Karatsuba Algorithm Week 1

$$X = \underset{a}{56}\ \underset{b}{78}$$
$$Y = \underset{c}{12}\ \underset{d}{34}$$

Step1: $a \cdot c$ --- ①

Step2: $b \cdot d$ --- ②

Step3: $(a+b)(c+d)$ = -- ③

Step4: ③ - ② - ① =

Step5: ① × 10000 + ② + ③ × 100 = $X \cdot Y$

⇒ There is better algorithm to solve question even for simple integer mutiplication

Recursive Algorithm for multiplying single digitnumber

```
int multiply(int x, int y){
    if (x==0|| y==0) return 0
    return x+ multiply(x, y-1)
```

e.g. $x=2$ $y=2$

1st: multiply (2,2) → 4
  return 2+ multiply(2,1)

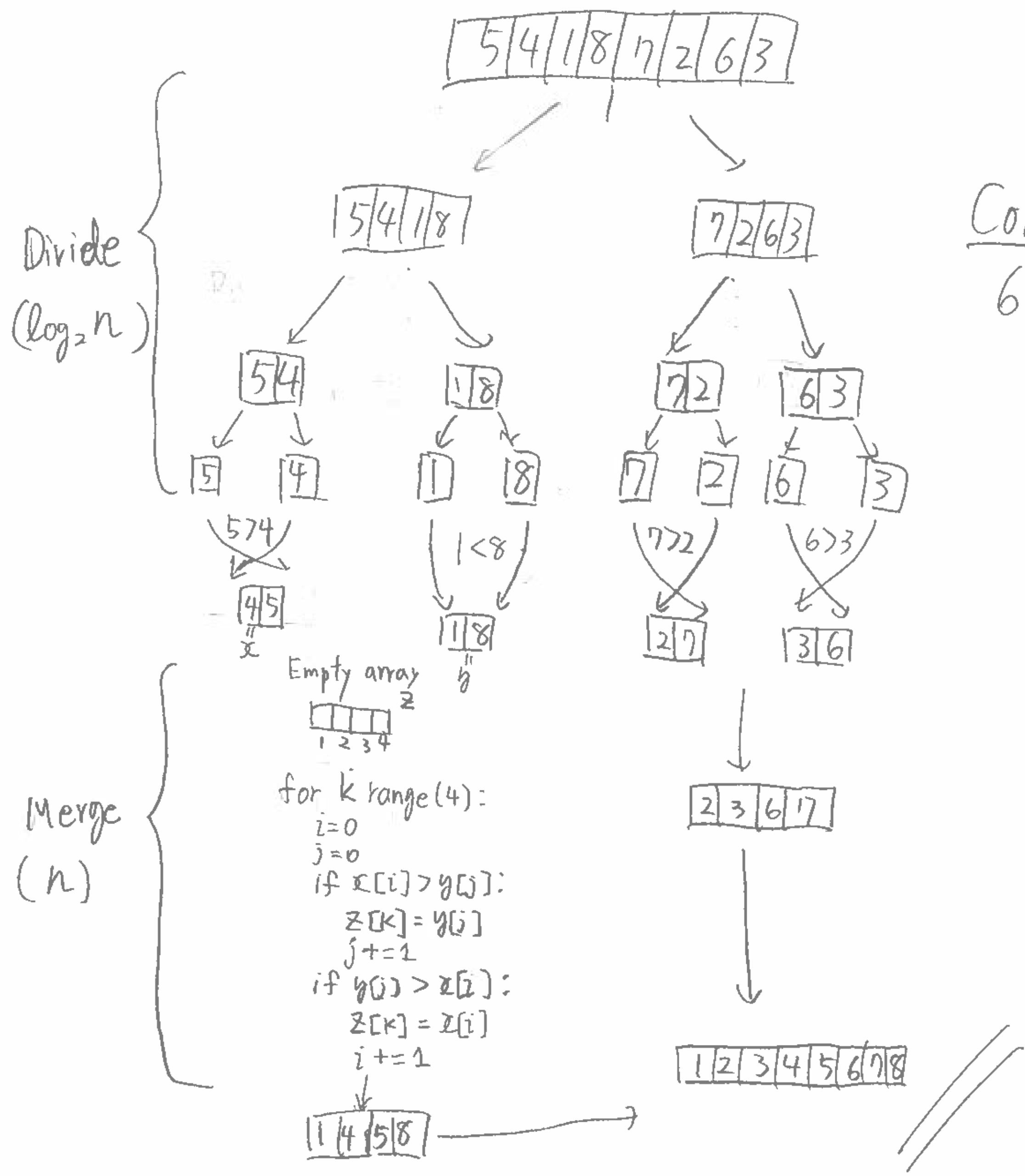
2nd: multiply (2,1) → 2
  return 2+ multiply (2,0)

3rd: multiply (2,0) → 0
  return 0

merge sort

- One of the good example of "Devide & Conquer" paradigm.
- Better than "Selection sort", "Insertion sort" and "Bubble sort" which all have complexity of n^2.

Merge sort is recursive algorithm



Complexity

$6n \log_2 n + 6n$

```
for k range(4):
    i=0
    j=0
    if x[i] > y[j]:
        z[k] = y[j]
        j+=1
    if y[j] > x[i]:
        z[k] = x[i]
        i+=1
```

Type of analysis

① Worst case analysis ··· Longest running time (Upper bound)
② Average analysis ··· Average running time under distributed input
③ Bench marks ··· Practical input running time

Require domain knowledge.

What we care

High order term > Low order term // Constant term

① Easier
② Constants depends on architecture/Compiler
③

Asymptotic analysis : Focus on large input

e.g. : $\underbrace{6n \cdot \log_2 n + 6n}_{\text{merge sort}}$ better than $\underbrace{\frac{1}{2}n^2}_{\text{Insertion sort}}$

will be satisfied if and only if (iff) n is sufficiently large.

Fast Algorithm

⇒ Worst-case running time grows slowly with an input size.

Big O notation (Way to design algorithm)

Why using asymptotic analysis?

⇒ Supress the constant and low-order terms.

Too system depend — irrelevant for large input

e.g. Merge sort's $6n\log_2 n + 6n$ turn to be $n \cdot \log_2 n$.

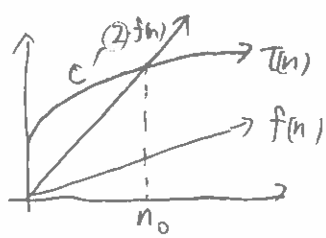
$O(n \cdot \log_2 n)$

- If $T(n) = a_k \cdot n^k + a_{k-1} \cdot n^{k-1} + \cdots a_1 \cdot n + a_0$

$T(n) = O(n^k)$ — Only the highest order matters.

- $T(n) = 2^{n+10} = O(2^n)$

English: T(n) will be bounded above by constant multiple of f(n)



Upper bound of T(n) is higher than f(n) but lower than 2·f(n)

⇒ $T(n) = O(f(n))$

Formal:

$T(n) = O(f(n))$ will be established iff there exist constants c, n_0 such satisfy

$$T(n) \le c \cdot f(n) \text{ for all } n > n_0$$

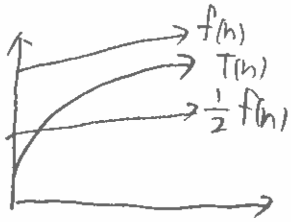
The point that an input become sufficiently large

※ C and n_0 are independent from n //

Ω Notation

$T(n) = \Omega(f(n))$ iff exist constants c, n_0 such that

$T(n) \geq c \cdot f(n) \quad \forall n \geq n_0$



$\Rightarrow$ Tell the limit of how fast is function can be.

θ Notation

$T(n) = \theta(f(n))$ iff $T(n) = O(f(n))$ and $T(n) = \Omega(f(n))$

$\Rightarrow C_1 \cdot f(n) \leq T(n) \leq C_2 \cdot f(n)$

When the Big O and Ω Notation is same.

$\Rightarrow$ How slow can be and how fast can be is same.