Complexity Analysis and Recursion

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)

memory

  - 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

(ไม่เน้นในวิชานี้)
space complex ของ AVL เท่ากับ O(n)
ขึ้นกับจำนวนinput

```cpp
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;
}
```

Most executed line

n = v.size()

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

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

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

```
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

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

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

เน้น

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

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

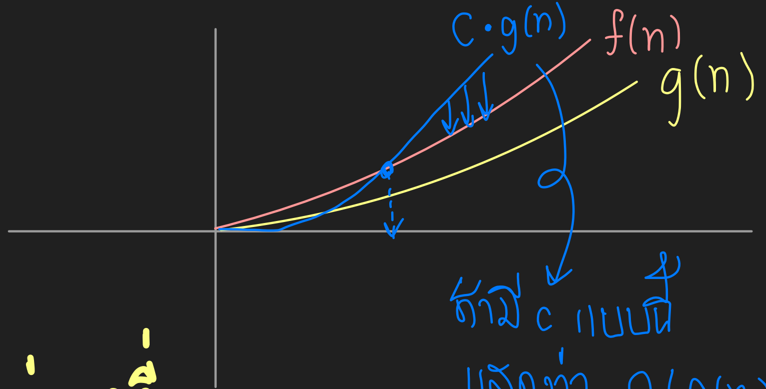
Asymptotic Notation

- 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)

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

Another Definition



แสดงว่า $O(g(n)) = \{f(n)\}$

- Using set builder notation

ค่าคงที่

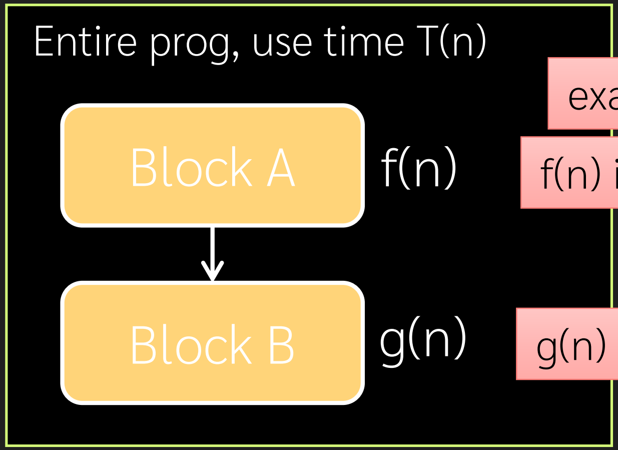
- $O(g(n)) = \{\, f(n) \mid$ there exists $c > 0$ and $n_0 >= 0$ such that $f(n) <= cg(n)$ for $n >= n_0 \,\}$
- $\Theta(g(n)) = \{\, f(n) \mid$ there exists $c_1 > 0$, $c_2 > 0$ and $n_0 >= 0$ such that $c_1 g(n) <= f(n) <= c_2 g(n)$ for $n >= n_0 \,\}$

Best Case, Worst case

- 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

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



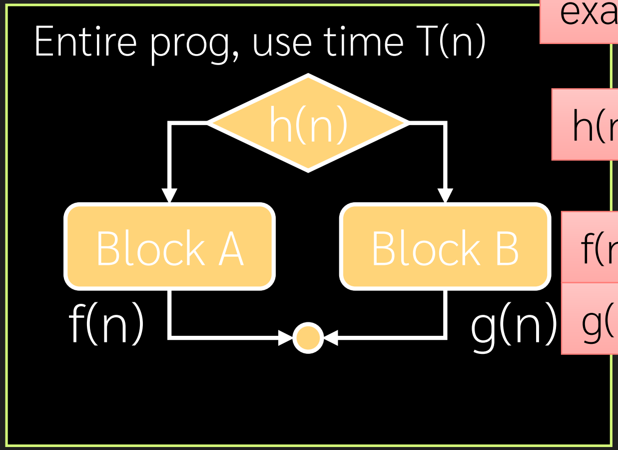
example

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

g(n) is $O(n^2)$

Sequential
T(n) = 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
T(n) = max(h(n)+f(n), h(n) + g(n))

$\max\left(\cancel{\Theta(n^2) + O(n^2)}, \Theta(n^2) + O(n^2) \right)$ $\Theta(n^2)$

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 $O(\sqrt{n})$

Example : Euclid’s GCD

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

Most executed line

หาแผน Euclid

a จะมากกว่า b (ถ้า b<a มันจะทำการสลับค่า)

a b tmp = 6

10 6

ค่าของ a mod b 4

6 4

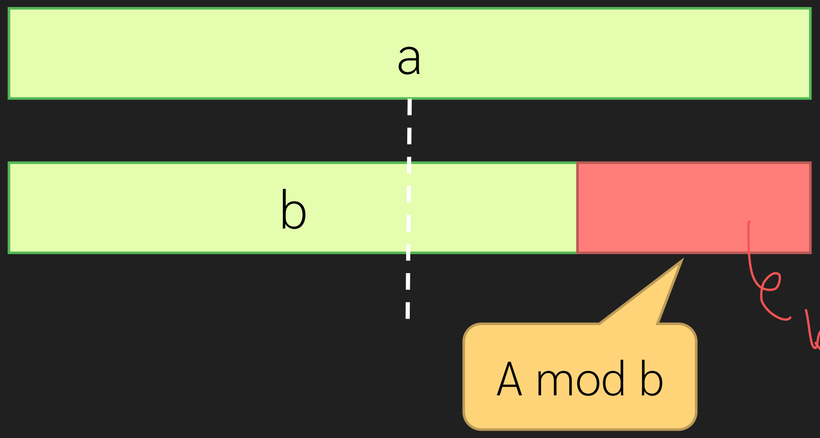
- How many iteration?
  - While loop runs until the result of mod is 0
  - The divisor (b) changes into dividend (a)
  - The remainder (a mod b) changes into the divisor (b)
- Proposition: The number of iteration is, at most, log(~~max~~ min(a,b))
  - Because b reduces by at least half

กรณีเลวร้ายสุด คือ b ลดลงทีละ "หาร 2"

How many iteration?

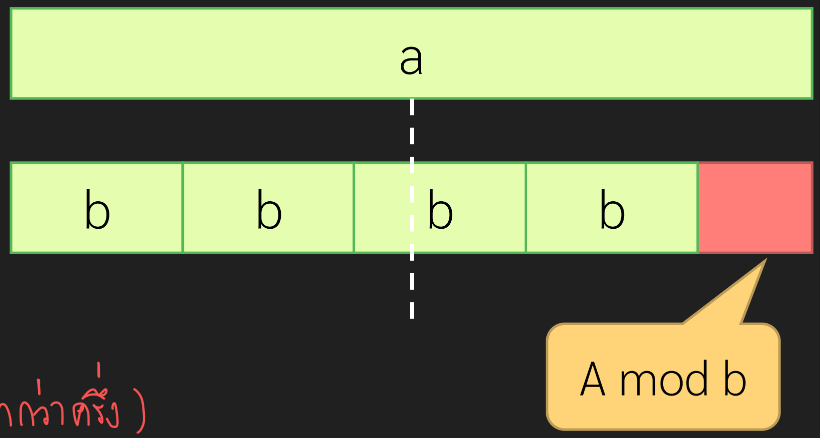
```
def gcd(a, b) {
  while (b > 0) {
      tmp = b
      b   = a mod b
      a   = tmp
  }
  return a
}
```

Case 1: b > a/2



b ในรอบถัดไป
(สังเกต! ลดลงมากกว่าครึ่ง)

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

- 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

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

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

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

Actual work

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

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

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

Master Method is easiest but not applicable to all cases

Example: Recursive GCD using Summation

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

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

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

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

draw_tri(1,5)	1
draw_tri(2,5)	2
draw_tri(3,5)	3
draw_tri(4,5)	4
draw_tri(5,5)	5

$$T(1) = T(2) + 1$$
$$T(2) = T(3) + 2$$
$$T(3) = T(4) + 3$$
$$\dots$$
$$T(max) = T(max + 1) + n$$
$$T(max + 1) = 0$$

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

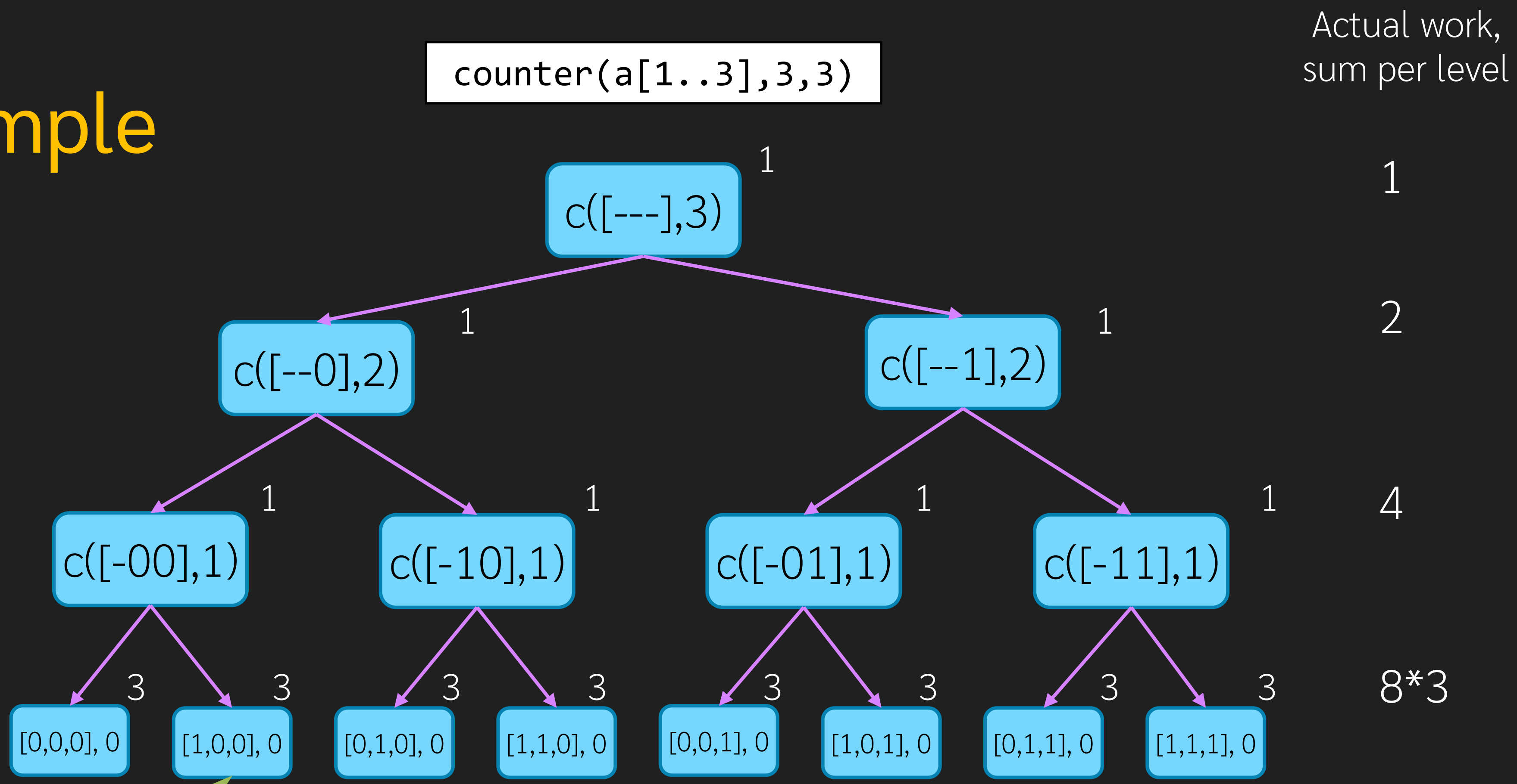
$$\sum_{1}^{max} i = \frac{n(n+1)}{2} = \theta(n^2)$$

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

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

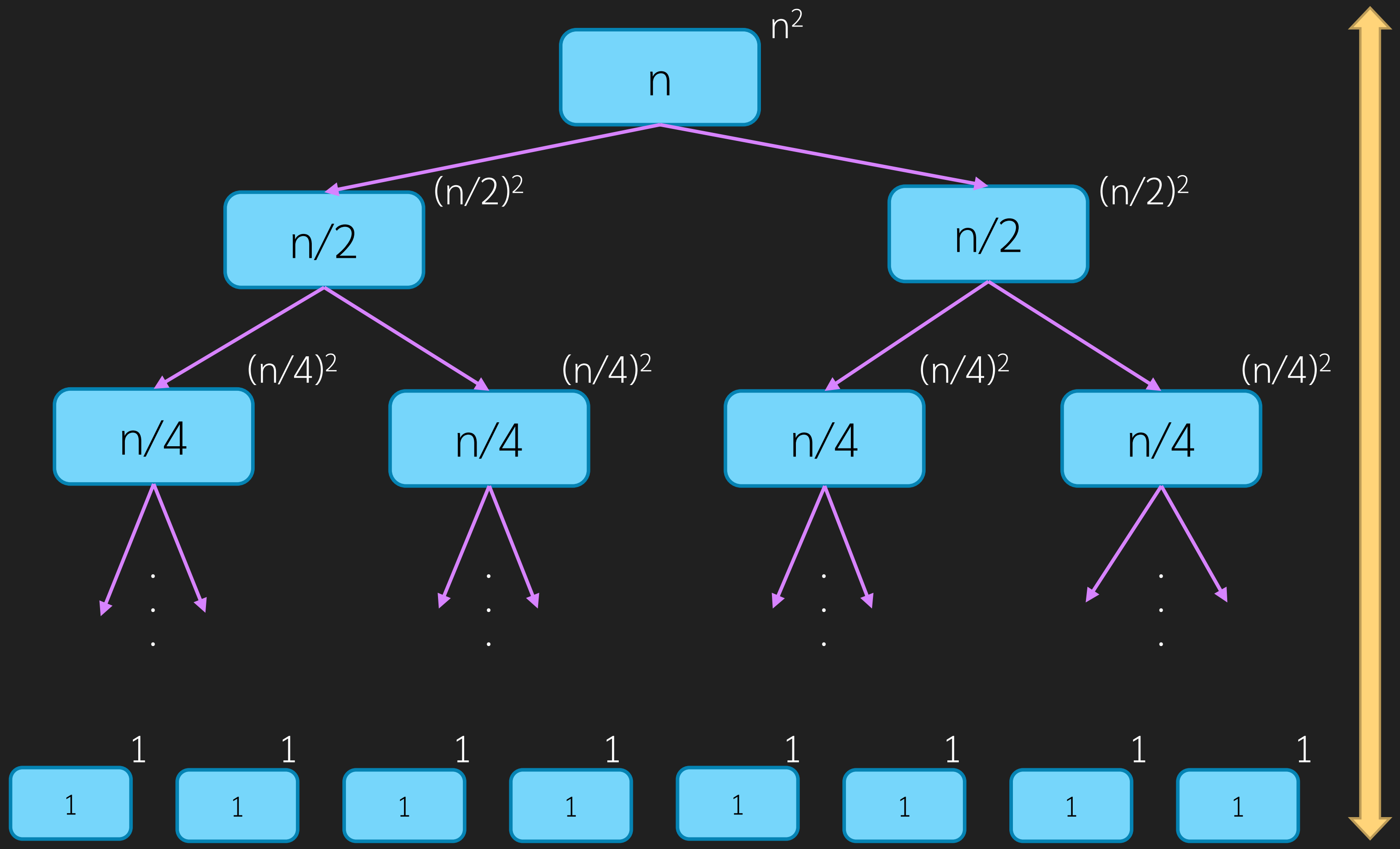
$$\begin{aligned} T(n) &= 2T(n-1)+1 \\ 2T(n-1) &= 4T(n-2)+2 \\ 4T(n-2) &= 8T(n-3)+4 \\ &\ldots \\ 2^{\mathrm{i}}T(n-i) &= n^{i-1}T(n-i-1)+2^{i} \\ &\ldots \\ 2^{\mathrm{n}-1}T(n-(n-1)) &= 2^{n}T(n-n)+2^{n-1} \\ 2^{n}T(n-n) &= 2^{n} * n \end{aligned}$$

Sum both sides of the equation,
Recursive terms cancel out

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

More Example

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



There is lg(n) level

Actual work,
sum per level

n^2

$n^2/2^1$

$n^2/2^2$

⋮

n

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

<= 2

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

Master Method

- 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

T(0) = 1

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

Where c = $\log_b(a)$

Relation	c	case	result
T(n) = 2T(n/2)+n	c = $\log_2 2$ = 1	(2)	$\theta(n \log n)$
T(n) = T(n/2)+n	c = $\log_2 1$ = 0	(3)	$\theta(n)$
T(n) = 10T(n/3)+n^2	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

- $T(n) = 2T(n-1)$
  - Size of N does not scale as a ratio
- $T(N) = 3T(n/4) + 6T(n/8) + 1$
  - Summation of different size
- $T(N) = 2nT(n/3) + n$
  - Number of sub-problem is not constant

Actual Version of Master Method

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

With following condition:
a >= 1
b > 1
T(0) = 1

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

Where c = $\log_b(a)$

Insight of Master Method

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

How size changes at each level

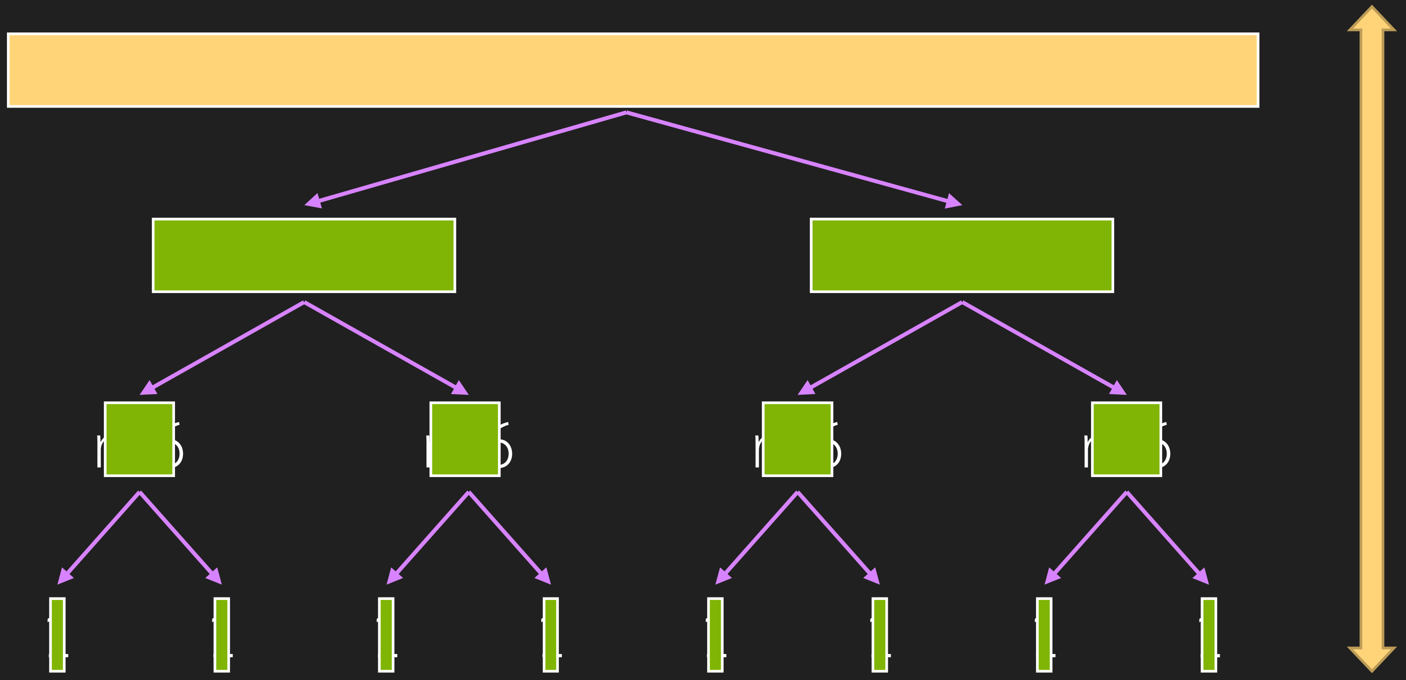
Actual work

- 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

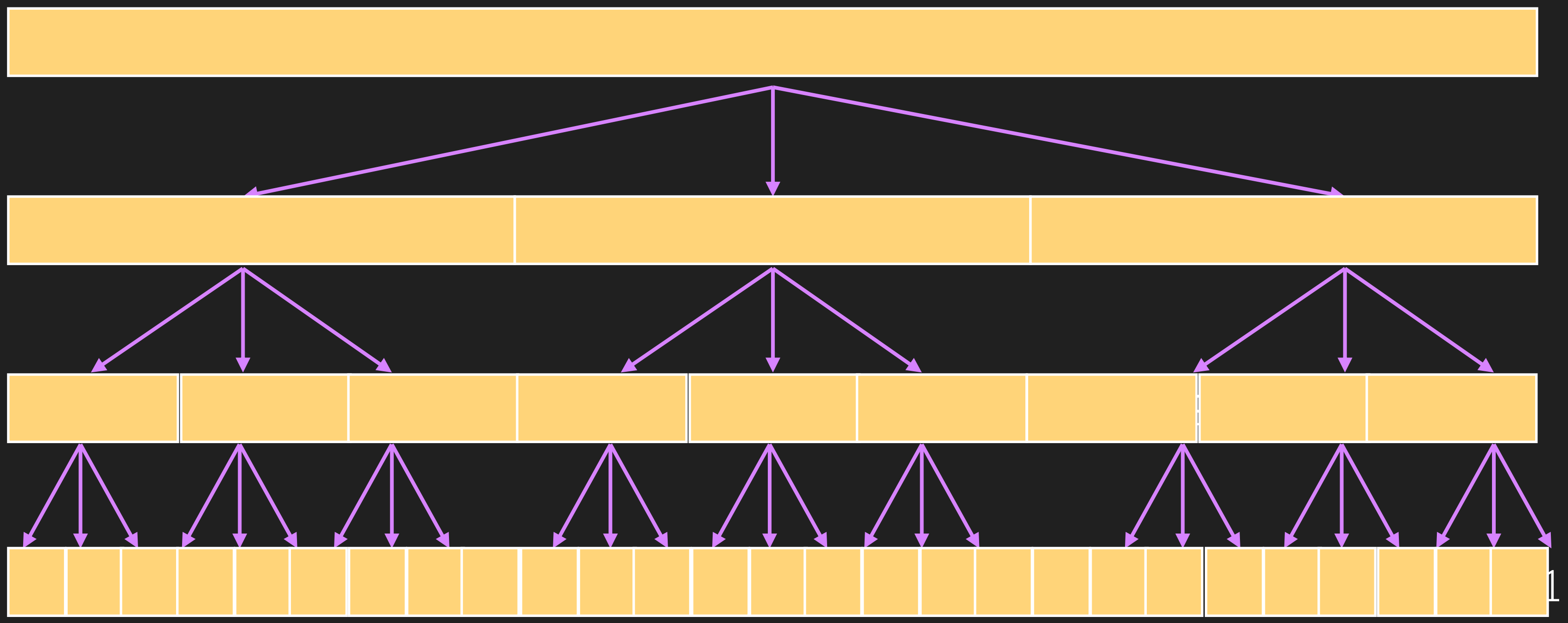
2^{k}n/4^k

There is k = log$_4$(n) level

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

Example

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



Actual work, sum per level

n

$3n/3^1$

$3^2n/3^2$

$3^kn/3^k$

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

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