; j++)  — (N) N+1
    {
      C[i,j] = 0;  — (N) (N) (1)
      for ( k=0; k<n; k++)  — (N) (N) N+1
      {
        C[i+j] = C[i+j] + A[i+k] * B[k,j];  — (N) (N) (N)
      }
    }
  }
}
```

time complexity

$N+1$

$(N)(N+1) \quad -N^2+N \;\Big]\; N^2+2N+1$

$(N)(N)(1) \quad -N^2$

$(N)(N)(N+1) \quad -N^3+N^2 \;\Big]\; 2N^3+N^2 \;\Big]\; 2N^3+2N^2$

$(N)(N)(N) \quad -N^3$

$2N^3+3N^2+2N+1$

$f(N) = 2n^3+3n^2+2n+1 \rightarrow \boxed{O(N^3)}$

Space complexity

A - N^2

B - N^2

n - 1

i - 1

j - 1

k - 1

C - N^2

Matrixes are Always going to be N^2 in space complexity no matter if they are being added or multipled!

$S(N) = 3N^2 + 4$

$\boxed{O(N^2)}$

1.5.1 time complexity

① for (i = 0; i < n; i++) - N+1

{

statement; - N

}

$f(N) = N+2 \rightarrow \boxed{O(N)}$ ✓

② for (i = n; i > 0; i--) - N (since the constants DONT matter due to the fact we only look at the biggest notation)

{

statement; - N

}

O(N) ✓

$\therefore$ = therefore

$\because$ = since

1.5.2 Time Complexity Example #2

⑤

```
for (i=1; i<n; i=i*2)
{
    Statement;
}
```

i
1
$1 \times 2 = 2$
$2 \times 2 = 4 \ (2^2)$
$4 \times 2 = 8 \ (2^3)$
⋮
⋮
2^k

Assume $i >= n$

$\because i = 2^k$

$\therefore 2^k >= n$

$2^k = n$

$k = \log_2 N$ $\quad O(\log_2 N)$

If it is incrementing like this, we can assume it will go up $\log_2 N$

BtW Reminder:

$$\log_a b = c \rightarrow a^c = b$$

NEW METHOD!

⑥

```
for (i=1; i<=n; i++)
{
  statement;
}
```

$i = \underbrace{1+1+1+1 \ldots +1}_{k} = n$

$k = n$

```
for (i=1; i<n; i=i*2)
{
  statement;
}
```

$i = \underbrace{1 \times 2 \times 2 \times 2 \ldots \times 2}_{k} = n$

$2^k = n$

$\hookrightarrow \boxed{k = \log_2 N}$

⑦

```
for (i=1; i<n; i=i*2)
{
  statement;
}
```

$n = 8$

statement; ——— $\log n$

i
1
$1 \times 2 = 2$
$2 \times 2 = 4$
$4 \times 2 = 8$ X

3 times repeats

$\log 8 = 3$

$\log_2 2^3 = 3 \quad \log_2 2 = 3$

So when we have $\log n$ for the time complexity always write it as $\lceil \log n \rceil$ so it rounds up since in this example we had done $n = 10$ there would be no value 10 it goes straight from 8 to 16. So to combat decimals we need to round up using $\lceil \rceil$.

1.5.3 Time complexity of while and if #3

Analysis of if and while

While loops continue til the condition is true, and when it becomes false it stops!

① `i=0;` —— 1
`while (i<n)` —— N+1
`{`
`Statement;` —— N
`i++;` —— N (since it is increasing everytime the loop runs)
`}`

$f(N) = 3n + 2$

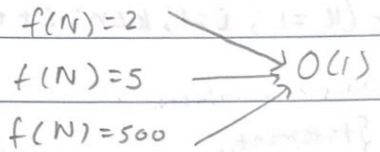
$\Theta(N)$

② `a=1;`
`while (a<b)`
`{`
`Statement;`
`a = a * 2;`
`}`

$O(\log N)$

a	terminate
1	$a \geq b$
$1 \times 2 = 2$	$\because a = 2^k$
2	$2^k \geq b$
$2 \times 2 = 4\ (2^2)$	$2^k = b$
$2^2 \times 2 = 2^3$	$k = \log_2 b$
⋮	
2^k	

1.6 Classes of functions



Types of time complexity

$O(1)$ — constant

$O(\log N)$ — logrithemic

$O(N)$ — linear

$O(N^2)$ — quadratic

$O(N^3)$ — cubic

$O(2^N)$ — exporetdial

1.7 Compare Class of Functions

Written increasing order of weightage!

$$1 < \log N < \sqrt{N} < N < N \log N < N^2 < N^3 < \ldots < 2^N < 3^N \ldots < N^N$$

	$\log N$	N	N^2	2^N
N=1	0	1	1	2
N=2	1	2	4	4
N=8	3	8	64	256

1.8.1 Asymptotic Notations Big O - omega - Theta #1

Asymptotic Notations

O	big-oh	upper bound
Ω	big-omega	lowerbound
Θ	theta	average bound

Big-oh

The function $f(n) = O(g(n))$ if and only if $\exists$ (there exists) constants C and N_0, such that $f(n) \leq C * g(n)$ $\forall$ $n \geq n_0$ (for all)

ex: $f(n) = 2n+3$

$2n+3 \leq 10n$ $\quad n \geq 1$ $\quad \therefore f(n) = O(n)$

$f(N)$ $\quad C$ $\quad g(N)$

to simplify it just do $2n+3 \leq 2n+3n \rightarrow 2n+3 \leq 5n$ for all values of $n \geq 1$

Easiest way to do this is make all of these terms, multiples of N.

you can we do $2n+3 \leq 2n^2+3n^2 \rightarrow 2n+3 \leq 5n^2$; its still true, but now big O Notation is equal to N^2

$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 \ldots < 2^n < 3^n \ldots < N^N$

lower bound ↑ Average bound upper bound

So anything Average or upper bound would work, but not lower bound.

When doing Big O, try to use the closest function!

Omega Ω Notation

The function $f(n) = \Omega(g(n))$ if and only if there is constants C and N_0, such that $f(n) \geq C * g(n)$ for all $n \geq N_0$

Example: $f(N) = 2N+3$

$2N+3 \geq 1 * N$ for all $N \geq 1$

$(f(N))$ ↑ ↑ $g(N)$

C

$\therefore f(N) = \Omega(N)$

You can use lower bounds and Average bound functions, But NOT upper bound

Make sure to use the nearest one, since that's what is most useful, and helps us out the most!

Theta Notation

The function $f(N) = \Theta(g(n))$ if and only if there exits constants C_1, C_2, and N_0, such that $C_1 * g(N) \leq f(N) \leq C_2 * g(N)$

example: $f(N) = 2N+3$

$1*N \leq 2N+3 \leq 5*N$

C_1 ↑ $g(N)$ ↑ $f(N)$ ↑ C_2 ↑ $g(N)$

$\therefore f(N) = \Theta(N)$ Average bound!

You cannot use another function other than N itself!

This is NOT the same thing as best case or worst case of an Algorithm!

1.9 Properties of Asymptotic Notations

Properties of Asymptotic Notations

General Properties

- if $f(N)$ is $O(g(N))$ then $a * f(N)$ is $O(g(N))$

  example: $f(N) = 2N^2 + 5$ is $O(N^2)$

  then $7 * F(N) = 7(2n^2+5)$

  $= 14n^2 + 35$ is $O(N^2)$

Basically if you know the Big O. of a function, and then you multiple it with something Big O, will remain the same! This property also works for Ω & Θ.

Reflexive Property

- if $f(N)$ is given then $f(N)$ is $O(f(N))$

example: $f(N) = N^2$ then $O(N^2)$

This is for upper bond in the ex, but it can be for lower bond too (Ω)

Transitive

if $f(N)$ is $O(g(N))$ and $g(N)$ is $O(h(N))$, then $f(N) = O(h(N))$

Example: $f(N) = N$ $g(N) = N^2$ $h(N) = N^3$

N is $O(N^2)$ & N^2 is $O(N^3)$ then N is $O(N^3)$

Works for all O, Ω, and Θ.

Symmetric

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

Example: $f(N) = N^2$ $g(N) = N^2$

$f(N) = \Theta(N^2)$

$g(N) = \Theta(N^2)$

ONLY For Theta Notation!

Transpose Symmetric (works only for O & Ω)

if $f(N) = O(g(N))$ then $g(N) = \Omega(f(N))$

example: $f(N) = N$ $\quad g(N) = N^2$

then h is $O(N^2)$ and N^2 is $\Omega(N)$

Other Things

① If $f(N) = O(g(N))$ and $f(N) = \Omega(g(N))$ then $f(N) = \Theta(g(N))$

if both upper and lower bound is the same, then it is same for Average bound.

② if $f(N) = O(g(N))$ and $D(N) = O(e(N))$ then $f(N) + D(N) = ?$

example:

$f(N) = N = O(N)$

$D(N) = N^2 = O(N^2)$

$F(N) + D(N) = N + N^2$

↳ $O(N^2)$

Its only $O(N^2)$ and NOT $O(N + N^2)$ since we only care about the biggest term!

③ if $f(N) = O(g(N))$ example:

and $D(N) = O(e(N))$ $F(N) = N$

then, $F(N) * D(N) = ?$ $D(N) = N^2$

$F(N) * D(N) = N * N^2 = N^3$

$\therefore O(N^3)$, just multiply it!

1.10.1 Comparison of Functions #1

N	N^2	$<$	N^3
2	$2^2 = 4$		$2^3 = 8$
3	$3^2 = 9$		$3^3 = 27$
4	$4^2 = 16$		$4^3 = 64$

2nd Method

h^2 N^3

Apply Log on both sides

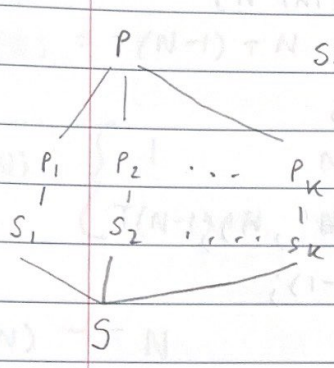
$\log N^2$ $\log N3$

$2 \log N$ $3 \log N$

$\log ab = \log a + \log b$
$\log a/b = \log a - \log 1$
$\log a^b = b \log a$
$a^{\log_c b} = b^{\log_c a}$
$a^b = n$ then $b = \log_a N$

(N^2)

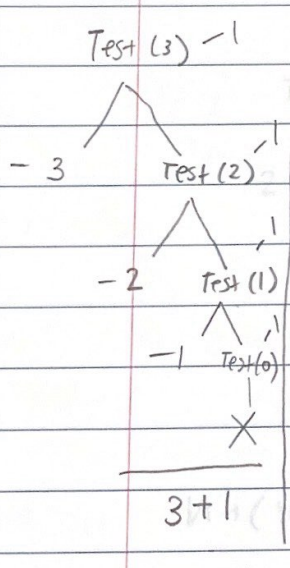
2 Divide and Conquer



So Basically we take a problem divide it up into multiple sub-problems find solutions for those problems, and then from there using those sub-problem solution combine them to find the actual solution.

It has to be same type of problem. If P is sorting the subproblems ALSO have to be solving! Its recursive in nature.

2.1.1 Recurrence Relations ($T(N) = T(n-1) + 1$) #1




```
void Test(int n)
{
  if (n > 0)
  {
    print("%d", n); ← 1 unit of time
    Test(n-1);
  }
}
```

Test (3)

f(N) = N+1 calls

O(N)

$$T(N) = \begin{cases} 1 & n = 0 \\ T(N-1) + 1 & N > 0 \end{cases}$$

$T(N)$ —— void Test (int n)

{

if (n > 0)

{

1 —— print("%d", n);

$T(N-1)$ —— Test(n-1);

}

}

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

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

$T(N) = T(N-2) + 2$

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

$T(N) = T(N-3) + 3$

continue for k times $T(N) = T(N-1) + 1$

The reason, we didn't take 1 unit of time and make it $T(N) = T(N-1) + 2$ for if (n > 0) is due to the fact we can just round up to 1 to make it easier for ourselves!

Since $T(N) = T(N-1) + 1$

$\therefore T(N-1) = T(N-2) + 1$

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

$T(N) = T(N-k) + k$ —— Assume $N - k = 0$

$\therefore N = k$

$\therefore T(N) = T(N-N) + N$

$T(N) = T(0) + N$

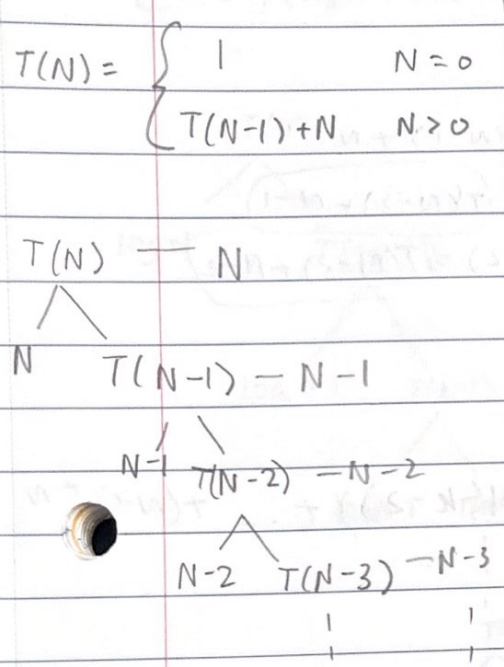
$T(N) = 1 + N$

$\boxed{\Theta(N)}$

2.1.2 Recurrence Relations ($T(N) = T(N-1) + N$) #2

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

$$T(N) = \begin{cases} 1 & N = 0 \\ T(N-1) + N & N > 0 \end{cases}$$




```
T(N) ——  Void Test (int N)
         {
1 ——       If (N>0)
           {
N+1 ——       For (i=0; i<n; i++)
             {
N ——           printf("%d", n);
             }
T(N-1) ——    Test (n-1)
           }
```

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

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

$\approx N$

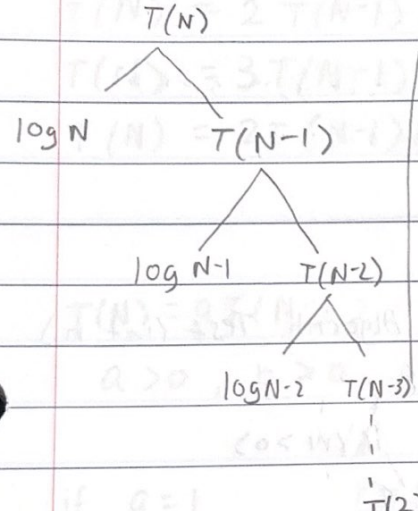
$0+1+2+3 \ldots + N-1+N = \frac{N(N+1)}{2}$

$T(N) = \frac{(N+1)N}{2} \approx N^2$

$\Theta(N^2)$

2.1.3 Recurrence Relation ($T(N) = T(N-1) + \log N$) #3

$$T(N) = \begin{cases} 1 & N = 0 \\ T(N-1) + \log N & N > 0 \end{cases}$$



$T(N)$ — Void Test (int n)

{

if (N>0)

{

$T(\log_2 N)$ — for (i=1; i<n; i=i*2)

{

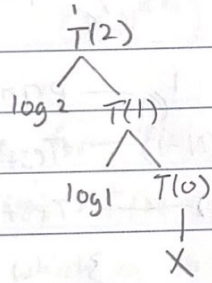
printf("%d", i);

}

$T(N-1)$ — Test (N-1);

$T(N) = T(N-1) + \log N$}

}



$\log N + \log N-1 + \log N-2 \ldots + \log 2 + \log 1$

$\log [n * (n-1) * \ldots * 2 * 1]$

$\log N!$

NO Theta

$O(N \log N)$

$T(N) = T(N-1) + 1$ — $O(N)$

$T(N) = T(N-1) + N$ — $O(N^2)$

$T(N) = T(N-1) + \log N$ — $O(N \log N)$

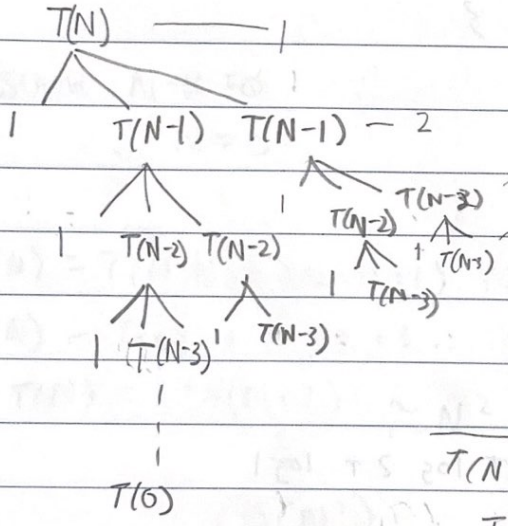
$T(N) = T(N-1) + N^2$ — $O(N^3)$

$T(N) = T(N-2) + 1$ — $\frac{n}{2}$ $O(N)$

$T(N) = T(N-100) + N$ — $O(N^2)$

$T(N) = 2\,T(N-1) + 1$

$$T(N) = \begin{cases} 1 & n = 0 \\ 2T(N-1)+1 & n > 0 \end{cases}$$




```
T(N) — Algorith Test (int n)
{
    if (N > 0)
    {
        1 — printf("%d", n);
        T(N-1) — Test (N-1);
        T(N-1) — Test (N-1);
    }
}
```

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

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

2^k Assume $N=k$, $O(2^k)$

$1 + 2 + 2^2 + 2^3 + 2^4 + \ldots + 2^k = 2^{k+1} - 1$

2.2 Masters Theorm Decreasing Function

$T(N) = T(N-1) + 1 \rightarrow O(N)$

$T(N) = T(N-1) + N \rightarrow O(N^2)$

$T(N) = T(N-1) + \log N \rightarrow O(N \log N)$

$T(N) = 2T(N-1) + 1 \rightarrow O(2^N)$

$T(N) = 3T(N-1) + 1 \rightarrow O(3^N)$

$T(N) = 2T(N-1) + N \rightarrow O(N 2^N)$

$T(N) = aT(N-b) + f(N)$

$a > 0$, $b > 0$ and $f(N) = O(N^k)$ where $k \geq 0$

if $a = 1$ $\quad O(N^{k+1})$ or $O(N * f(N))$

↳ for example for $T(N) = T(N-1) + 1$ its $1 * N$ to get $O(N)$ or for $T(N) = T(N-1) + \log N$ is $O(N \log N)$ since $(N * f(N))$ which in this case was $\log N$

if $a > 1$ $\quad O(n^k * a^{N/b})$ or $O(f(N)\, a^{N/B})$

↳ for example $T(N) = 2T(N-1) + N$ its $O(N 2^N)$ due to the fact $a = 2$, and N is in the $f(N)$ spot.

if $a < 1$ $\quad O(N^k)$

$O(f(N))$