Lab 3
Assembly Lab II
Exercise & Report Format

Video Link : https://youtu.be/XG8P4stxnco

Outline

1. Exercise Overview
2. Exercise 1 ~ 4
3. Format of the lab report
4. File Structure of submission
5. Appendix : Introduction of merge sort

Exercise Overview

CO2021 Lab3 – Merge Sort : https://hackmd.io/s4kYmwloRR-UhOSPo-_WTg?view
CO2022 Lab3 – Merge Sort : https://hackmd.io/6bzzx4znRgGEsBHG9lJlgg?both

There are 4 exercises for Lab3

1. Variable Division
2. Power
3. Factorial
4. Merge Sort

1. Finish the C code 2. Finish the assembly code 3. Screenshot the result from memory and paste into report	1. Finish the assembly code 2. Screenshot the result from memory and paste into report 3. Record a video to explain your program

[Requirements]

- Submit codes (C & Assembly) and a result screenshot Report and a Video to explain Merge Sort

Assembly Code :

- You need to write complete and detailed comments in your assembly codes like Lab3 ppt
- You need to implement the main function assembly code yourself (TA will provide test data)

Video

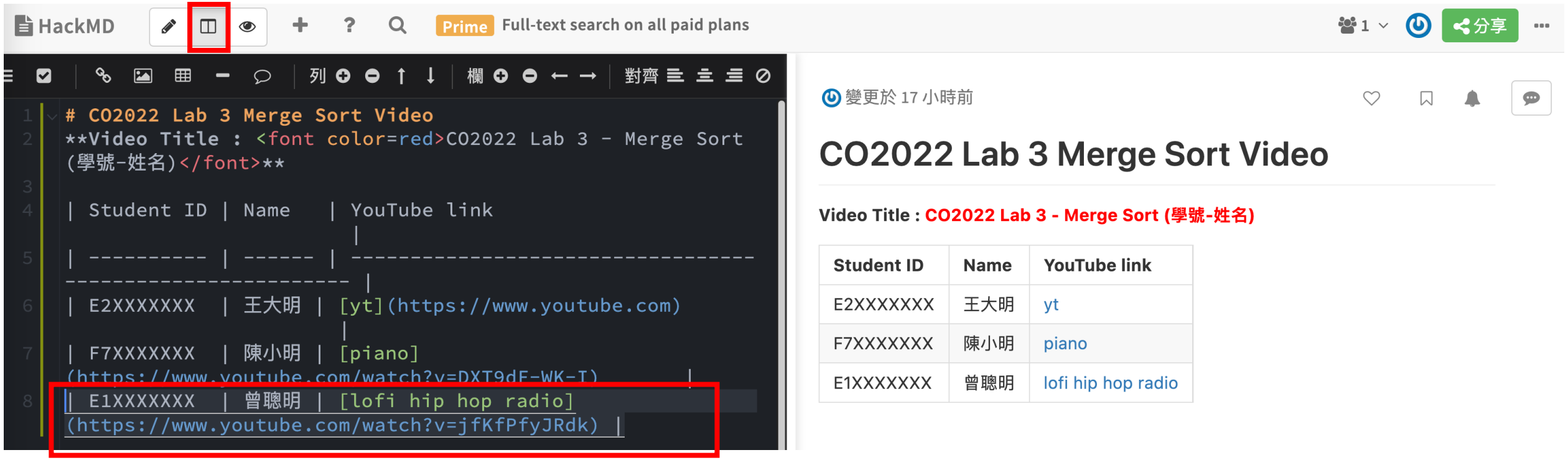
1. (Optional) You can make some ppt to show your idea before explaining the code
2. Show your idea from your assembly code (like Lab3 Lecture Video)
   (You can execute your code with breakpoints while explaining to prove your idea is right)
3. Show the Caller Saved & Callee Saved you did (Explain why you need to save them)
4. Run all test data and show they are all correct

HackMD Tutorial

CO2021 Lab3 – Merge Sort : https://hackmd.io/s4kYmwloRR-UhOSPo-_WTg?view
CO2022 Lab3 – Merge Sort : https://hackmd.io/6bzzx4znRgGEsBHG9lJlgg?both

- Share your video here if you want

1. Click this

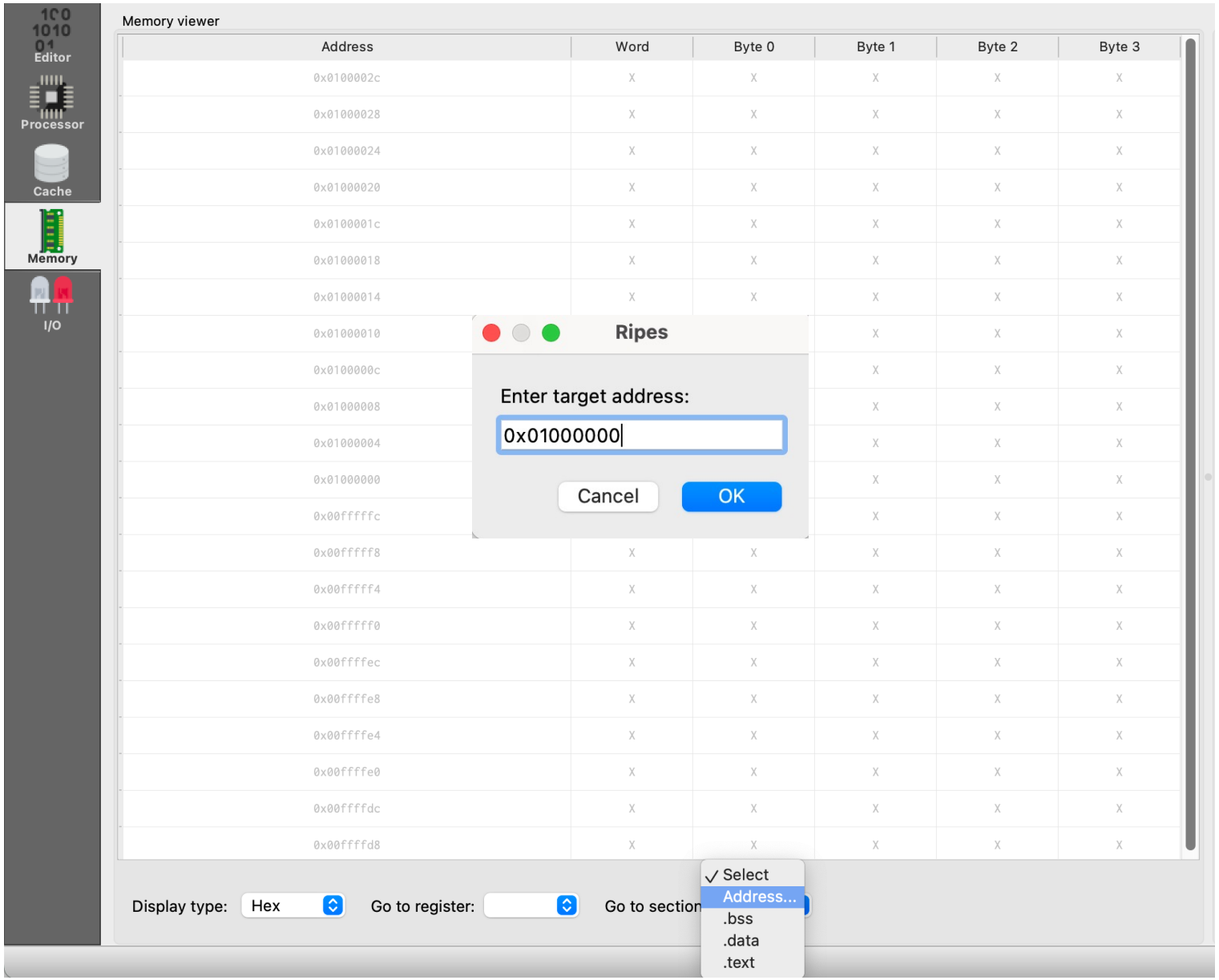


2. Copy and paste below
3. Modify as your profile

Exercise Overview – Result Location

- .text starts from 0x00000000 (default)
- .data starts from 0x10000000 (default)

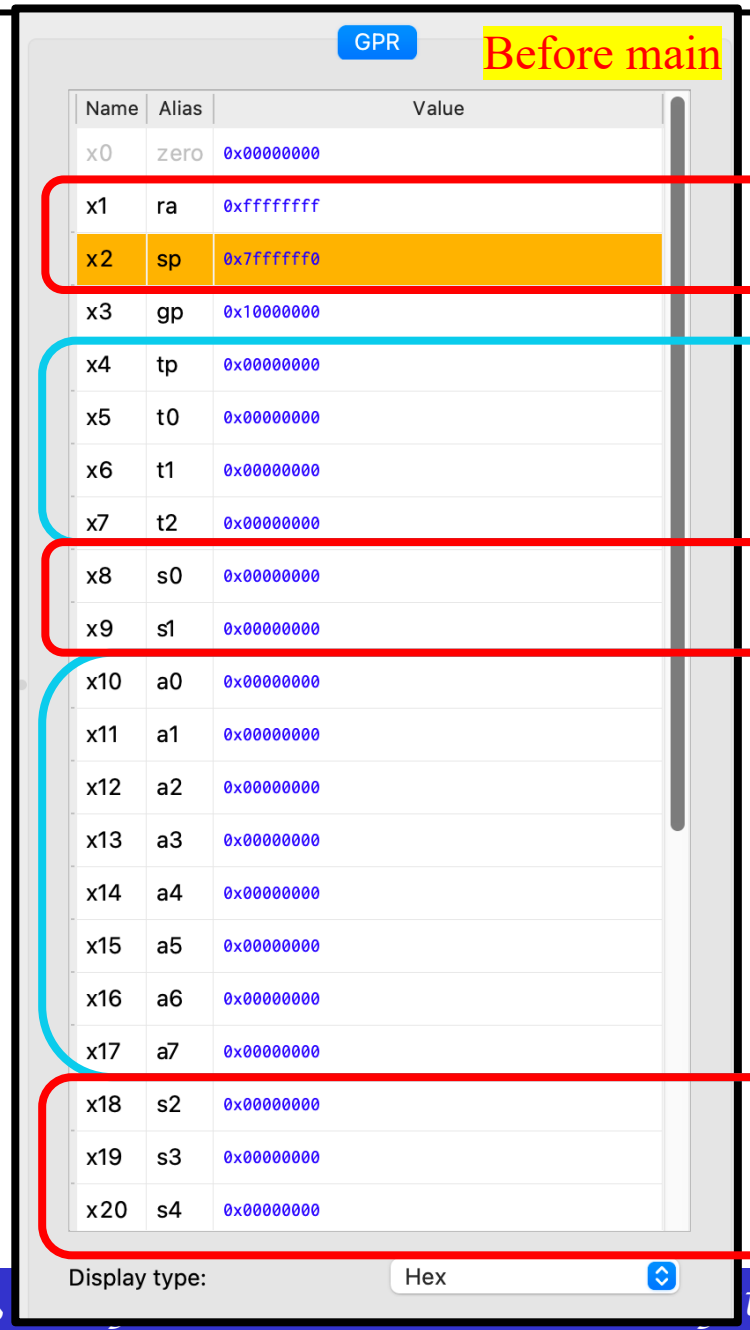
The answer of this lab
Please place them starting at 0x01000000

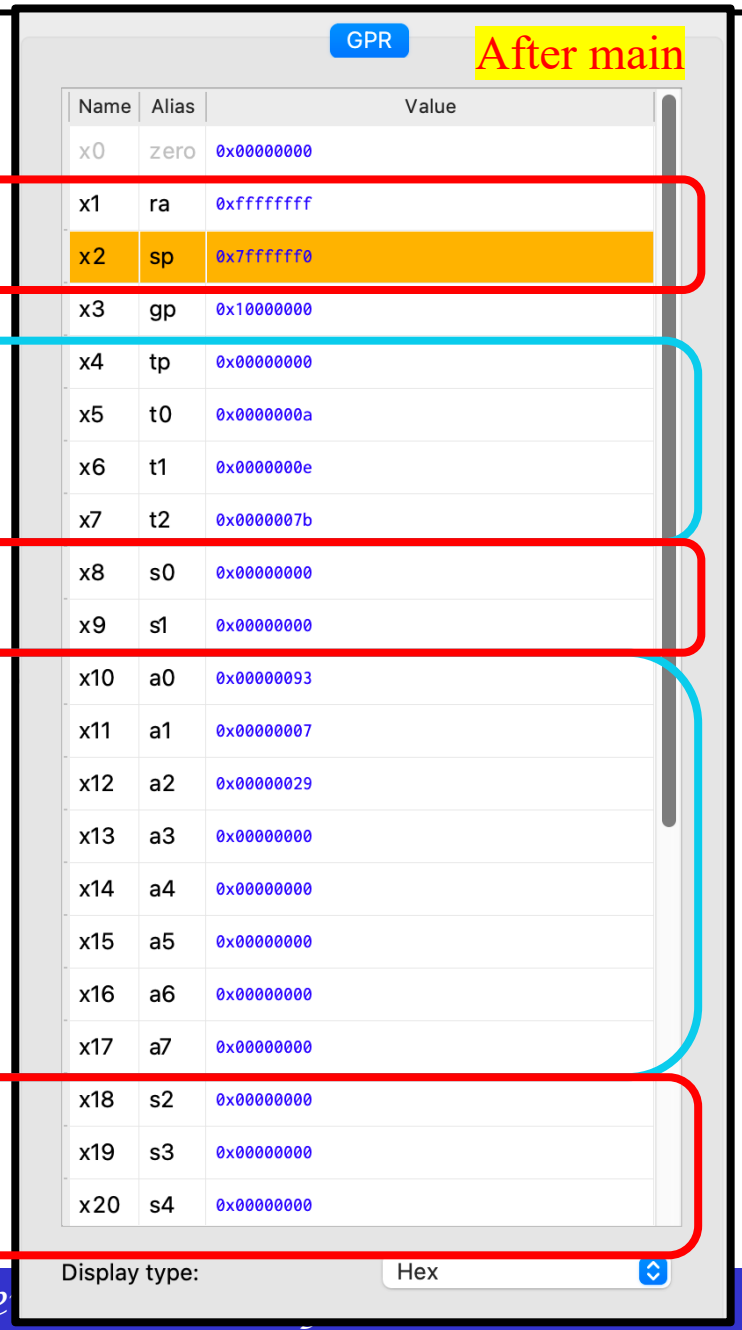


Exercise Overview – How to check whether Callee Saved is correct

```
1  .data
2  # ...
3
4  .text
5  setup:
6      li    ra, -1            # ra = -1
7      li    sp, 0x7ffffff0    # sp = 0x7ffffff0
8
9  main:
10     # ...
11
12     ret
13
```

Callee saved registers & ra need to be the same
Others can be changed at will





Exercise 1 : Variable Division

```c
int div(int dividend, int divisor) {
    // ...
}

int main() {
    int num_test = * (int *) 0x10000000;
    int *test =      (int *) 0x10000004;
    int *answer =    (int *) 0x01000000;

    for (int i = 0 ; i < num_test ; i++) {
        // test i
        int result;
        int valid = 1;
        // test{i} from memory;
        int dividend = *(test++);
        int divisor = *(test++);
        if (divisor == 0)
            valid = 0;
        else
            result = div(dividend, divisor);
        *(answer++) = valid;
        *(answer++) = result;
    }
    return 0;
}
```

Finish the C code by yourself

[Div Function]

1. No need to consider *Variable* / 0
   - We block this case at the Caller of div function

- You need to execute all test data at once (See the main function)

- Screenshot the result from memory start from address 0x01000000 and paste it into the report
  (30 answers = 2 x 15) (30 * 4 = 120 Bytes)
  (0x01000000 ~ 0x01000078)

```c
.data
num_test: .word 15
test1: .word 8, 0
test2: .word 0, 8    0 / 8
test3: .word 0, -3   0 / -3
test4: .word 4, 4
test5: .word 8, 4
test6: .word 29, 5
test7: .word 3, 4
test8: .word -8, 4
test9: .word -11, 4
test10: .word -3, 4
test11: .word 7, -4
test12: .word 3, -4
test13: .word -9, -6
test14: .word -5, -5
test15: .word -3, -5

.text
setup:
    li    ra, -1
    li    sp, 0x7ffffff0
main:
```

Exercise 2 : Power

```c
int power(int base, int exponent) {
    // ...
}
```

Finish the C code by yourself

```c
int main() {
    int num_test = * (int *) 0x10000000;
    int *test =      (int *) 0x10000004;
    int *answer =    (int *) 0x01000000;

    for (int i = 0 ; i < num_test ; i++) {
        // test i
        int result;
        int valid = 1;
        // test{i} from memory;
        int base = *(test++);
        int exponent = *(test++);
        if (base == 0 && exponent <= 0)
            valid = 0;
        else
            result = power(base, exponent);
        *(answer++) = valid;
        *(answer++) = result;
    }
    return 0;
}
```

- $2 \times 3 = 2 + 2 + 2$ => loop of addition
- $2^3 = 2 \times 2 \times 2$ => loop of multiplication

[Power Function]

1. No need to consider $0^{exp \leq 0}$
   - We block this case at the Caller of power function
2. $0^{positive} = 0$
3. $Variable^0 = 1$
4. $Variable^{negative} = (int)\ 0$
   - $1^{negative} = 1$

- You can use the example code "Variable Multiplication" to support implementation
- You need to execute all test data at once (See the main function)
- Screenshot the result from memory start from address 0x01000000 and paste it into the report
  (36 answers = 2 x 18) (36 * 4 = 144 Bytes)
  (0x01000000 ~ 0x01000090)

```
.data
num_test: .word 18
test1: .word 0, 0   0^0
test2: .word 0, -2
test3: .word 0, 7
test4: .word 7, 0   7^0
test5: .word -3, 0
test6: .word 1, 4
test7: .word 1, 0
test8: .word 1, -5   1^(-5)
test9: .word -1, 5
test10: .word -1, 4
test11: .word -1, 0
test12: .word -1, -5
test13: .word -1, -4
test14: .word 2, 3
test15: .word 2, -3
test16: .word -3, 5
test17: .word -3, 6  (-3)^6
test18: .word -3, -3

.text
setup:
    li    ra, -1
    li    sp, 0x7fff
main:
```

Exercise 3 : Factorial

```c
int factorial(int n) {
    // ...
}
```

Finish the C code by yourself

```c
int main() {
    int num_test = * (int *) 0x10000000;
    int *test =      (int *) 0x10000004;
    int *answer =    (int *) 0x01000000;

    for (int i = 0 ; i < num_test ; i++) {
        // test i
        // test{i} from memory;
        int n = *(test++);
        int result = factorial(n);
        *(answer++) = result;
    }
    return 0;
}
```

[Factorial Function]

1. Factorial (n < 0) => -1
2. Factorial (n = 0) => 0! = 1
3. Factorial (n > 0) => n!

```
.data
num_test: .word 5
test1: .word -10
test2: .word 0
test3: .word 1
test4: .word 5
test5: .word 10

.text
setup:
    li    ra, -1
    li    sp, 0x7ffffff0
main:
```

- You need to implement it by recursion
- You can use the example code “Variable Multiplication” to support implementation
- You need to execute all test data at once (See the main function)
- Screenshot the result from memory start from address 0x01000000 and paste it into the report
  (5 answers = 1 x 5) (5 * 4 = 20 Bytes)
  (0x01000000 ~ 0x01000014)

Exercise 4 : Merge Sort (Please see Appendix)

```c
int main(){
    int num_test = * (int *) 0x10000000;
    int *size =      (int *) 0x10000004;
    int *test =      (int *) 0x10000004 + num_test;
    int *answer =    (int *) 0x01000000;

    for (int i = 0 ; i < num_test ; i++) {
        // test i
        int test_size = *(size++);
        mergesort(test, 0, test_size-1);

        // Write answer
        for (int j = 0 ; j < test_size ; j++) {
            *(answer++) = *(test++);
        }
    }
    return 0;
}
```

- You need to implement it by recursion
- You need to execute all test data at once
  (See the main function)
- Screenshot the result from memory
  start from address 0x01000000 and paste it into the report
- Record a video to explain

```
.data
num_test: .word 3
TEST1_SIZE: .word 34
TEST2_SIZE: .word 19
TEST3_SIZE: .word 29
test1: .word 3,41,18,8,40,6,45,1,18,10,24,46,37,23,43,12,3,37,0,15,11,49,47,27,23,30,16,10,45,39,1,23,40,38
test2: .word -3,-23,-22,-6,-21,-19,-1,0,-2,-47,-17,-46,-6,-30,-50,-13,-47,-9,-50
test3: .word -46,0,-29,-2,23,-46,46,9,-18,-23,35,-37,3,-24,-18,22,0,15,-43,-16,-17,-42,-49,-29,19,-44,0,-18,23

.text
setup:
    li    ra, -1
    li    sp, 0x7ffffff0
main:
```

Sorted Array 1 : 0x01000000 ~ 0x01000088 (136 Bytes)
Sorted Array 2 : 0x01000088 ~ 0x010000d4 (76 Bytes)
Sorted Array 3 : 0x010000d4 ~ 0x01000148 (116 Bytes)

Question List

Q1 :
Why the immediate of jal & branch doesn't include imm[0] ?
What's the advantage of this design?
Why we need jalr ?

Q2 : Please explain RISC-V calling convention in your word

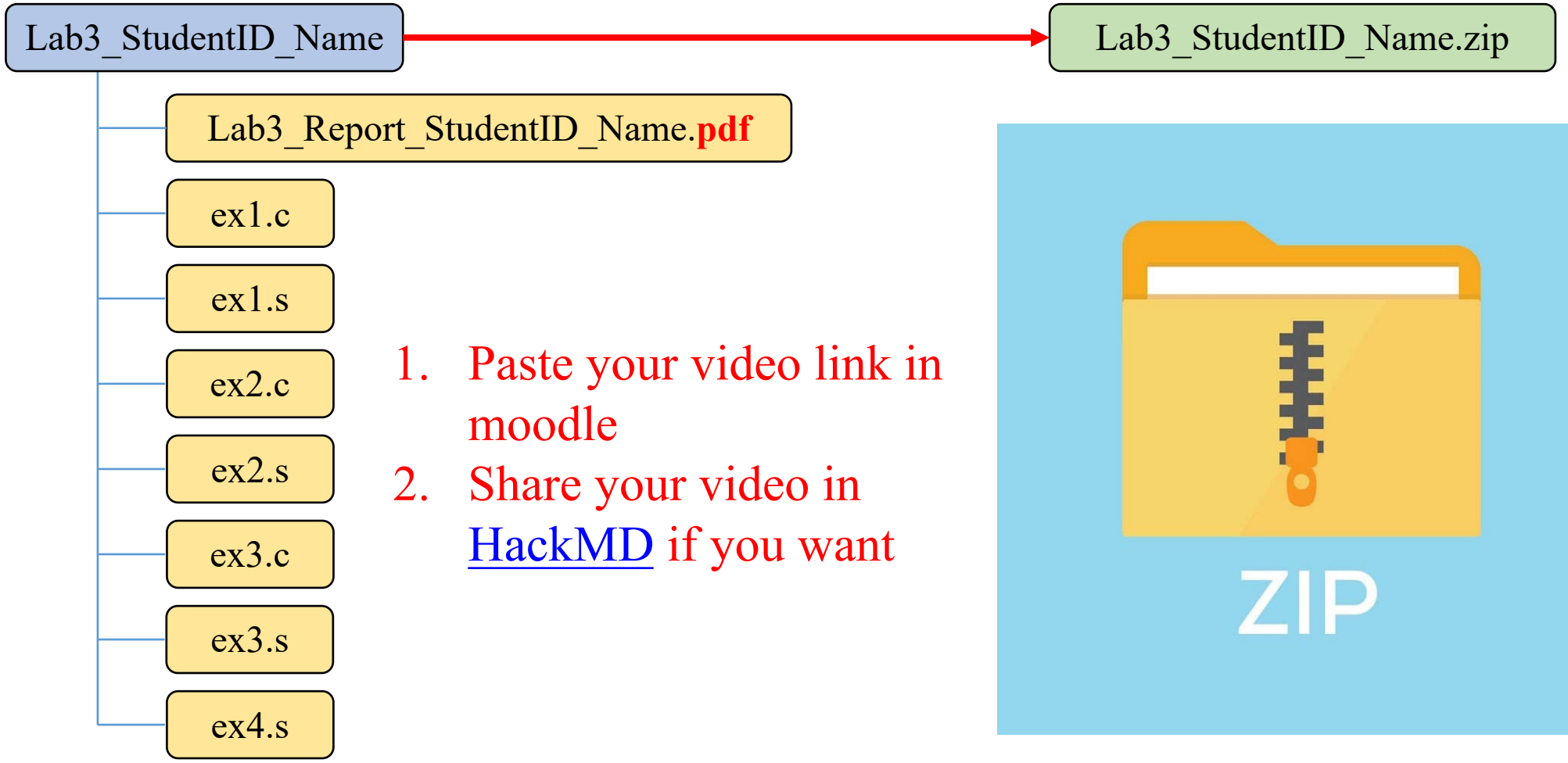
Q3 : Why there is no bgt(u) / ble(u) in standard RV32I ?

Q4 :
We can find that auipc, jal, branch are all PC-relative instructions.
What's the advantage of PC-relative ?

Format of the Lab Report (PDF !!!)

- Cover (There is a default format of the report on the Moodle.)
- Content of the report
    1. Answer “Question List”.
    2. Screenshot the results of ex1 ~ ex4 from RIPES memory starting at address 0x01000000

File structure for submission



1. Paste your video link in moodle
2. Share your video in HackMD if you want

Merge Sort

Time complexity of Sort

by divide and conquer method

Sort Algorithm	Insertion	Bubble	Selection	Quick	Merge	Heap	Bogo
Average Case	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(nlogn)$	$O(nlogn)$	$O(nlogn)$	$O((n+1)!)$
Worst Case	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(nlogn)$	$O(nlogn)$	$O(\infty)$
Stable	Yes	Yes	No	No	Yes	No	No

What is stable?

$\overset{*}{5}6\overset{\wedge}{5}3$ sorting → $3\overset{*}{5}\overset{\wedge}{5}6$ (Stable)

$\overset{*}{5}6\overset{\wedge}{5}3$ → $3\overset{\wedge}{5}\overset{*}{5}6$ (Not stable)

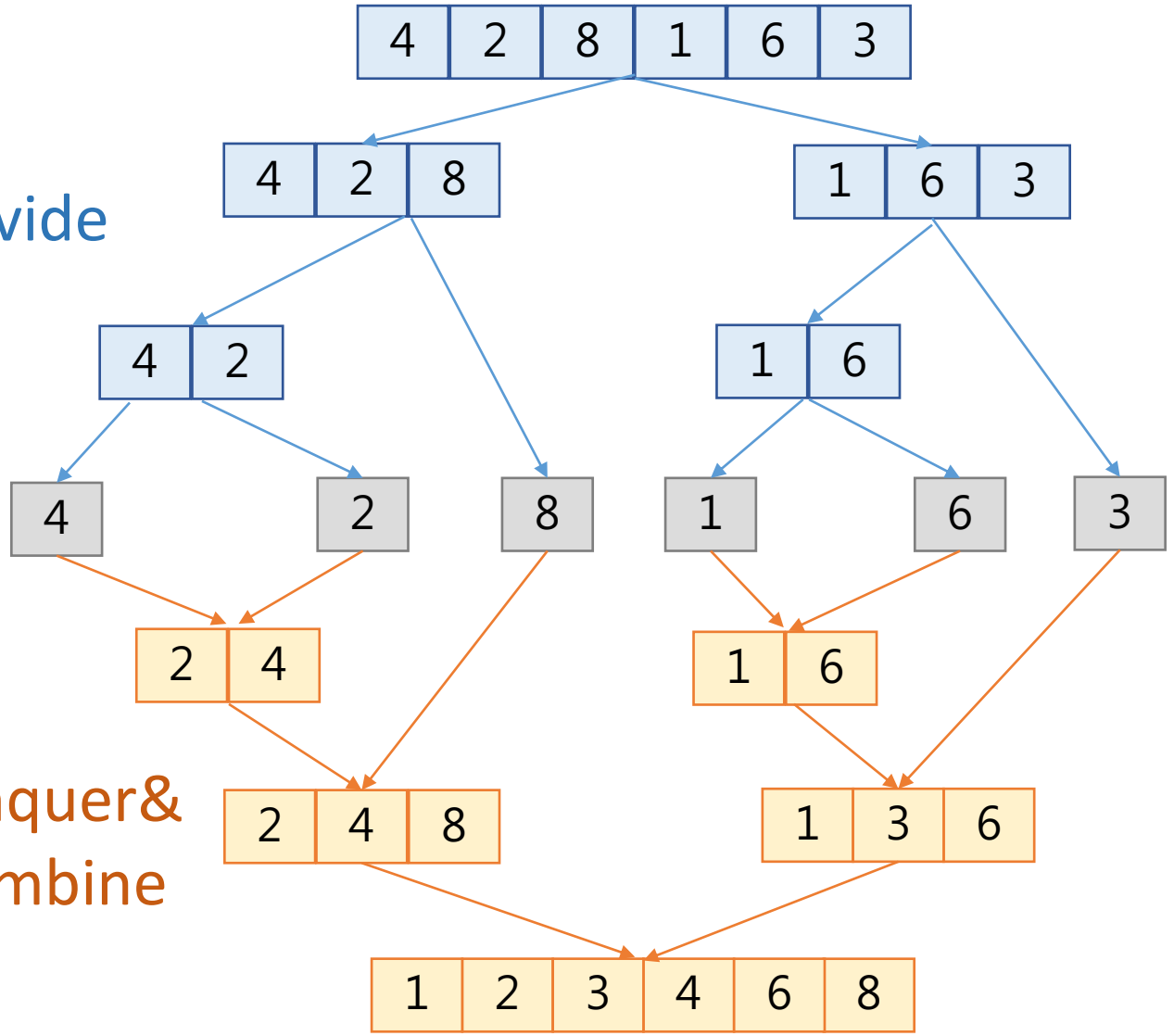
15 Sorting Algorithms in 6 Minutes
https://youtu.be/kPRA0W1kECg

Merge sort - Divide & Conquer

1. Divide : The original problem is divided into several smaller, relatively independent subproblems of the same form as the original problem.

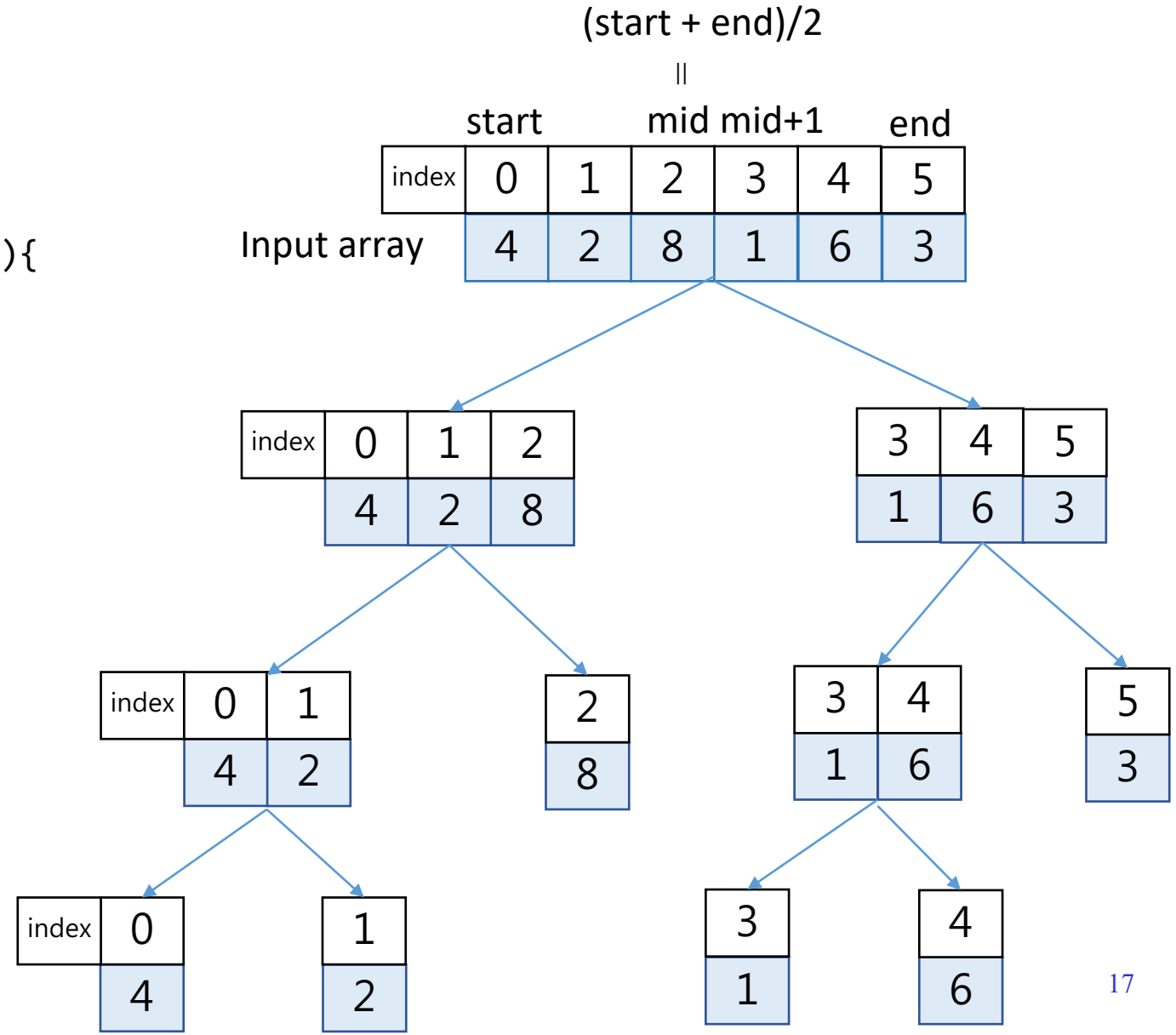
2. Conquer: If the subproblems are small and easy to solve, then solve them directly. Otherwise, solve each subproblem in a recursion.

3. Combine: Combine the solutions of each subproblem into the solution of the original problem.



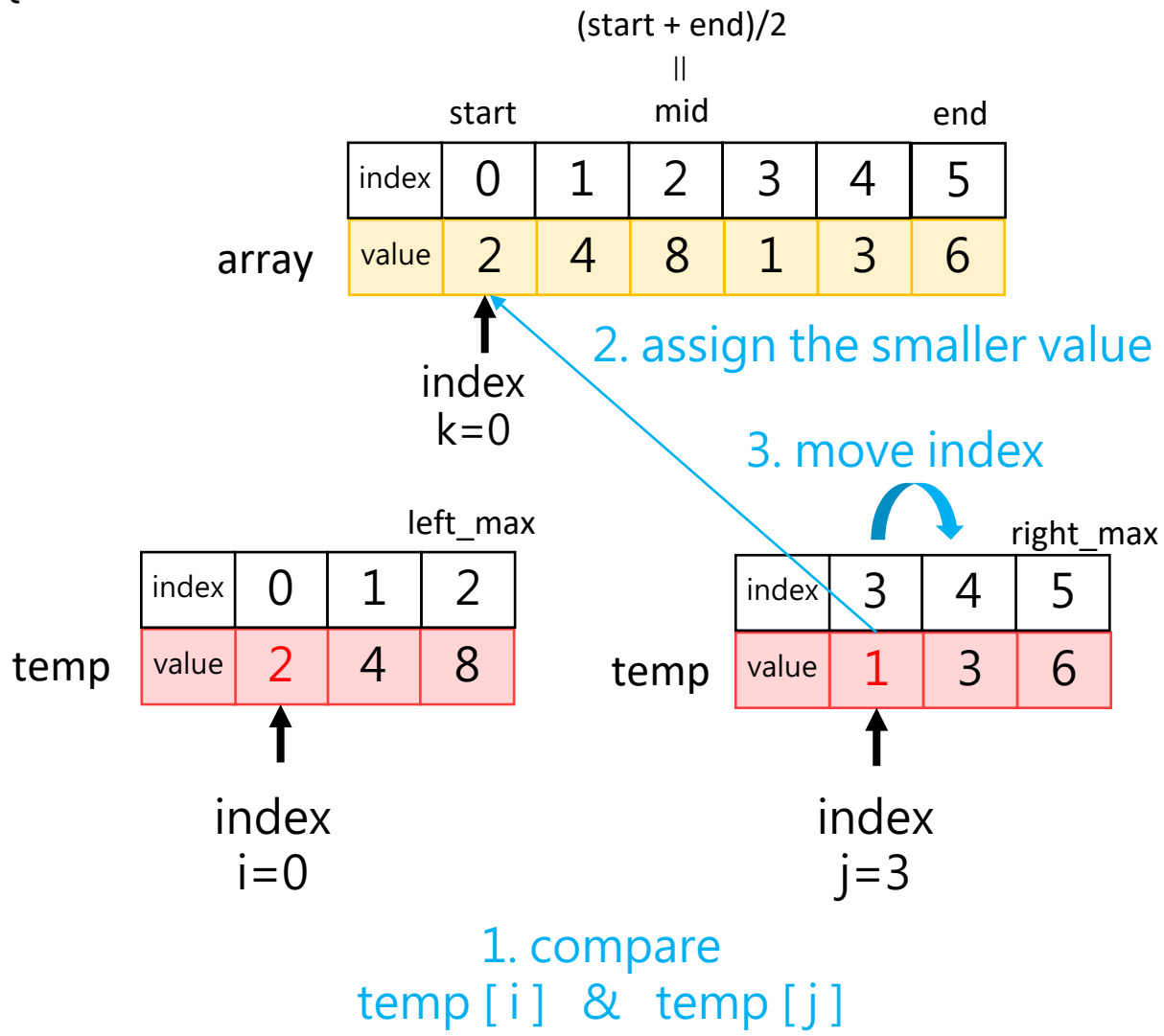
Merge sort - Divide & Conquer

```
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //divided to the left
        mergesort(array, start, mid);
        //divided to the right
        mergesort(array, mid+1, end);
        //solve the problem
        merge(array, start, mid, end);
    }
}
```



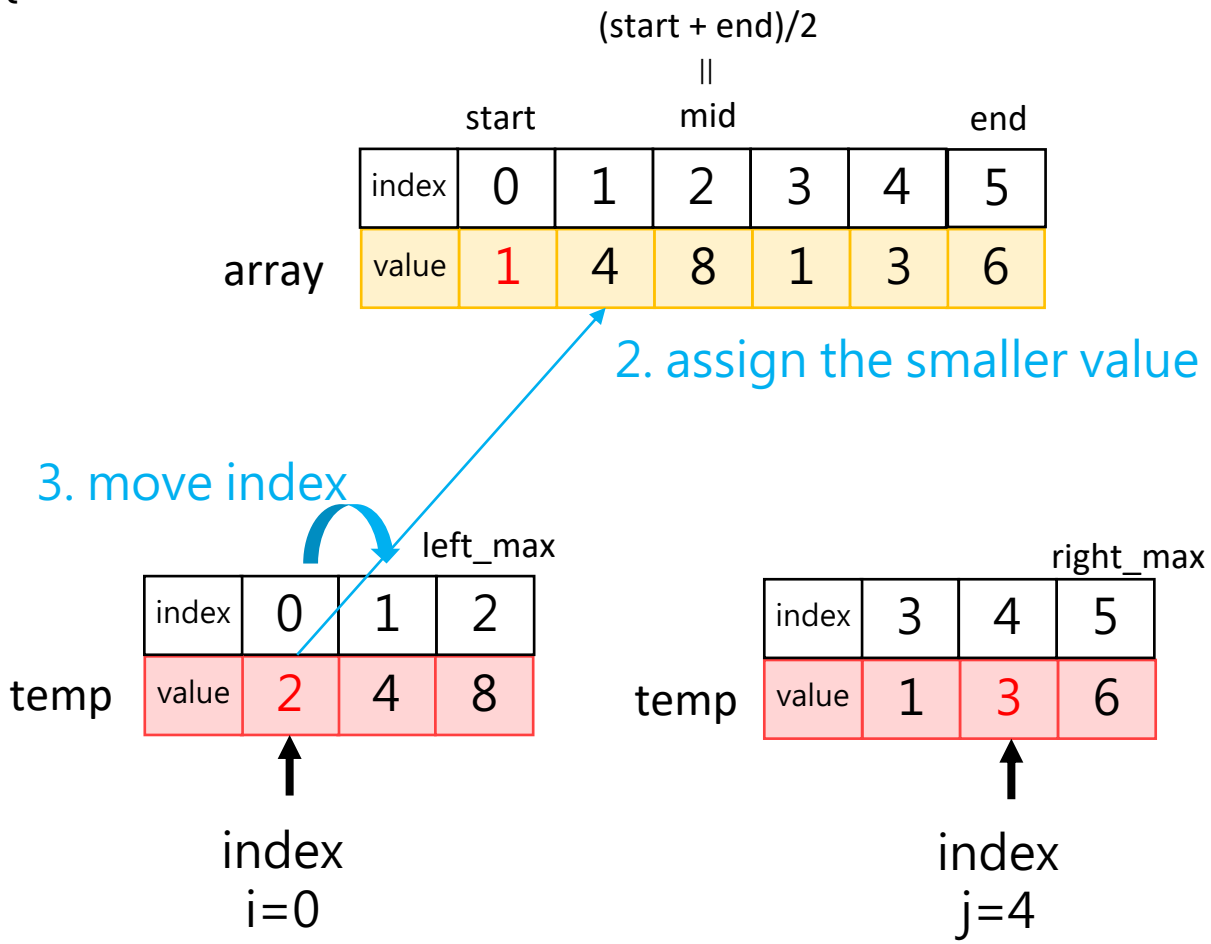
Merge sort - Conquer : Merge function

```
merge(int *array, int start, int mid, int end) {
    while (i <= left_max && j <= right_max) {
        if (temp[ i ] <= temp[ j ]) {
            array[ k ] = array[ i ];
                k++;
                i++;
        }
        else {
            array[ k ] = temp[ j ];
            k++;
            j++;
        }
    }
    while(i <= left_max) {
        array[ k ] = temp[ i ];
            k++;
            i++;
    }
    while(j <= right_max) {
        array[ k ] = temp[ j ];
        k++;
        j++;
    }
}
```



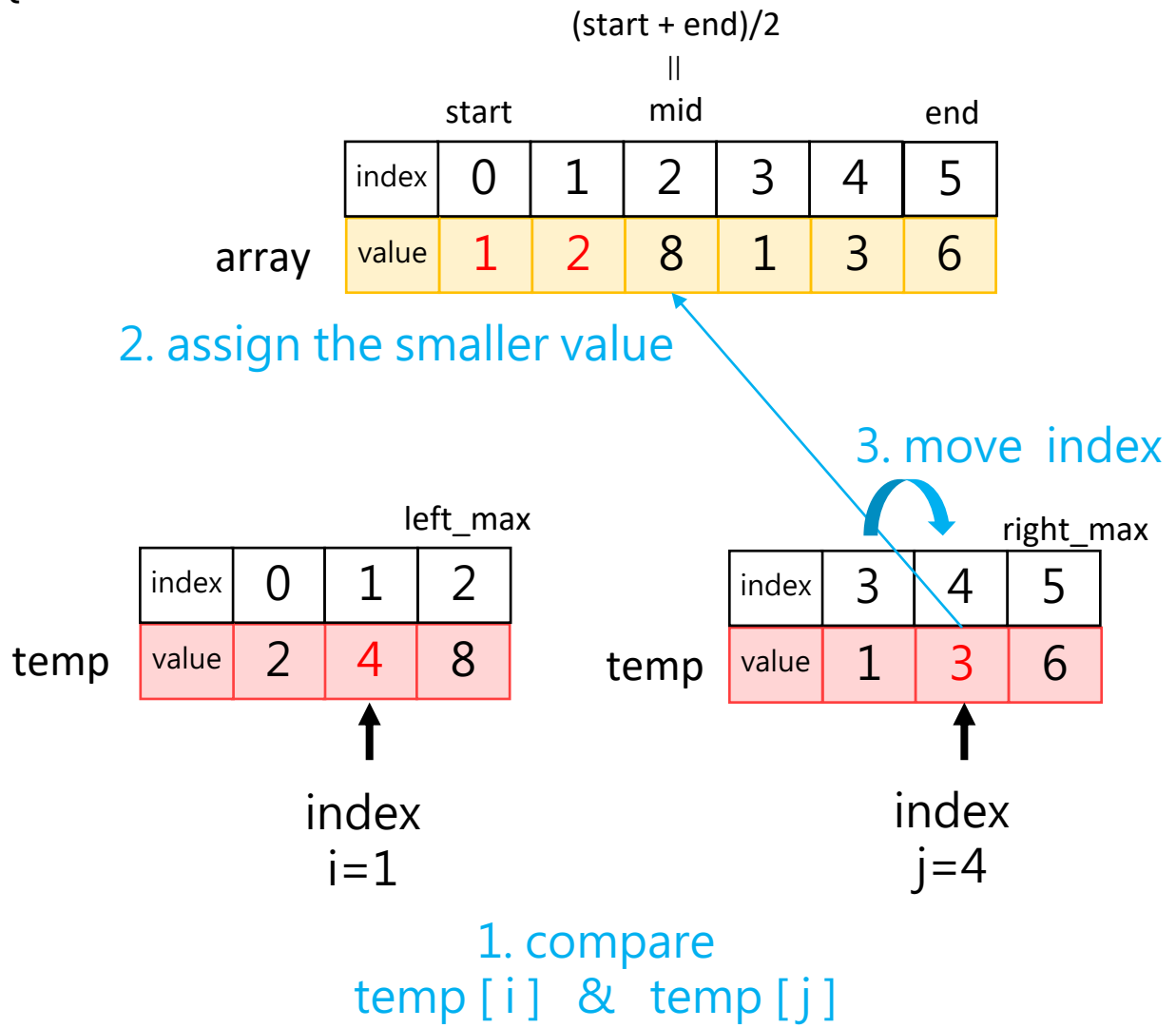
Merge sort - Conquer : Merge function

```
merge(int *array, int start, int mid, int end) {
    while (i <= left_max && j <= right_max) {
        if (temp[ i ] <= temp[ j ]) {
            array[ k ] = array[ i ];
                k++;
                i++;
        }
        else {
            array[ k ] = temp[ j ];
            k++;
            j++;
        }
    }
    while(i <= left_max) {
        array[ k ] = temp[ i ];
            k++;
            i++;
    }
    while(j <= right_max) {
        array[ k ] = temp[ j ];
        k++;
        j++;
    }
}
```



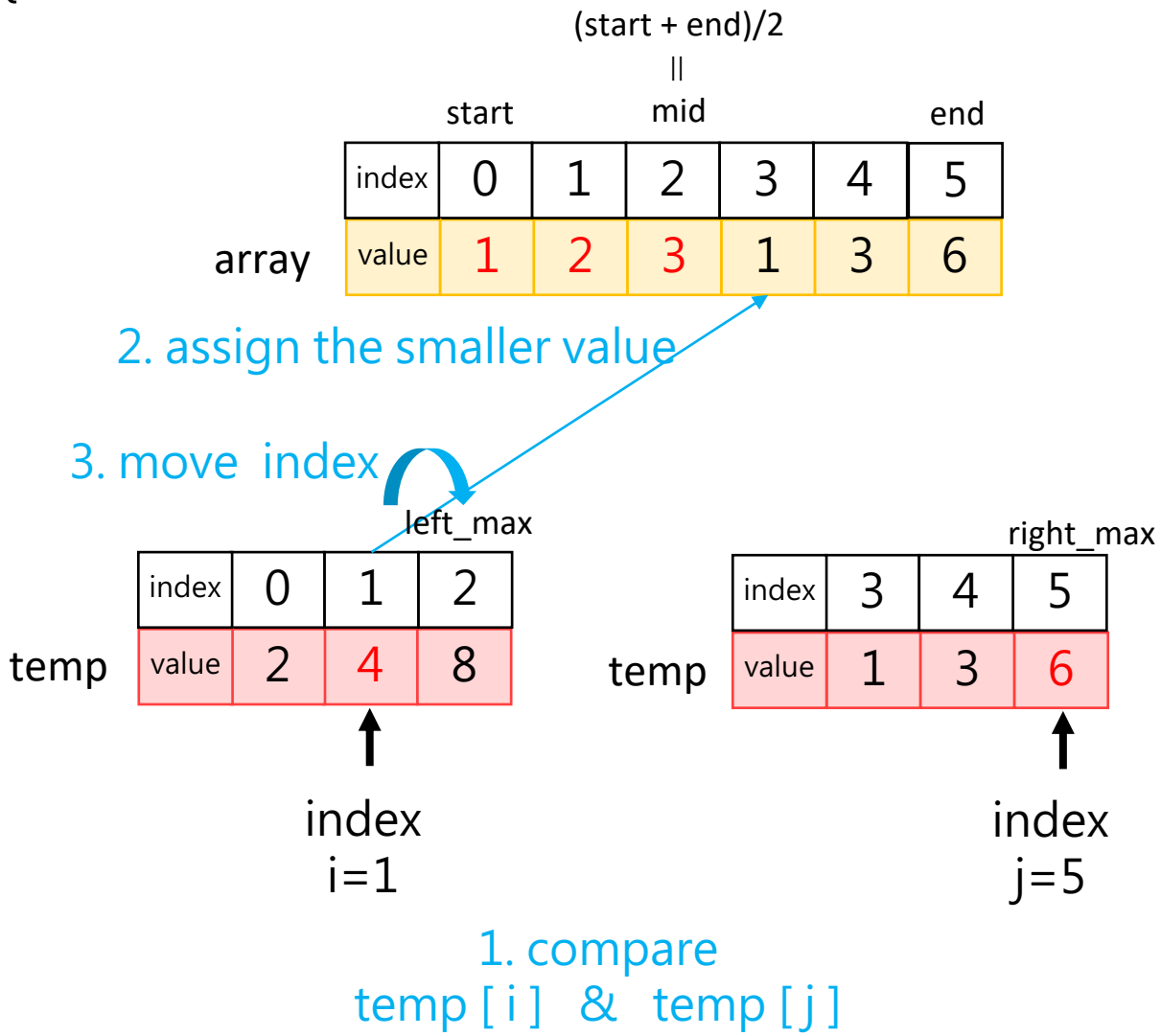
Merge sort - Conquer : Merge function

```
merge(int *array, int start, int mid, int end) {
    while (i <= left_max && j <= right_max) {
        if (temp[ i ] <= temp[ j ]) {
            array[ k ] = array[ i ];
                k++;
                i++;
        }
        else {
            array[ k ] = temp[ j ];
            k++;
            j++;
        }
    }
    while(i <= left_max) {
        array[ k ] = temp[ i ];
            k++;
            i++;
    }
    while(j <= right_max) {
        array[ k ] = temp[ j ];
        k++;
        j++;
    }
}
```



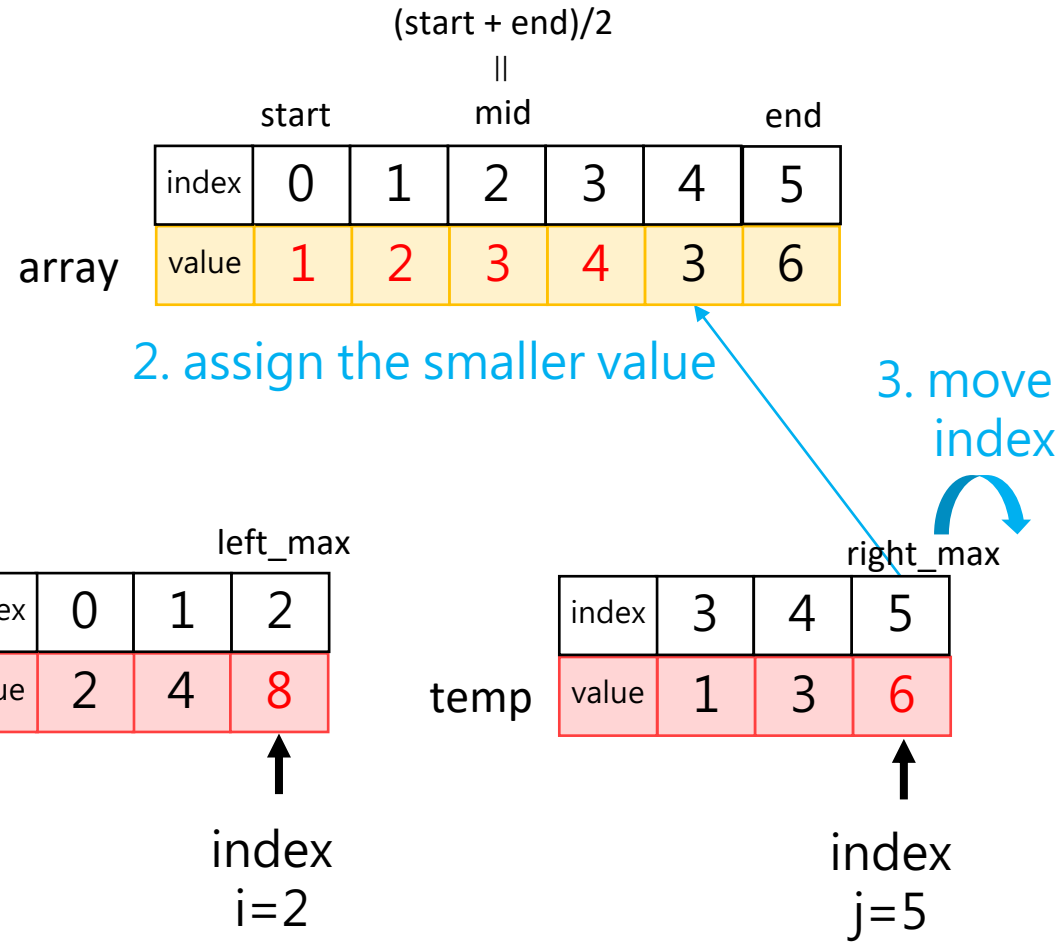
Merge sort - Conquer : Merge function

```
merge(int *array, int start, int mid, int end) {
    while (i <= left_max && j <= right_max) {
        if (temp[ i ] <= temp[ j ]) {
            array[ k ] = array[ i ];
                k++;
                i++;
        }
        else {
            array[ k ] = temp[ j ];
            k++;
            j++;
        }
    }
    while(i <= left_max) {
        array[ k ] = temp[ i ];
            k++;
            i++;
    }
    while(j <= right_max) {
        array[ k ] = temp[ j ];
        k++;
        j++;
    }
}
```



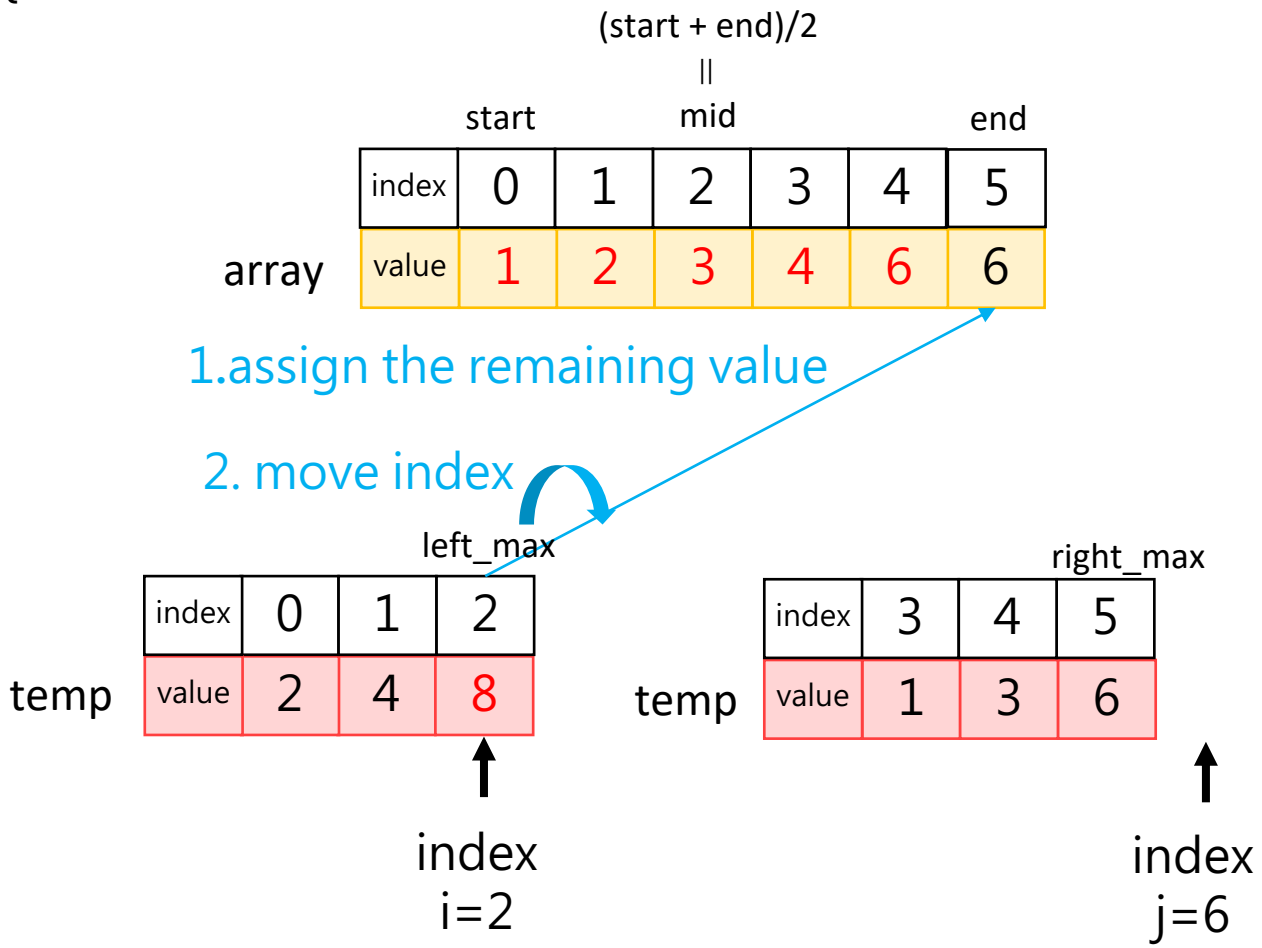
Merge sort - Conquer : Merge function

```
merge(int *array, int start, int mid, int end) {
    while (i <= left_max && j <= right_max) {
        if (temp[ i ] <= temp[ j ]) {
            array[ k ] = array[ i ];
                k++;
                i++;
        }
        else {
            array[ k ] = temp[ j ];
            k++;
            j++;
        }
    }
    while(i <= left_max) {
        array[ k ] = temp[ i ];
            k++;
            i++;
    }
    while(j <= right_max) {
        array[ k ] = temp[ j ];
        k++;
        j++;
    }
}
```



Merge sort - Conquer : Merge function

```
merge(int *array, int start, int mid, int end) {
    while (i <= left_max && j <= right_max) {
        if (temp[ i ] <= temp[ j ]) {
            array[ k ] = array[ i ];
                k++;
                i++;
        }
        else {
            array[ k ] = temp[ j ];
            k++;
            j++;
        }
    }
    while(i <= left_max) {
        array[ k ] = temp[ i ];
            k++;
            i++;
    }
    while(j <= right_max) {
        array[ k ] = temp[ j ];
        k++;
        j++;
    }
}
```



Merge sort - Conquer : Merge function

```
merge(int *array, int start, int mid, int end) {
    while (i <= left_max && j <= right_max) {
        if (temp[ i ] <= temp[ j ]) {
            array[ k ] = array[ i ];
                k++;
                i++;
        }
        else {
            array[ k ] = temp[ j ];
            k++;
            j++;
        }
    }
    while(i <= left_max) {
        array[ k ] = temp[ i ];
            k++;
            i++;
    }
    while(j <= right_max) {
        array[ k ] = temp[ j ];
        k++;
        j++;
    }
}
```

(start + end)/2
||
start mid end

	index	0	1	2	3	4	5
array	value	1	2	3	4	6	8

left_max

	index	0	1	2
temp	value	2	4	8

index
i=3

right_max

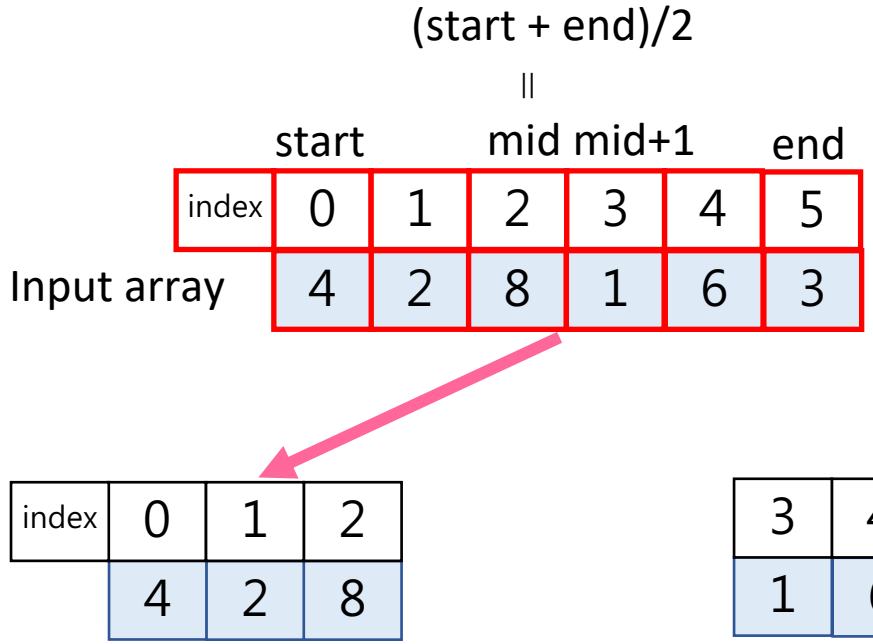
	index	3	4	5
temp	value	1	3	6

index
j=6

finish merge function

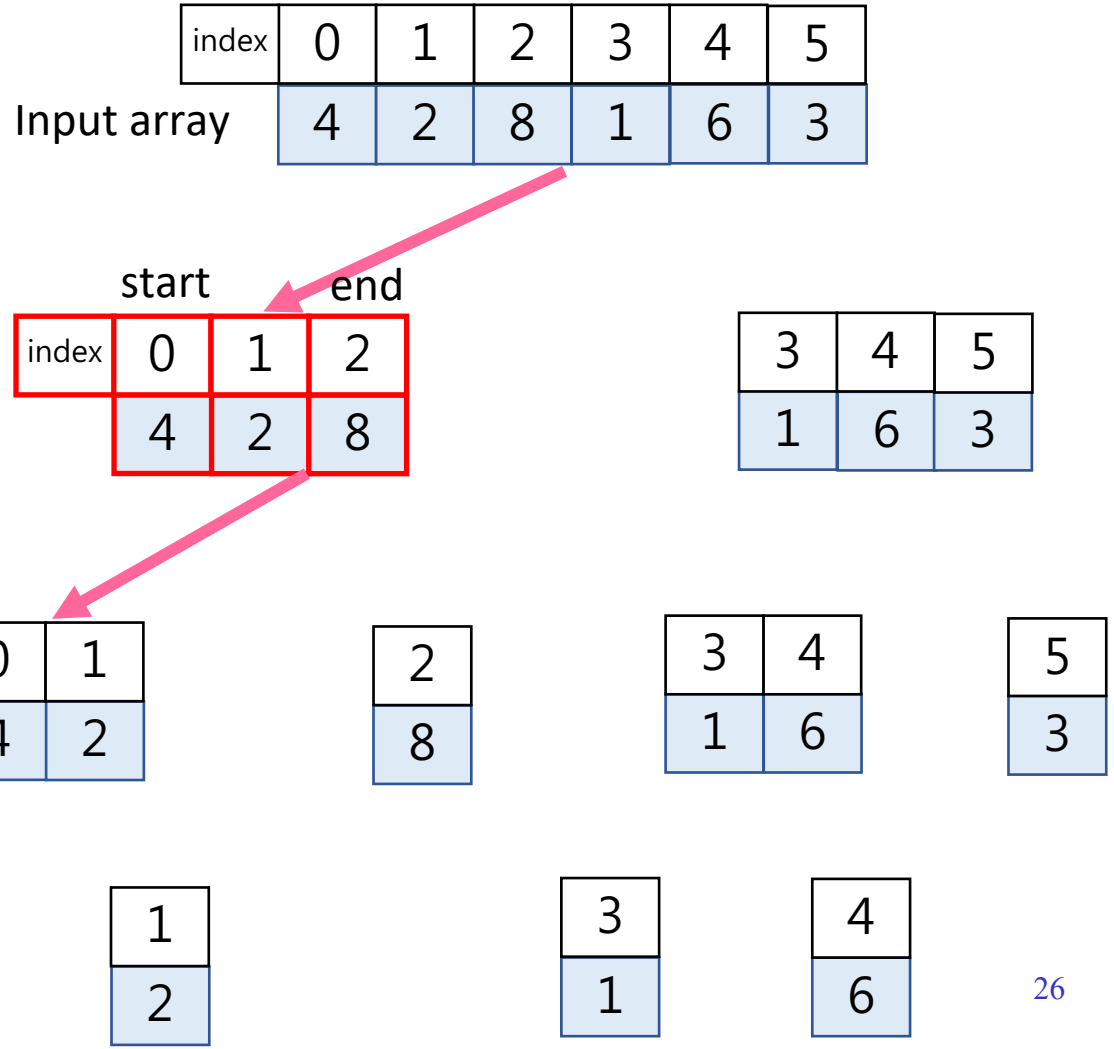
Merge sort - Divide & Conquer

```
//                                  0          5
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                      0     2
        mergeSort(array, start, mid);
        //                      3     5
        mergeSort(array, mid+1, end);
        //                  0     2     5
        merge(array, start, mid, end);
    }
}
```



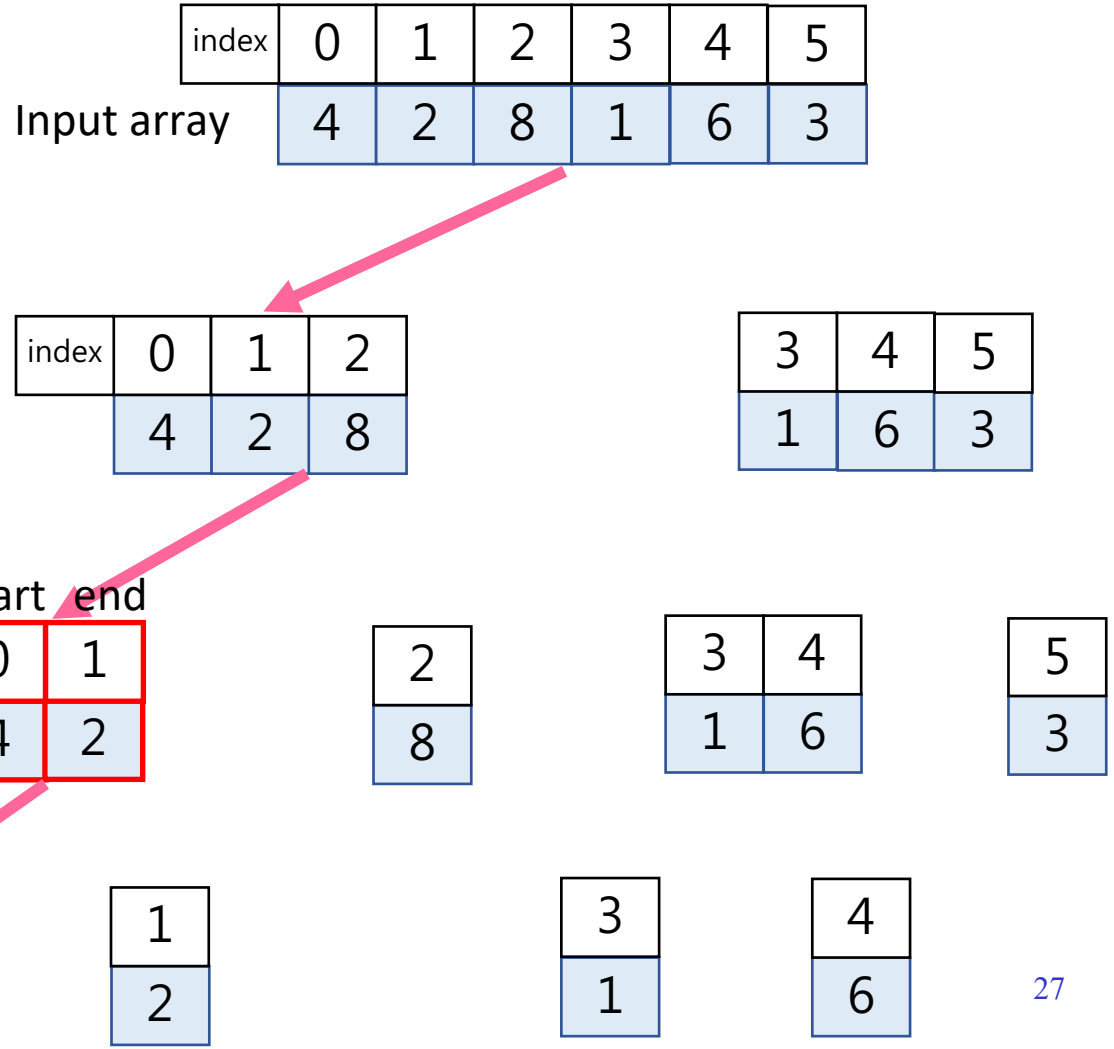
Merge sort - Divide & Conquer

```
//                                  0         2
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                      0     1
        mergeSort(array, start, mid);
        //                      2     2
        mergeSort(array, mid+1, end);
        //                  0     1     2
        merge(array, start, mid, end);
    }
}
```



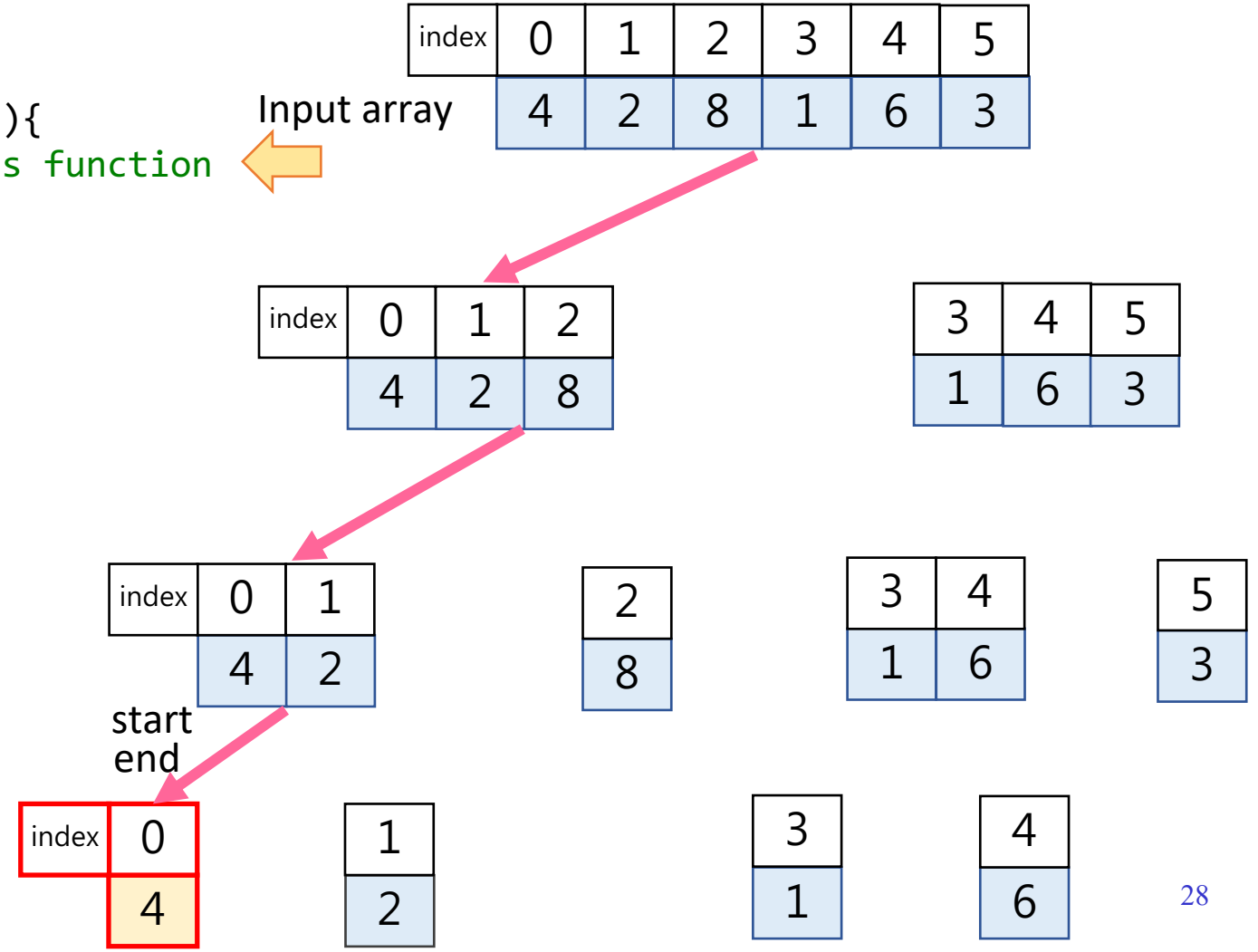
Merge sort - Divide & Conquer

```
//                                  0         1
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                      0     0
        mergeSort(array, start, mid);
        //                      1     1
        mergeSort(array, mid+1, end);
        //                  0     0     1
        merge(array, start, mid, end);
    }
}
```



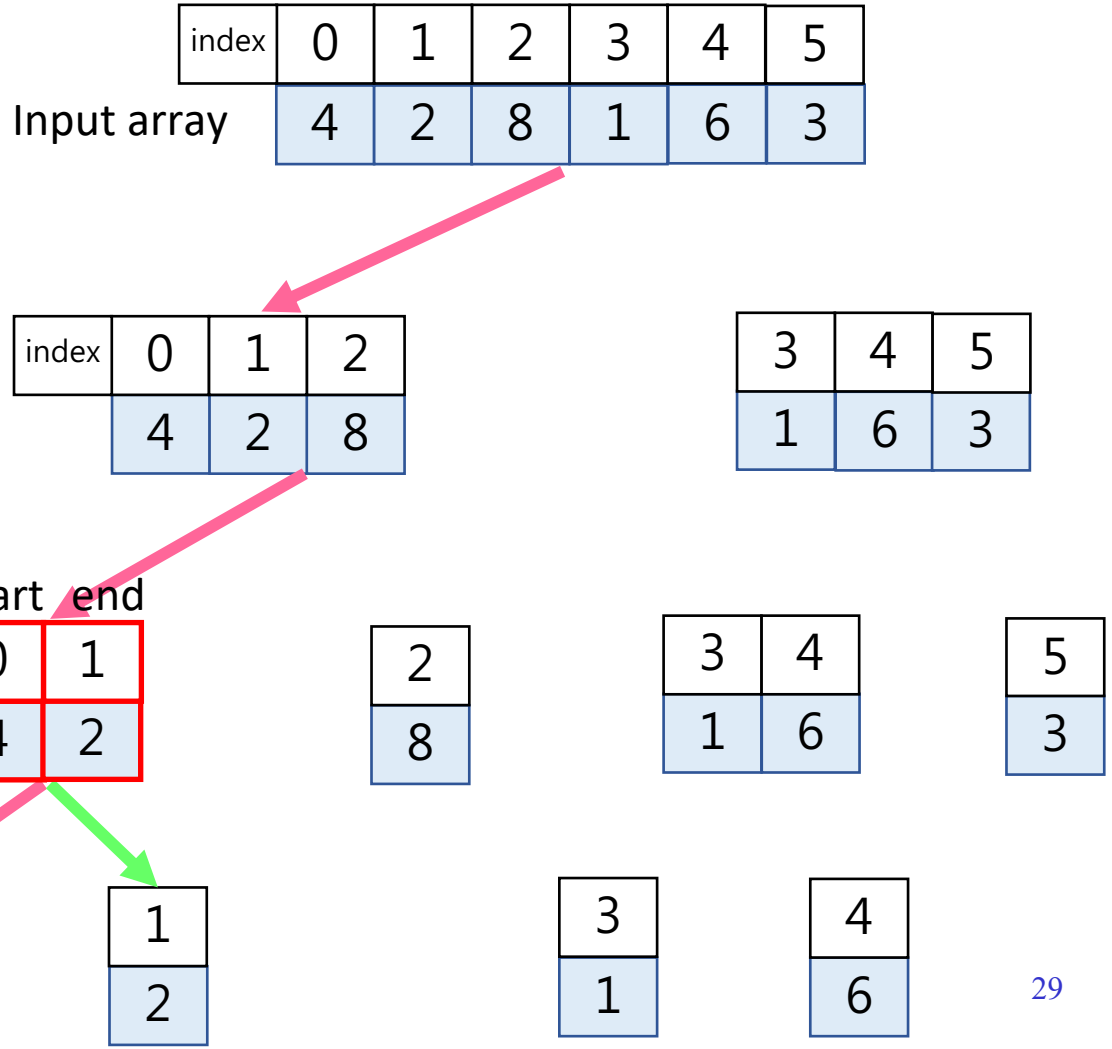
Merge sort - Divide & Conquer

```
//                                  0         0
void mergeSort(int *array, int start, int end){
    if(start < end){//0 < 0 return to previous function
        int mid = (end + start)/2;
//
        mergeSort(array, start, mid);
//
        mergeSort(array, mid+1, end);
//
        merge(array, start, mid, end);
    }
}
```



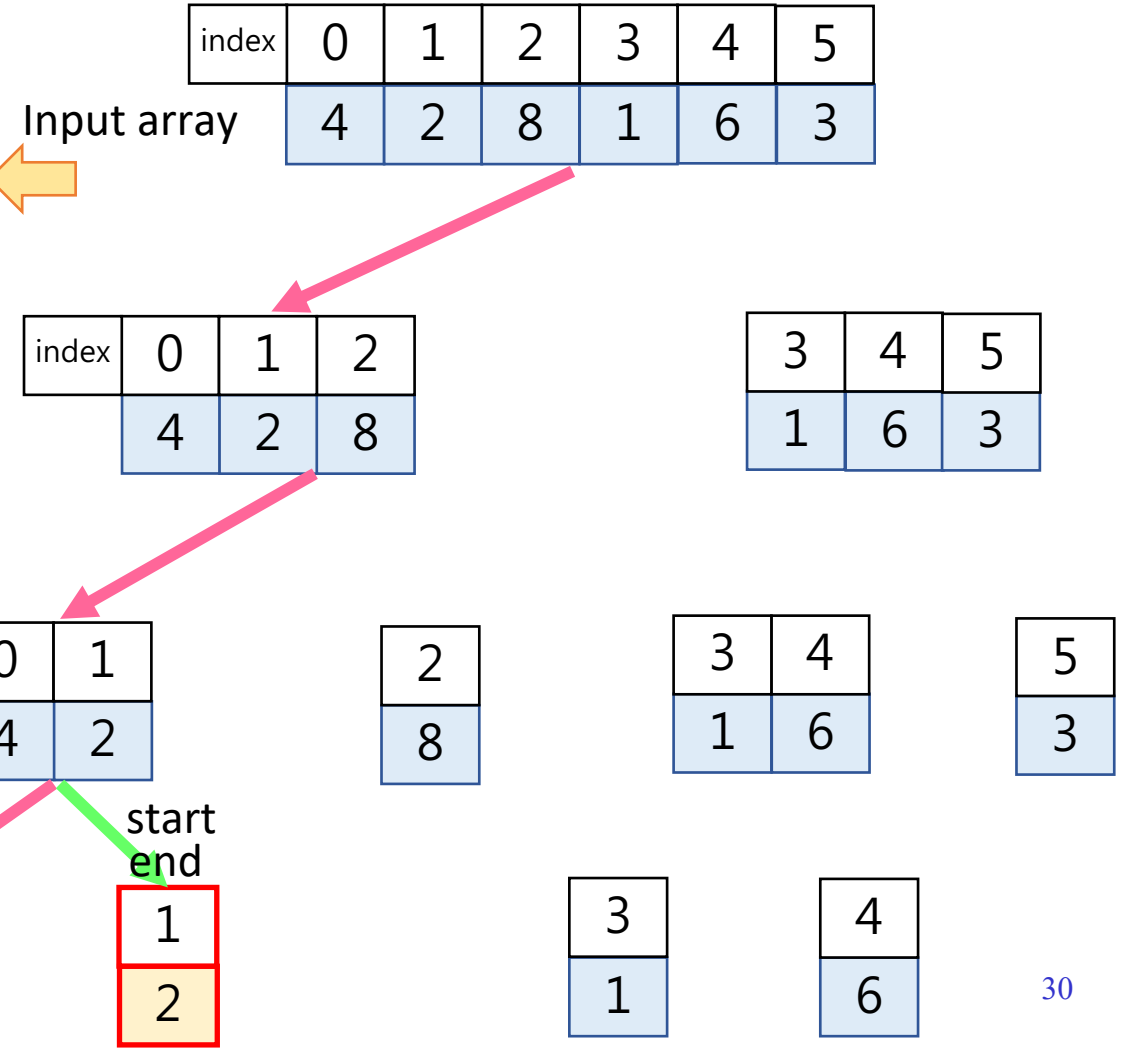
Merge sort - Divide & Conquer

```
//                                  0         1
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                  0     0
        mergeSort(array, start, mid);
        //                  1     1
        mergeSort(array, mid+1, end);
        //                0     0    1
        merge(array, start, mid, end);
    }
}
```



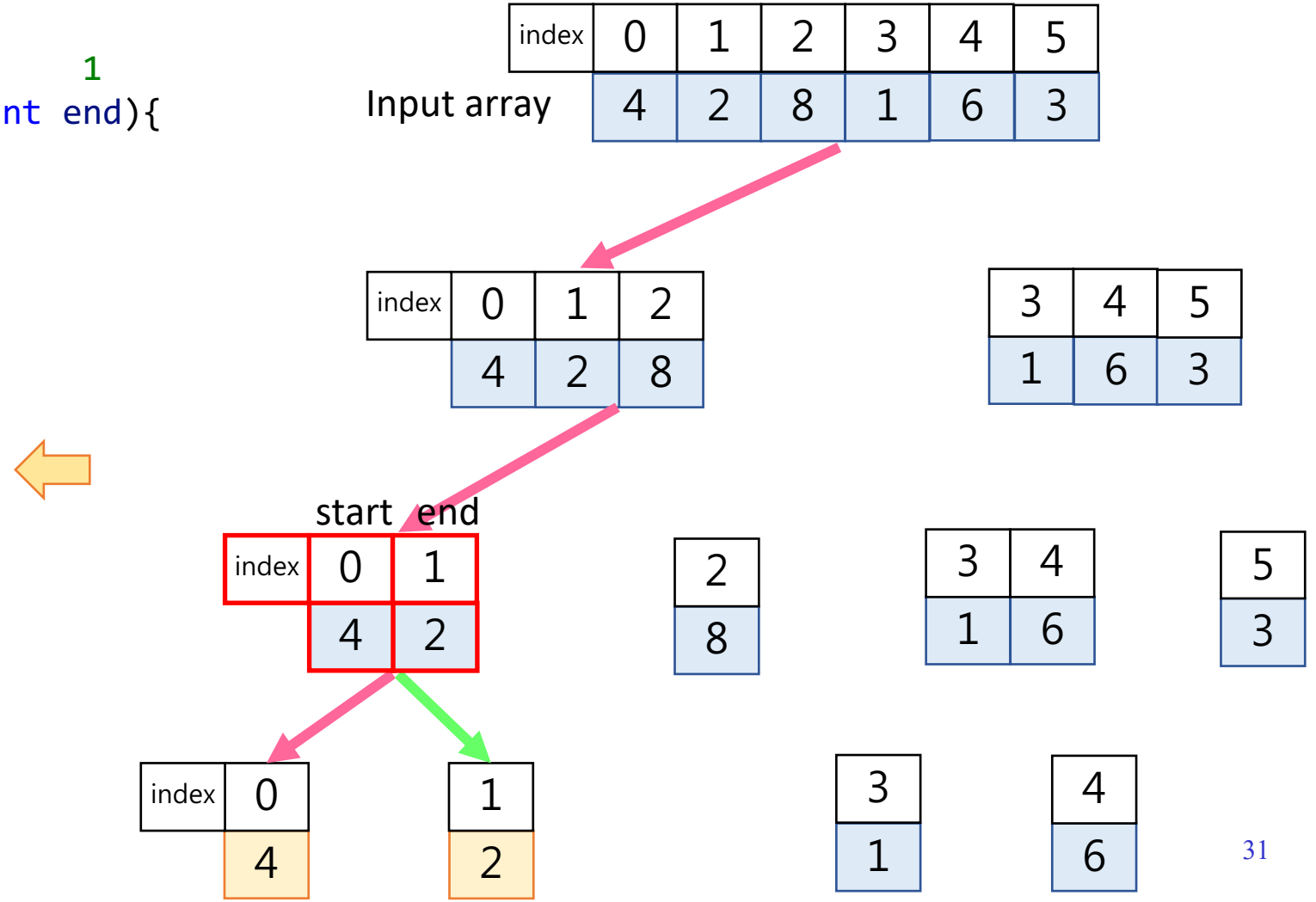
Merge sort - Divide & Conquer

```
//                                  1         1
void mergeSort(int *array, int start, int end){
    if(start < end){//1 < 1 return to previous function
        int mid = (end + start)/2;
//
        mergeSort(array, start, mid);
//
        mergeSort(array, mid+1, end);
//
        merge(array, start, mid, end);
    }
}
```



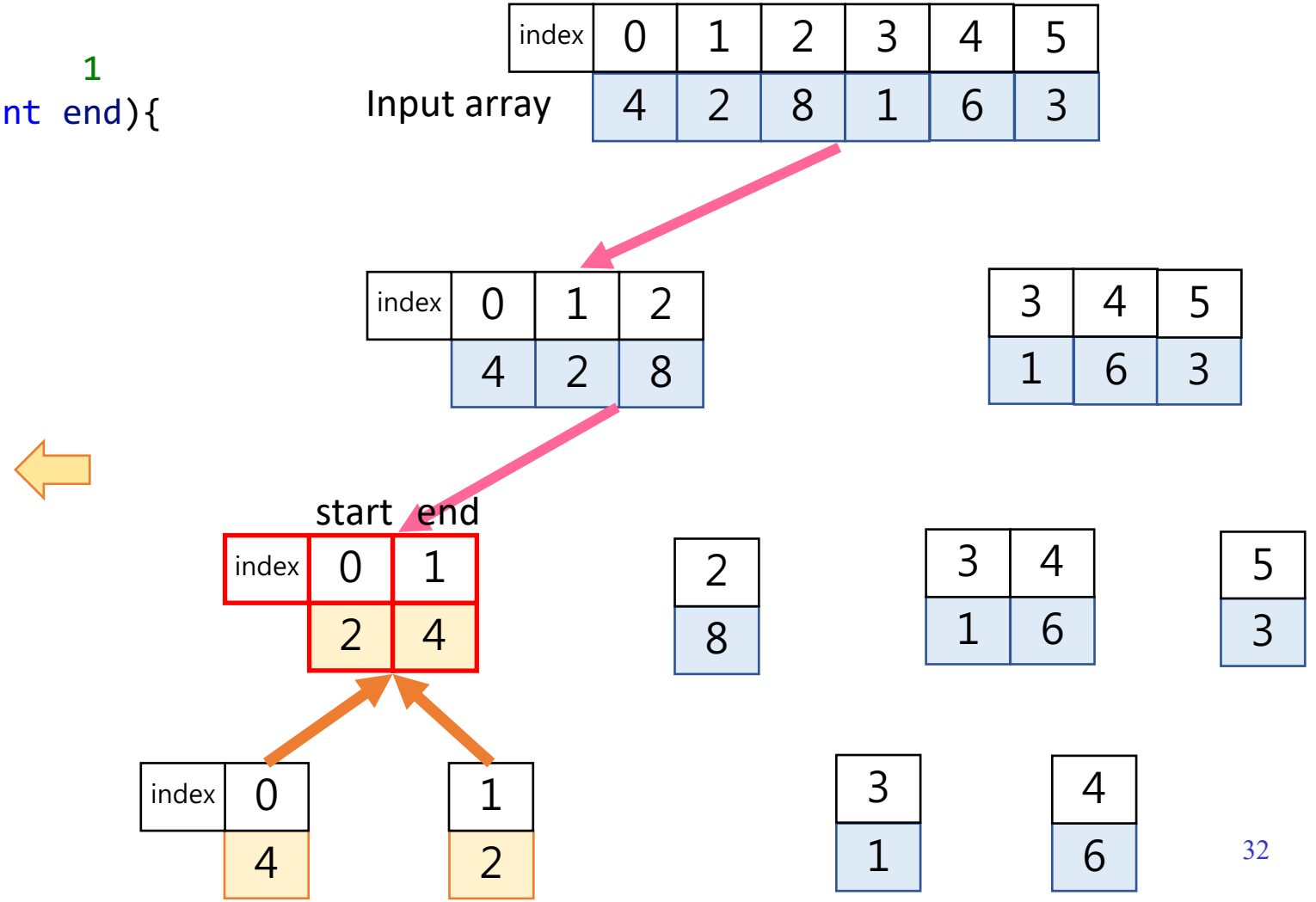
Merge sort - Divide & Conquer

```
//                                  0         1
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                    0     0
        mergeSort(array, start, mid);
        //                    1     1
        mergeSort(array, mid+1, end);
        //                  0     0    1
        merge(array, start, mid, end);
    }
}
```



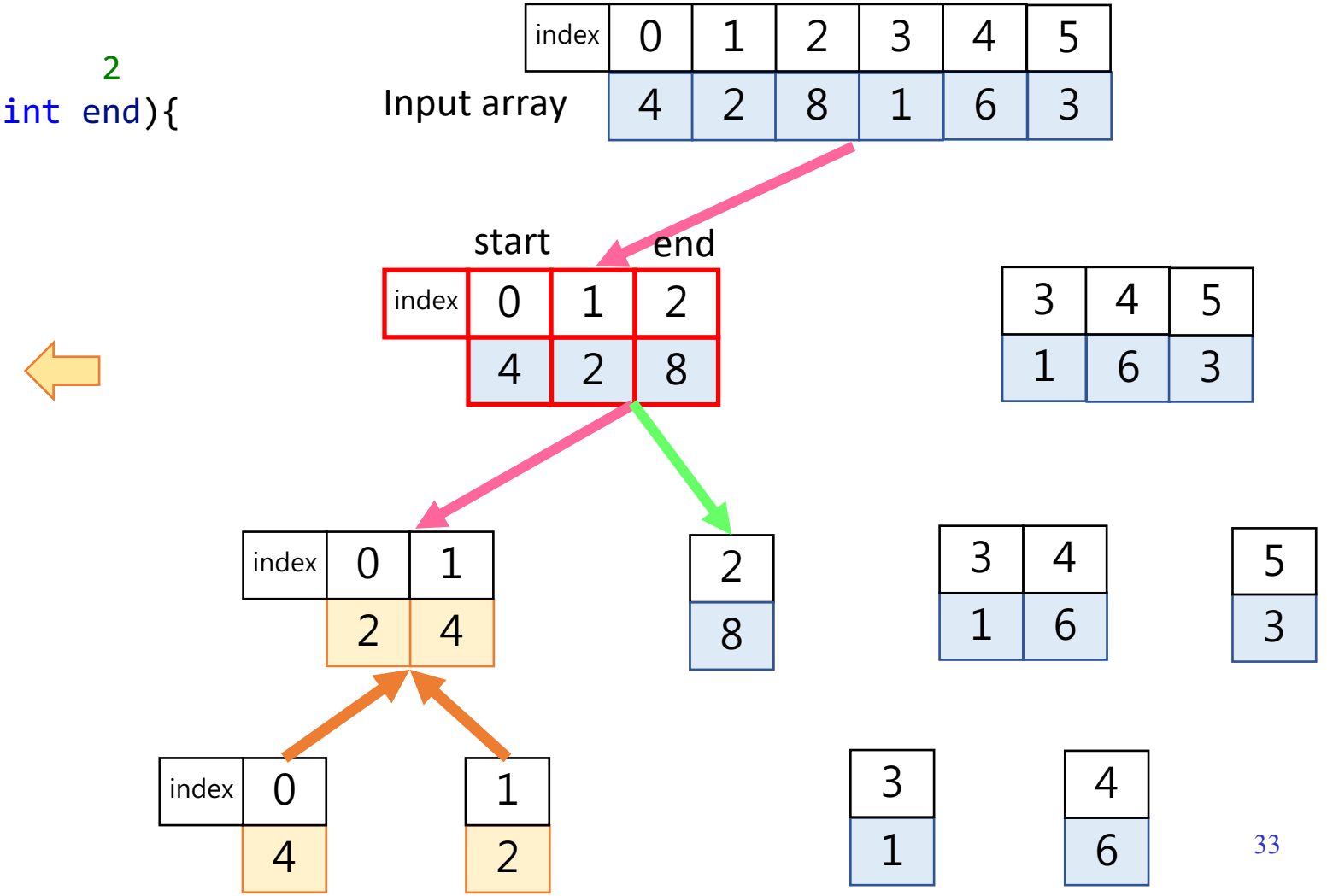
Merge sort - Divide & Conquer

```
//                                  0         1
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                    0     0
        mergeSort(array, start, mid);
        //                    1     1
        mergeSort(array, mid+1, end);
        //                  0     0    1
        merge(array, start, mid, end);
    }
}
```



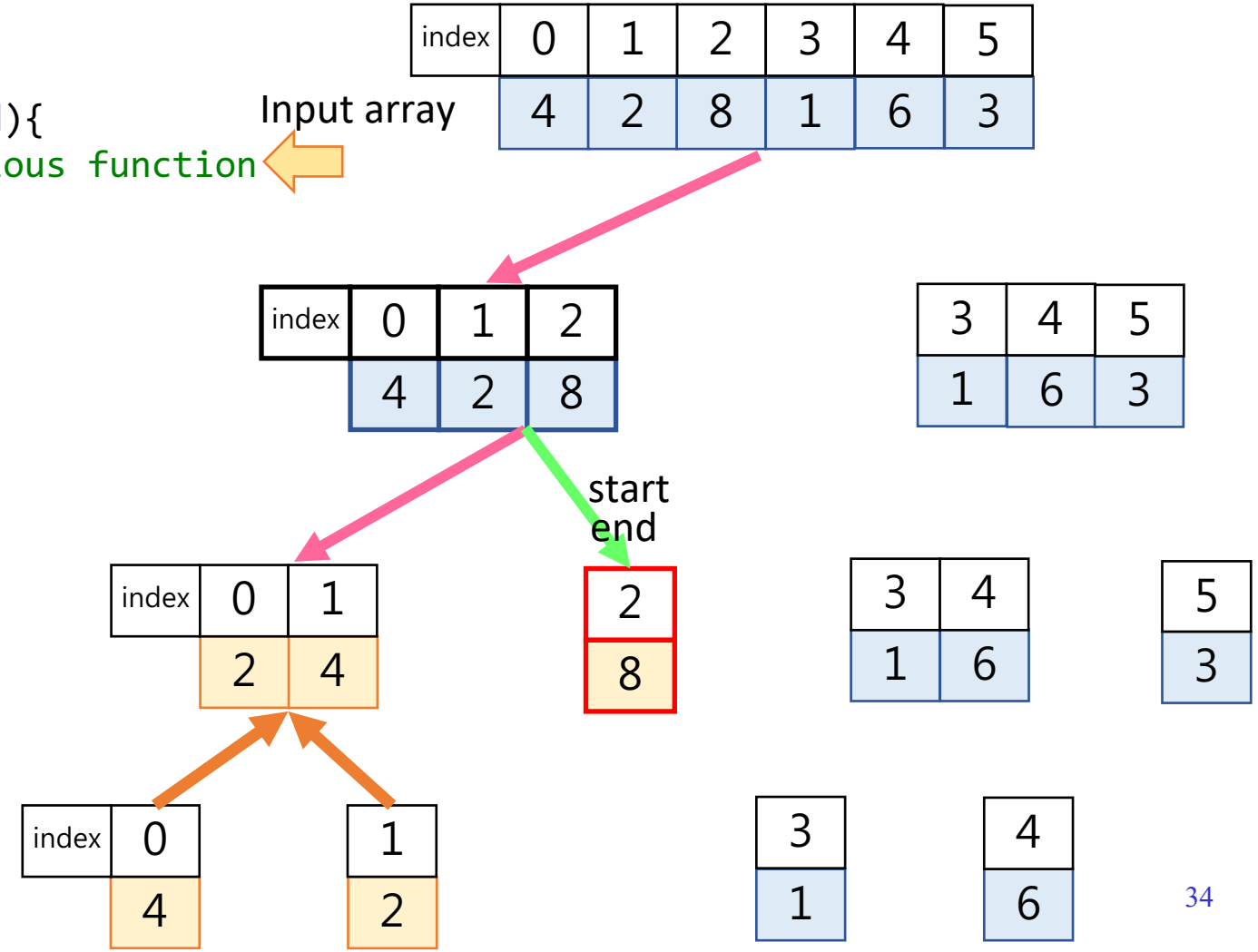
Merge sort - Divide & Conquer

```
//                                  0          2
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                      0     1
        mergeSort(array, start, mid);
        //                      2     2
        mergeSort(array, mid+1, end);
        //                  0     1     2
        merge(array, start, mid, end);
    }
}
```



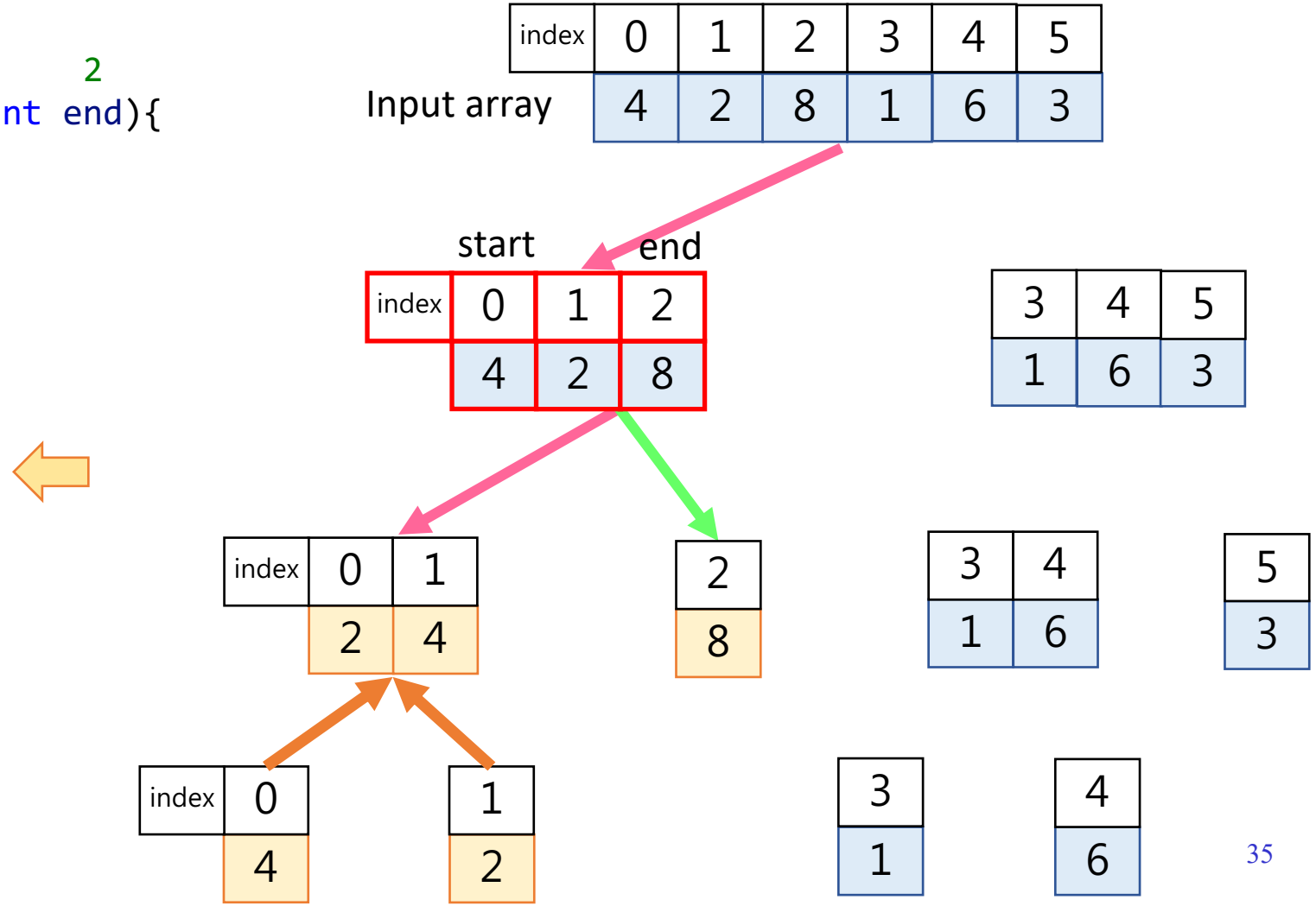
Merge sort - Divide & Conquer

```
//                                  2          2
void mergeSort(int *array, int start, int end){
    if(start < end){}{//2 < 2 return to previous function
        int mid = (end + start)/2;
//
        mergeSort(array, start, mid);
//
        mergeSort(array, mid+1, end);
//
        merge(array, start, mid, end);
    }
}
```



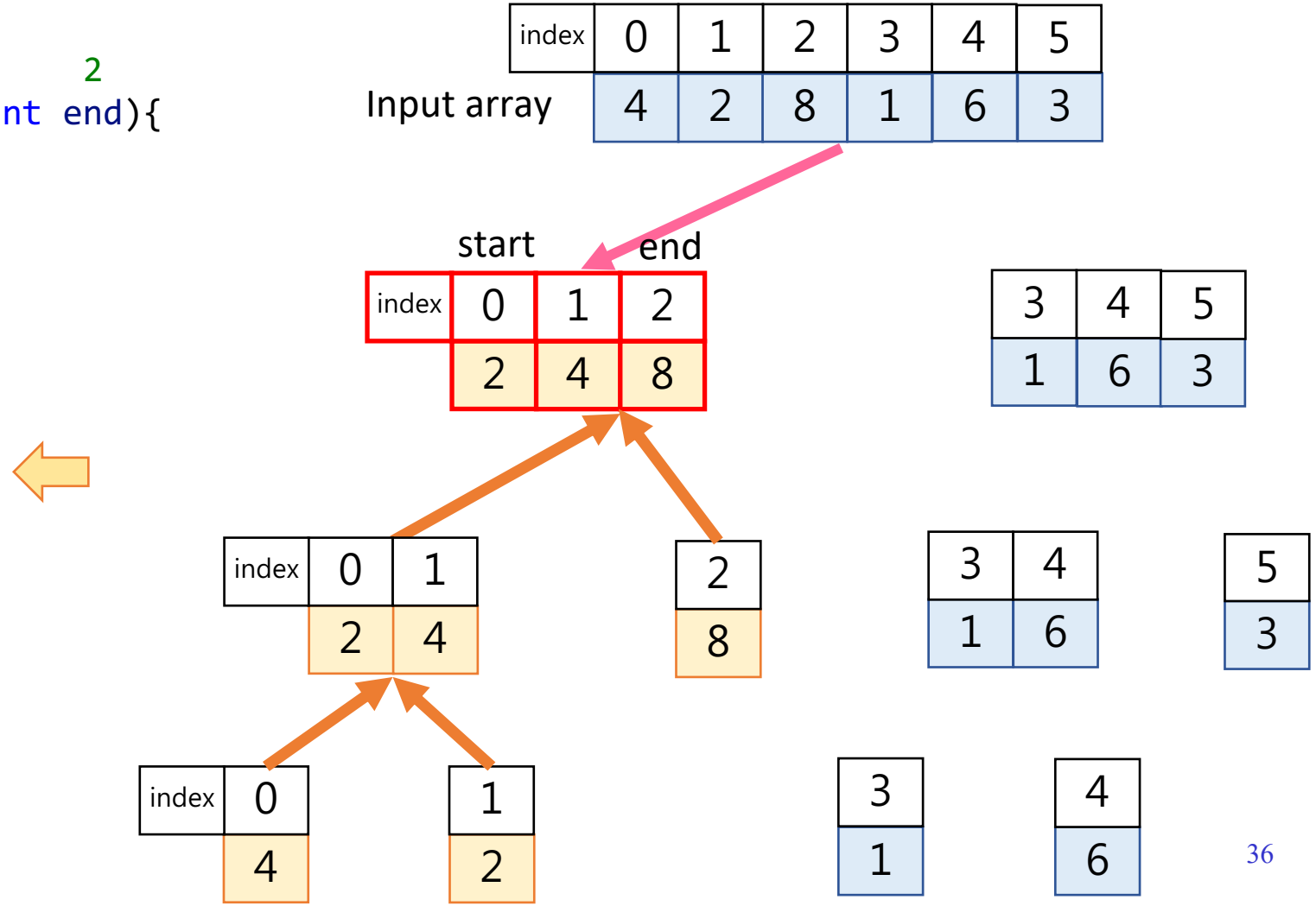
Merge sort - Divide & Conquer

```
//                                  0          2
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                  0     1
        mergeSort(array, start, mid);
        //                  2     2
        mergeSort(array, mid+1, end);
        //                0     1     2
        merge(array, start, mid, end);
    }
}
```



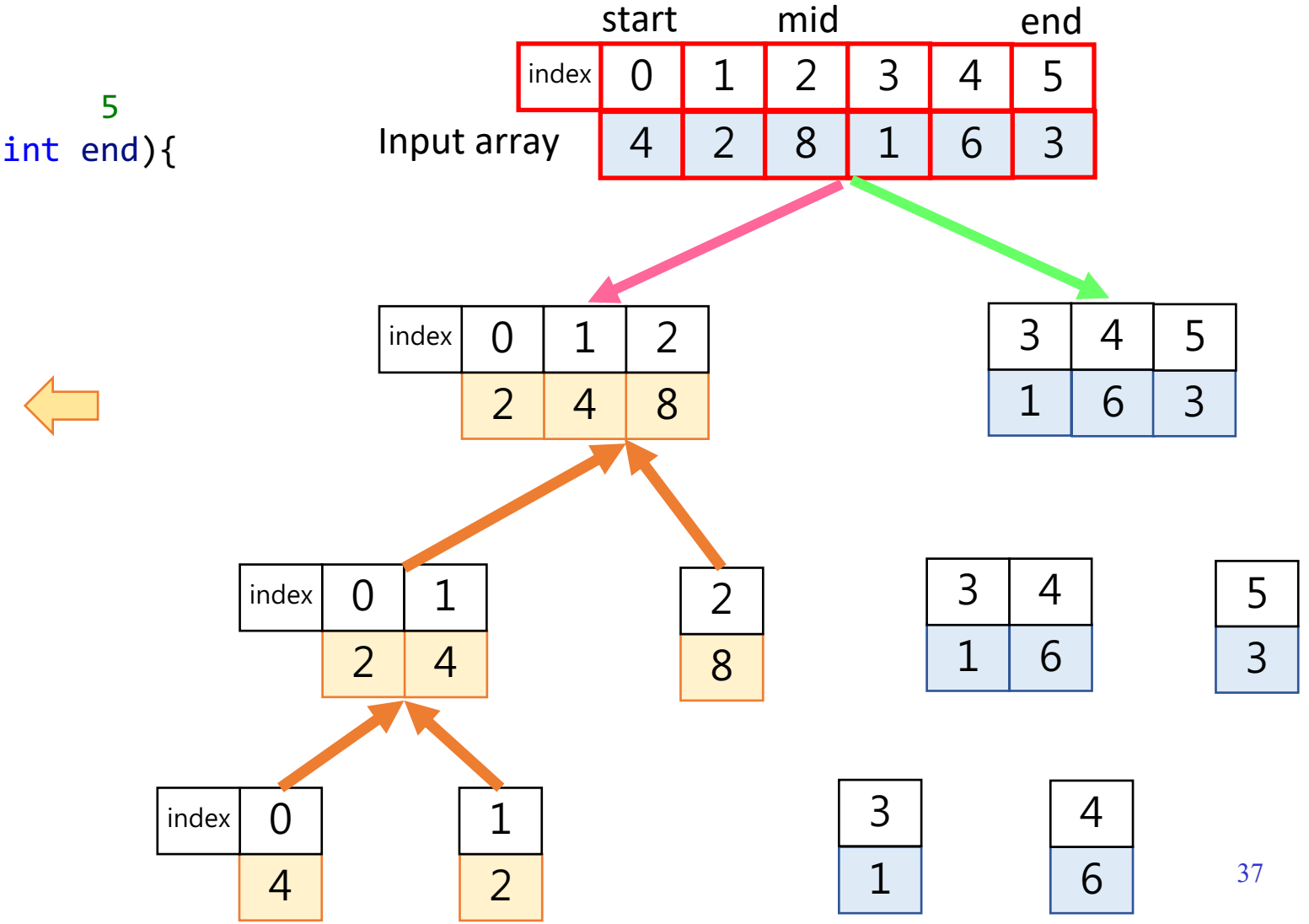
Merge sort - Divide & Conquer

```
//                                  0          2
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                  0     1
        mergeSort(array, start, mid);
        //                  2     2
        mergeSort(array, mid+1, end);
        //                0     1     2
        merge(array, start, mid, end);
    }
}
```



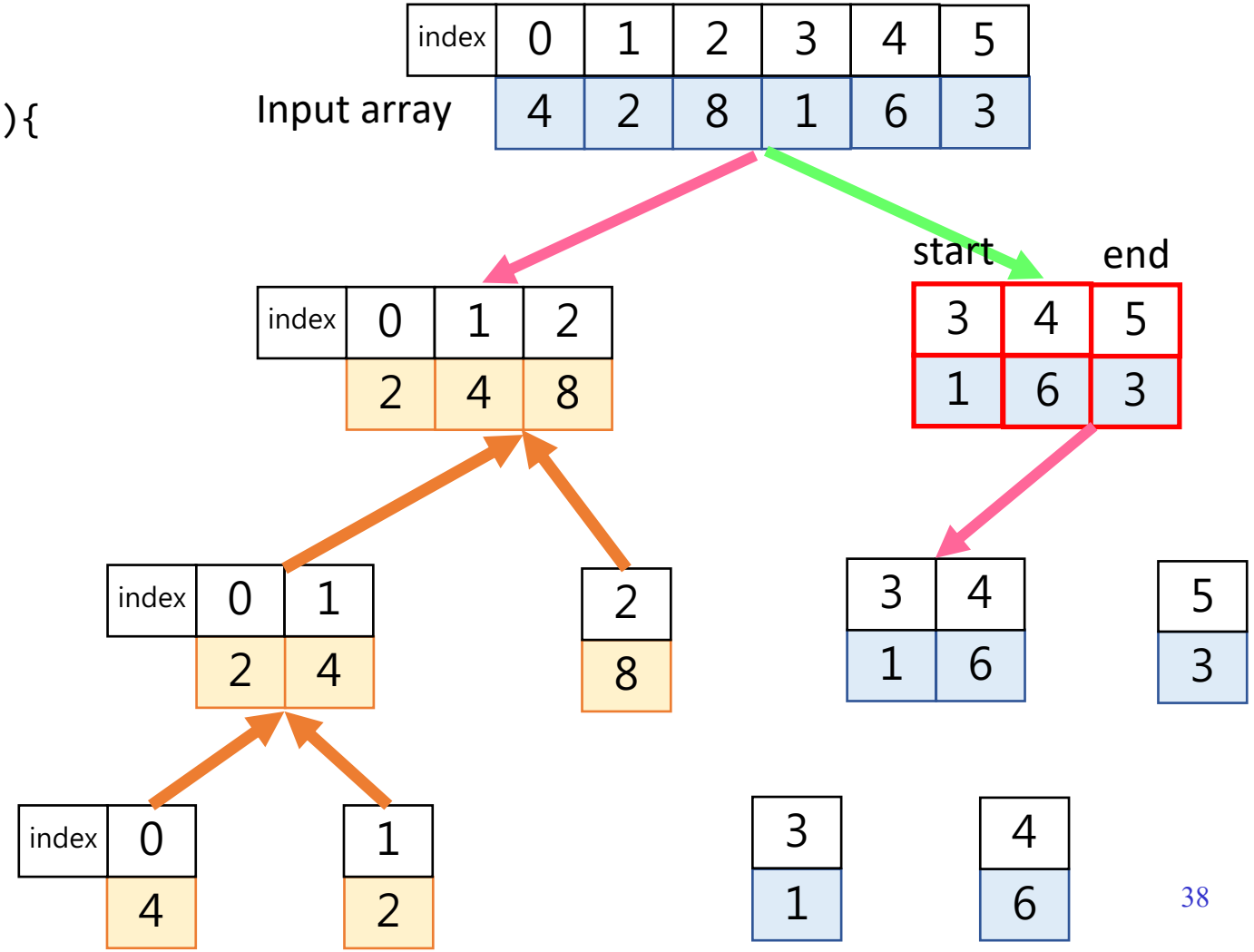
Merge sort - Divide & Conquer

```
//                                  0          5
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                      0     2
        mergeSort(array, start, mid);
        //                      3     5
        mergeSort(array, mid+1, end);
        //                  0     2     5
        merge(array, start, mid, end);
    }
}
```



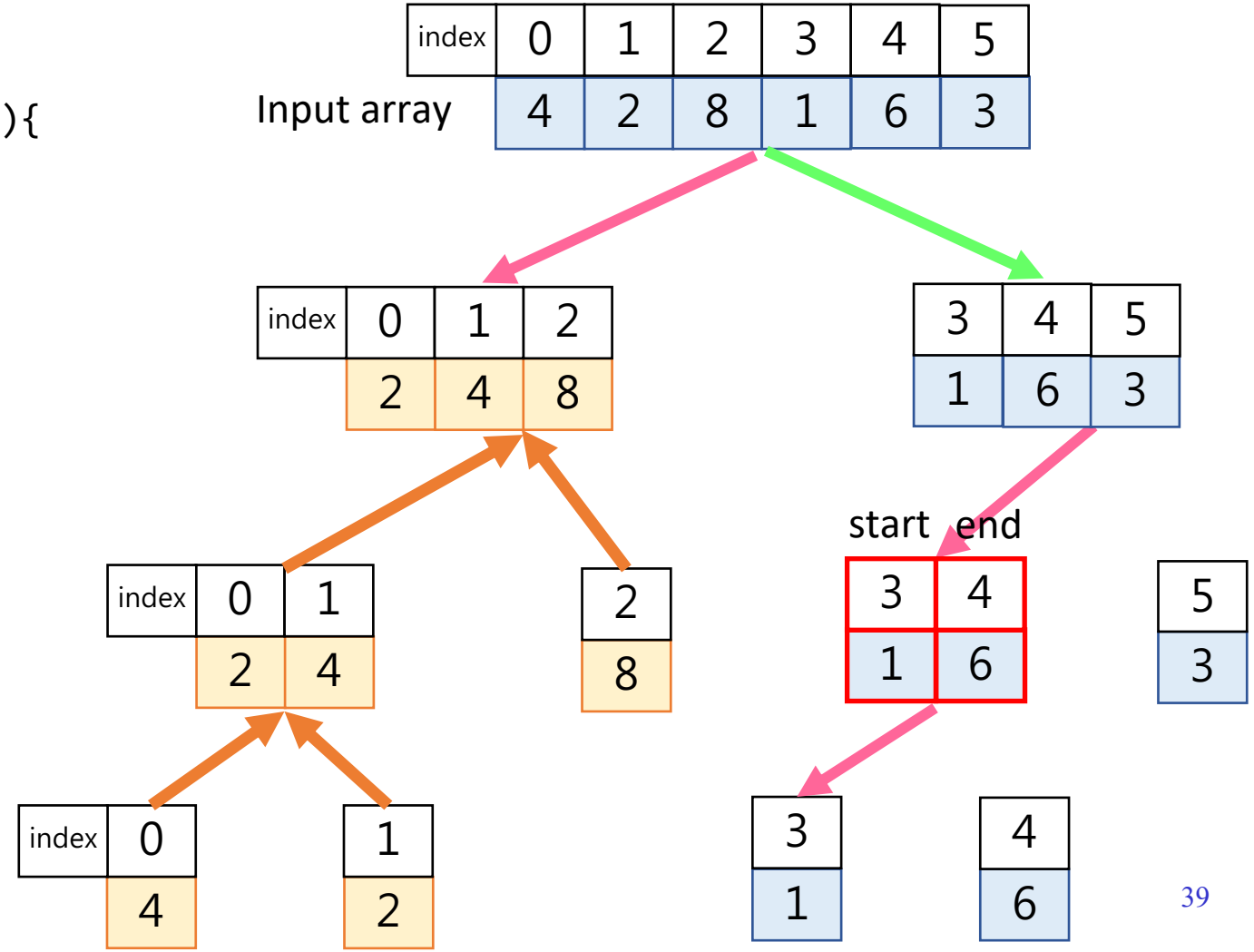
Merge sort - Divide & Conquer

```
//                                  3          5
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                  3     4
        mergeSort(array, start, mid);
        //                  5     5
        mergeSort(array, mid+1, end);
        //                3     4     5
        merge(array, start, mid, end);
    }
}
```



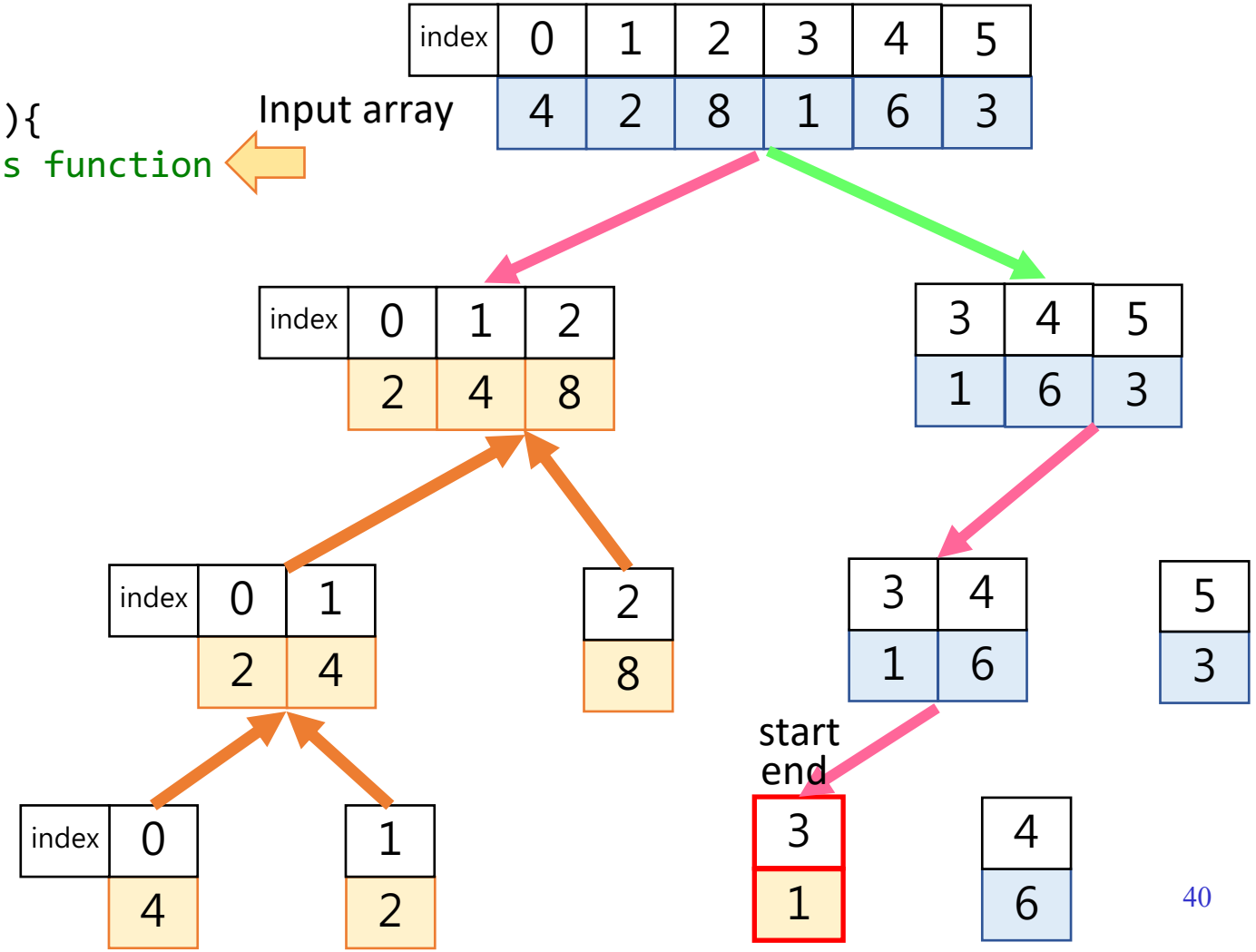
Merge sort - Divide & Conquer

```
//                                  3         4
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                  3     3
        mergeSort(array, start, mid);
        //                  4     4
        mergeSort(array, mid+1, end);
        //                3     3    4
        merge(array, start, mid, end);
    }
}
```



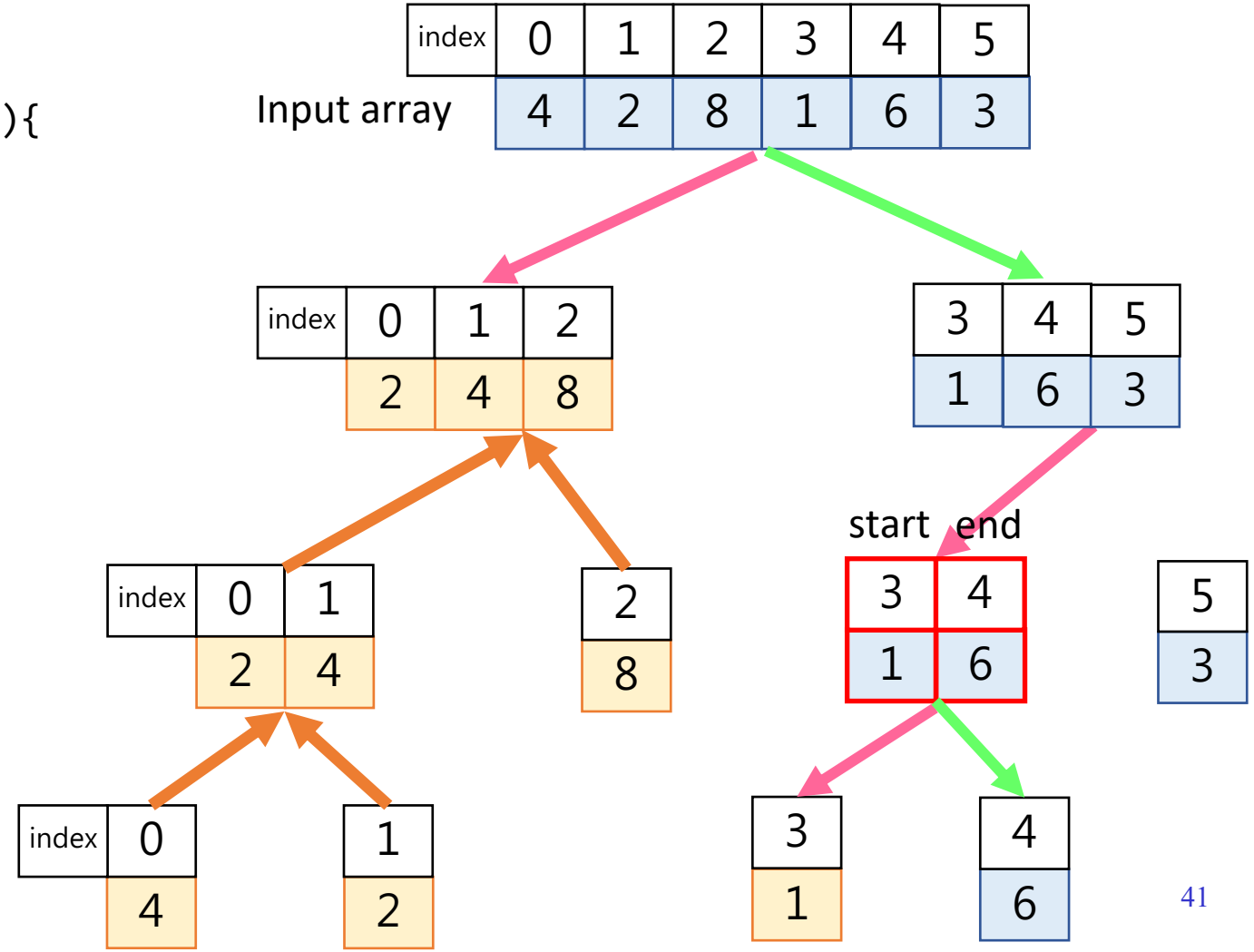
Merge sort - Divide & Conquer

```
//                                  3         3
void mergeSort(int *array, int start, int end){
    if(start < end){//3 < 3 return to previous function
        int mid = (end + start)/2;
//
        mergeSort(array, start, mid);
//
        mergeSort(array, mid+1, end);
//
        merge(array, start, mid, end);
    }
}
```



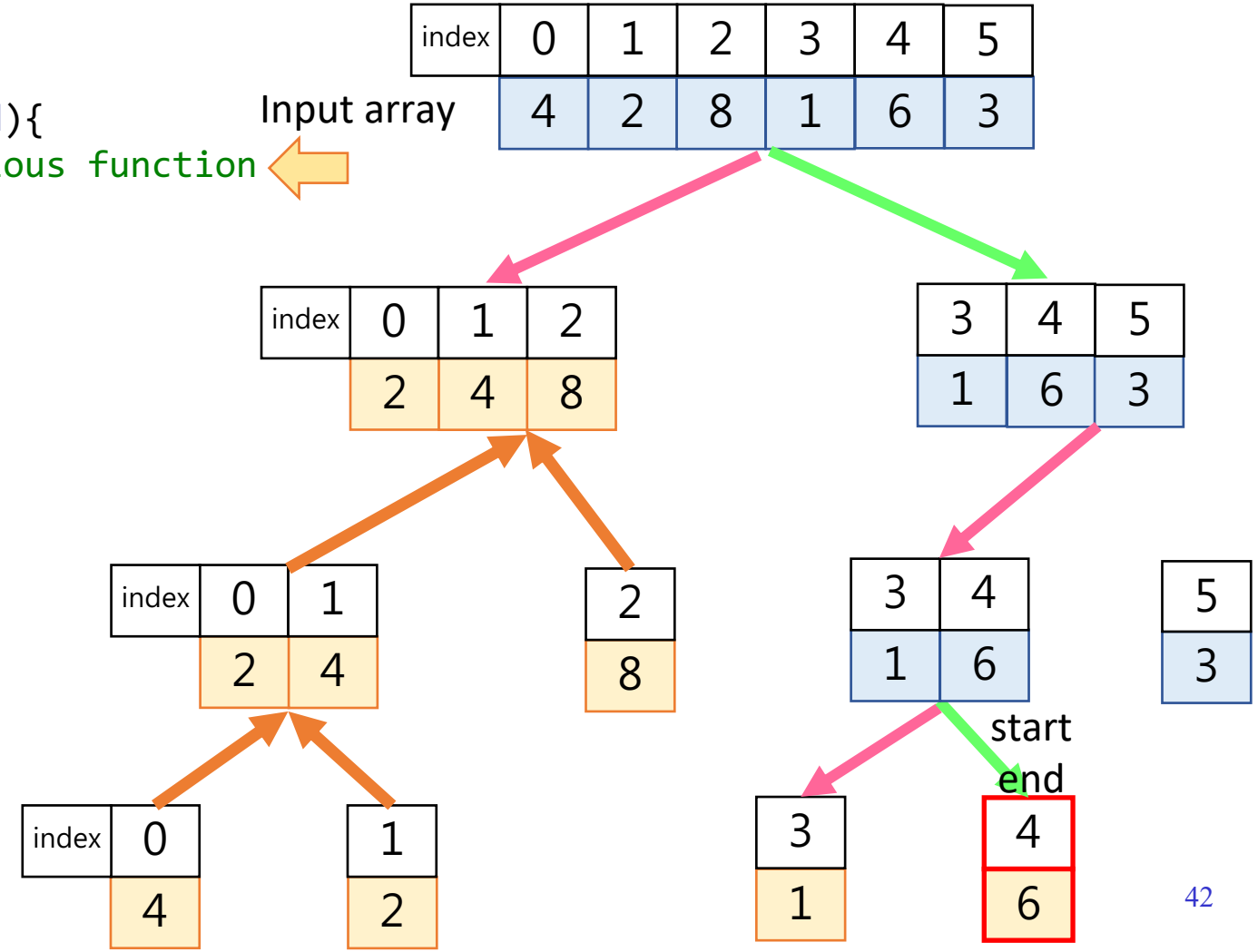
Merge sort - Divide & Conquer

```
//                                  3          4
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                    3    3
        mergeSort(array, start, mid);
        //                    4     4
        mergeSort(array, mid+1, end);
        //                  3     3    4
        merge(array, start, mid, end);
    }
}
```



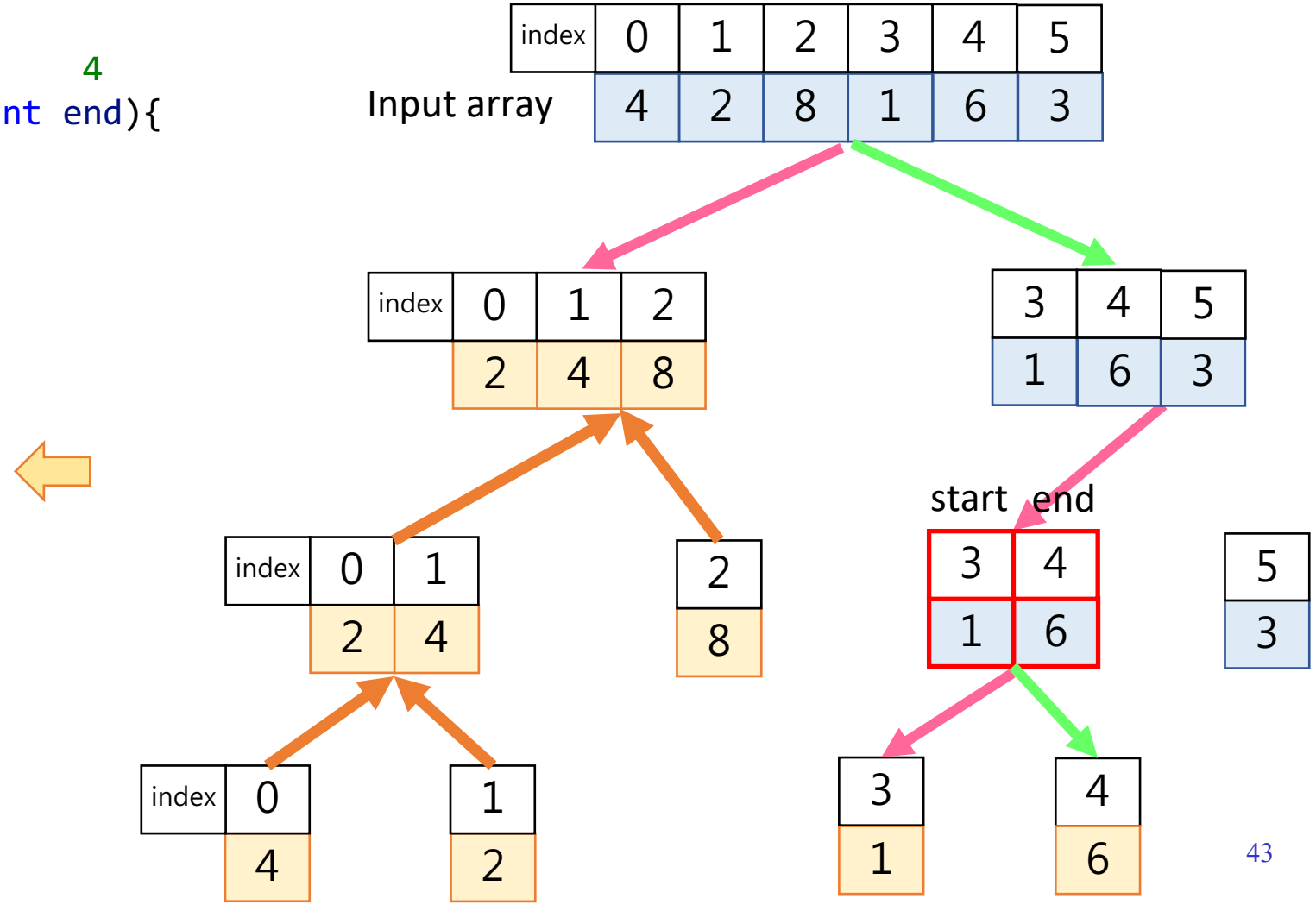
Merge sort - Divide & Conquer

```
//                                  4         4
void mergeSort(int *array, int start, int end){
    if(start < end){}{//4 < 4 return to previous function
        int mid = (end + start)/2;
//
        mergeSort(array, start, mid);
//
        mergeSort(array, mid+1, end);
//
        merge(array, start, mid, end);
    }
}
```



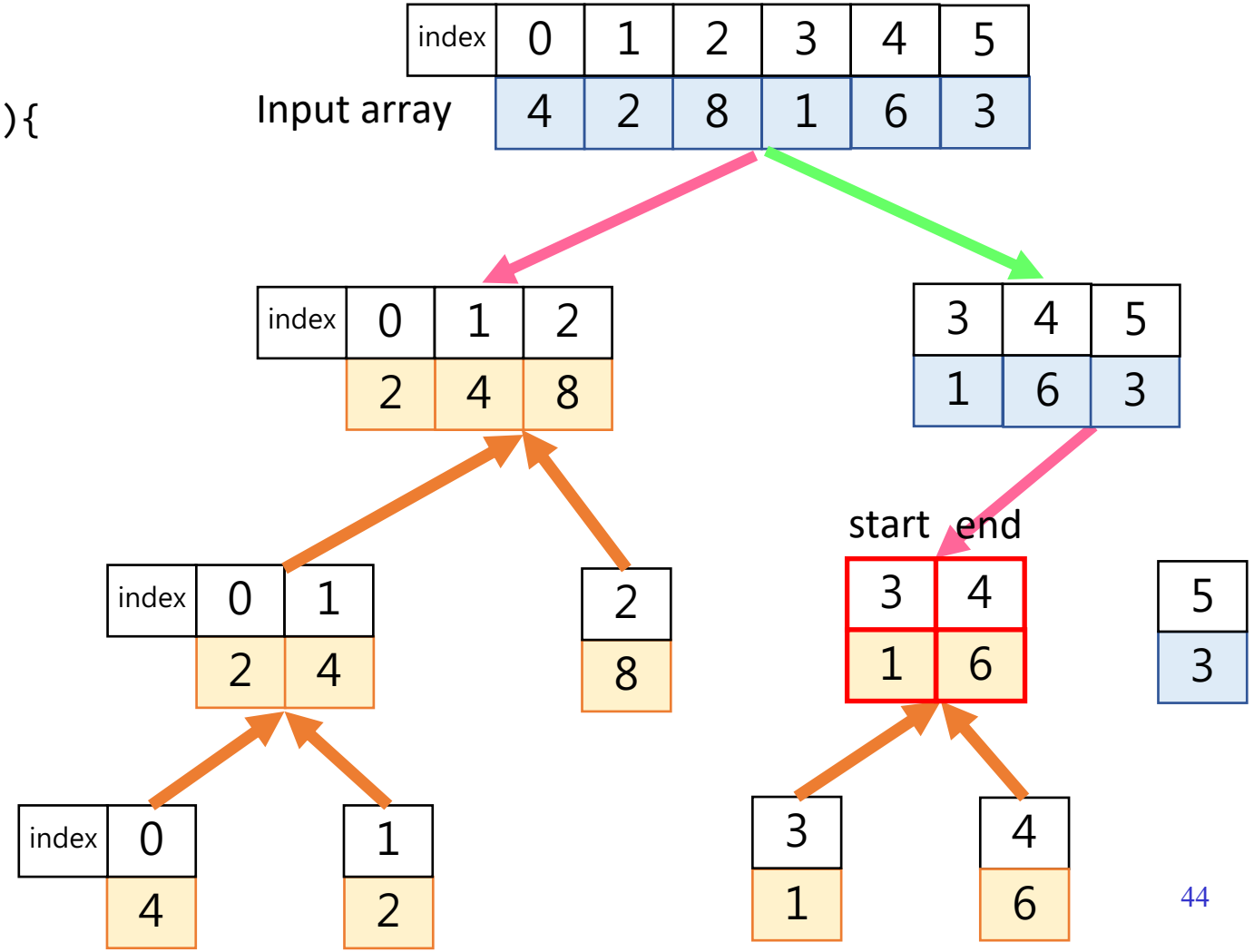
Merge sort - Divide & Conquer

```
//                                  3          4
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                      3     3
        mergeSort(array, start, mid);
        //                      4     4
        mergeSort(array, mid+1, end);
        //                  3     3     4
        merge(array, start, mid, end);
    }
}
```



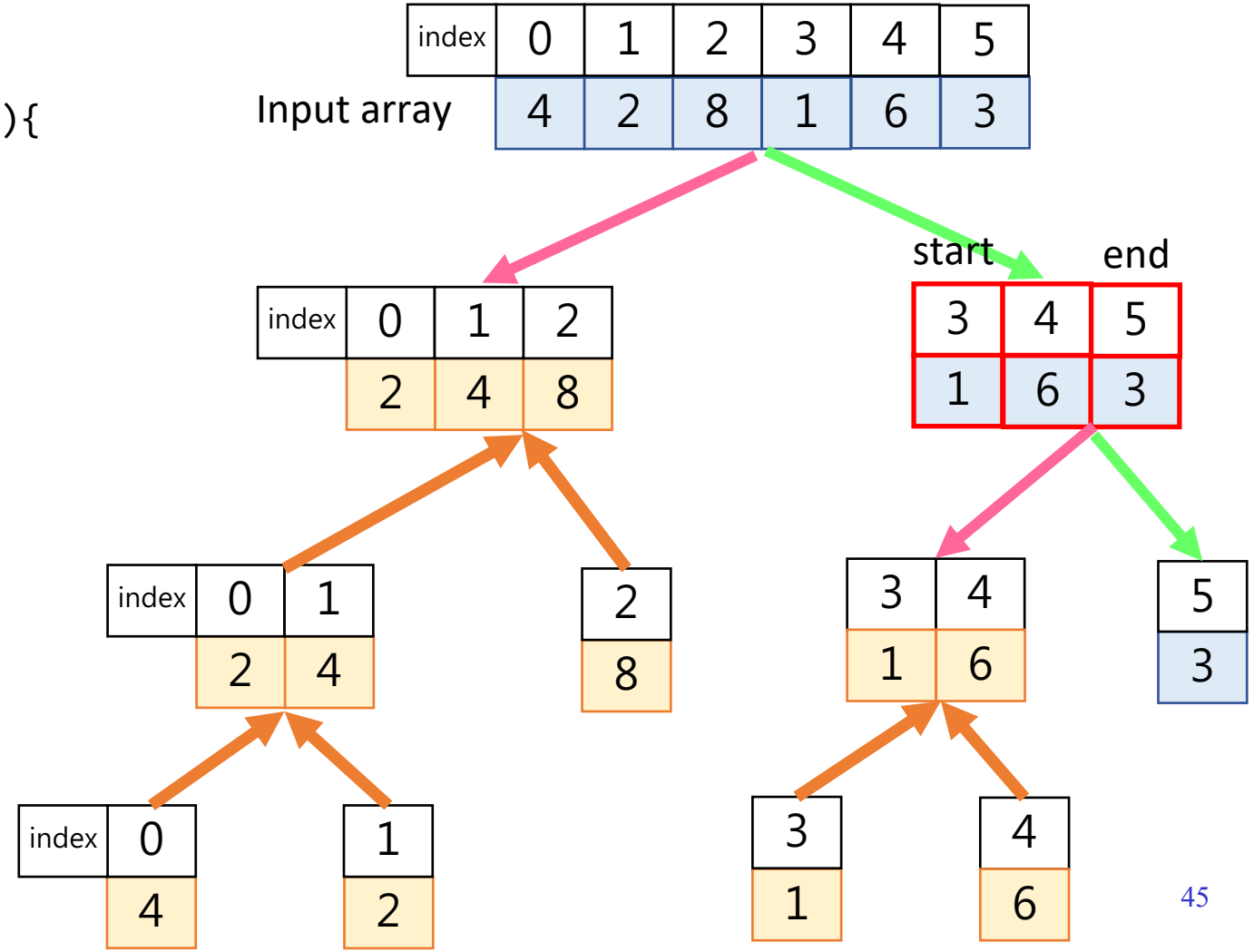
Merge sort - Divide & Conquer

```
//                                  3         4
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                  3     3
        mergeSort(array, start, mid);
        //                  4     4
        mergeSort(array, mid+1, end);
        //                3     3     4
        merge(array, start, mid, end);
    }
}
```



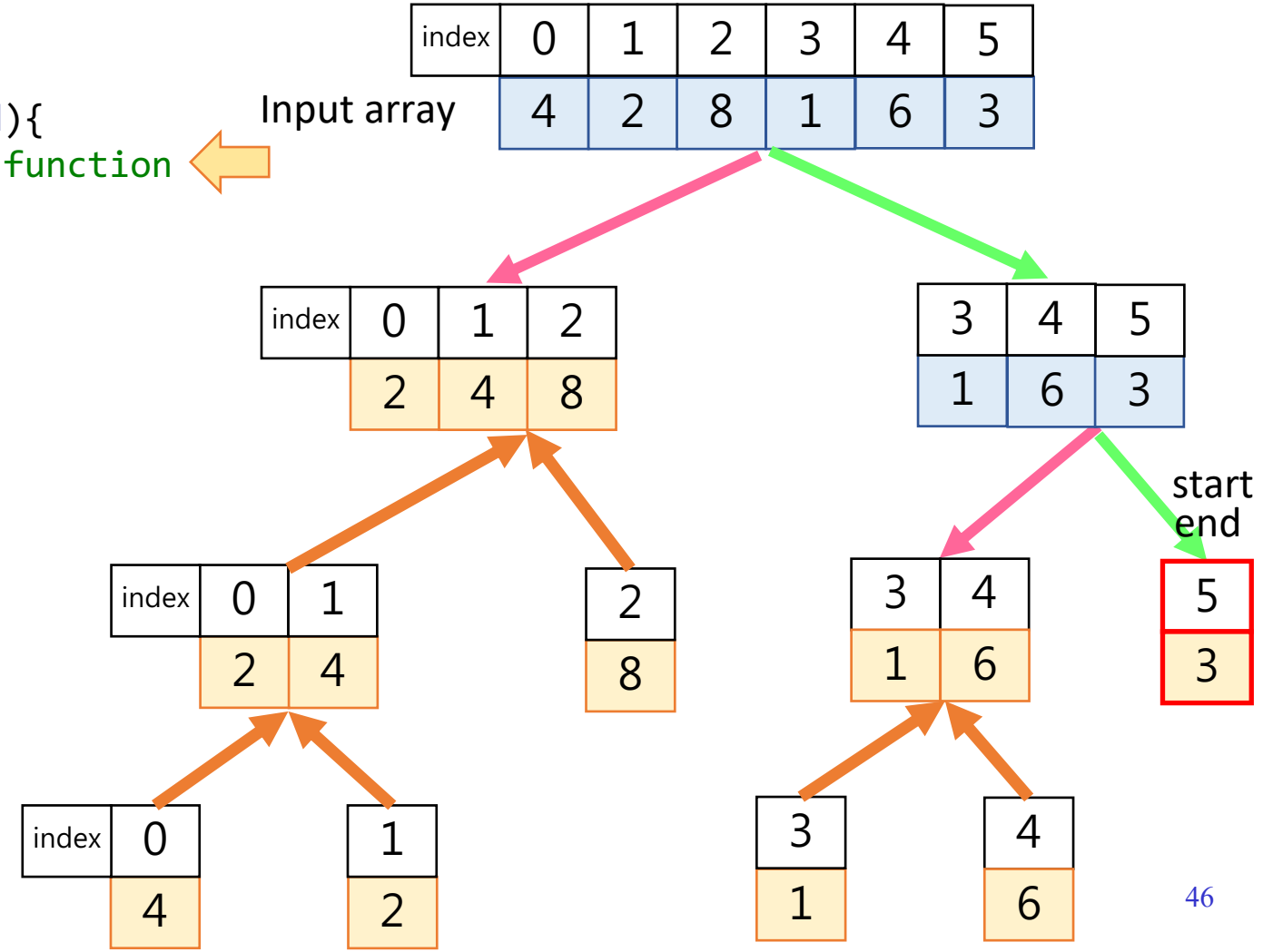
Merge sort - Divide & Conquer

```
//                                  3          5
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                  3     4
        mergeSort(array, start, mid);
        //                  5     5
        mergeSort(array, mid+1, end);
        //                3     4     5
        merge(array, start, mid, end);
    }
}
```



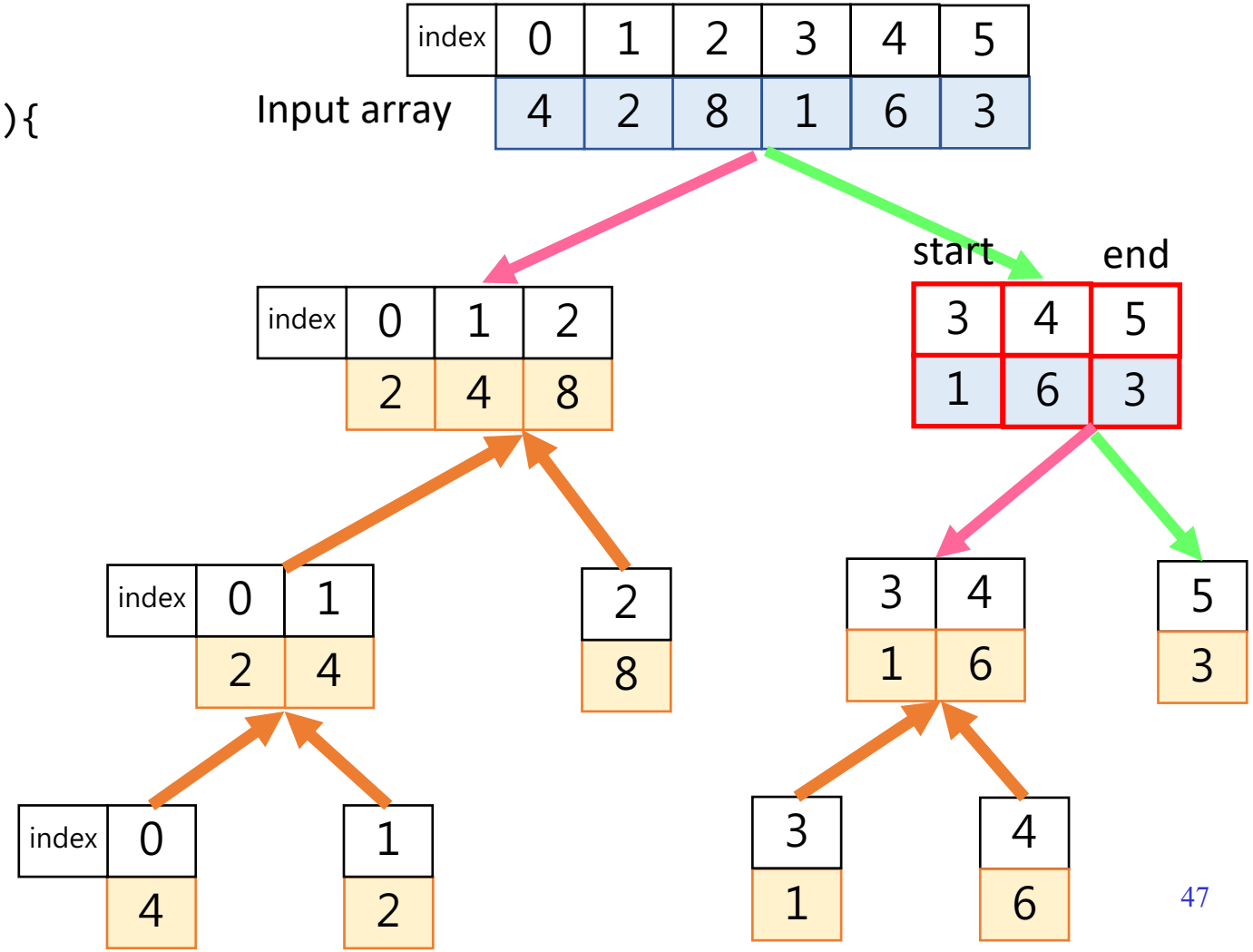
Merge sort - Divide & Conquer

```
//                                  5         5
void mergeSort(int *array, int start, int end){
    if(start < end){//5 < 5 back to previous function
        int mid = (end + start)/2;
//
        mergeSort(array, start, mid);
//
        mergeSort(array, mid+1, end);
//
        merge(array, start, mid, end);
    }
}
```



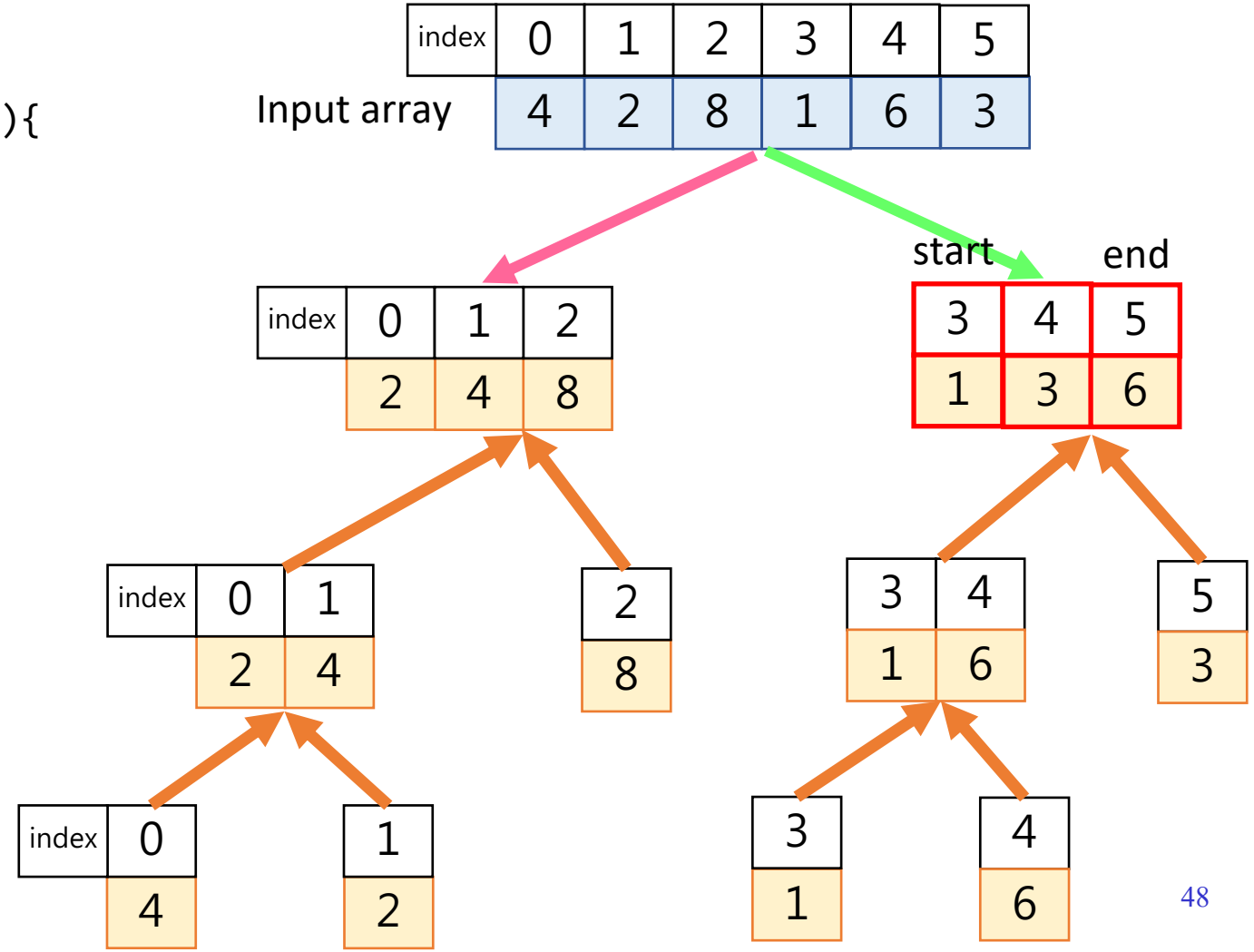
Merge sort - Divide & Conquer

```
//                                  3          5
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                  3     4
        mergeSort(array, start, mid);
        //                  5     5
        mergeSort(array, mid+1, end);
        //                0     3     5
        merge(array, start, mid, end);
    }
}
```



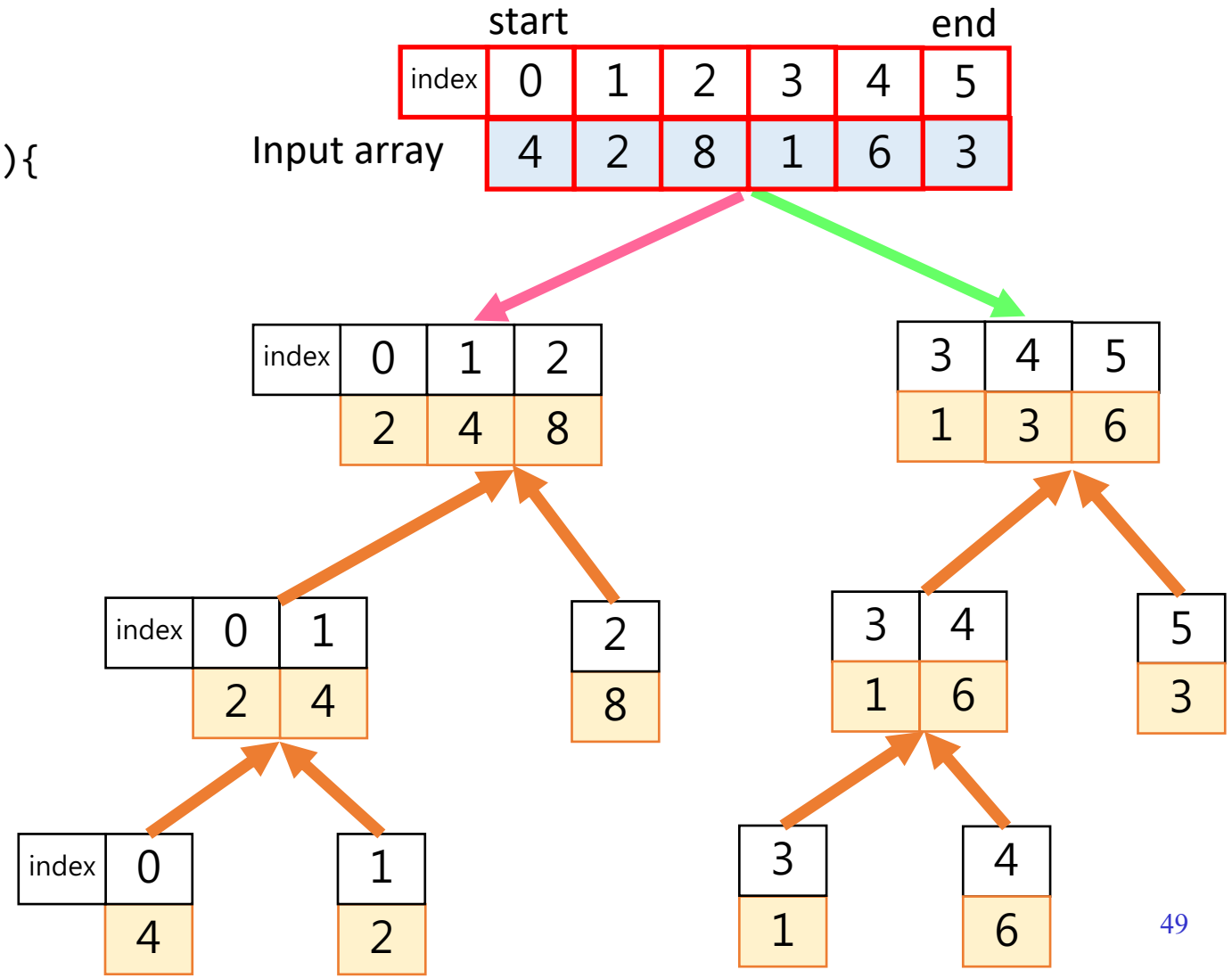
Merge sort - Divide & Conquer

```
//                                3          5
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                    3     4
        mergeSort(array, start, mid);
        //                    5     5
        mergeSort(array, mid+1, end);
        //                0     3     5
        merge(array, start, mid, end);
    }
}
```



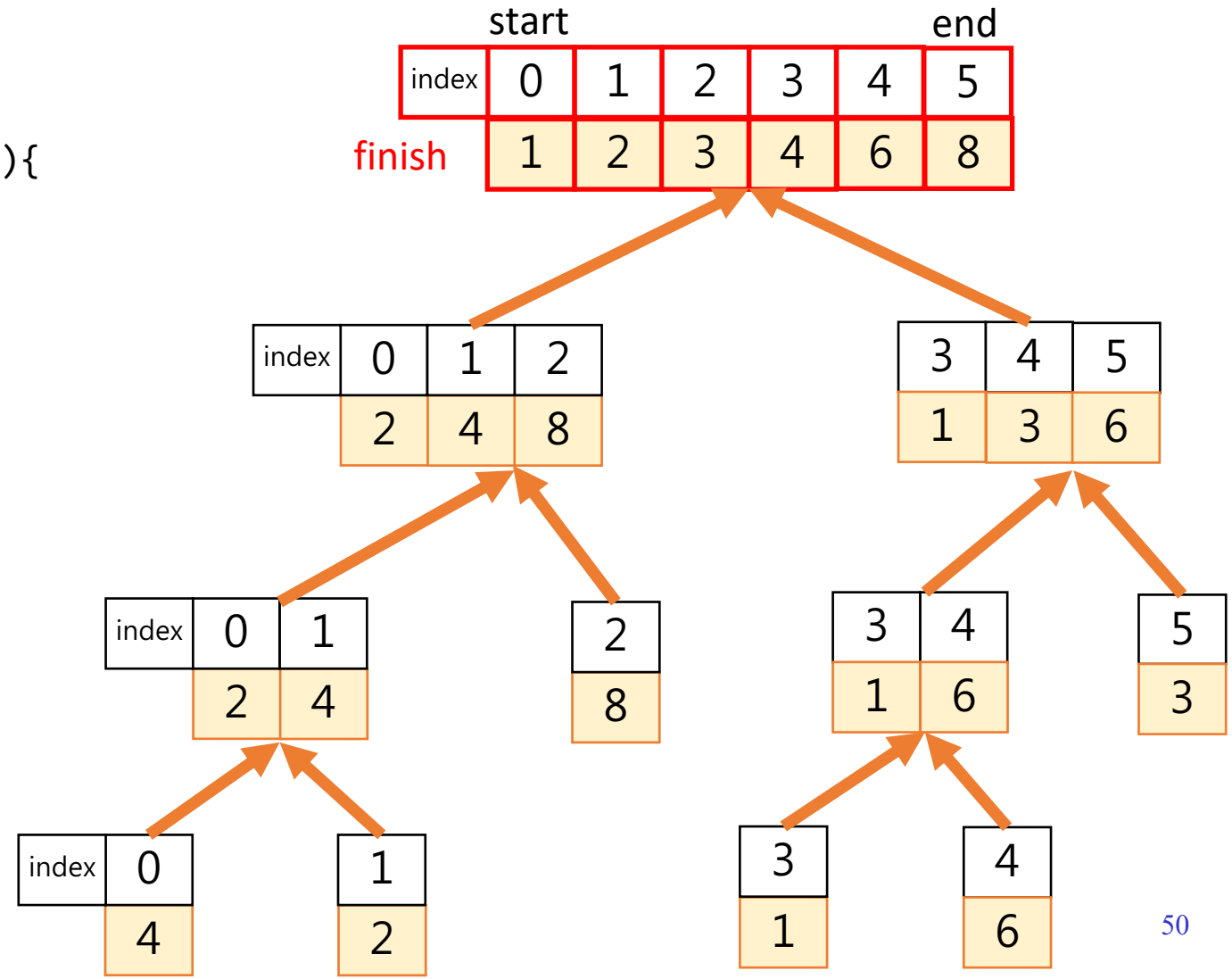
Merge sort - Divide & Conquer

```
//                                  0          5
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                      0     2
        mergeSort(array, start, mid);
        //                      3     5
        mergeSort(array, mid+1, end);
        //                  0     2     5
        merge(array, start, mid, end);
    }
}
```



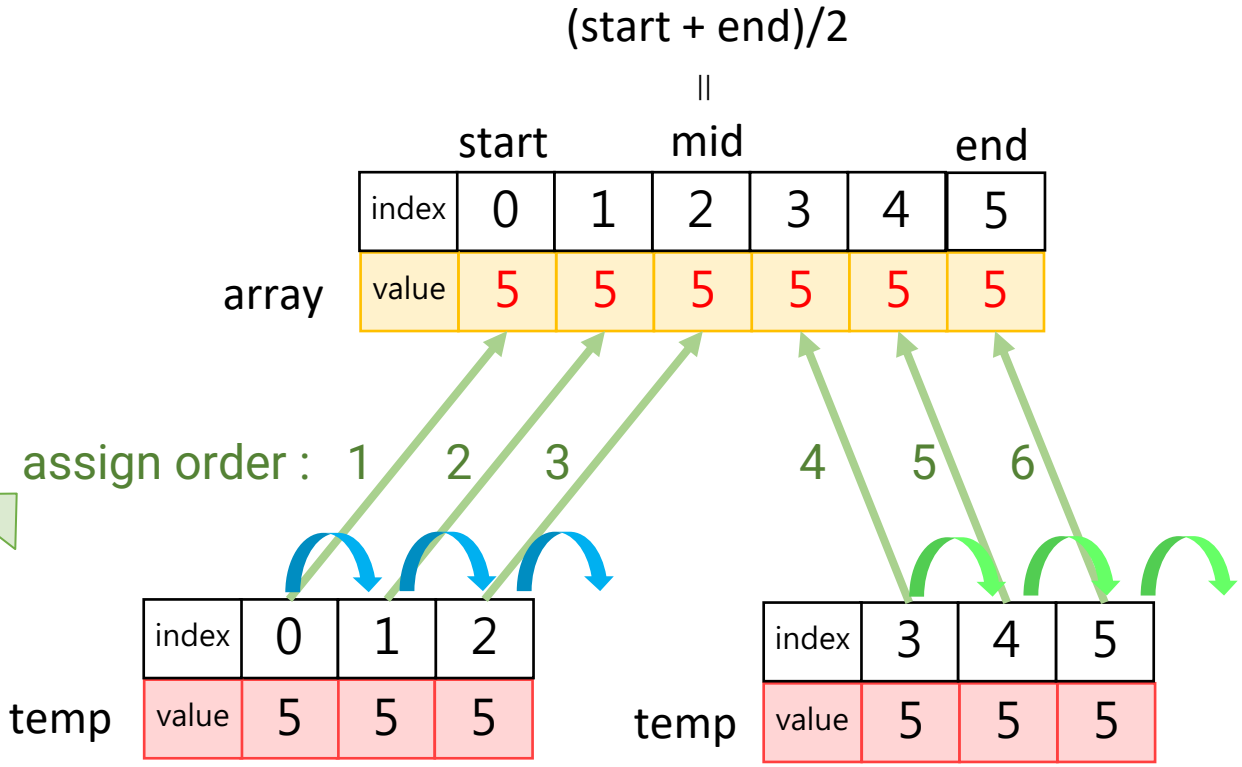
Merge sort - Divide & Conquer

```
//                                  0          5
void mergeSort(int *array, int start, int end){
    if(start < end){
        int mid = (end + start)/2;
        //                      0     2
        mergeSort(array, start, mid);
        //                      3     5
        mergeSort(array, mid+1, end);
        //                  0     2     5
        merge(array, start, mid, end);
    }
}
```



Merge sort - Why is it stable ?

```
merge(int *array, int start, int mid, int end) {
    while (i <= mid && j <= end) {
        if (temp[ i ] <= temp[ j ]) {
            array[ k ] = array[ i ];
                k++;
                i++;
        }
        else {
            array[ k ] = temp[ j ];
            k++;
            j++;
        }
    }
    while(i <= mid) {
        array[ k ] = temp[ i ];
            k++;
            i++;
    }
    while(j <= end) {
        array[ k ] = temp[ j ];
        k++;
        j++;
    }
}
```



Merge sort - Time complexity of merge function

merge function

array

index	0	1	2	3	4	5
value	1	2	3	4	6	8

$N = x + y = 6$

Best compare times : x

temp

index	0	1	2
value	1	2	3

x = 3

temp

index	3	4	5
value	4	6	8

y = 3

Worst compare times : x + y - 1

temp

index	0	1	2
value	2	4	8

x = 3

temp

index	3	4	5
value	1	3	6

y = 3

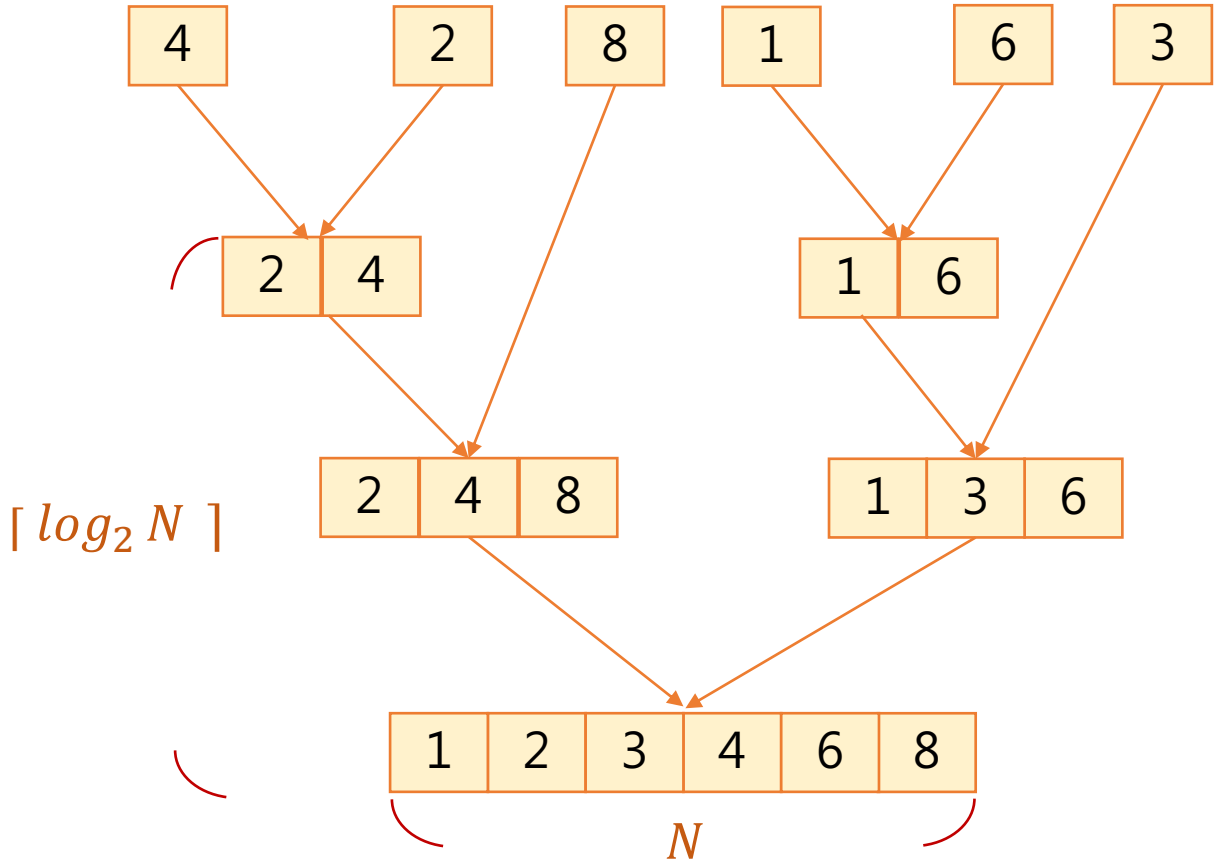
Time complexity of merge function : $O(N)$

Merge sort - Time complexity

Compare times

Merge function in every layer : $O\ (\ N\ )$

Number of layers :
$\lceil\ log_2 N\ \rceil \Rightarrow O\ (\ log_2 N\ )$



Time complexity :

$O\ (\ N\ ) * O\ (\ log_2 N\ ) = O\ (\ N\ log_2 N\ )$