Information Technology

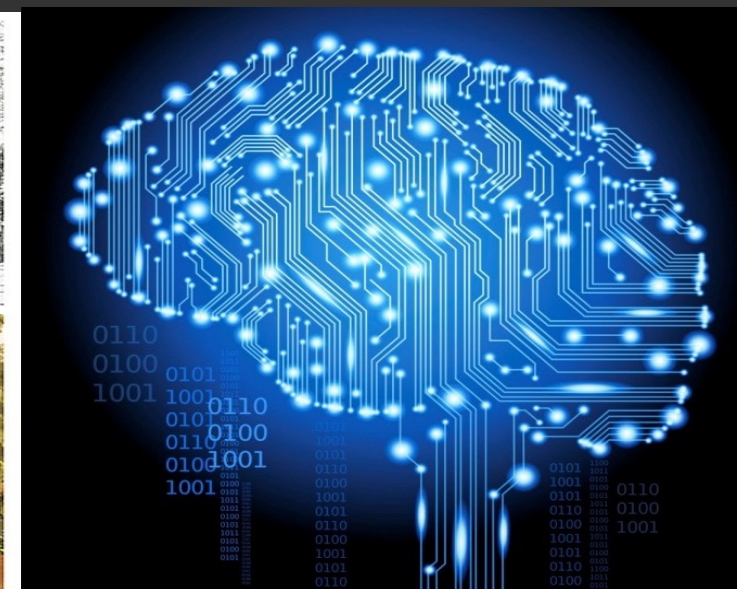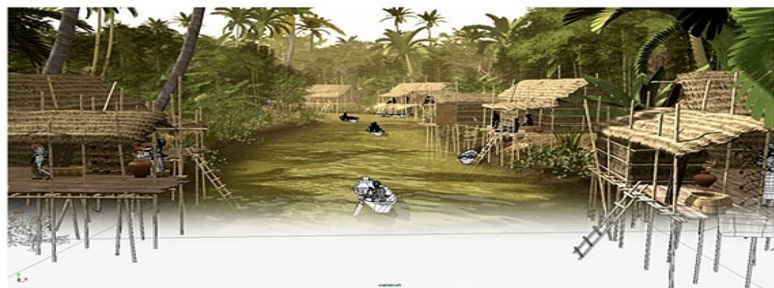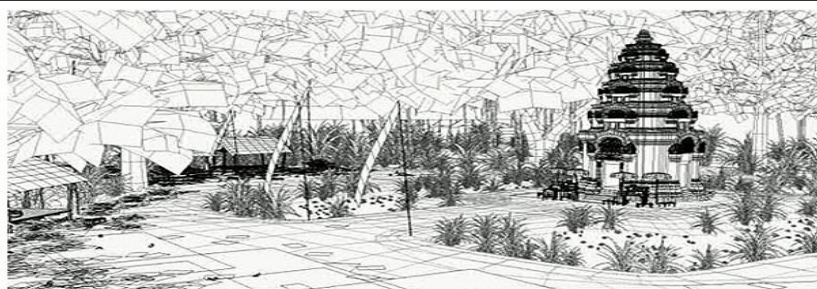# Recursive Sorts

Prepared by Maria Garcia de la Banda
Updated by Brendon Taylor

Overview

- **To review what a "divide and conquer" algorithm is**
- **To review in more depth two different "divide and conquer" sorting algorithms:**
  - Merge Sort
  - Quick Sort

- **To be able to implement them and compare their efficiency for different classes of inputs**
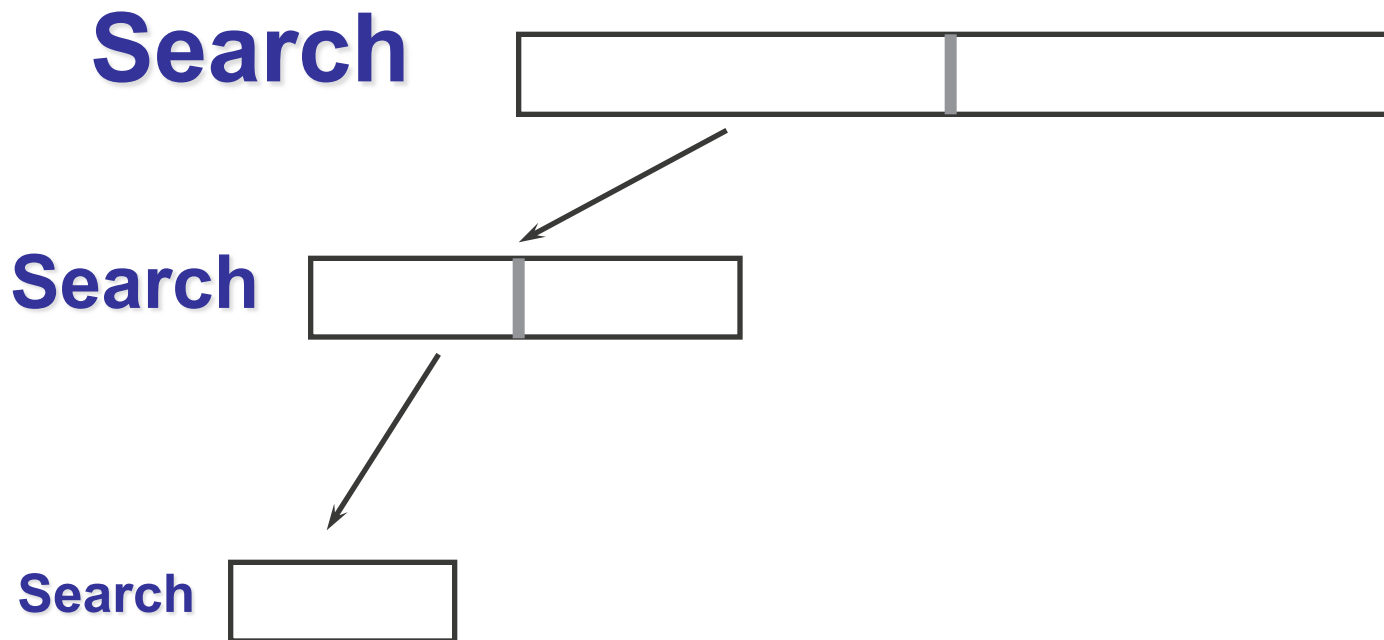
Divide and Conquer

Divide-and-conquer algorithms

- **One common recursive strategy: divide-and-conquer**
  - Split the original problem into subproblems
  - Solve each subproblem independently
  - Combine their solutions to yield the final solution

- **Such algorithms can be very efficient, especially when the subproblems have roughly the same size**
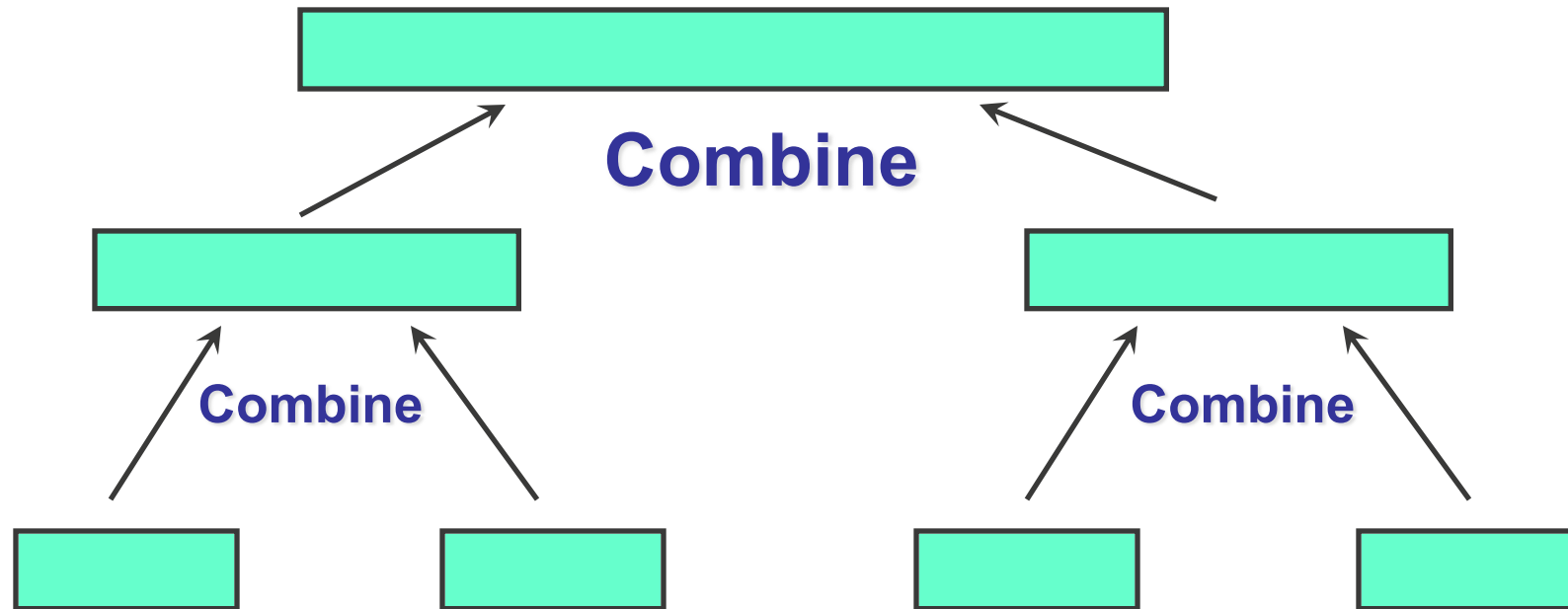
Searching by Divide and Conquer

Recall: Binary Search

Sorting arrays by Divide and Conquer
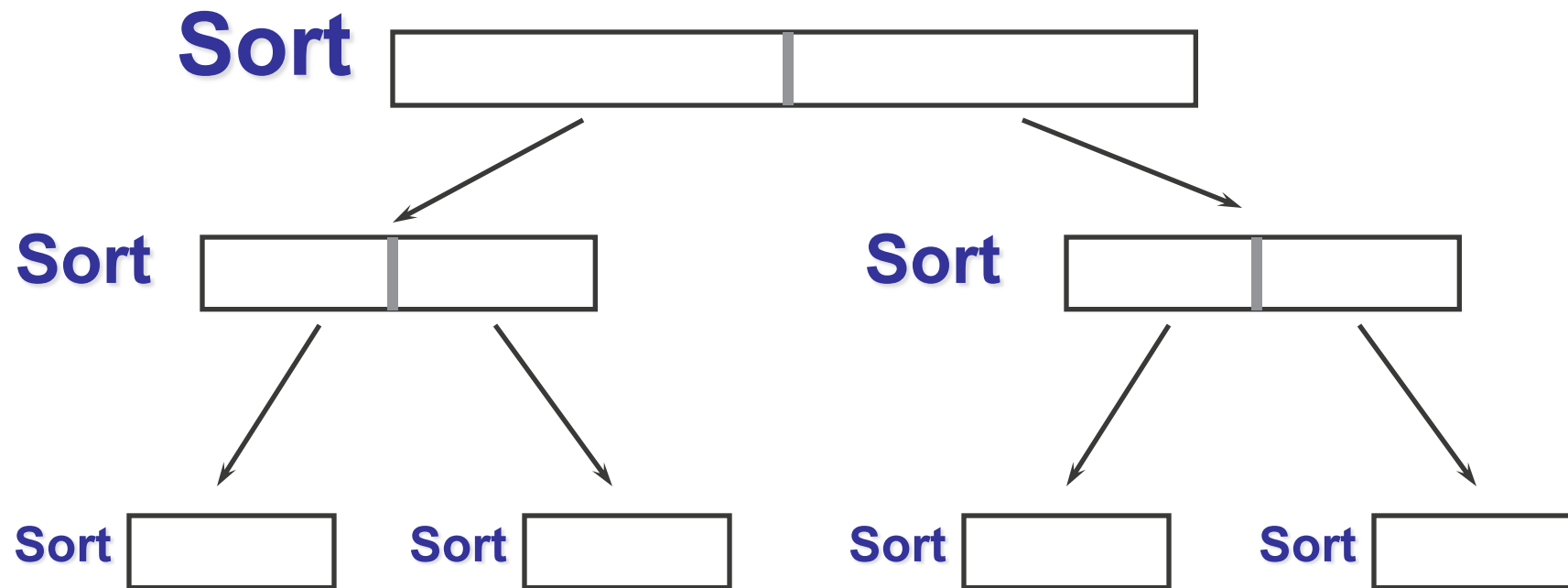
Sorting arrays by Divide and Conquer

Sorting arrays by Divide and Conquer

- **The general strategy is (usually – not always -- for an array or a list implemented with arrays):**

```
def sort(array: ArrayR) -> None:
    if len(array) > 1:
        split(array, first_part, second_part)
        sort(first_part)
        sort(second_part)
        combine(first_part, second_part)
```

Why len > 1?

If len <=1, it is already sorted

- **Merge Sort has a simple split and a complex combine**
- **Quick Sort has a complex split and a simple combine**

Merge Sort

Merge Sort

- **The split is trivial: splits the array into two halves**
- **The combination is non-trivial: merges two sorted halves into a sorted array**

Split

Sort

Sort

Merge

Merge Sort

- **For this it requires a temporary array of the same size as the original array**

array:

$$len(array)$$

tmp:

Merge Sort

array:

merge_sort

tmp:

MONASH University

Merge Sort

array:

tmp:

MONASH University
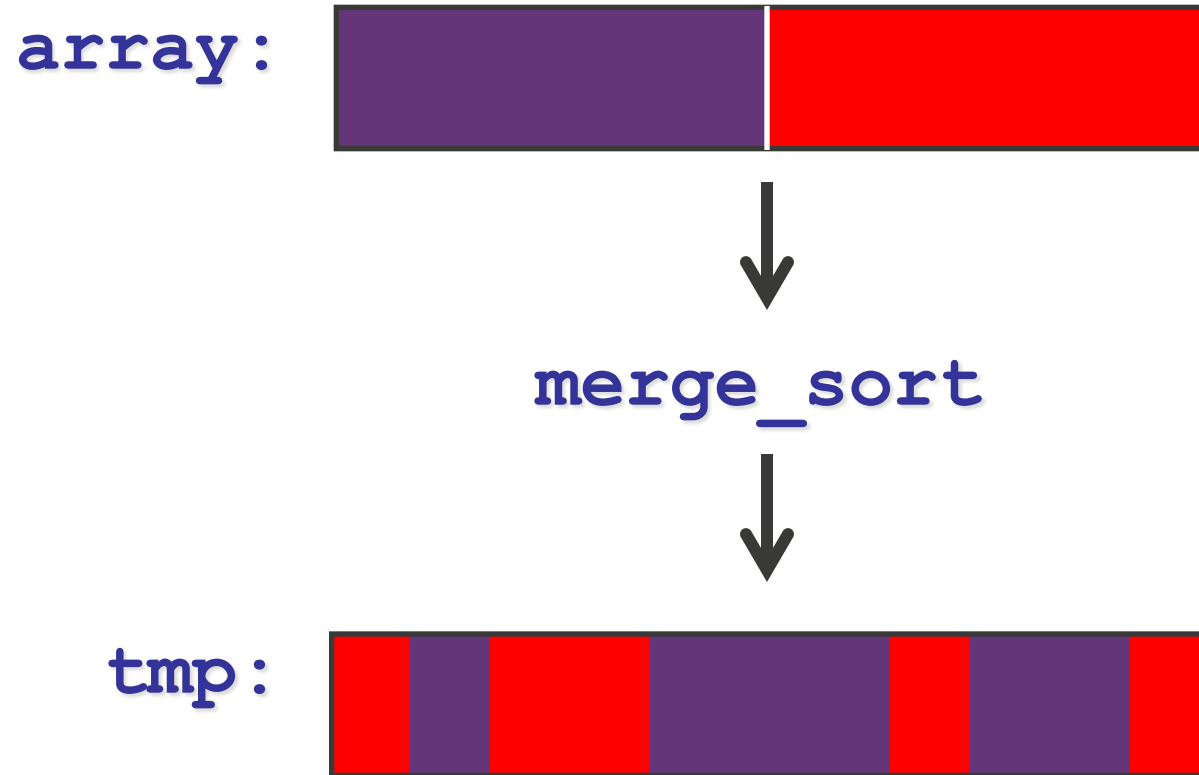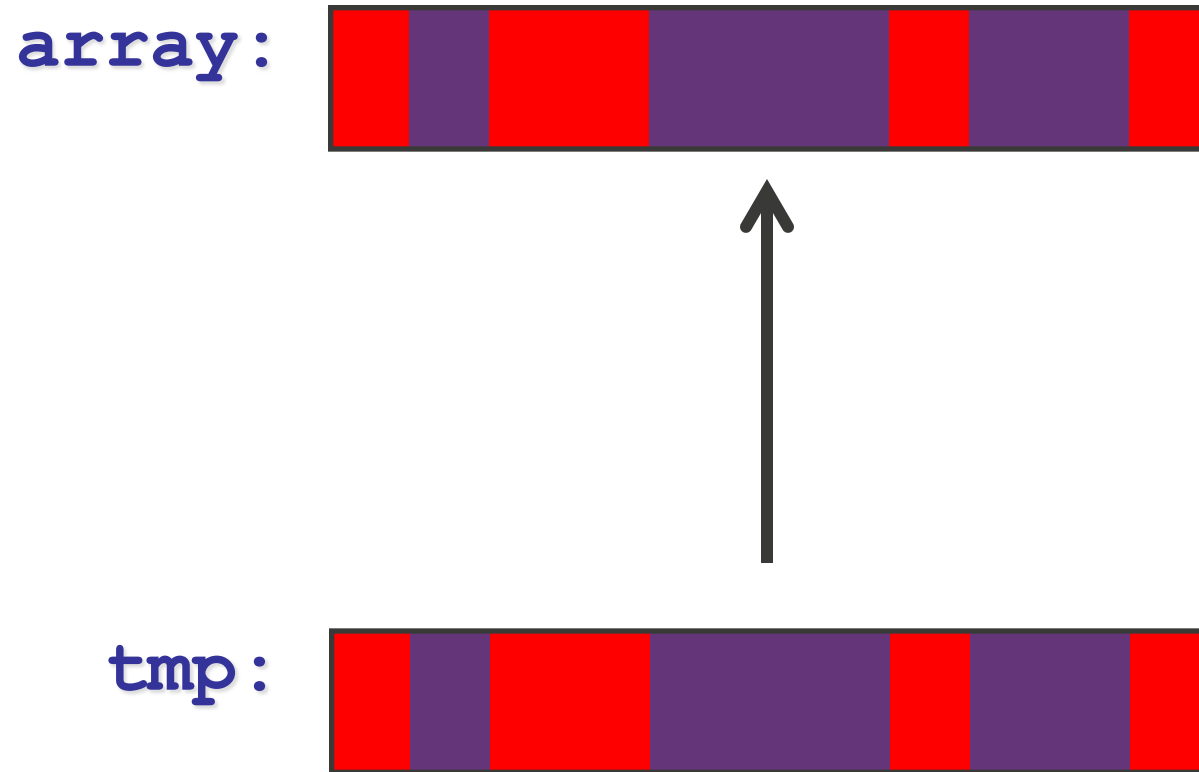
Merge Sort

array:

Merge Sort method

- **So we need:**
  - A temporary array to store the merged (partial) solution
  - Markers for the part of the array I am looking at (no need for new array)

```
def merge_sort(array: ArrayR) -> None:

    tmp = ArrayR(len(array))

    start = 0

    end = len(array)-1

    merge_sort_aux(array, start, end, tmp)
```

The temporary array I need

Frontiers to the part of the array I am currently looking at

Auxiliary method with all arguments I need

merge_sort_aux method

```python
def merge_sort_aux(array: ArrayR, start: int, end: int, tmp: T) -> None:
    if not start == end: # 2 or more still to sort
        mid = (start + end)//2
        # split into two halves
        merge_sort_aux(array, start, mid, tmp)
        merge_sort_aux(array, mid+1, end, tmp)
        # merge
        merge_arrays(array, start, mid, end, tmp)
        # copy tmp back into the original
        for i in range(start, end+1):
            array[i] = tmp[i]
```

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

b: | 10 | 14 | 22 | 43 | 50 |

start: 0   **mid: 5**   **end: 10**

tmp: | | | | | | | | | | | |

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

ia=0

b: | 10 | 14 | 22 | 43 | 50 |

ib=6

tmp: | 3 | | | | | | | | | | |

k=0

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

ia=1

b: | 10 | 14 | 22 | 43 | 50 |

ib=6

tmp: | 3 | 5 | | | | | | | | | |

k=1

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

ia=2

b: | 10 | 14 | 22 | 43 | 50 |

ib=6

tmp: | 3 | 5 | 10 | | | | | | | | |

k=2

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

ia=2

b: | 10 | 14 | 22 | 43 | 50 |

ib=7

tmp: | 3 | 5 | 10 | 14 | | | | | | | |

k=3

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

ia=2

b: | 10 | 14 | 22 | 43 | 50 |

ib=8

tmp: | 3 | 5 | 10 | 14 | 15 | | | | | | |

k=4

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

ia=3

b: | 10 | 14 | 22 | 43 | 50 |

ib=8

tmp: | 3 | 5 | 10 | 14 | 15 | 22 | | | | | |

k=5

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

ia=3

b: | 10 | 14 | 22 | 43 | 50 |

ib=9

tmp: | 3 | 5 | 10 | 14 | 15 | 22 | 28 | | | | |

k=6

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

ia=4

b: | 10 | 14 | 22 | 43 | 50 |

ib=9

tmp: | 3 | 5 | 10 | 14 | 15 | 22 | 28 | 30 | | | |

k=7

MONASH University

Example: merge_arrays

`a:` | 3 | 5 | 15 | 28 | 30 | **32** |

`ia=5`

`b:` | 10 | 14 | 22 | **43** | 50 |

`ib=9`

`tmp:` | 3 | 5 | 10 | 14 | 15 | 22 | 28 | 30 | 32 | | |

`k=8`

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

ia=6

b: | 10 | 14 | 22 | 43 | 50 |

ib=9

tmp: | 3 | 5 | 10 | 14 | 15 | 22 | 28 | 30 | 32 | 43 | |

k=9

Example: merge_arrays

a: | 3 | 5 | 15 | 28 | 30 | 32 |

ia=6

b: | 10 | 14 | 22 | 43 | 50 |

ib=10

tmp: | 3 | 5 | 10 | 14 | 15 | 22 | 28 | 30 | 32 | 43 | 50 |

k=10

MONASH University

Example: merge_arrays

a: `3` `5` `15` `28` `30` `32`   b: `10` `14` `22` `43` `50`

ia=6   ib=11

Done!

tmp: `3` `5` `10` `14` `15` `22` `28` `30` `32` `43` `50`

k=11

MONASH University

merge_arrays method

```python
def merge_arrays(a: ArrayR, start: int, mid: int, end: int, tmp: T) -> None:
    ia = start
    ib = mid+1
    for k in range(start, end+1):
        if ia > mid: # half1 finished, copy half2
            tmp[k] = a[ib]
            ib += 1
        elif ib > end: # half2 finished, copy half1
            tmp[k] = a[ia]
            ia += 1
        elif a[ia] <= a[ib]: # a[ia] is the item to copy
            tmp[k] = a[ia]
            ia += 1
        else:
            tmp[k] = a[ib] # a[ib] is the item to copy
            ib += 1
```

30

MONASH University
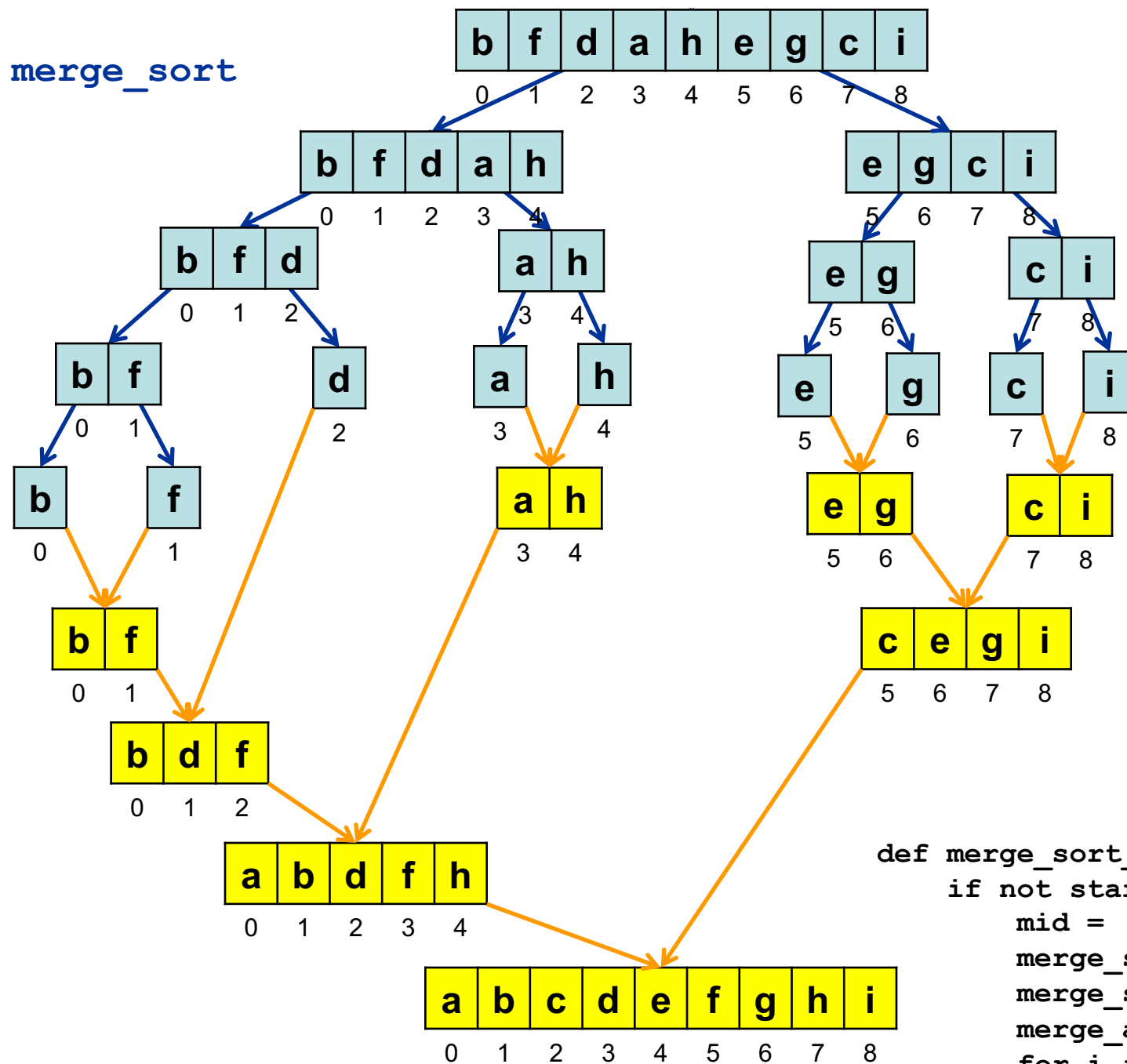
```
def merge_sort_aux(array, start, end, tmp):
    if not start == end: # 2+ still to sort
        mid = (start + end)//2
        merge_sort_aux(array, start, mid, tmp)
        merge_sort_aux(array, mid+1, end, tmp)
        merge_arrays(array,start,mid,end,tmp)
        for i in range(start,end+1):
            array[i] = tmp[i]
```

merge_sort

b	f	d	a	h	e	g	c	i
0	1	2	3	4	5	6	7	8

b	f	d	a	h
0	1	2	3	4

e	g	c	i
5	6	7	8

b	f	d
0	1	2

a	h
3	4

e	g
5	6

c	i
7	8

b	f
0	1

d
2

a
3

h
4

e
5

g
6

c
7

i
8

b
0

f
1

a	h
3	4

e	g
5	6

c	i
7	8

b	f
0	1

c	e	g	i
5	6	7	8

b	d	f
0	1	2

a	b	d	f	h
0	1	2	3	4

a	b	c	d	e	f	g	h	i
0	1	2	3	4	5	6	7	8
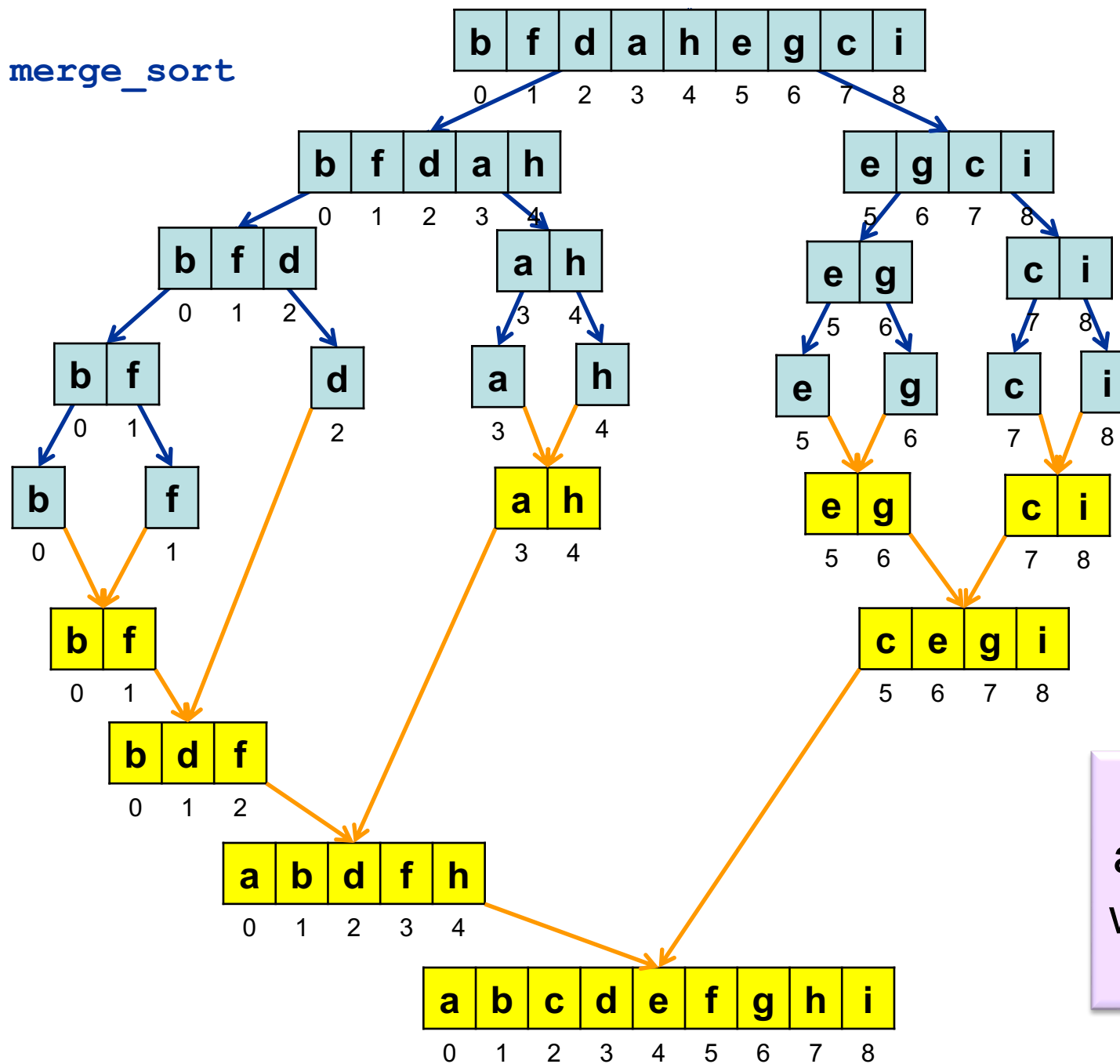
Depth is log(N), and at each level we do N constant operations

32

Merge Sort Analysis

- **This is typically the method that you would naturally use when sorting a pile of books, CDs cards, etc.**

- **Most of the work is in the merging**

- **Uses more space than other sorts**

- **Takes O(n log(n))*CompEq, where n is the number of elements in the array and CompEq is the complexity of the comparison**

- **Close to optimal in number of comparisons**
  - Good for languages where comparison is expensive

Quick Sort

Quick Sort

- **The split is non-trivial: partitions the array into two:**
  - One with all elements smaller than the pivot p
  - Another with all elements greater than the pivot
- **Combination is trivial: append with pivot in the middle**
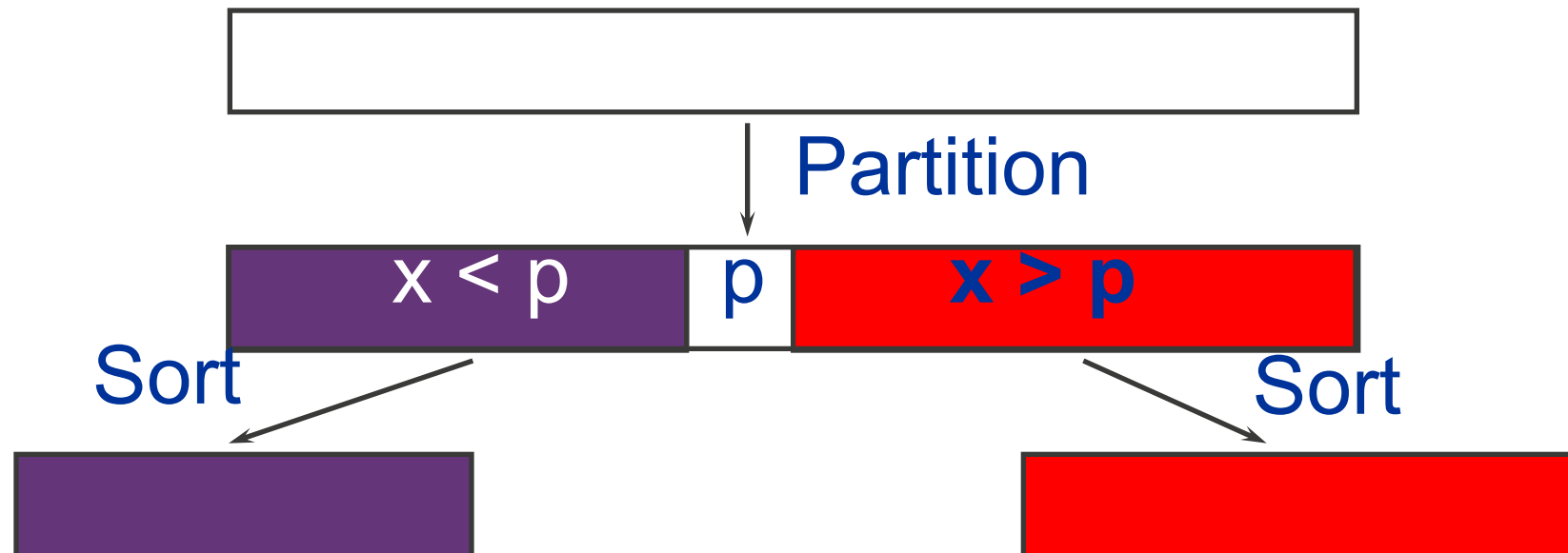
Quick Sort: choosing the pivot

- **It must be an element in the array**
- **Ideally, its value is the median of all values**
  - This is does not mean is in the middle position! (unless the list is already sorted)

- **Why: best when the subarrays are about the same size**
- **Hitch: how to pick the median element? (or similar)**
- **One strategy: pick a small sample of elements (e.g. first, last, mid) and choose the median of these**
- **Or you could just pick a random element**
- **First element is usually a bad idea: presorted input**

MONASH University

Example: Quick Sort

array: | 5 | 89 | 35 | 14 | 24 | 15 | 37 | 13 | 20 | 7 | 70 |

start:0   **end:10**

Example: Quick Sort

array:

| 5 | 89 | 35 | 14 | 24 | **15** | 37 | 13 | 20 | 7 | 70 |

Say we randomly choose 15 as the pivot.

It happens to be in the middle of the list, but that's just a coincidence!

Example: Quick Sort

array: 5 | 89 | 35 | 14 | 24 | **15** | 37 | 13 | 20 | 7 | 70

partition:

| 5 | 89 | 35 | 14 | 24 | 15 | 37 | 13 | 20 | 7 | 70 |

↓

| 7 | 14 | 5 | 13 | 15 | 35 | 37 | 89 | 20 | 24 | 70 |
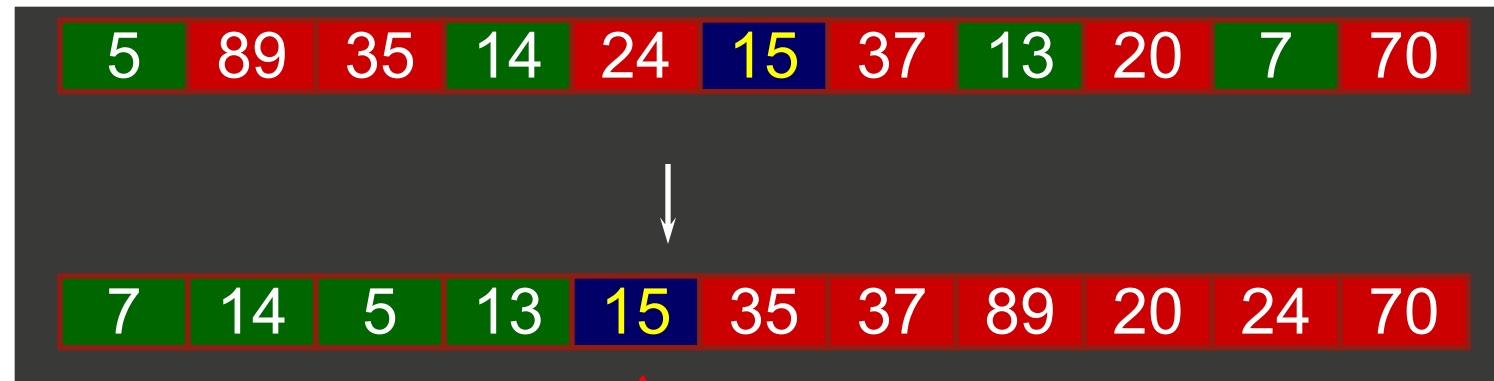
boundary: 4

Quick Sort method

```python
def quick_sort(array: ArrayR) -> None:
    start = 0
    end = len(array)-1
    quick_sort_aux(array, start, end)

def quick_sort_aux(array: ArrayR, start: int, end: int) -> None:
    if start < end:
        boundary = partition(array, start, end)
        quick_sort_aux(array, start, boundary-1)
        quick_sort_aux(array, boundary+1, end)
```
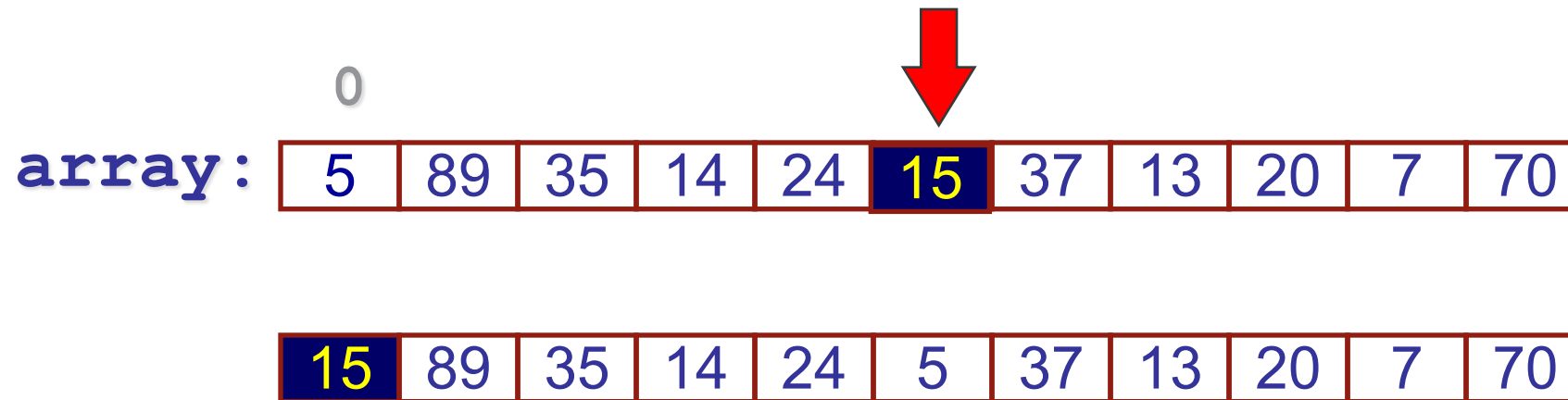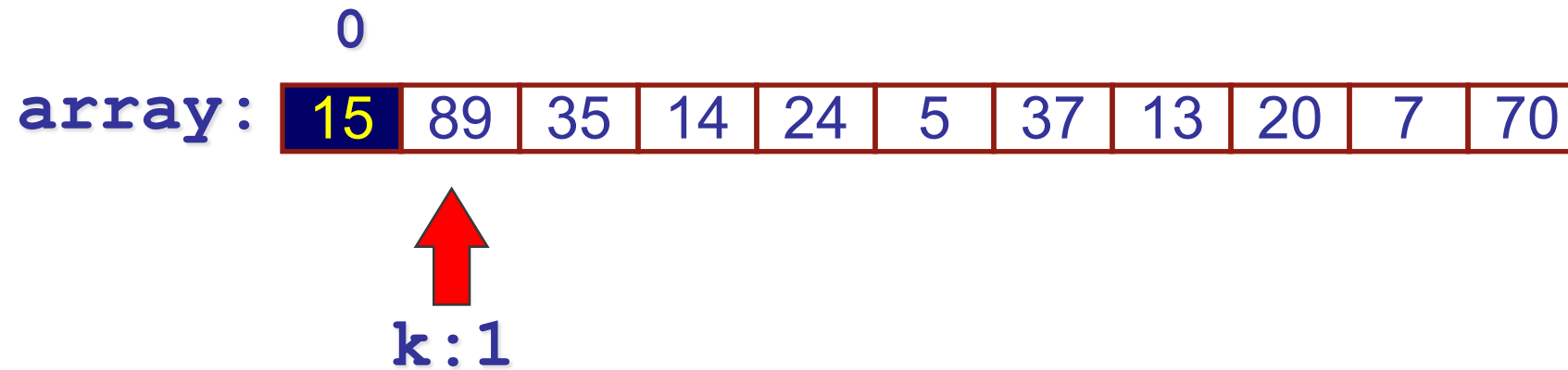
But no need for a temporary array this time

Again, the frontiers to the part of the array I am currently looking at

MONASH University

Partition: Checklist
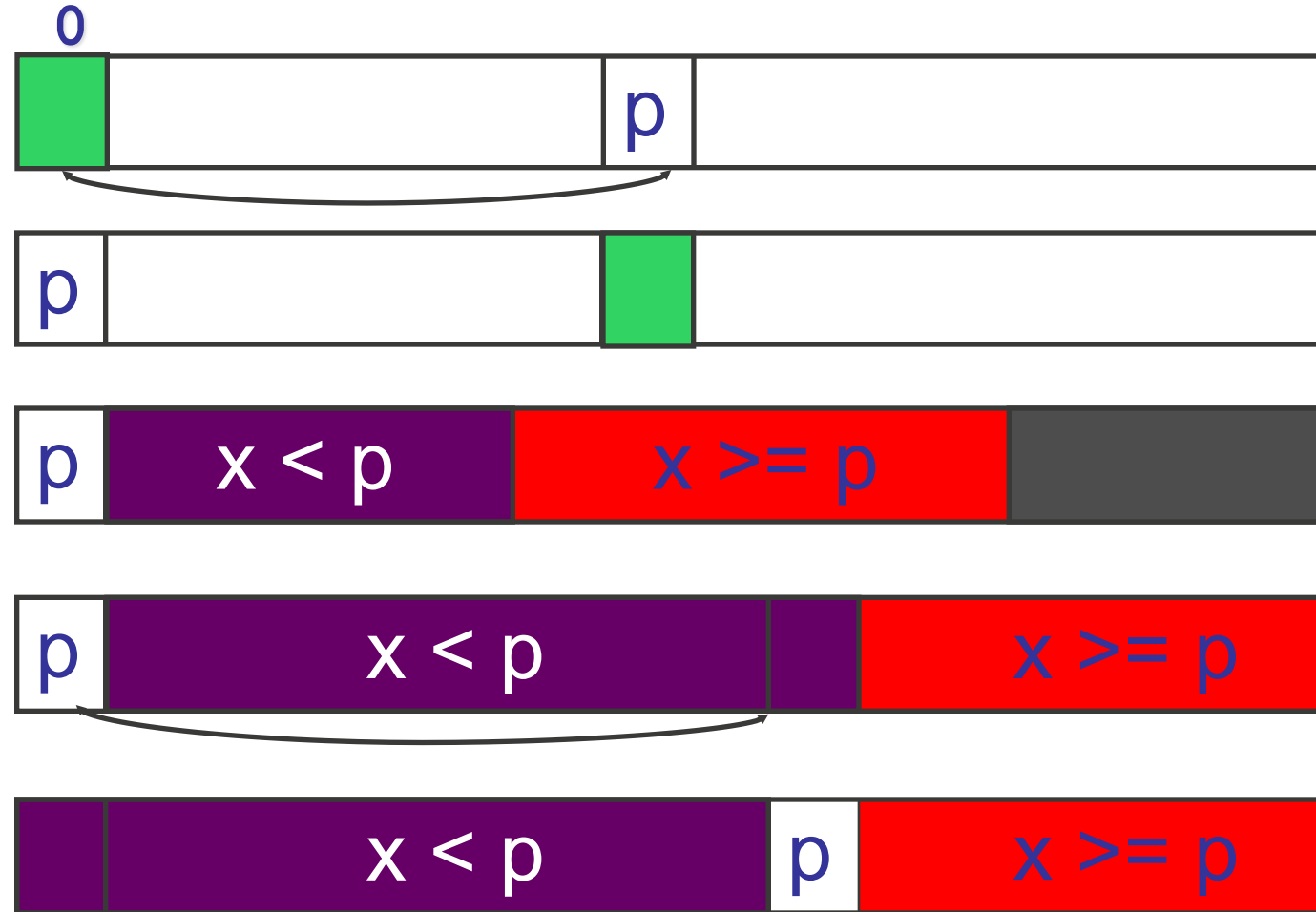
Example: Partition

randomly pick element in position 5

0

array: | 5 | 89 | 35 | 14 | 24 | **15** | 37 | 13 | 20 | 7 | 70 |

| **15** | 89 | 35 | 14 | 24 | 5 | 37 | 13 | 20 | 7 | 70 |

Example: Partition

0

array: | 15 | 89 | 35 | 14 | 24 | 5 | 37 | 13 | 20 | 7 | 70 |

k:1

Example: Partition

boundary:0

array: | 15 | 89 | 35 | 14 | 24 | 5 | 37 | 13 | 20 | 7 | 70 |

k:1

Example: Partition

boundary:0

array: | 15 | 89 | 35 | 14 | 24 | 5 | 37 | 13 | 20 | 7 | 70 |

k:2

Example: Partition

boundary:0

array: | 15 | 89 | 35 | 14 | 24 | 5 | 37 | 13 | 20 | 7 | 70 |

k:3

Example: Partition

boundary:1

array: | 15 | 14 | 35 | 89 | 24 | 5 | 37 | 13 | 20 | 7 | 70 |

k:3

Example: Partition

boundary:1

array: | 15 | 14 | 35 | 89 | 24 | 5 | 37 | 13 | 20 | 7 | 70 |

k:4

Example: Partition

boundary:1

array: | 15 | 14 | 35 | 89 | 24 | 5 | 37 | 13 | 20 | 7 | 70 |

k:5

Example: Partition

boundary:2

array: | 15 | 14 | 5 | 89 | 24 | 35 | 37 | 13 | 20 | 7 | 70 |

k:5

Example: Partition

boundary:2

array: | 15 | 14 | 5 | 89 | 24 | 35 | 37 | 13 | 20 | 7 | 70 |

k:6

Example: Partition

boundary:2

array: | 15 | 14 | 5 | 89 | 24 | 35 | 37 | 13 | 20 | 7 | 70 |

k:7 *etc...*

Example: Partition

boundary:4

array: | 15 | 14 | 5 | 13 | 7 | 35 | 37 | 89 | 20 | 24 | 70 |

k:11

MONASH University

Example: Partition

Example: Partition

pivot now in correct position

array: | 7 | 14 | 5 | 13 | **15** | 35 | 37 | 89 | 20 | 24 | 70 |

x < 15   x >= 15

Example: Partition

| 7 | 14 | 5 | 13 | 15 | 35 | 37 | 89 | 20 | 24 | 70 |

quick_sort   quick_sort

| 7 | 5 | 13 | 14 |

| 20 | 24 | 35 | 37 | 70 | 89 |
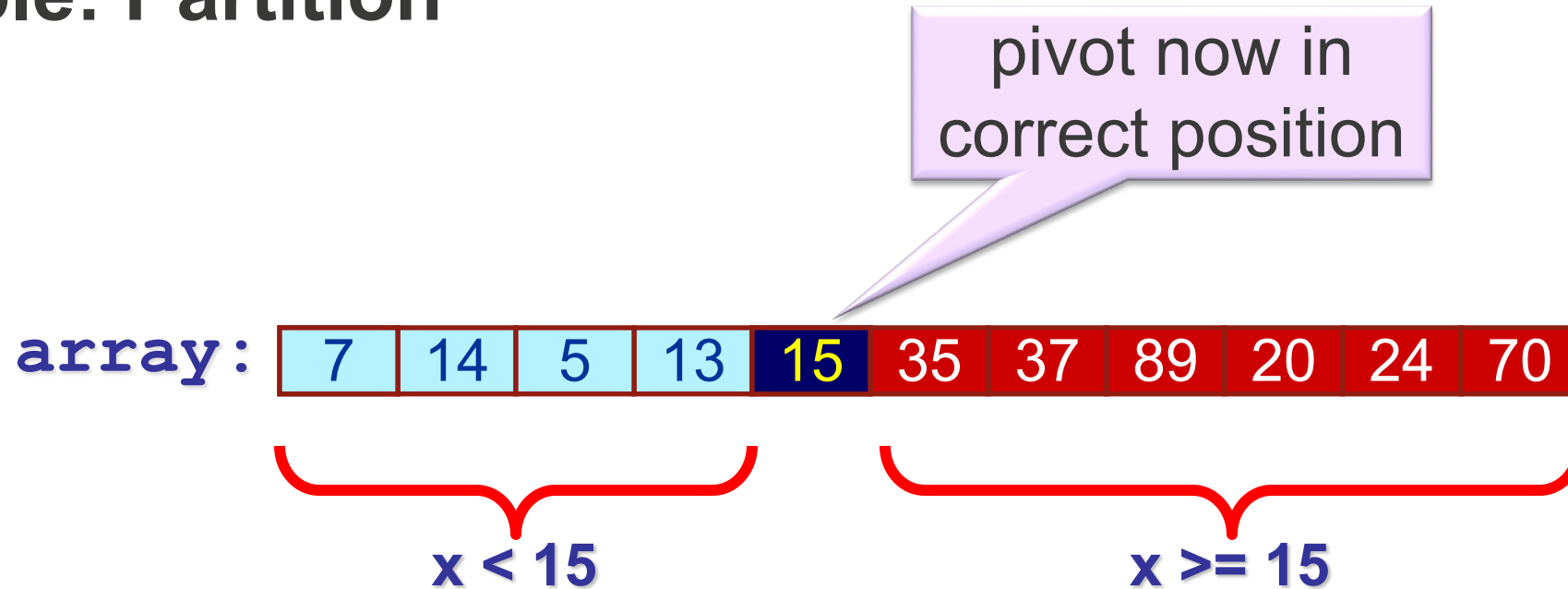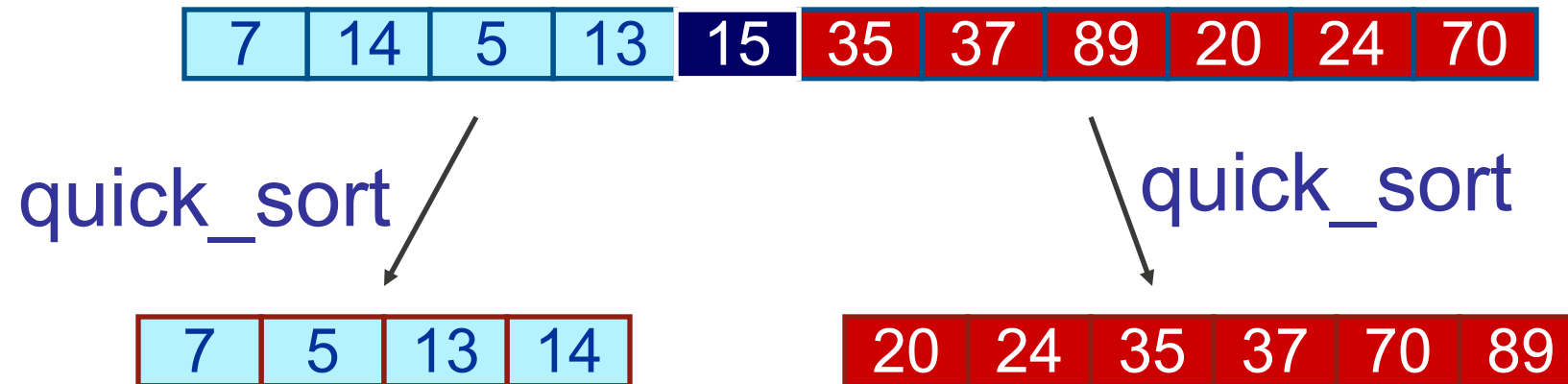
Partition Method

```
def partition(array: ArrayR, start: int, end: int) -> int:
    mid = (start+end)//2
    pivot = array[mid]
    swap(array, start, mid)

    boundary = start

    for k in range(start+1, end+1):
        if array[k] < pivot:
            boundary += 1
            swap(array, k, boundary)

    swap(array, start, boundary)
    return boundary
```

For simplicity, let me choose mid element here…

quick_sort with pivot being the first element

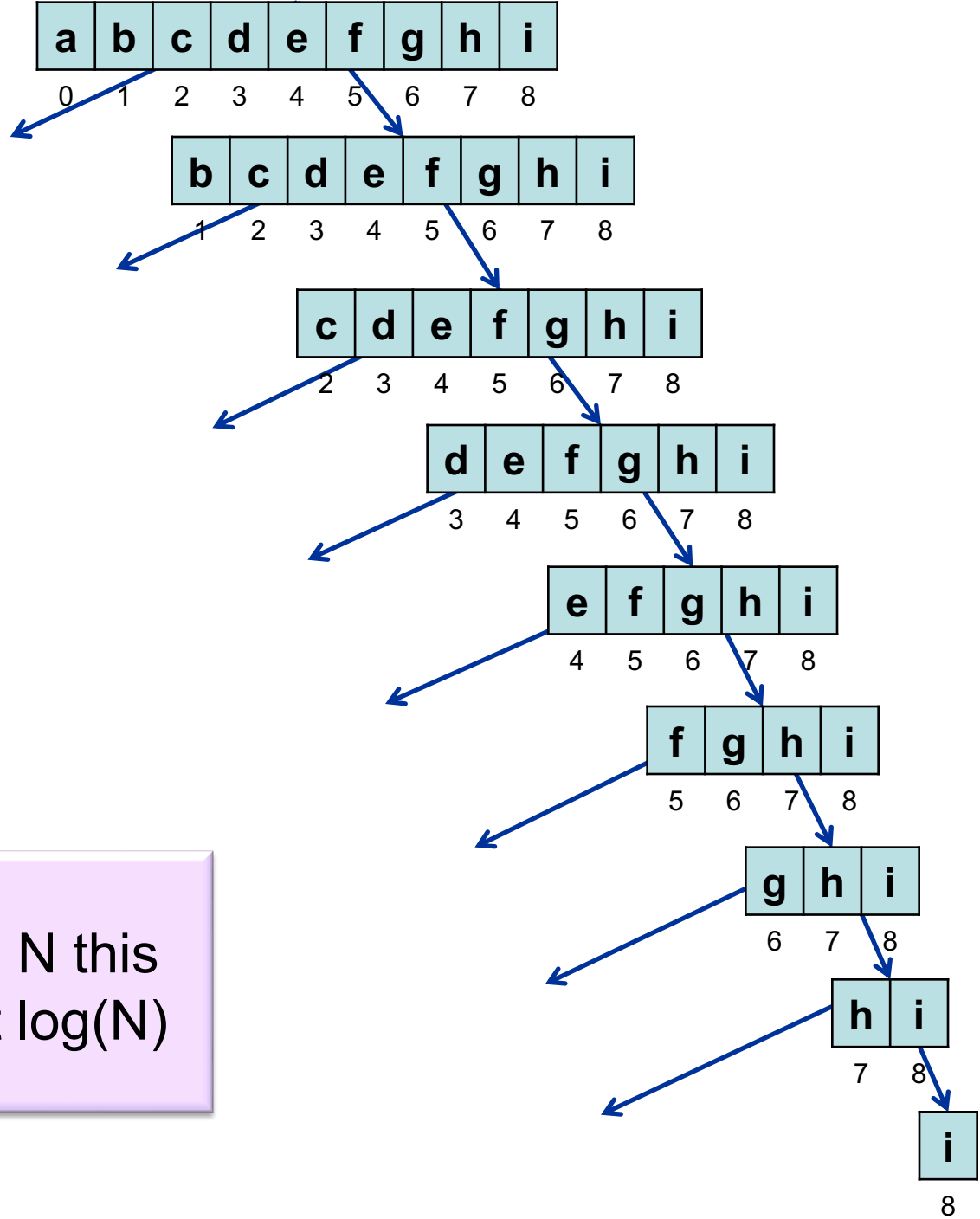| a	b	c	d	e	f	g	h	i
0	1	2	3	4	5	6	7	8

b	c	d	e	f	g	h	i
1	2	3	4	5	6	7	8

c	d	e	f	g	h	i
2	3	4	5	6	7	8

d	e	f	g	h	i
3	4	5	6	7	8

e	f	g	h	i
4	5	6	7	8

f	g	h	i
5	6	7	8

g	h	i
6	7	8

h	i
7	8

i
8

Depth is N this time, not log(N)

58

Quick Sort Analysis

- **Most of the work done in partitioning**

- **Remember: careful about choice of pivot**
  - Example: 2, 4, 6, 7, 3, 1, 5 what happens if we use the mid element as pivot?
  - Often used: median among 3 random

- **Complexity?**
  - Best case takes **O(n log(n))*CompEq** time
    - Always pick median as pivot
  - Worst case takes **O(n^2)*CompEq** time
    - Always pick min/max as pivot

- **Main advantage over merge ksort:**
  - no need to copy back (so quicksort has a smaller constant if the pivot is good)

- **Still, for small arrays (≤20) it has significant overhead (due to the recursion); for those better use insertion sort**

Summary

- **Divide and Conquer algorithms**
  - Relationship with recursive algorithms
- **Merge Sort**
  - Easy split
  - Complex part: merge method
- **Quick Sort**
  - Complex split: partition method
  - Easy combination