Complexity Analysis and Recursion

Review and Master Theorem

Nattee Niparnan

Complexity Analysis Review

- We measure and compare growth of resource usage (time, memory) of the algorithm with respect to size of input (N)
  - By using growth, we focus our measurement on long term trend (large input) while disregarding unimportant detail
  - This can be done by counting major primitive instruction as a function of N and comparing the function with other function

```
int test2(vector<int> v) {
  int sum = 0;
  for (int i = 0;i < v.size();i++)
    for (int j = i+1;j < v.size();j++)
      sum += v[i] + v[j];
  return sum;
}
```

n : v.size()

Most executed line

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

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

T(n) grows similar to n^2

Counting the most executed instruction

- First, we have to identify the most executed instruction (line) in the code
  - Must be a primitive instruction (arithmetic operation, basic comparison, assignment operation but not a function call to a complex function)

```
void erase(iterator it) {
  while((it+1)!=end()) {
    *it = *(it+1);
    it++;
  }
  mSize--;
}
```

Most executed line

Mostly, it is the one in the innermost loop

ขึ้นอยู่กับข้อมูล

```
tree_iterator& operator++() {
  if (ptr->right == NULL) {
    node *parent = ptr->parent;
    while (parent != NULL &&
           parent->right == ptr) {
      ptr = parent;
      parent = ptr->parent;
    }
    ptr = parent;
  } else {
    ptr = ptr->right;
    while (ptr->left != NULL)
      ptr = ptr->left;
  }
  return (*this);
}
```

Sometime, it is separated

Counting the most executed line

- Calculate how many time it should be executed, with respect to N (size of input)
- Mostly, we derive a summation that use N

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

$$\sum_{i=0}^{n} \boxed{a^i} = \frac{1}{1-a} \quad (\text{when } a < 1)$$

$$\sum_{i=0}^{n} \boxed{c^i} = \frac{c^{n+1}-1}{c-1}$$

$$\sum_{i=0}^{n} \log(i) = \log(n!)$$

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

$\frac{1}{1-c}$

$\log(i) = \log(n!)$

Asymptotic Notation

สัญญากร เชิงเส้นกำกับ

- To emphasize on long term trend, we use asymptotic notation

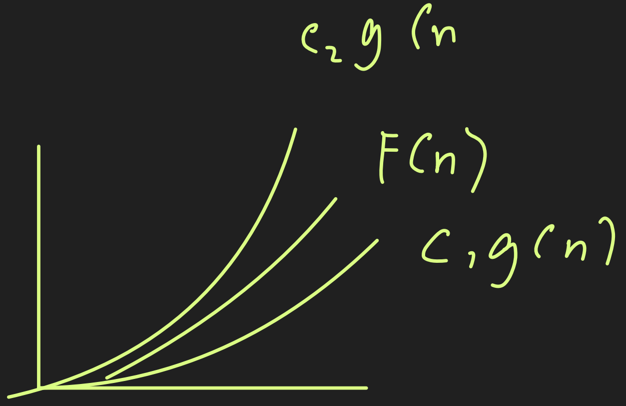
Notation	Meaning
$O(g(N))$	Set of all functions T(n) that grows not faster than g(N)
$\theta(g(N))$	Set of all functions T(n) that grows equal to g(N)
$\Omega(g(N))$	Set of all functions T(n) that grows not slower than g(N)
$o(g(N))$	Set of all functions T(n) that grows slower than g(N)
$\omega(g(N))$	Set of all functions T(n) that grows faster than g(N)

$$\boxed{\lim_{n\to\infty}\left(\frac{f(n)}{g(n)}\right)} \begin{cases} 0 \checkmark & \text{f(n) grows slower than g(n)} \\ c \checkmark & \text{f(n) grows similar than g(n)} \\ \infty & \text{f(n) grows faster than g(n)} \end{cases}$$

Another Definition

- Using set builder notation
- $O(g(n)) = \{ f(n) \mid \text{there exists } c > 0 \text{ and } n_0 >= 0$
  $\text{such that } f(n) \boxed{<=} cg(n) \text{ for } n >= n_0 \}$
- $\Theta(g(n)) = \{ f(n) \mid \text{there exists } c_1 > 0,\ c_2 > 0 \text{ and } n_0 >= 0$
  $\text{such that } c_1 g(n) <= f(n) <= c_2 g(n) \text{ for } n >= n_0 \}$

Best Case, Worst case

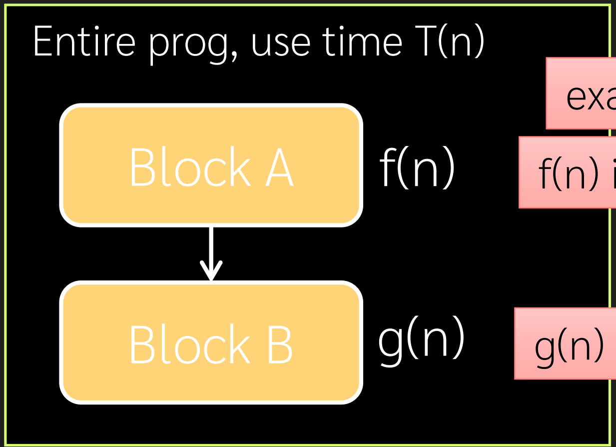


Θ

- We prefer a tight bound (θ)
- However, some cases cannot be described by a tight bound due to the nature of the algorithm and the input
  - We use upper bound (O) to simplify
  - Instead of saying that a function grows exactly as g(n), we say that that function does not grows beyond g(n) which is the worst case
  - Example, insertion sort, it is possible that the algorithm runs very fast (best case in O(n)) but its worst case is O(n^2)

Some shortcut from code ทางลัด (ไม่ดี) → ข้อยกเว้นเยอะ

- Beware! Exceptions happens
- Let h(n) is the instruction count of the entire program which has several Block-X



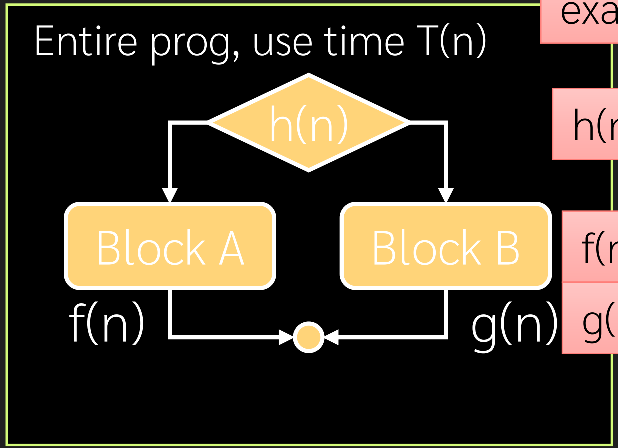
example

f(n) is $\theta(n)$

g(n) is O(n^2)

Sequential
O(f(n) + g(n))

T(n) = O($\theta(n)$ + O(n^2))
= O(n^2)



example

h(n) is $\theta(n^2)$

f(n) is O(n^2)

g(n) is O(n^2)

T(n) = $\theta(n^2)$

Conditional
max(h(n)+f(n), h(n) + g(n))

Some shortcut from code

- Simply count the time the innermost operation happens with respect to N
- Remove multiplicative constant and any other term with lesser growth
  - $T(N) = 5n^3-2n+4$ is $O(n^3)$
  - $T(N) = n * (n + \log n)$ is $O(n^2)$
  - $T(N) = n \log n/n + \sqrt{n}$ is $O(\sqrt{n})$

ดูเทอม
ที่ใหญ่
สุด

Example : Euclid’s GCD

```
def gcd(a, b) {
  while (b > 0) {
      tmp = b
      b   = a mod b
      a   = tmp
  }
  return a
}
```

assume b มากกว่า a

Most executed line

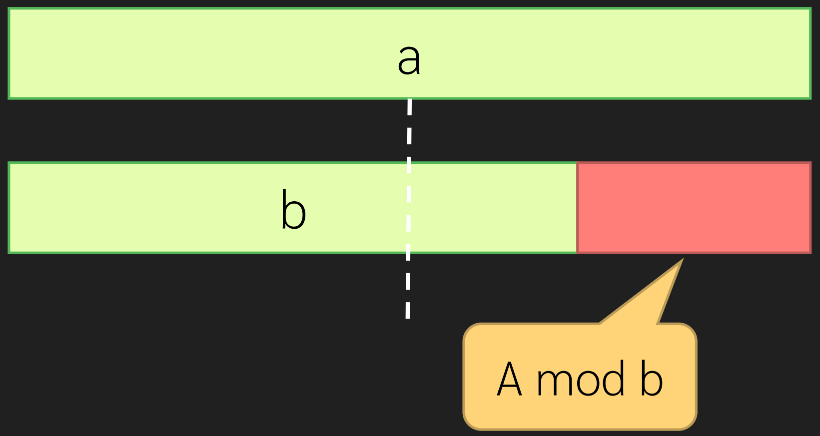
- How many iteration?
  - While loop runs until the result of mod is 0
  - The divisor (b) changes into dividend (a)
  - The remainder (a mod b) changes into the divisor (b)
- Proposition: The number of iteration is, at most, log(~~max~~ min(a,b))
  - Because b reduces by <u>at least half</u>

How many iteration?

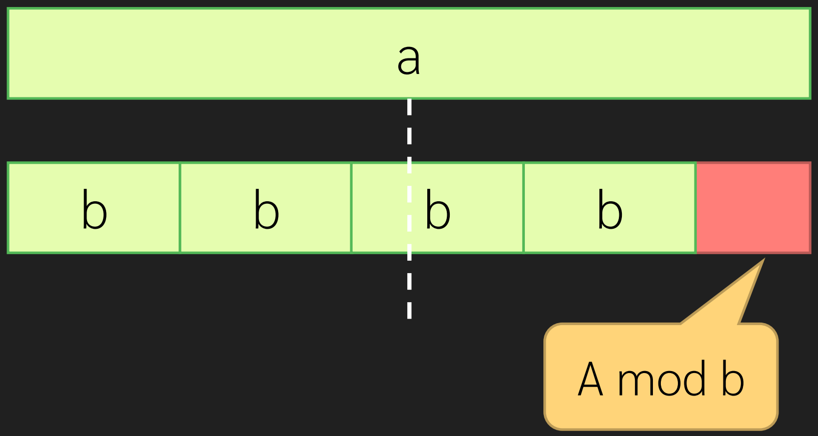
O(lgn)

```
def gcd(a, b) {
  while (b > 0) {
      tmp = b
      b   = a mod b
      a   = tmp
  }
  return a
}
```

Case 1: b > a/2



Case 2: b < a/2



- Remainder becomes divisor
- The divisor (b) reduce by at least half
- Loop until divisor is zero, Hence O(log n)

Recursive Program and Analysis

Recursive Program

- Function that calls itself
- Has 2 parts
  - Separate by terminating condition
  - Terminating case (trivial case)
    - The case that has no recursion
    - Without this one, program will always call itself and never finish
  - Recursion case
    - The case that call itself (sub-problem)

Terminating condition

```
// calculate sum 0..n
int recur1(int n) {
  if (n <= 0) {
    // terminating case
    return 0;
  } else {
    // recursion case
    return recur1(n-1) + n;
  }
}
```

Different parameter, going toward termination

Example

```
void draw_tri(int level,int max) {
  if (level <= max) {
    for (int j = 0;j < level;j++)
      printf("*");
    printf("\n");
    draw_tri(level + 1,max);
  }
}
```

```
draw_tri(1,5):
*
**
***
****
*****
```

```
draw_tri(1,3):
*
**
***
```

Actual work

$$T(n) = \begin{cases} n + T(n+1) & ; n \le max \\ 0 & ; n > max \end{cases}$$

n equal to "level"

Recursive

- There is a terminating case, it does nothing
- What is the time complexity?
- For recursive program, usually T(N) is a recurrence relation, consisting of two parts
  - The actual work parts and recursive part
  - The initial condition for the recurrence relation depends on the terminating case of the recursive

Calculating Instruction Count function of a recursive program

รู้มาดังเยน

- Method 1: Using (non-)homogeneous linear recurrence relations (with knowledge from Discrete Structure class)
- Method 2: Using summation and substitution method
  - With help of recursion tree

$T(n) = aT(\sim)$

- Method 3: Use a Master Method (or Master Theorem)

เน้น

Master Method is easiest but not applicable to all cases

Example: Recursive GCD using Summation

$$\cancel{T(b)} = \cancel{T\left(\frac{b}{2}\right)} + 1$$
$$\cancel{T\left(\frac{b}{2}\right)} = \cancel{T\left(\frac{b}{4}\right)} + 1$$
$$\cancel{T\left(\frac{b}{4}\right)} = \cancel{T\left(\frac{b}{8}\right)} + 1$$
$$\ldots$$
$$\cancel{T\left(\frac{b}{2^{k-1}}\right)} = \cancel{T\left(\frac{b}{2^k}\right)} + 1$$
$$\cancel{T\left(\frac{b}{2^k}\right)} = 1$$

$2^k > b$

```
def gcd(a, b)
  if (b == 0)
    return a
  return gcd(b, a mod b)
end
```

worse case

$$T(b) = \begin{cases} T(b/2) & ; b > 0 \\ 1 & ; b = 0 \end{cases}$$

Assume worst case:
b reduce by half every time

Sum both sides of the equation,
Recursive terms cancel out

Result is T(b) = 1 log(b)

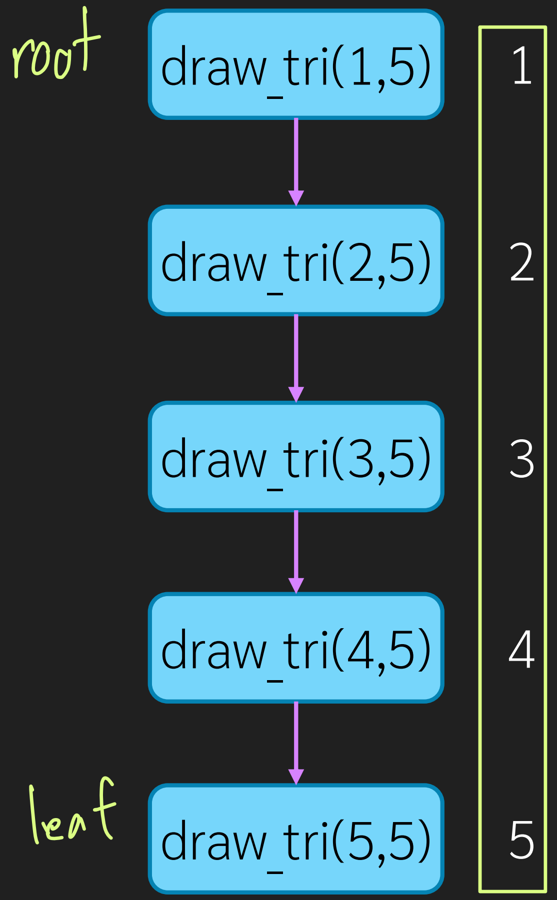
Recursion Tree

- A tree that helps understanding recursive program (and its recurrence relation)
- A node is each function call
  - Describe a parameter of a function in the node
  - Root node is the first call to this recursive function
  - Leaves nodes are ones that is terminating case (no more recursive call)
- An edge is a function call
  - Children of each node is a function that were called by this node
  - Order the children according to the order of call
- Write actual works done by each node and sum it

Recursion Tree Example

```
draw_tri(1,5)
```

Actual work



$$
\begin{aligned}
T(1) &= \cancel{T(2)} + 1\\
\cancel{T(2)} &= \cancel{T(3)} + 2\\
\cancel{T(3)} &= \cancel{T(4)} + 3\\
&\;\;\dots\\
T(\cancel{max}) &= T(\cancel{max} + 1) + n\\
T(\cancel{max} + 1) &= 0
\end{aligned}
$$

```
void draw_tri(int level,int max) {
  if (level <= max) {
    for (int j = 0;j < level;j++)
      printf("*"  );
    printf("\n");
    draw_tri(level + 1,max);
  }
}
```

$$T(n) = \begin{cases} n + T(n+1) & ; n \le max \\ 0 & ; n > max \end{cases}$$

n is “level”

Let n be equal to max

Sum of all works = 1+2+…+max

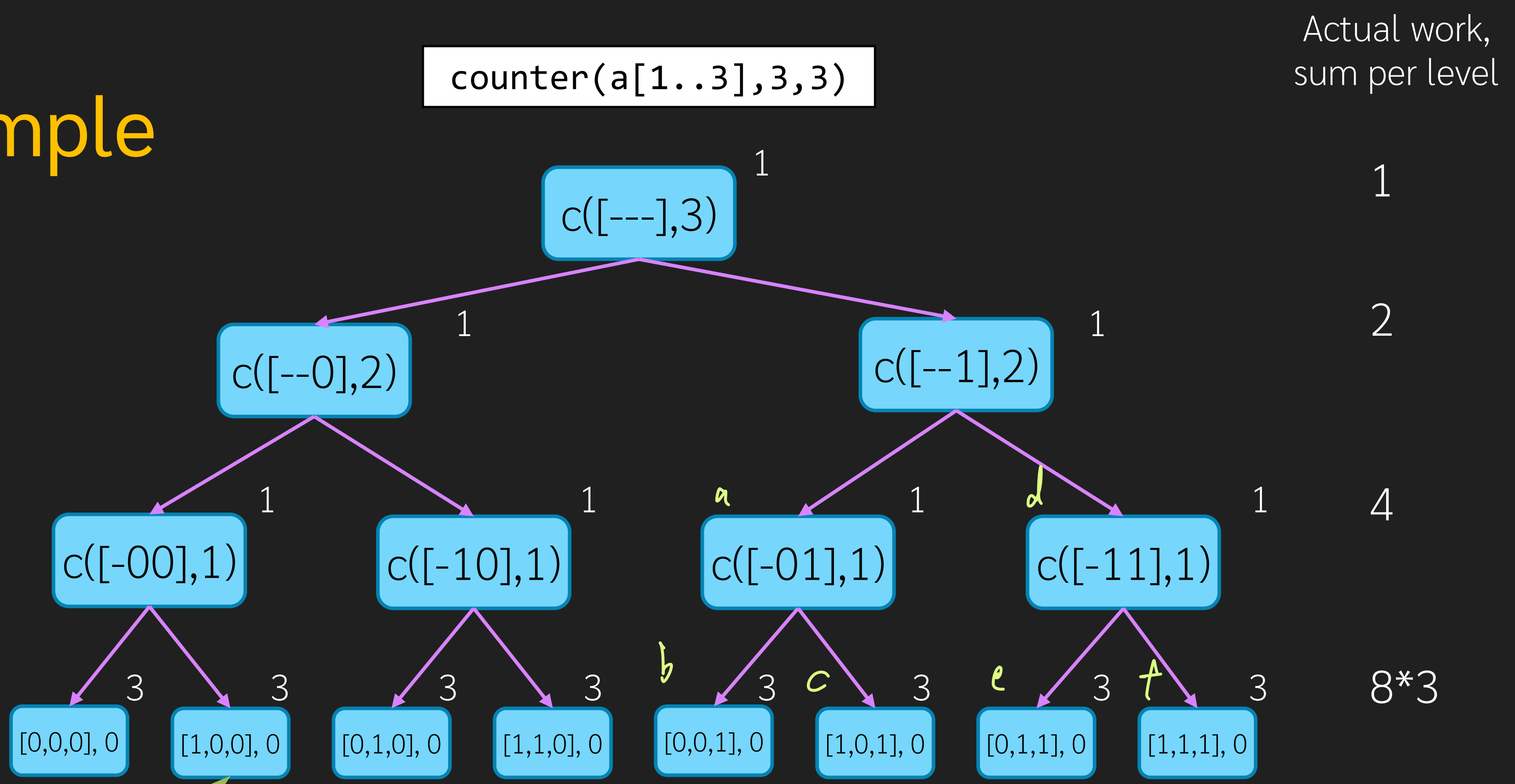
$$\sum_{1}^{max} i = \frac{n(n+1)}{2} = \theta(n^2)$$

Another Example

- Binary Counter

```
def counter(v[1..n], i, n)
  if (i > 0)
    v[i] = 0
    counter(v,i-1,n)

    v[i] = 1
    counter(v,i-1,n)
  else
    print v
  end
end
```



Shorthand version, write only relevant parameters, v and i

actual work of each node is 1 (no work)

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

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

$T(n) = \theta(n2^n)$

Solving binary counter with Method 2

$$
\begin{aligned}
T(n) &= 2T(n-1) + 1 \\
2T(n-1) &= 4T(n-2) + 2 \\
4T(n-2) &= 8T(n-3) + 4 \\
&\dots \\
2^{\mathrm{i}}T(n-i) &= n^{i-1}T(n-i-1) + 2^{i} \\
&\dots \\
2^{\mathrm{n}-1}T(n-(n-1)) &= 2^{n}T(n-n) + 2^{n-1} \\
2^{n}T(n-n) &= \boxed{2^{n} * n}
\end{aligned}
$$

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

Sum both sides of the equation,
Recursive terms cancel out

Result is $T(n) = \boxed{\sum_{i=0}^{n-1} 2^{i}} + \boxed{2^{n} * n}$

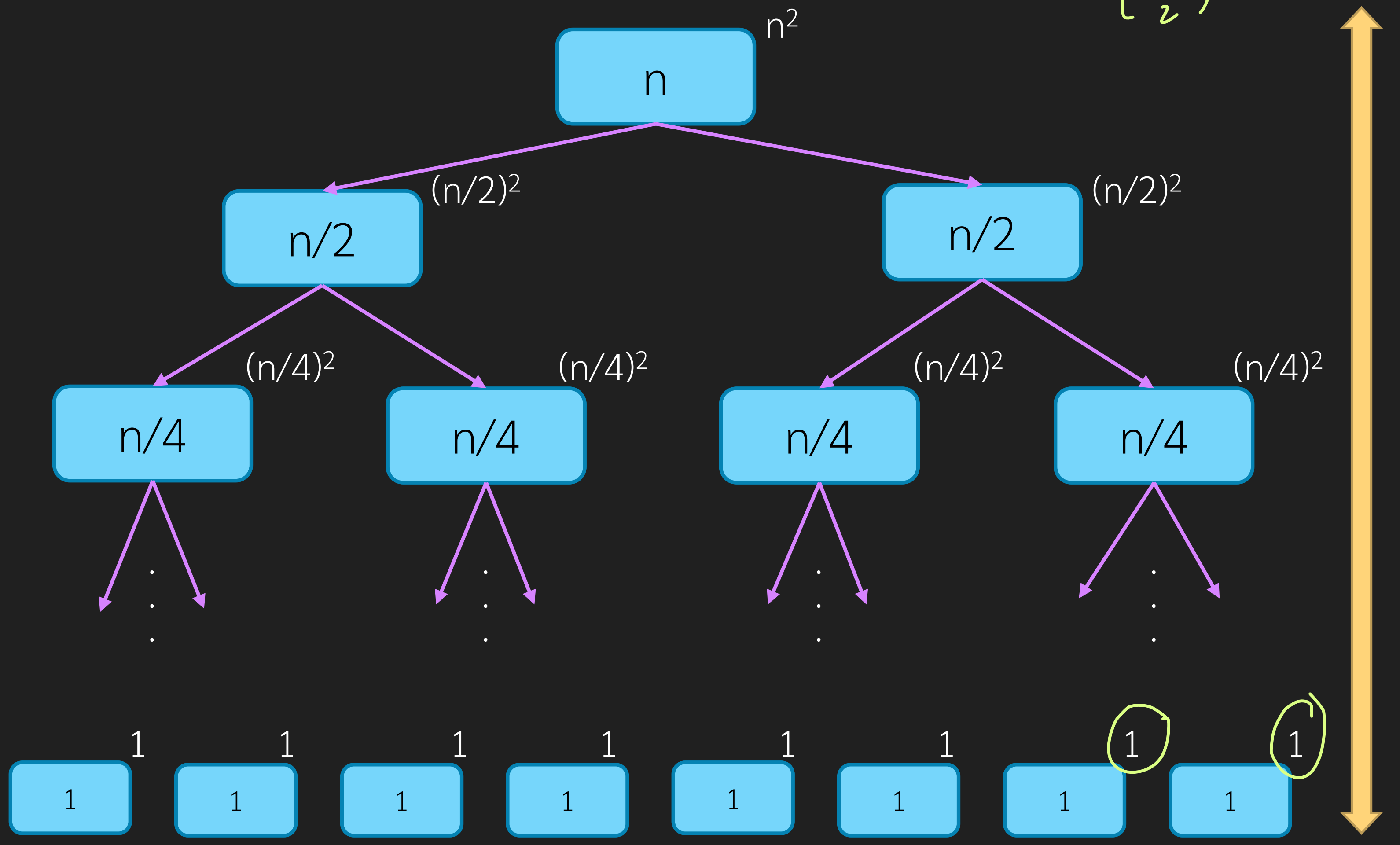
$$
= 2^{n} - 1 + 2^{n} * n
$$

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

More Example

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

$\left(\frac{n}{2}\right)^2$



There is lg(n) level

Actual work, sum per level

n^2

$n^2/2^1$

$n^2/2^2$

$\vdots$

n

$$\sum_{i=0}^{\log_2 n} \frac{n^2}{2^i} = n^2 \boxed{\sum_{i=0}^{\log_2 n} \frac{1}{2^i}}$$

$<= 2$

$$T(n) = \theta(n^2)$$

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

$F(1) = 1$

$F(0) = 0$

Master Method

$T(n) = aT(\frac{n}{b}) + \theta(n^d)$ $c = \log_b a$

$a \geq 1$
$b > 1$
$d \geq 0$

$T(0) = 1$

- Shortcut in solving some recurrent relation in this form

$$T(n) = aT(n/b) + \theta(n^d)$$

With following condition:

a >= 1

b > 1

d >= 0

$\boxed{T(0) = 1}$

$n^2 \log n$

$$T(n) = \begin{cases} \theta(n^c) & ; n^d < n^c \\ \theta(n^c \log n) & ; n^d = n^c \\ \theta(n^d) & ; n^d > n^c \end{cases}$$

เอาตัวมาก

$n^2 > n^{\log_4 (\cdot)}$

Where c = $\log_b(a)$

$\log_2 2 = 1$

Relation	c	case	result
T(n) = 2T(n/2)+n	c = $\log_2 2$ = 1	(2)	$\theta(n \log n)$
T(n) = 1T(n/2)+n	c = $\log_2 1$ = 0	(3)	$\theta(n)$
T(n) = 10T(n/3)+n^2 (d)	c = $\log_3 10$ > 2	(1)	$\theta(n^{\log_3 10})$
T(n) = 4T(n/2)+n^2	c = $\log_2 4$ = 2	(2)	$\theta(n^2 \log n)$

Exception to Master Method

- $T(n) = 2T(n\text{-}1)$
  - Size of N does not scale as a ratio
- $T(N) = 3T(n/4) + 6T(n/8) + 1$

  4 ไม่ได้
  - Summation of different size
- $T(N) = 2nT(n/3) + n$

  ไม่ได้
  - Number of sub-problem is not constant

Actual Version of Master Method

$f(n)$

$n^d < n^c$

$0 < \epsilon$ ขอให้โตกว่าอะไรซักอย่าง

$$T(n) = aT(n/b) + f(n)$$

With following condition:

a >= 1

b > 1

T(0) = 1

$$T(n) = \begin{cases} \theta(n^c) & ; f(n) = O(n^{c-\epsilon}) \\ \theta(n^c \log^{k+1} n) & ; f(n) = \theta(n^c \log^{k} n) \\ & ; f(n) = \Omega(n^{c+\epsilon}) \\ \theta(n^d) & af(n/b) \le kf(n) \\ & k < 1, n > n_0 \end{cases}$$

โตไม่ช้าไปกว่า

Where c = $\log_b(a)$

แก้ $T(n) = \sim + n \lg n$ ได้

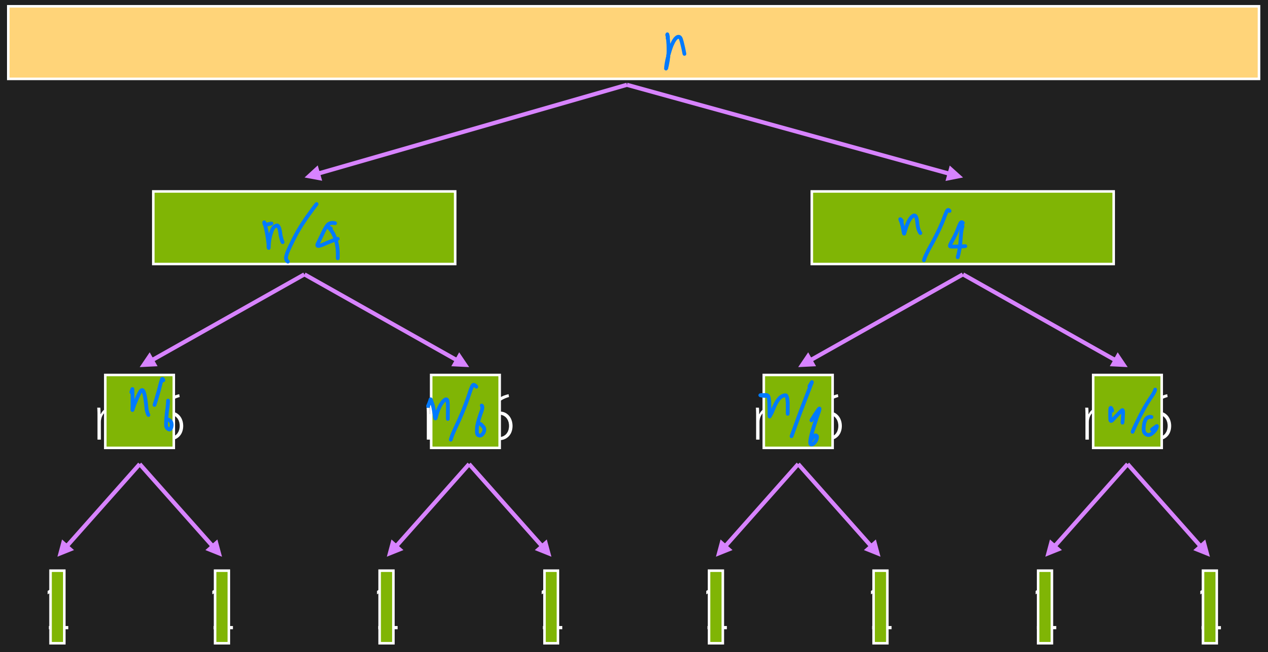
Insight of Master Method

$$T(n) = \underbrace{aT(n/b)}_{\text{How size changes at each level}} + \underbrace{\theta(n^d)}_{\text{Actual work}}$$

- 3 cases
  - When actual work dominates, the result depends on actual work
  - When recursion dominates, the result depends on recursion
  - When both are critically equal, special case

Example

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



Actual work, sum per level

n

$2n/4^1$

$4n/4^2$

$2^k n/4^k$

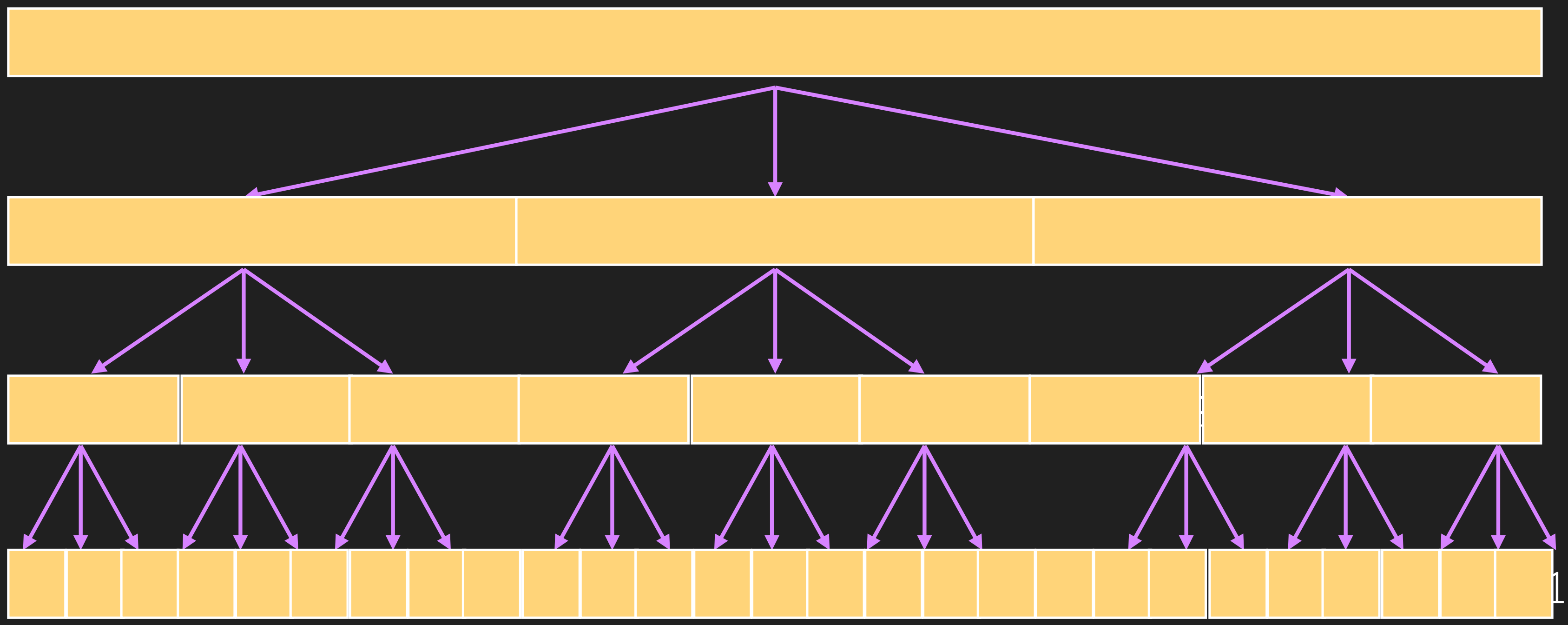
There is k = $\log_4(n)$ level

$$n \sum_{i=0}^{\log_4 n} \frac{2^i}{4^i} = n \overbrace{\sum_{i=0}^{\log_4 n} \frac{1}{2^i}}^{2}$$
$$\leq 2n$$
$$= \theta(n)$$

Example

$$T(n) = 3T\left(\frac{n}{3}\right) + n$$

Actual work,
sum per level



n

$3n/3^1$

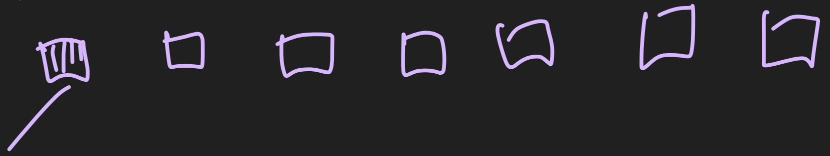
$3^2n/3^2$

$3^kn/3^k$

$$\sum_{i=0}^{\log_3 n} \frac{3^i n}{3^i} = n \sum_{i=0}^{\log_3 n} 1$$
$$= \theta(n \log n)$$

$T(n) = 16T\left(\frac{n}{4}\right) + n^2$

n^2



$n^2 \sum_{0}^{\log_4 n} \frac{16^n}{4^{2n}} = n^2 \log n$

$16 \left(\frac{n}{4}\right)^2$

$16^2 \left(\frac{n}{4 \cdot 4}\right)^2$ 4^2

$\frac{n^2}{(4^2)^2}$

Recursion Tree Summary

- Writing Recursion Tree helps us understand relation between recursion and actual work
- Help us solve recurrence relation by substitution and summation
- Use Master Method when the recurrence relation allow