COMP90038 Algorithms and Complexity

Lecture 11: Sorting with Divide-and-Conquer

(with thanks to Harald Søndergaard)

Toby Murray

toby.murray@unimelb.edu.au

DMD 8.17 (Level 8, Doug McDonell Bldg)

http://people.eng.unimelb.edu.au/tobym

@tobycmurray

Divide and Conquer

THE UNIVERSITY OF MELBOURNE

- We earlier studied recursion as a powerful problem solving technique.

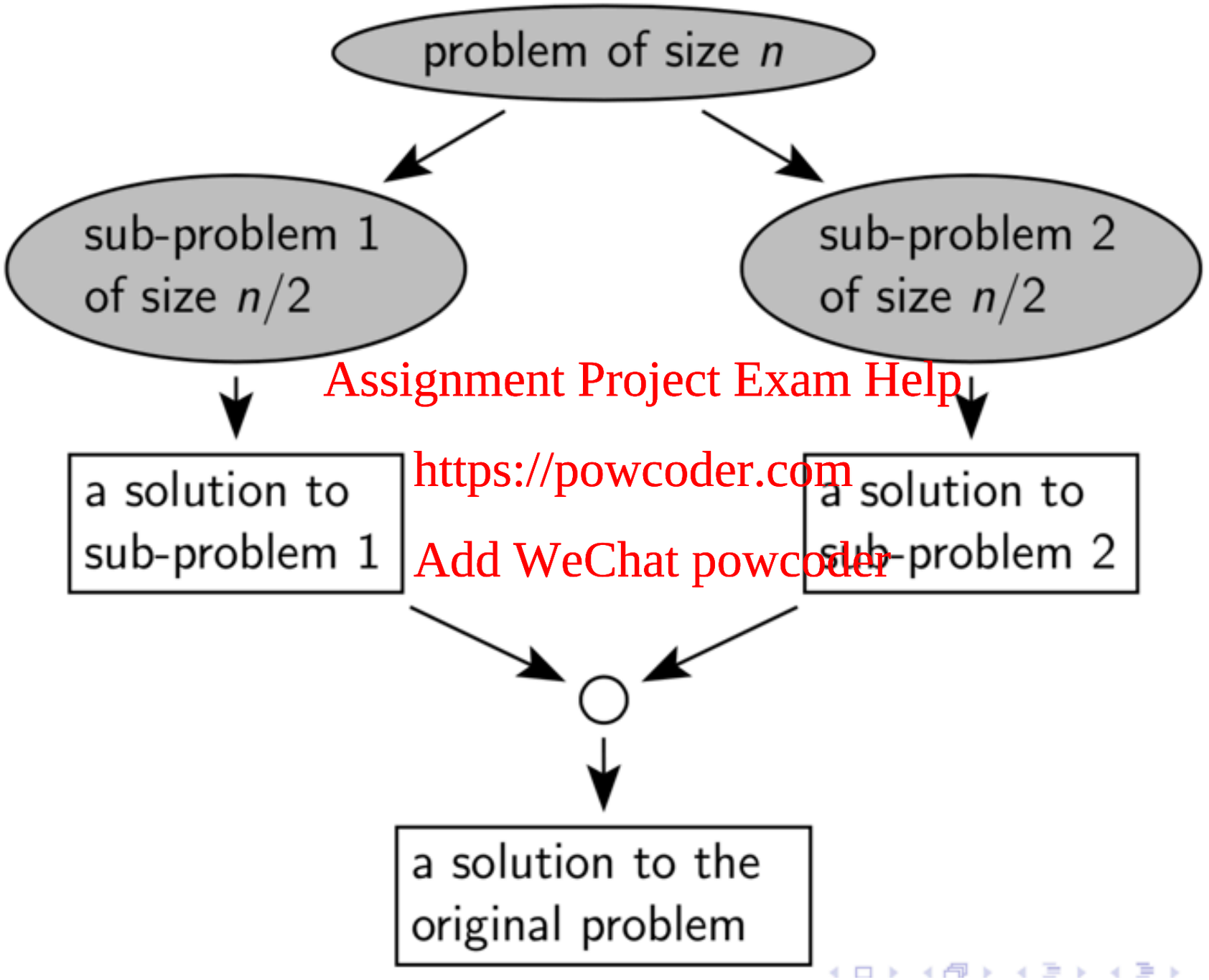
- The **divide-and-conquer** strategy tries to make the most of this idea:

1. Divide the given problem instance into smaller instances.

2. Solve the smaller instances recursively.
3. Combine the smaller solutions to solve the original instance.

- This works best when the smaller instances can be made to be of equal (or near-equal) size.

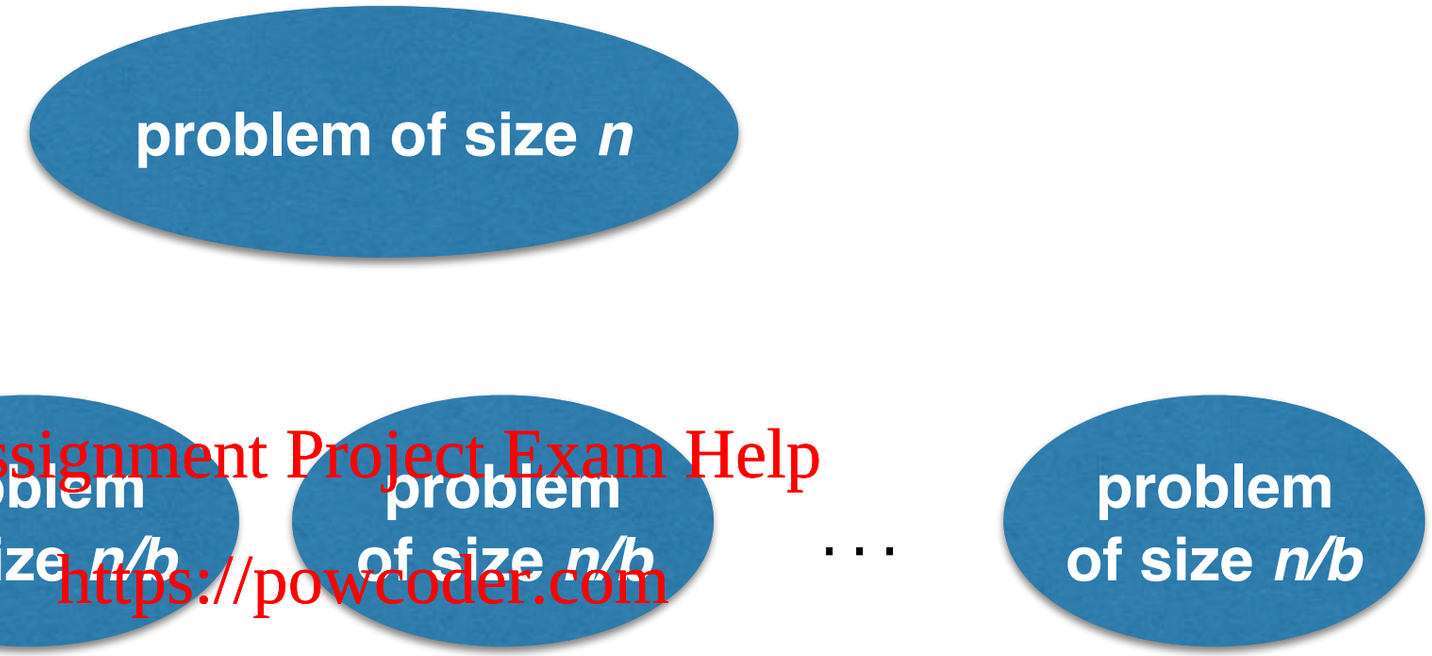
Split-Solve-and-Join Approach



Divide and Conquer Algorithms

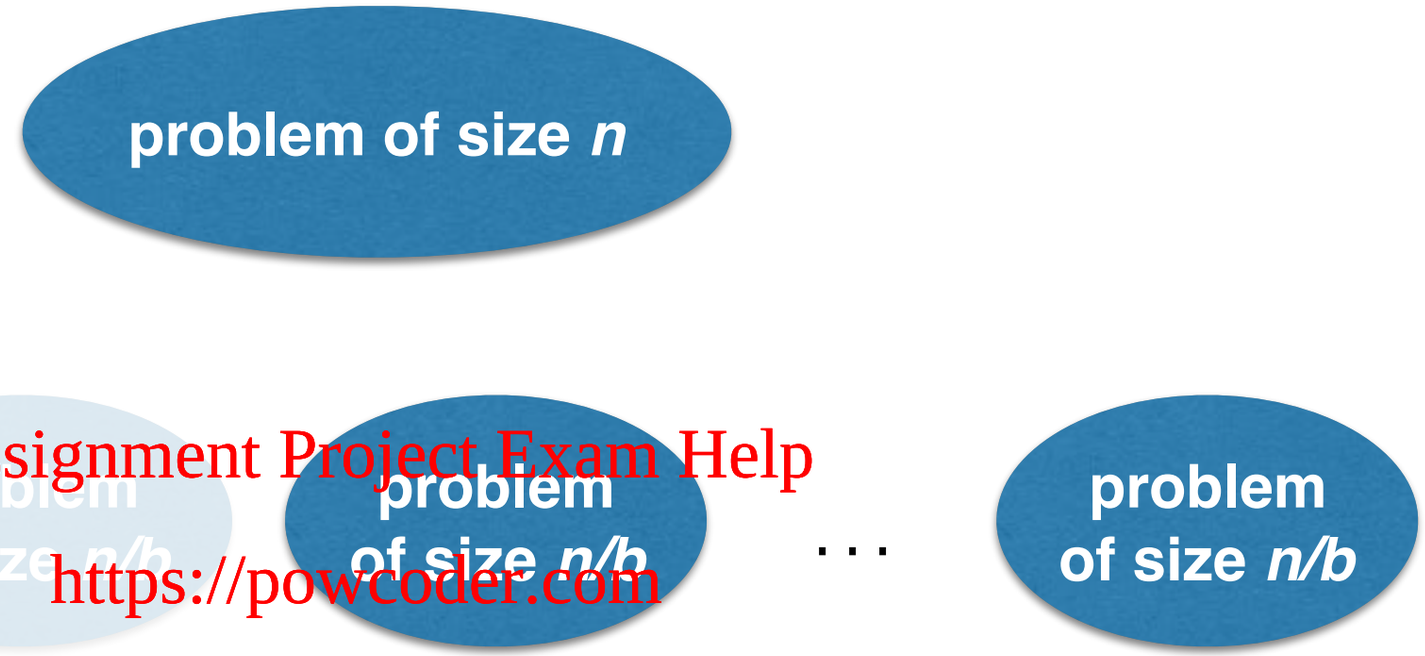
- We will discuss:
  - The Master Theorem
  - Mergesort
  - Quicksort
  - Tree traversal
  - Closest Pair revisited

Divide-and-Conqer General Case



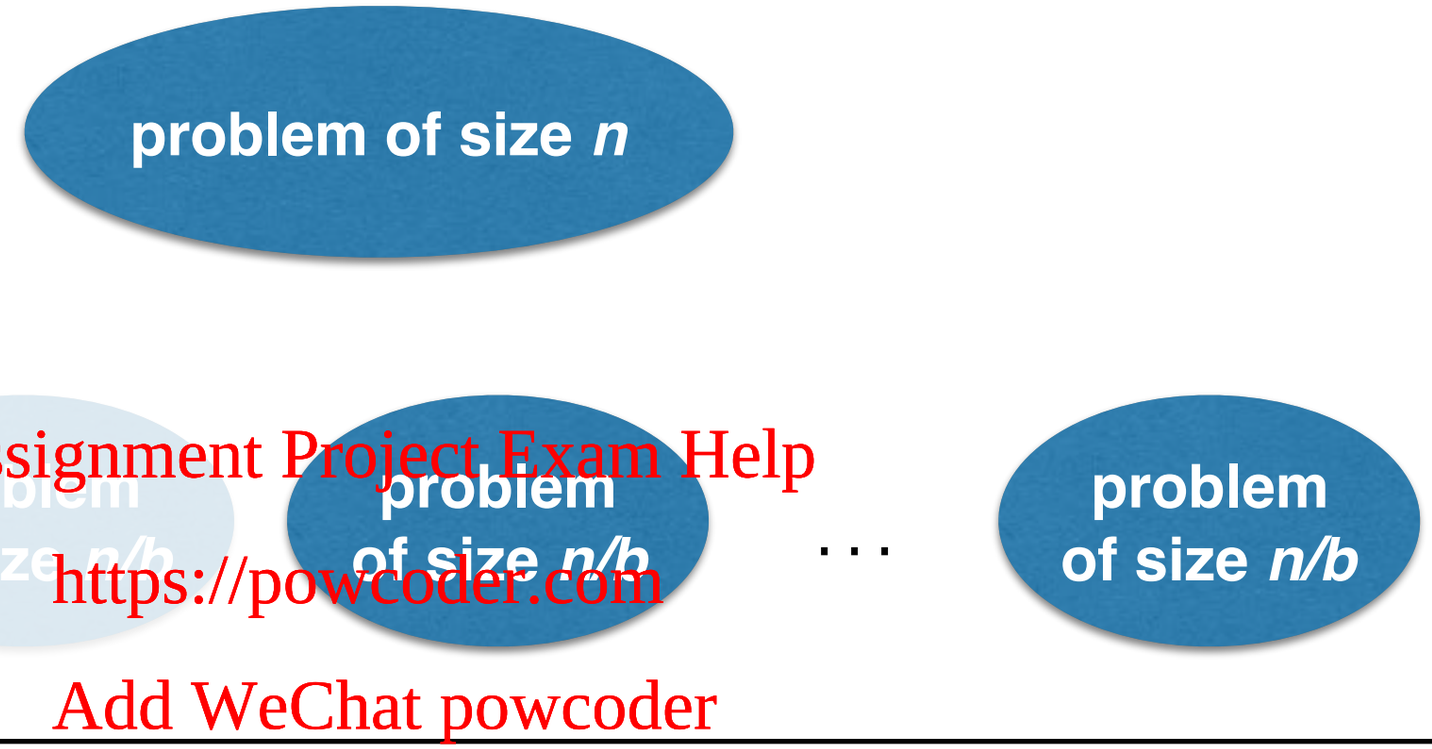
b sub-problems

Divide-and-Conqer General Case



only *a* sub-problems need to be solved

Divide-and-Conqer General Case



only *a* sub-problems need to be solved

Divide-and-Conquer Recurrences

THE UNIVERSITY OF MELBOURNE

- What is the time required to solve a problem of size *n* by divide-and-conquer?

- For the general case, assume we split the problem into *b* instances (each of size *n/b*), of which *a* need to be solved:

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

where $f(n)$ expresses the time spent on dividing a problem into *b* sub-problems and combining the *a* results.

- (A very common case is $T(n) = 2T(n/2) + n$.)

- How to find closed forms for these recurrences?

The Master Theorem

- (A proof is in Levitin's Appendix B.)

- For integer constants $a \geq 1$ and $b > 1$, and function f with $f(n) \in \Theta(n^d)$, $d \geq 0$, the recurrence

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

(with T(1) = c) has solutions, and

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- Note that we also allow a to be greater than b.

Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$ a = 2, b = 2, d = 1

Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$

a = 2, b = 2, d = 1

$a = b^d$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

So, by the Master Theorem, $T(n) \in \Theta(n \log n)$

Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$ $\quad$ a = 2, b = 2, d = 1

Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$ a = 2, b = 2, d = 1

$T(n) = 2(2T(n/4) + (n/2)) + n$

$1 \times n$

Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$ a = 2, b = 2, d = 1

$T(n) = 4T(n/4) + 2(n/2) + n$

Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$ a = 2, b = 2, d = 1

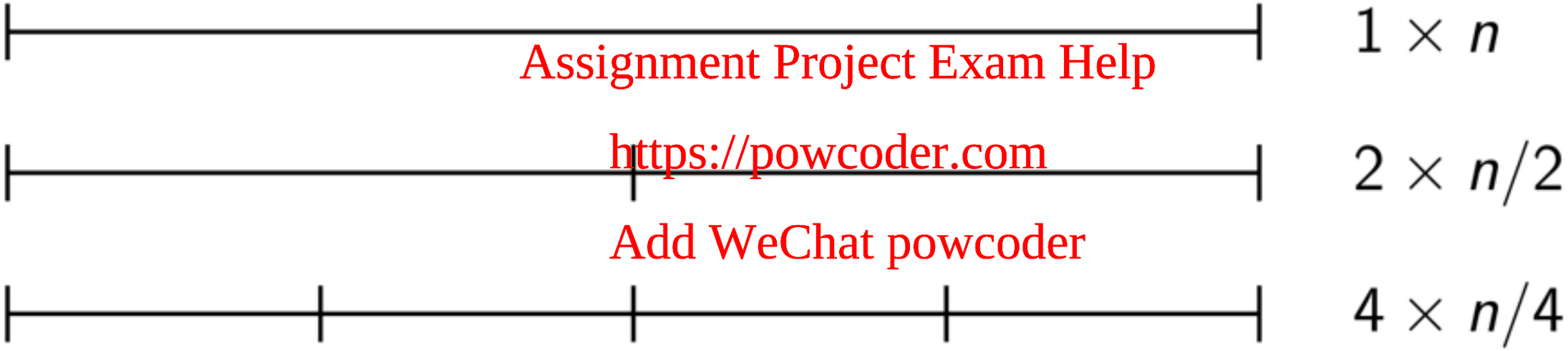
$T(n) = 4(2T(n/8) + n/4) + 2(n/2) + n$

Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$ a = 2, b = 2, d = 1

$T(n) = 8T(n/8) + 4(n/4) + 2(n/2) + n$

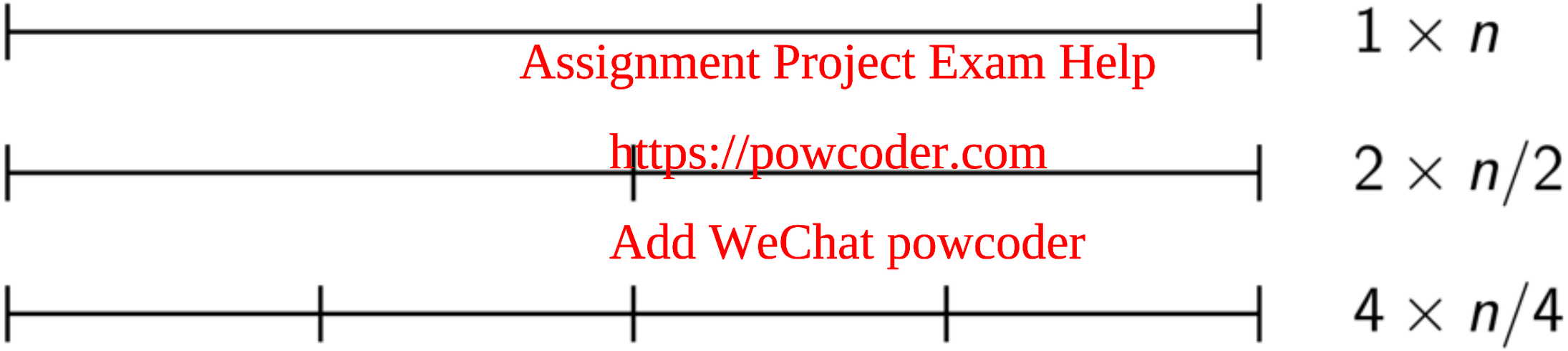


Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$ $\quad$ a = 2, b = 2, d = 1

$T(n) = 8(2T(n/16) + n/8) + 4(n/4) + 2(n/2) + n$

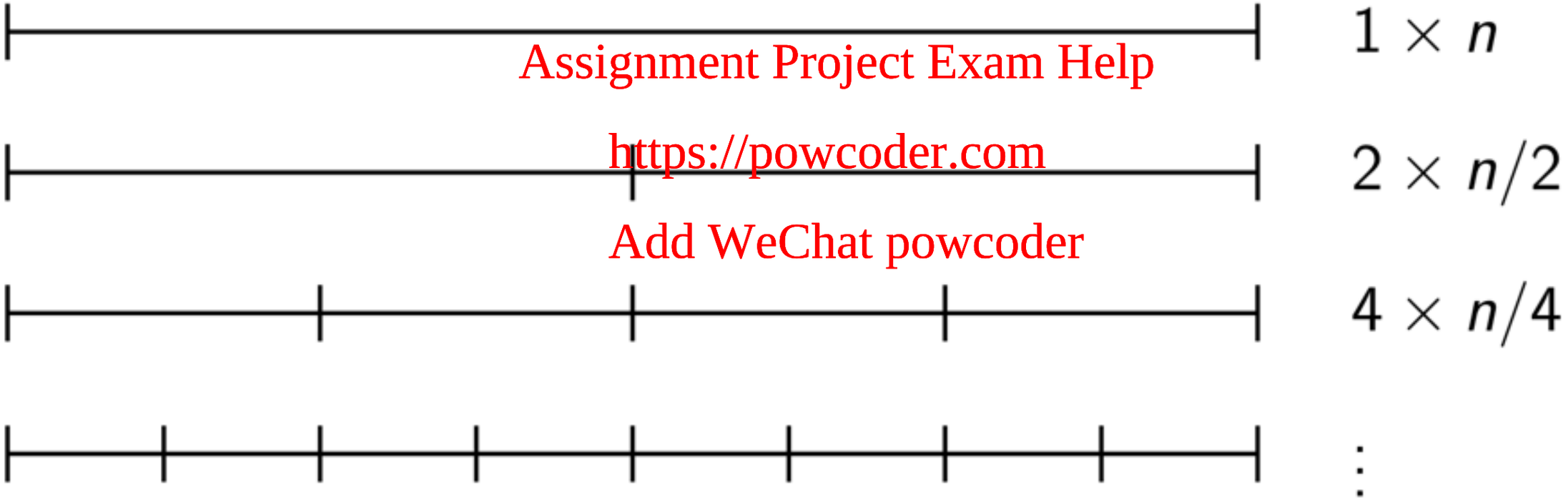


Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$ $\quad$ a = 2, b = 2, d = 1

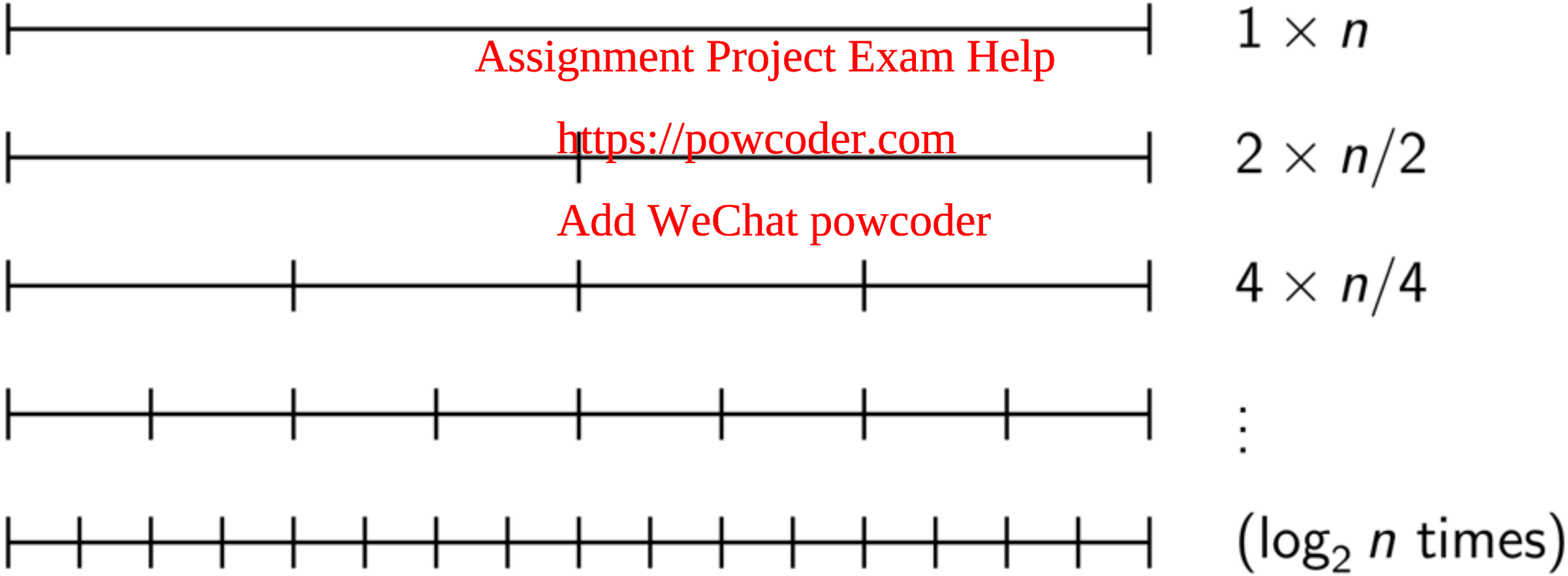
$T(n) = 16T(n/16) + 8(n/8) + 4(n/4) + 2(n/2) + n$



Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$ $\quad$ a = 2, b = 2, d = 1

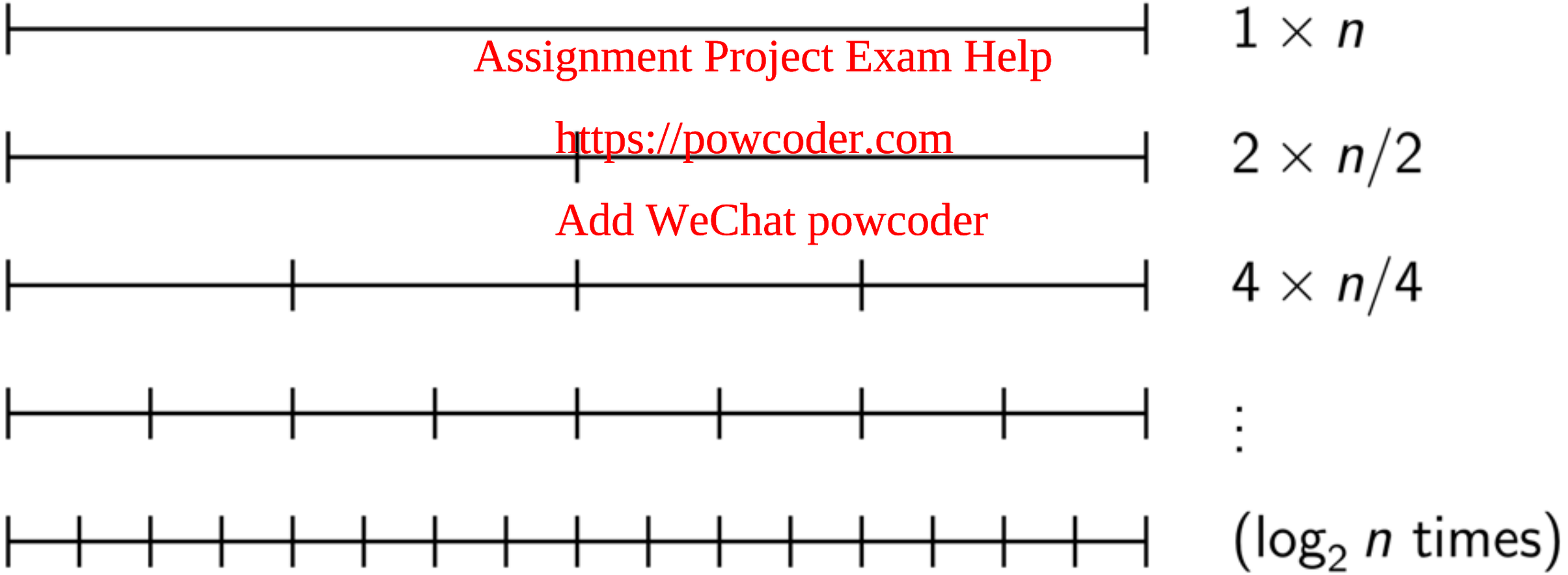


Master Theorem: Example 1

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n$ $\quad$ a = 2, b = 2, d = 1

$T(n) \in \Theta(n \log n)$



Master Theorem: Example 2

THE UNIVERSITY OF MELBOURNE

$T(n) = 4T(n/4) + n$

a = 4, b = 4, d = 1

$a = b^d$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

So, by the Master Theorem, $T(n) \in \Theta(n \log n)$

Master Theorem: Example 2

THE UNIVERSITY OF MELBOURNE

$T(n) = 4T(n/4) + n$ a = 4, b = 4, d = 1

Master Theorem: Example 2

THE UNIVERSITY OF MELBOURNE

$T(n) = 4T(n/4) + n$ a = 4, b = 4, d = 1

$T(n) = 4(4T(n/16) + (n/4)) + n$

Master Theorem: Example 2

THE UNIVERSITY OF MELBOURNE

$T(n) = 4T(n/4) + n$ a = 4, b = 4, d = 1

$T(n) = 16T(n/16) + 4(n/4) + n$

Master Theorem: Example 2

THE UNIVERSITY OF MELBOURNE

$T(n) = 4T(n/4) + n$ a = 4, b = 4, d = 1

$T(n) = 16T(n/16) + 4(n/4) + n$

Master Theorem: Example 2

THE UNIVERSITY OF MELBOURNE

$T(n) = 4T(n/4) + n$ a = 4, b = 4, d = 1

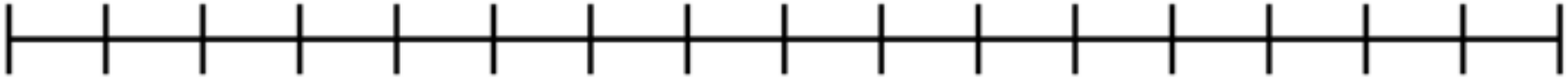
$T(n) = 16(4T(n/64) + n/16)\ 4(n/4) + n$

Master Theorem: Example 2

THE UNIVERSITY OF MELBOURNE

$T(n) = 4T(n/4) + n$ a = 4, b = 4, d = 1

$T(n) = 64T(n/64) + 16(n/16) + 4(n/4) + n$

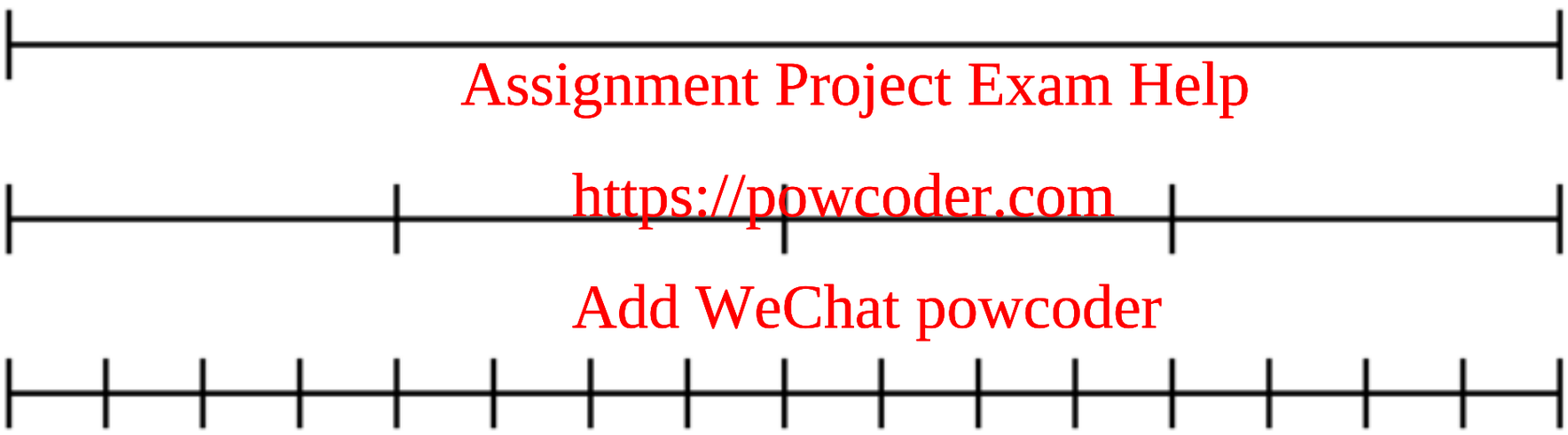


Master Theorem: Example 2

THE UNIVERSITY OF MELBOURNE

$T(n) = 4T(n/4) + n$ a = 4, b = 4, d = 1

$T(n) = 64T(n/64) + 16(n/16) + 4(n/4) + n$



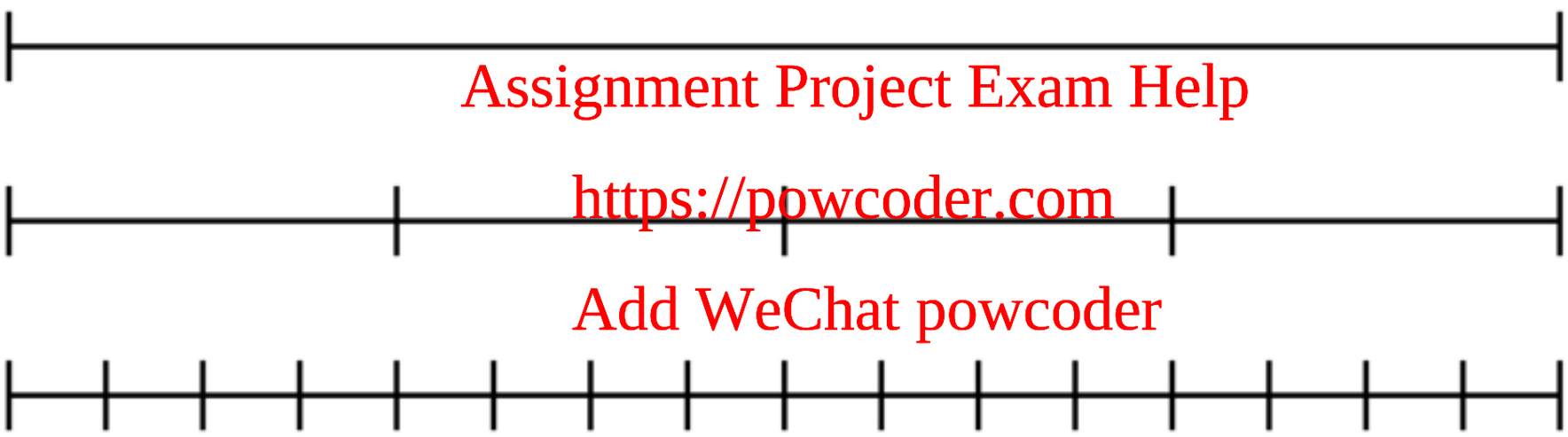
($\log_4$ n times)

Master Theorem: Example 2

THE UNIVERSITY OF MELBOURNE

$T(n) = 4T(n/4) + n$ a = 4, b = 4, d = 1

$T(n) = 64T(n/64) + 16(n/16) + 4(n/4) + n$



($\log_4$ n times)

Master Theorem: Example 2

THE UNIVERSITY OF MELBOURNE

$T(n) = 4T(n/4) + n$ a = 4, b = 4, d = 1

$T(n) = 64T(n/64) + 16(n/16) + 4(n/4) + n$

$T(n) \in \Theta(n \log n)$

Master Theorem: Example 3

THE UNIVERSITY OF MELBOURNE

$T(n) = T(n/2) + n$ a = 1, b = 2, d = 1

Master Theorem: Example 3

THE UNIVERSITY OF MELBOURNE

$T(n) = T(n/2) + n$

a = 1, b = 2, d = 1

$a < b^d$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

So, by the Master Theorem, $T(n) \in \Theta(n)$

Master Theorem: Example 3

THE UNIVERSITY OF MELBOURNE

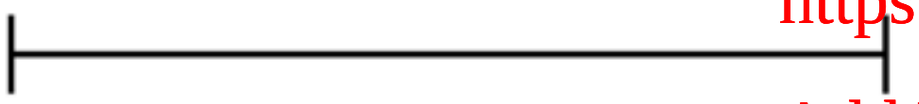
$T(n) = T(n/2) + n$ a = 1, b = 2, d = 1

Master Theorem: Example 3

THE UNIVERSITY OF MELBOURNE

$T(n) = T(n/2) + n$ a = 1, b = 2, d = 1

$T(n) = T(n/4) + n/2 + n$



Master Theorem: Example 3

THE UNIVERSITY OF MELBOURNE

$T(n) = T(n/2) + n$ a = 1, b = 2, d = 1

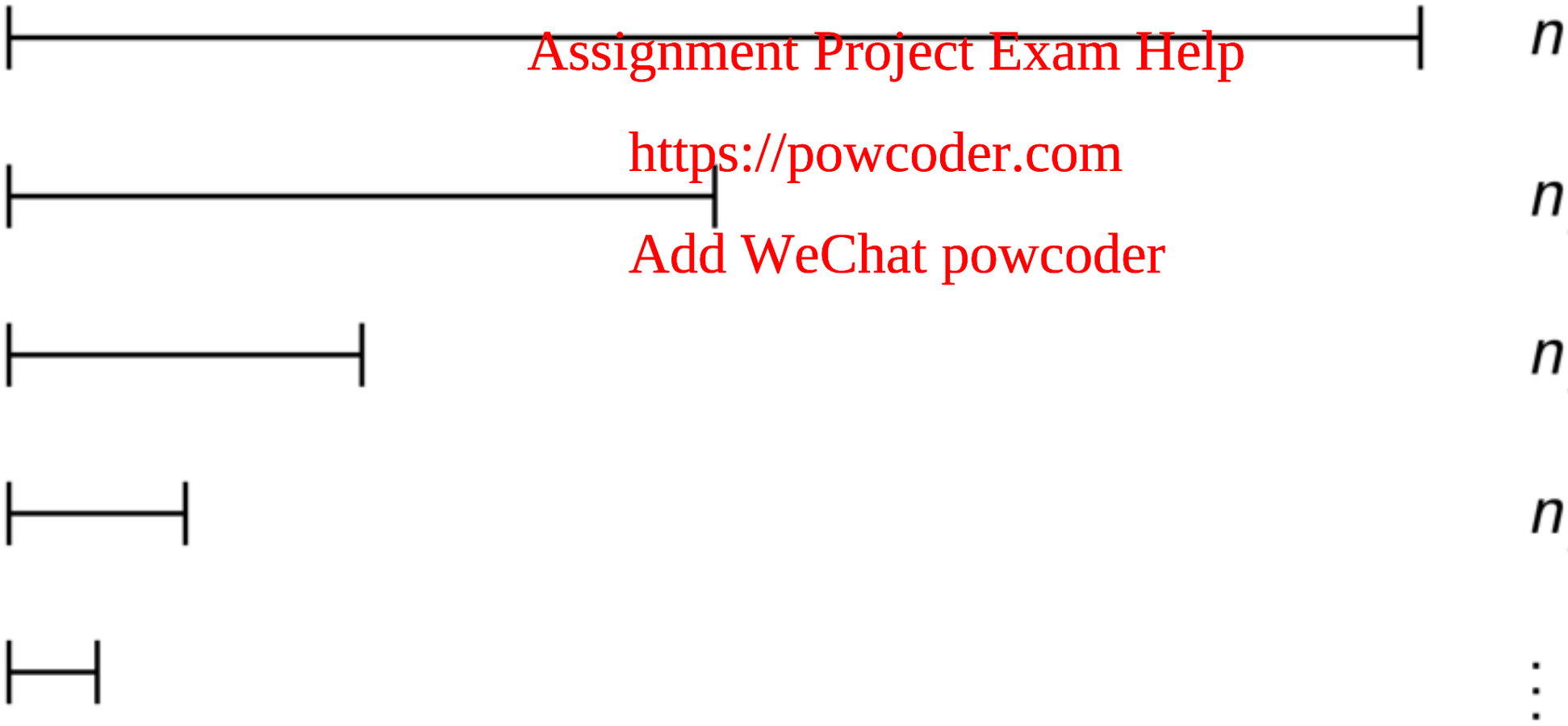
$T(n) = T(n/8) + n/4 + n/2 + n$

Master Theorem: Example 3

THE UNIVERSITY OF MELBOURNE

$T(n) = T(n/2) + n$ a = 1, b = 2, d = 1

$T(n) = T(n/8) + n/4 + n/2 + n$



Master Theorem: Example 3

THE UNIVERSITY OF MELBOURNE

$T(n) = T(n/2) + n$ a = 1, b = 2, d = 1

$T(n) = T(n/8) + n/4 + n/2 + n$

$T(n) \in \Theta(n)$

Master Theorem: Example 4

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n^2$ a = 2, b = 2, d = 2

Master Theorem: Example 4

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n^2$

a = 2, b = 2, d = 2

$a < b^d$

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

So, by the Master Theorem, $T(n) \in \Theta(n^2)$

Master Theorem: Example 4

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n^2$ a = 2, b = 2, d = 2

Master Theorem: Example 4

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n^2$ a = 2, b = 2, d = 2

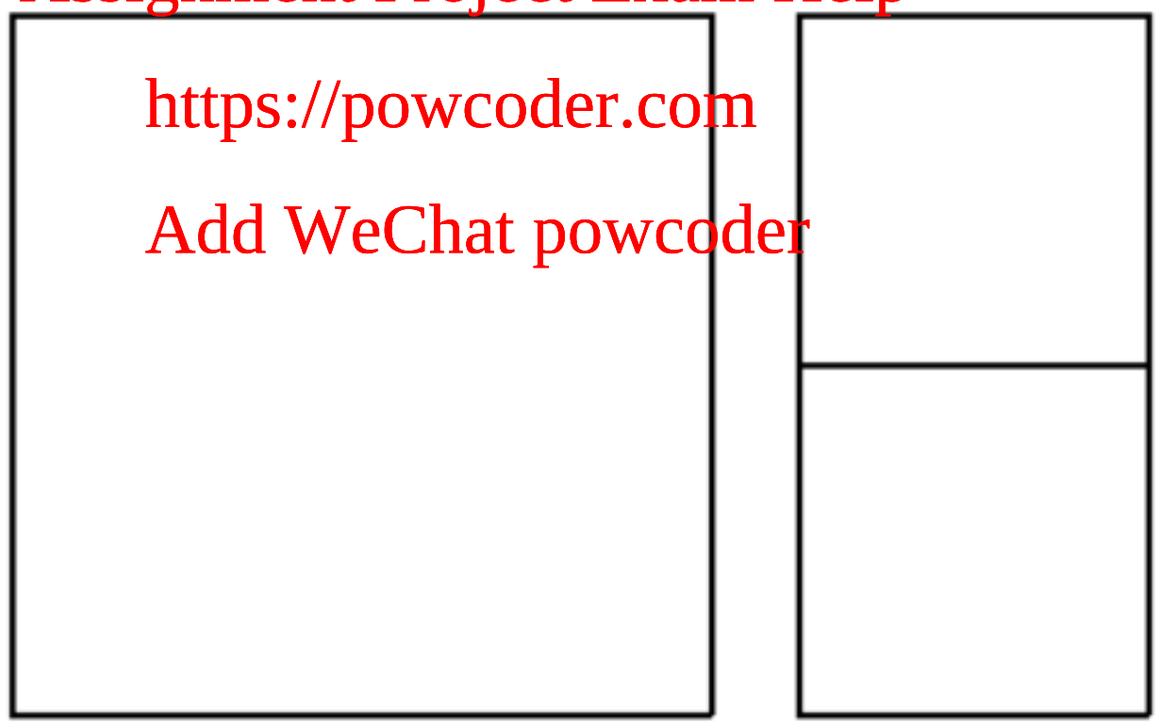
$T(n) = 2(2T(n/4) + (n/2)^2) + n^2$

Master Theorem: Example 4

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n^2$ a = 2, b = 2, d = 2

$T(n) = 4T(n/4) + 2(n/2)^2 + n^2$

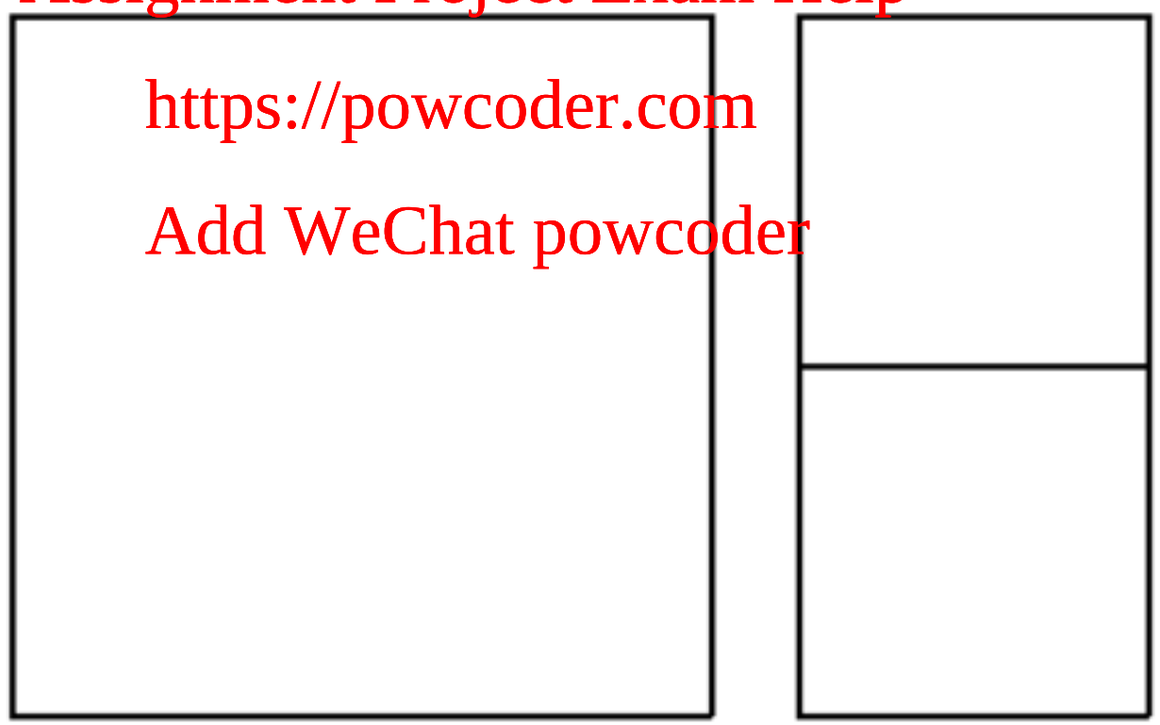


Master Theorem: Example 4

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n^2$ a = 2, b = 2, d = 2

$T(n) = 4(2T(n/8) + (n/4)^2) + 2(n/2)^2 + n^2$

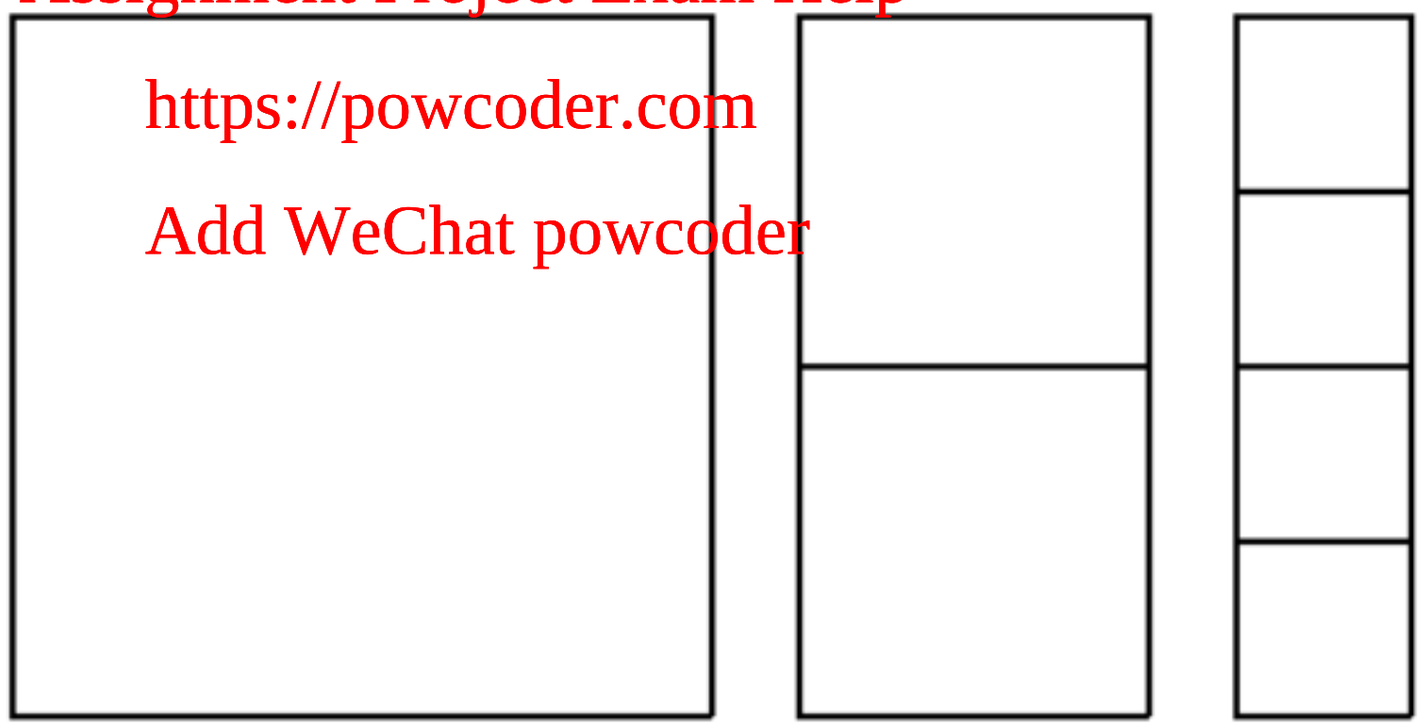


Master Theorem: Example 4

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n^2$ a = 2, b = 2, d = 2

$T(n) = 8T(n/8) + 4(n/4)^2 + 2(n/2)^2 + n^2$

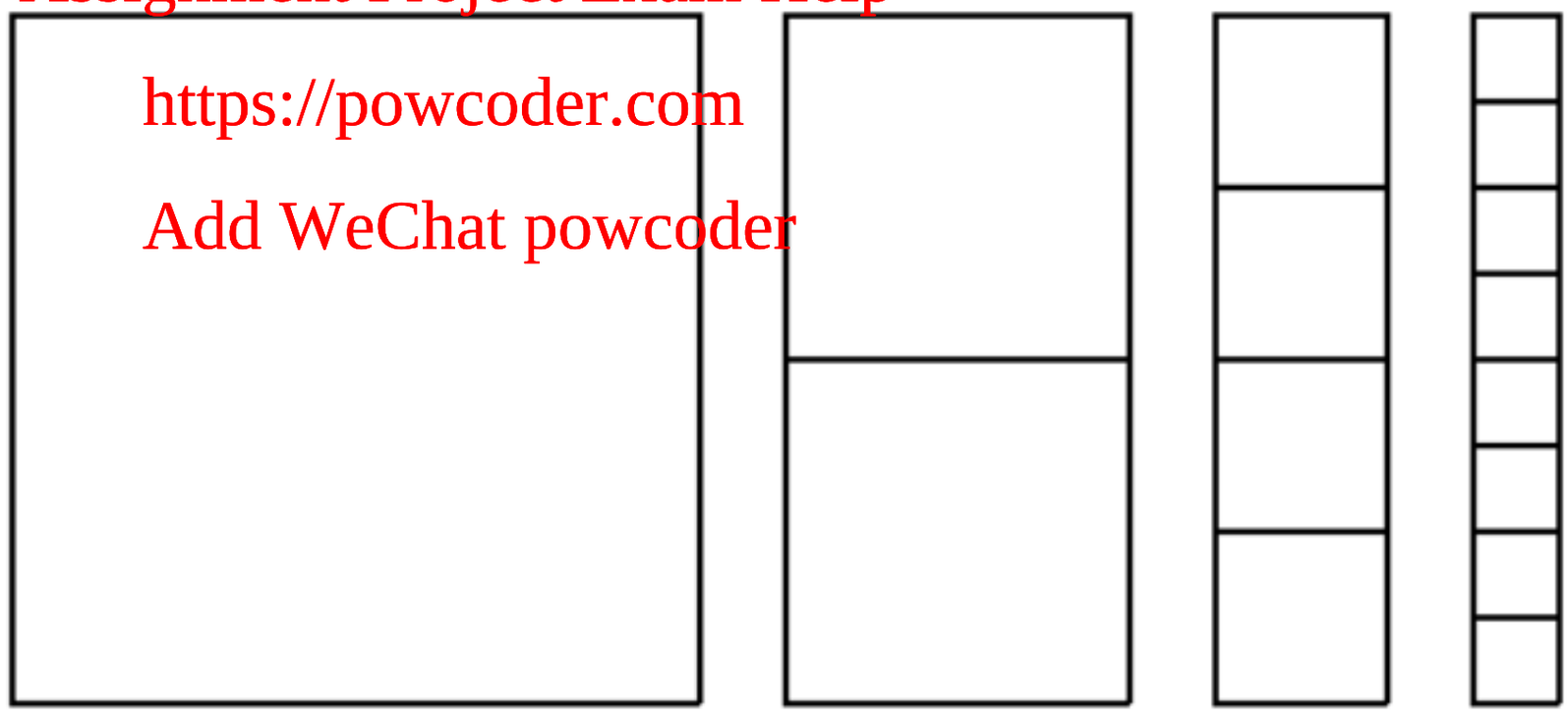


Master Theorem: Example 4

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n^2$ a = 2, b = 2, d = 2

$T(n) = 8T(n/8) + 4(n/4)^2 + 2(n/2)^2 + n^2$



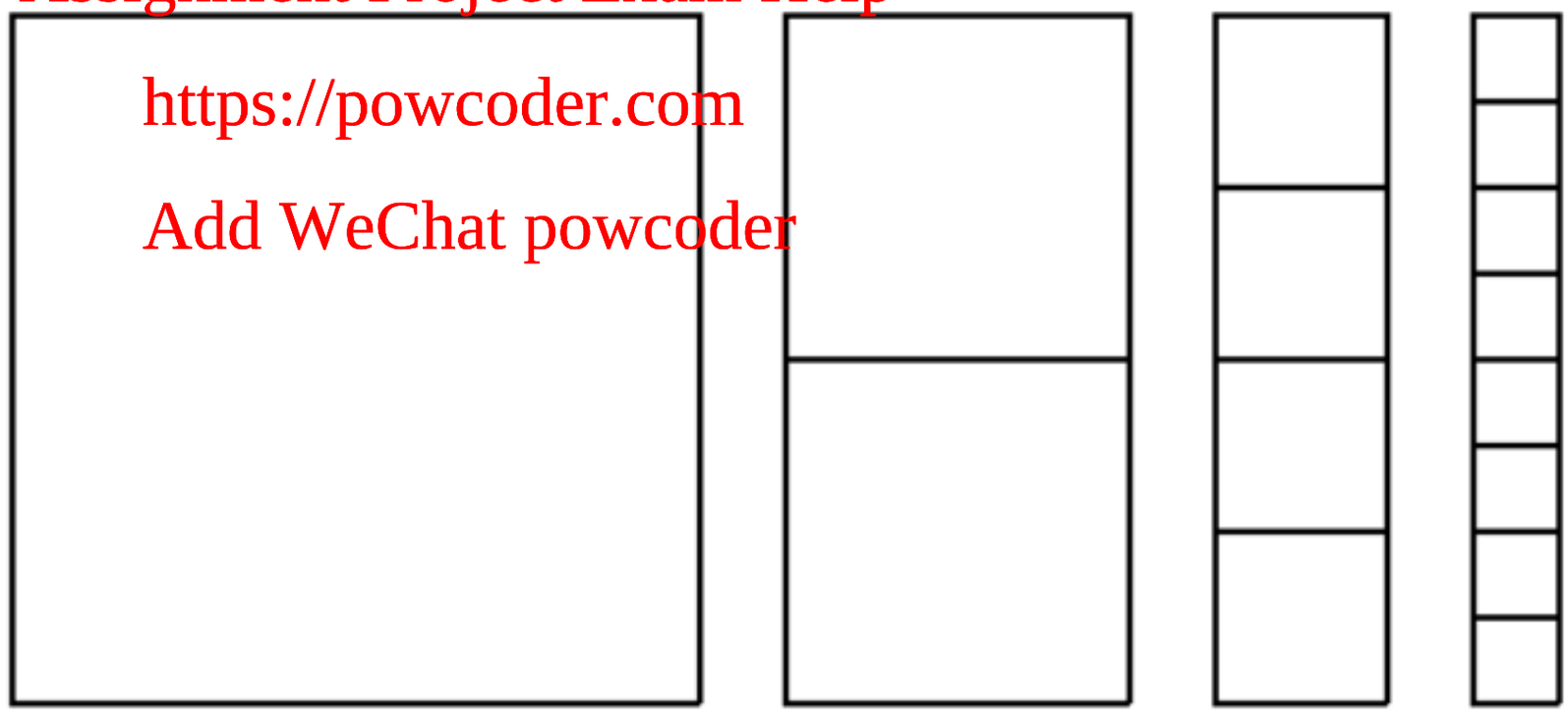
Master Theorem: Example 4

THE UNIVERSITY OF MELBOURNE

$T(n) = 2T(n/2) + n^2$ a = 2, b = 2, d = 2

$T(n) = 8T(n/8) + 4(n/4)^2 + 2(n/2)^2 + n^2$

$T(n) \in \Theta(n^2)$



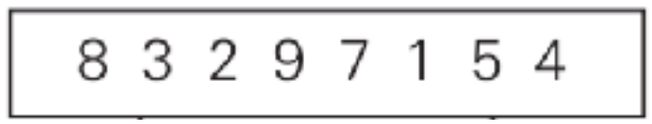
Mergesort

- Perhaps the most obvious application of divide-and-conquer:

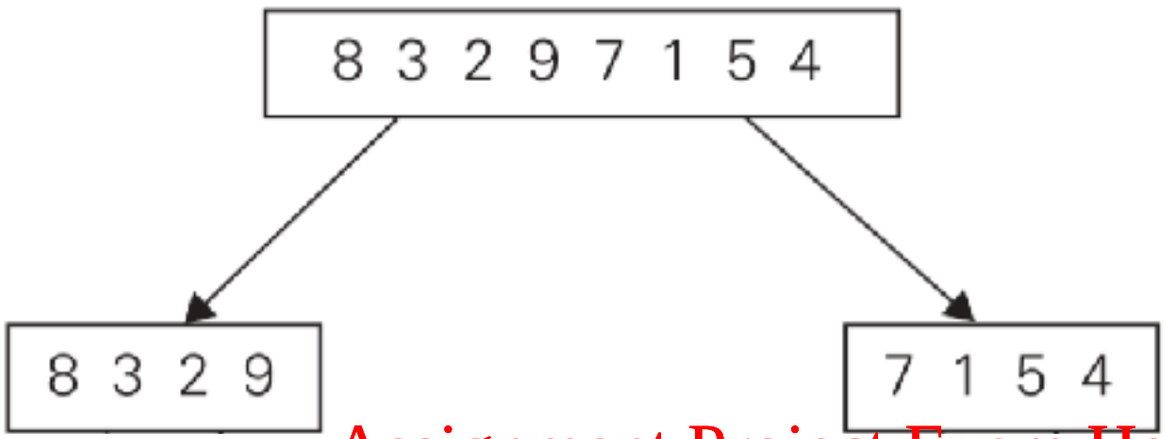
- To sort an array (or a list), cut it into two halves, sort each half, and merge the two results.


```
procedure MergeSort(A[], n)                                  ▷ Sort A[0]..A[n − 1]
    if n > 1 then
        for i ← 0 to ⌊n/2⌋ − 1 do                            ▷ Copy left half of A to B
            B[i] ← A[i]
        for i ← 0 to ⌈n/2⌉ − 1 do                            ▷ Copy right half of A to C
            C[i] ← A[⌊n/2⌋ + i]
        MergeSort(B, ⌊n/2⌋)                                  ▷ Sort B
        MergeSort(C, ⌈n/2⌉)                                  ▷ Sort C
        Merge(B, ⌊n/2⌋, C, ⌈n/2⌉, A)                        ▷ Merge B and C into A
```

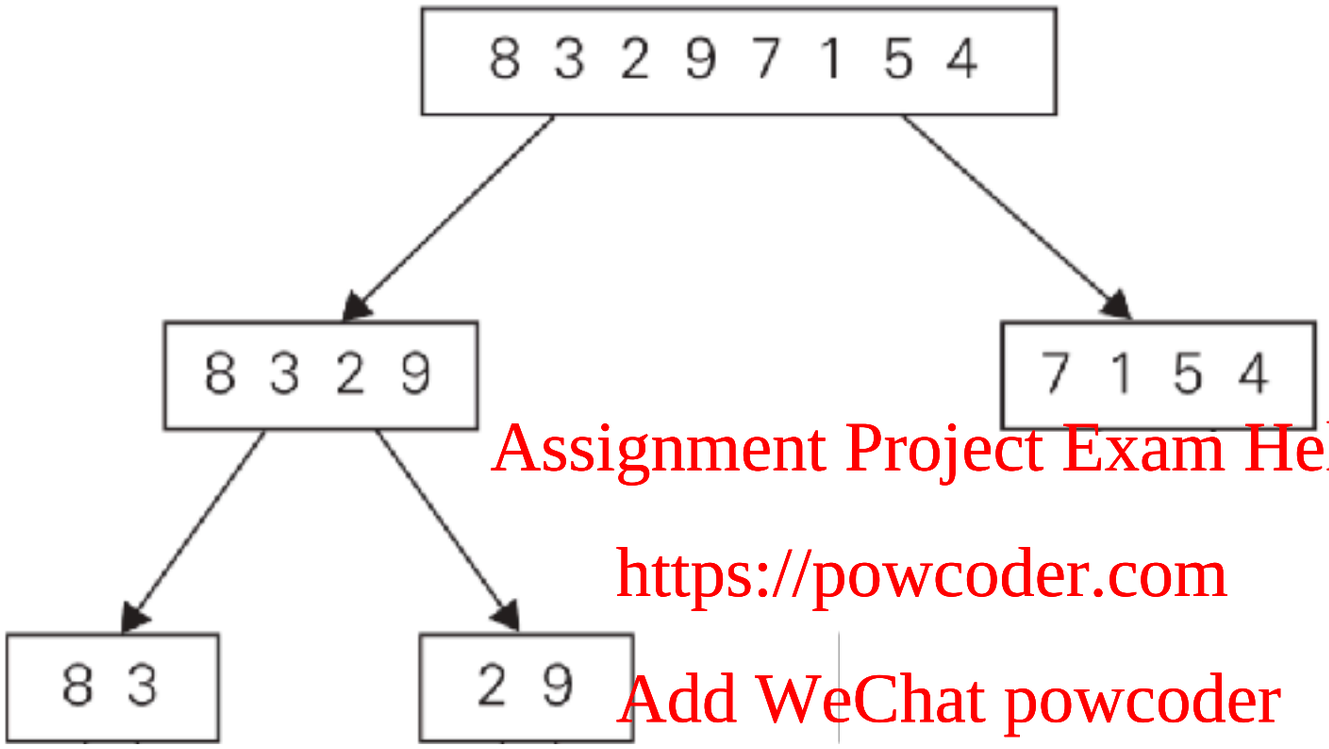
Mergesort



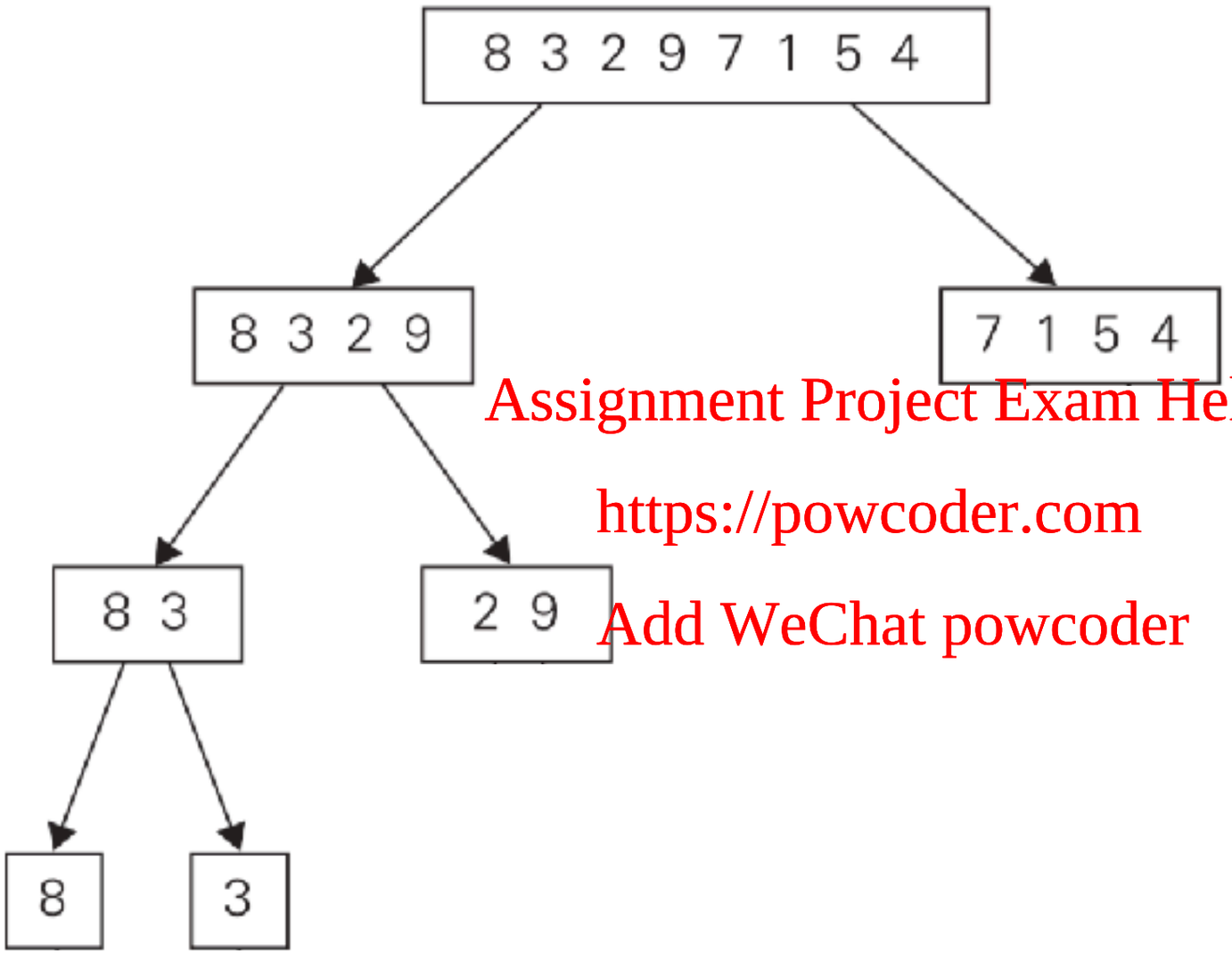
Mergesort



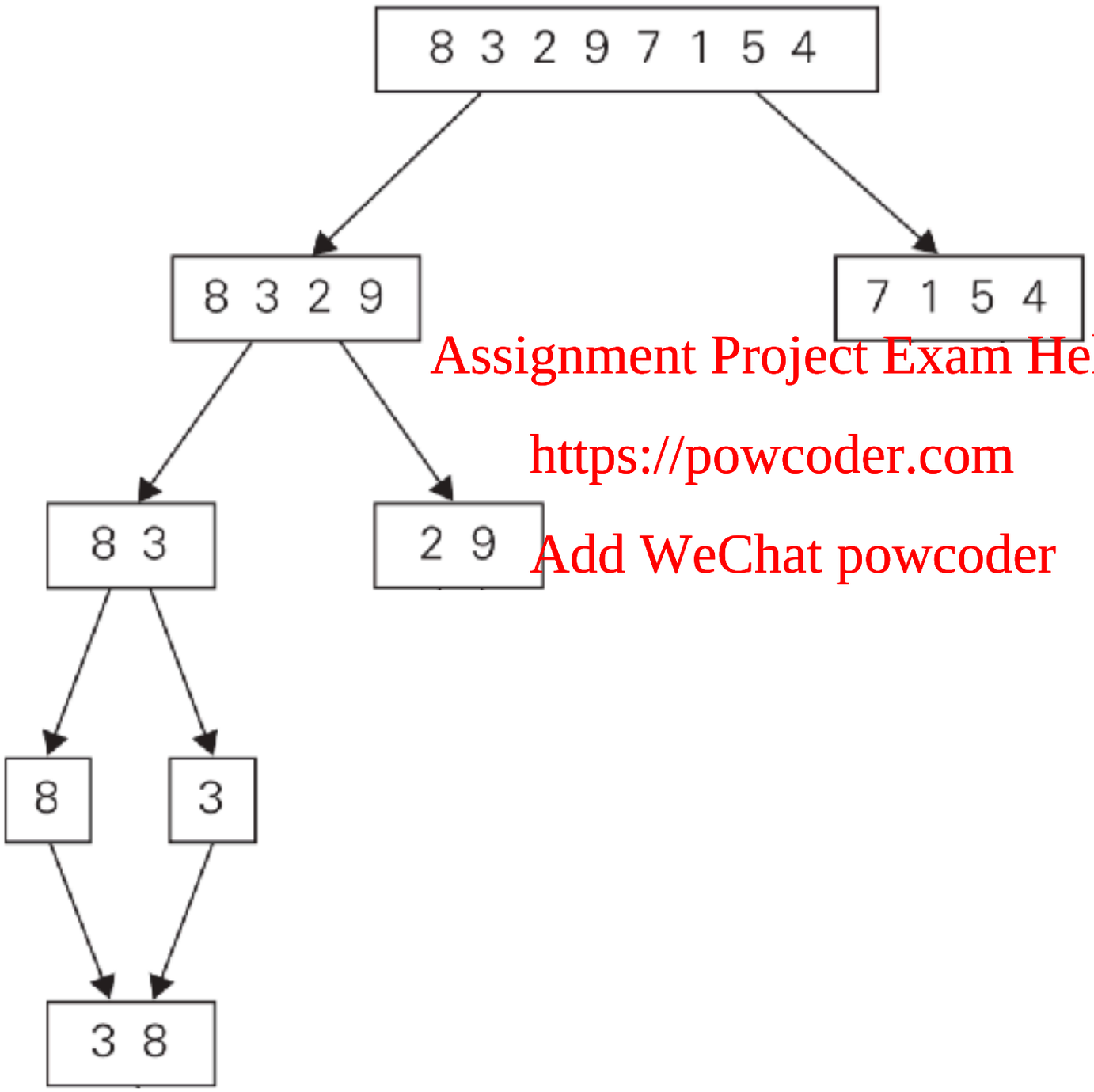
Mergesort



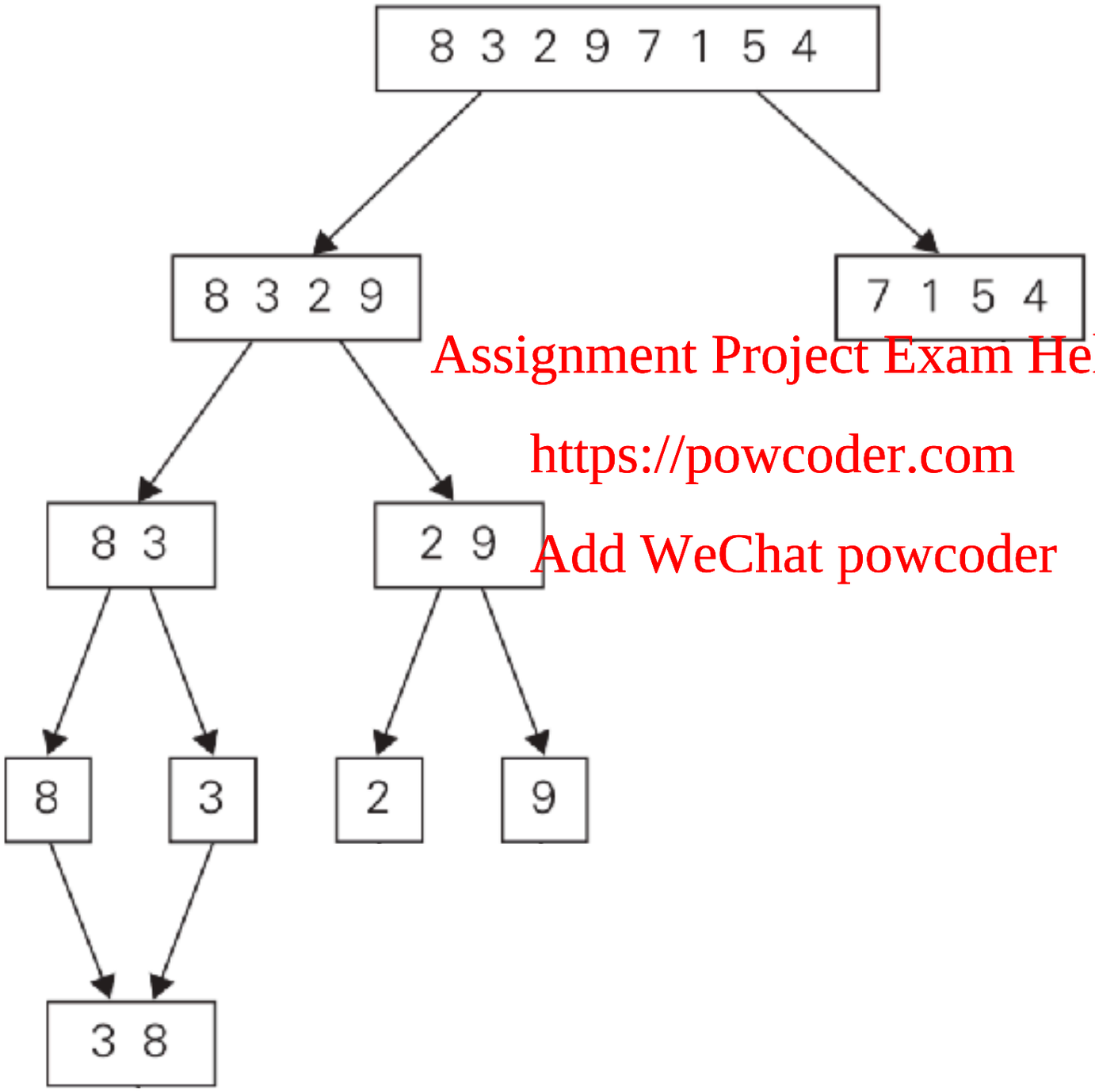
Mergesort



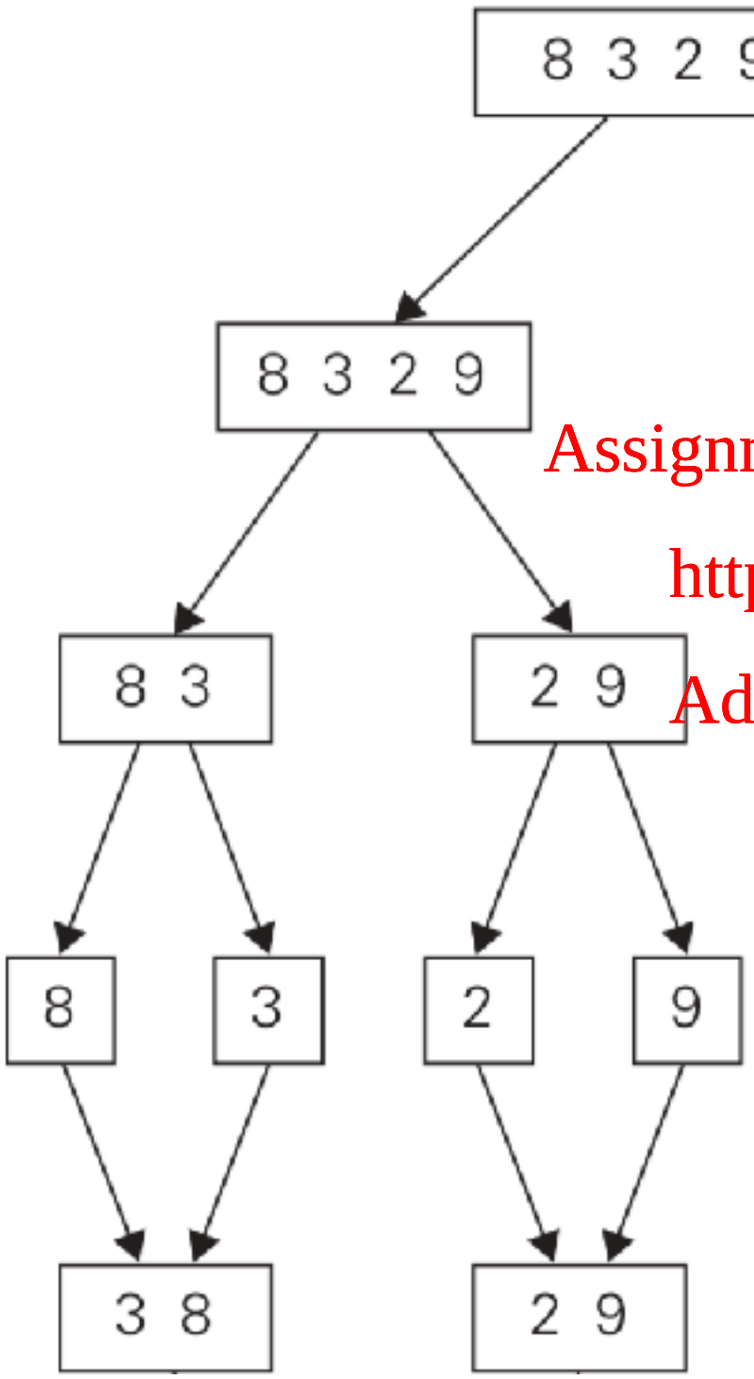
Mergesort



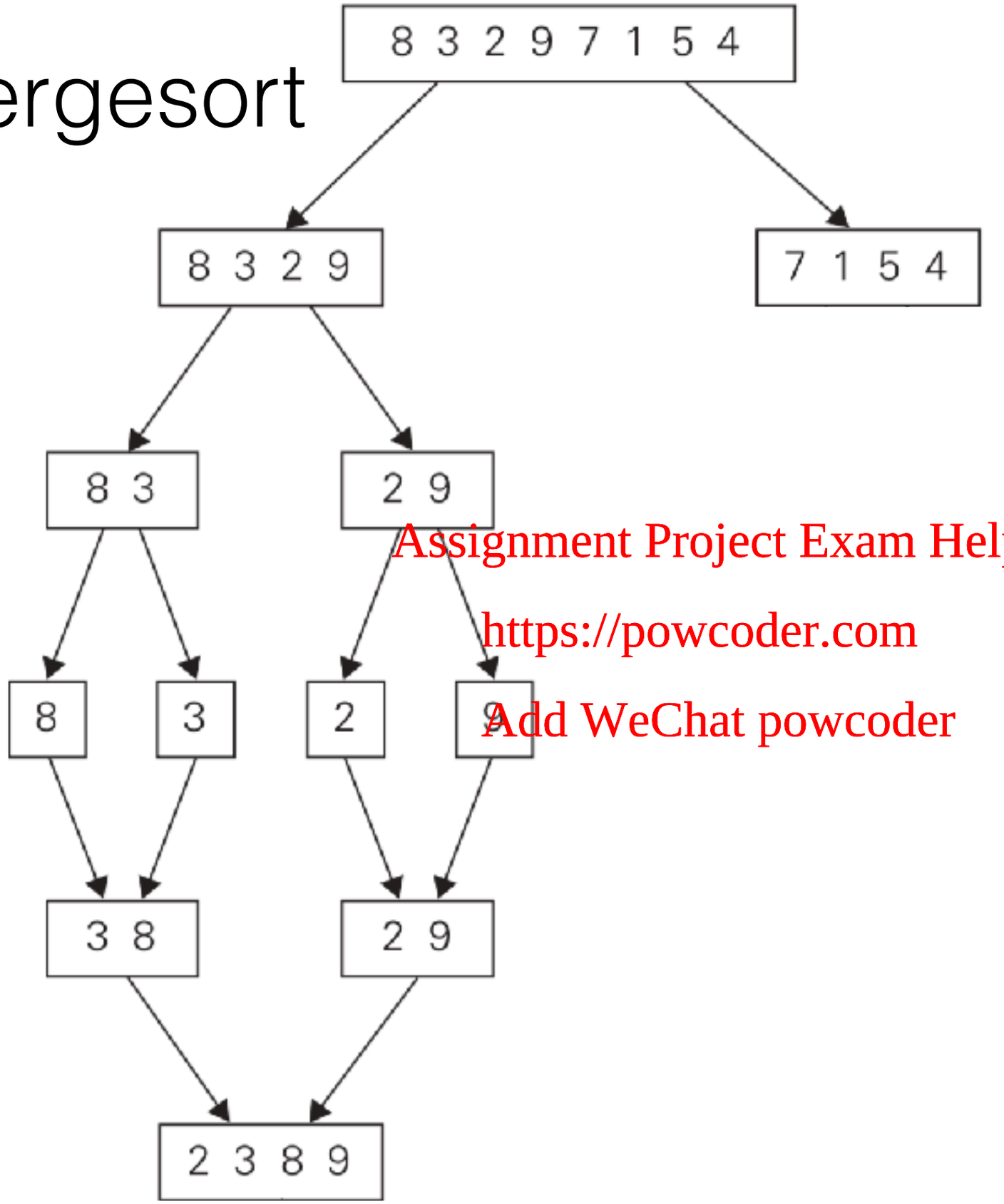
Mergesort



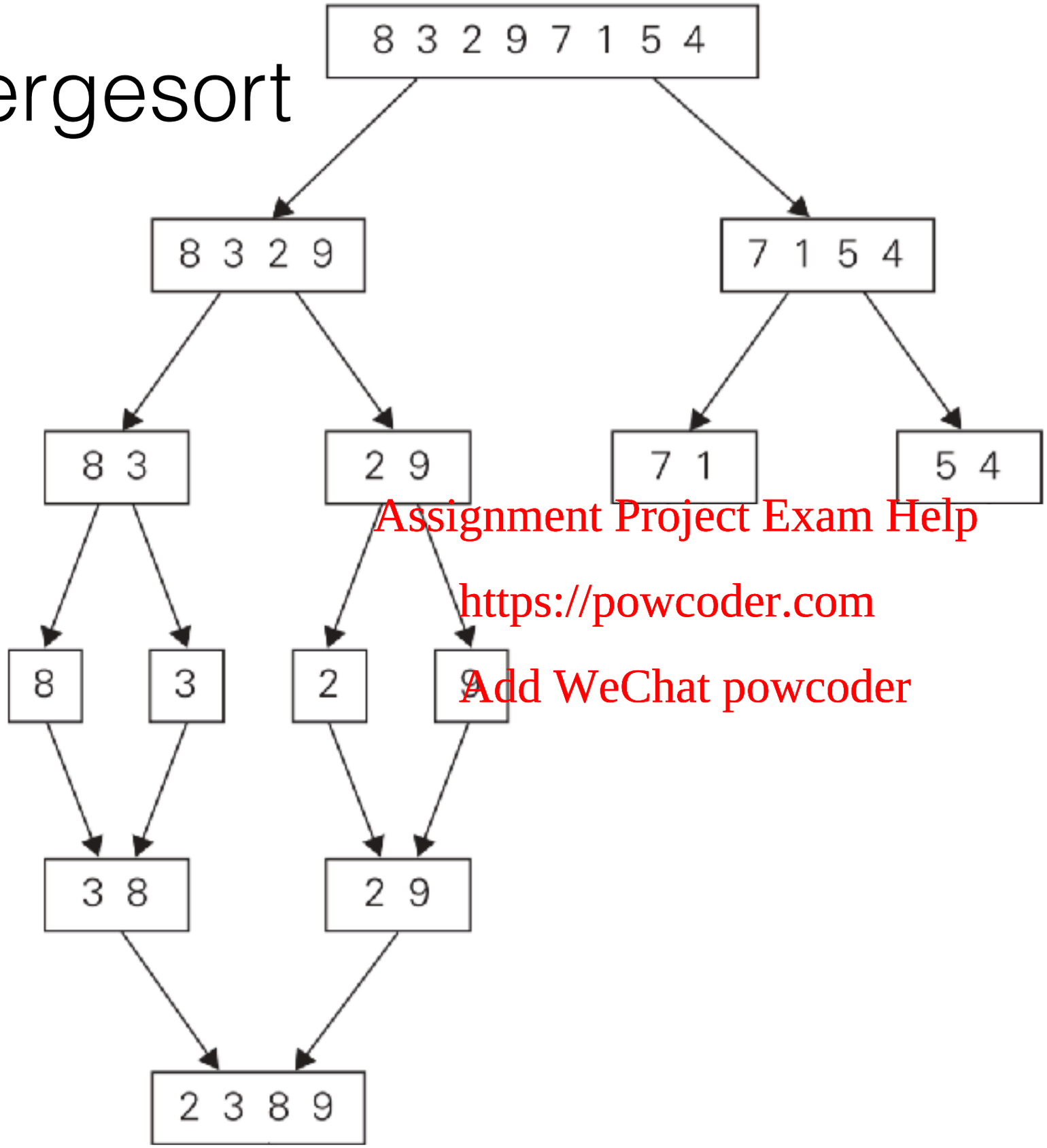
Mergesort



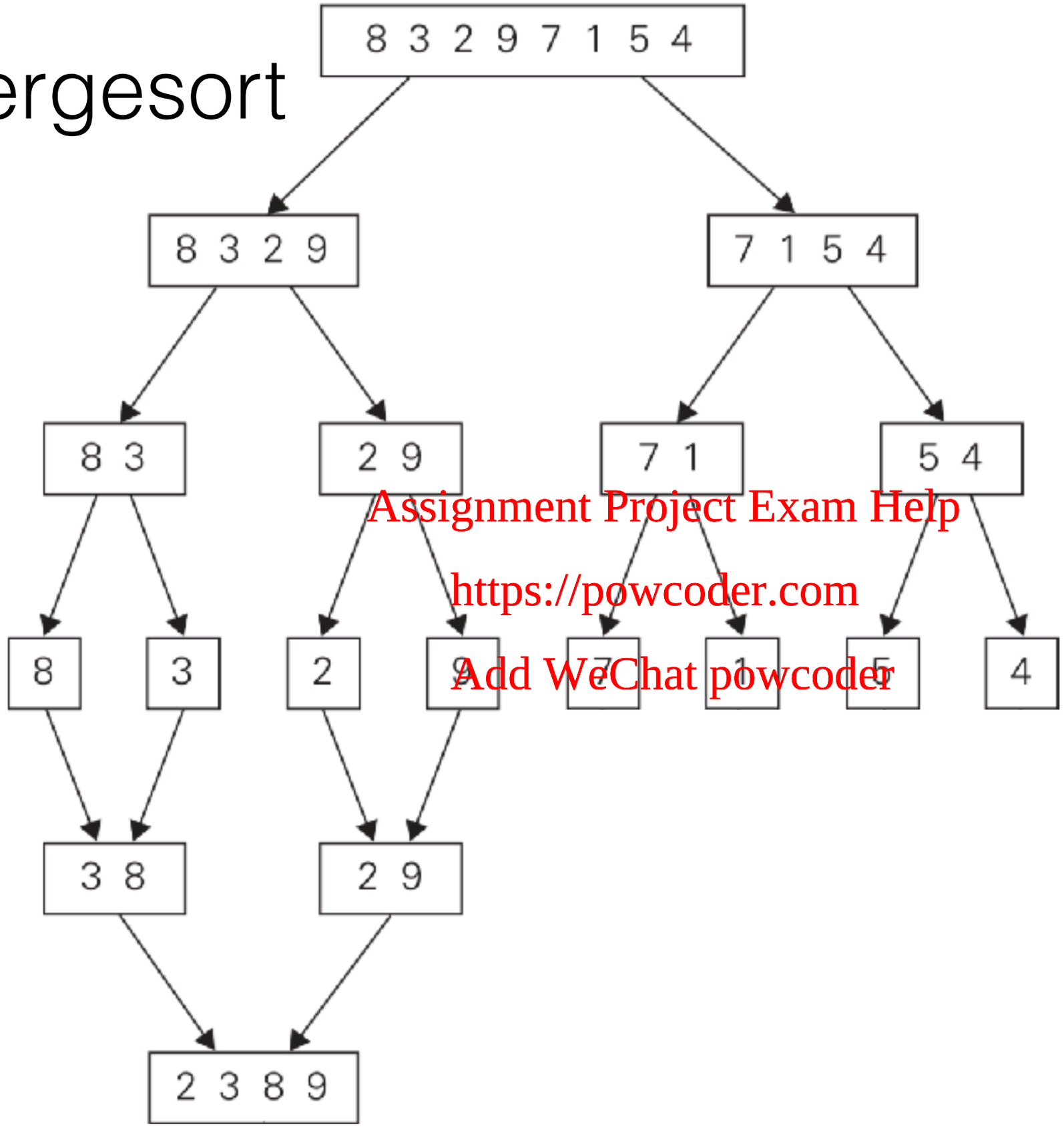
Mergesort



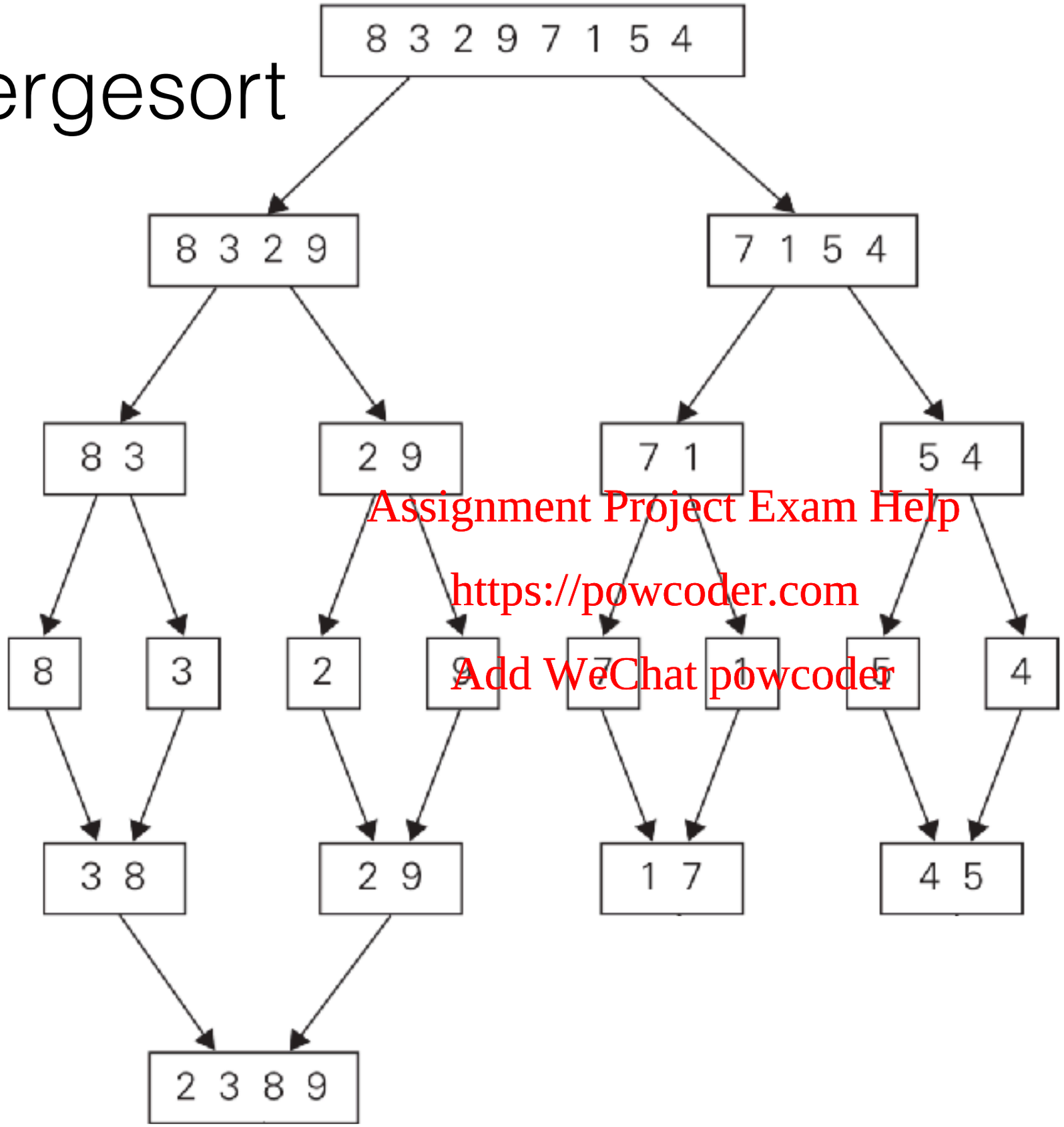
Mergesort



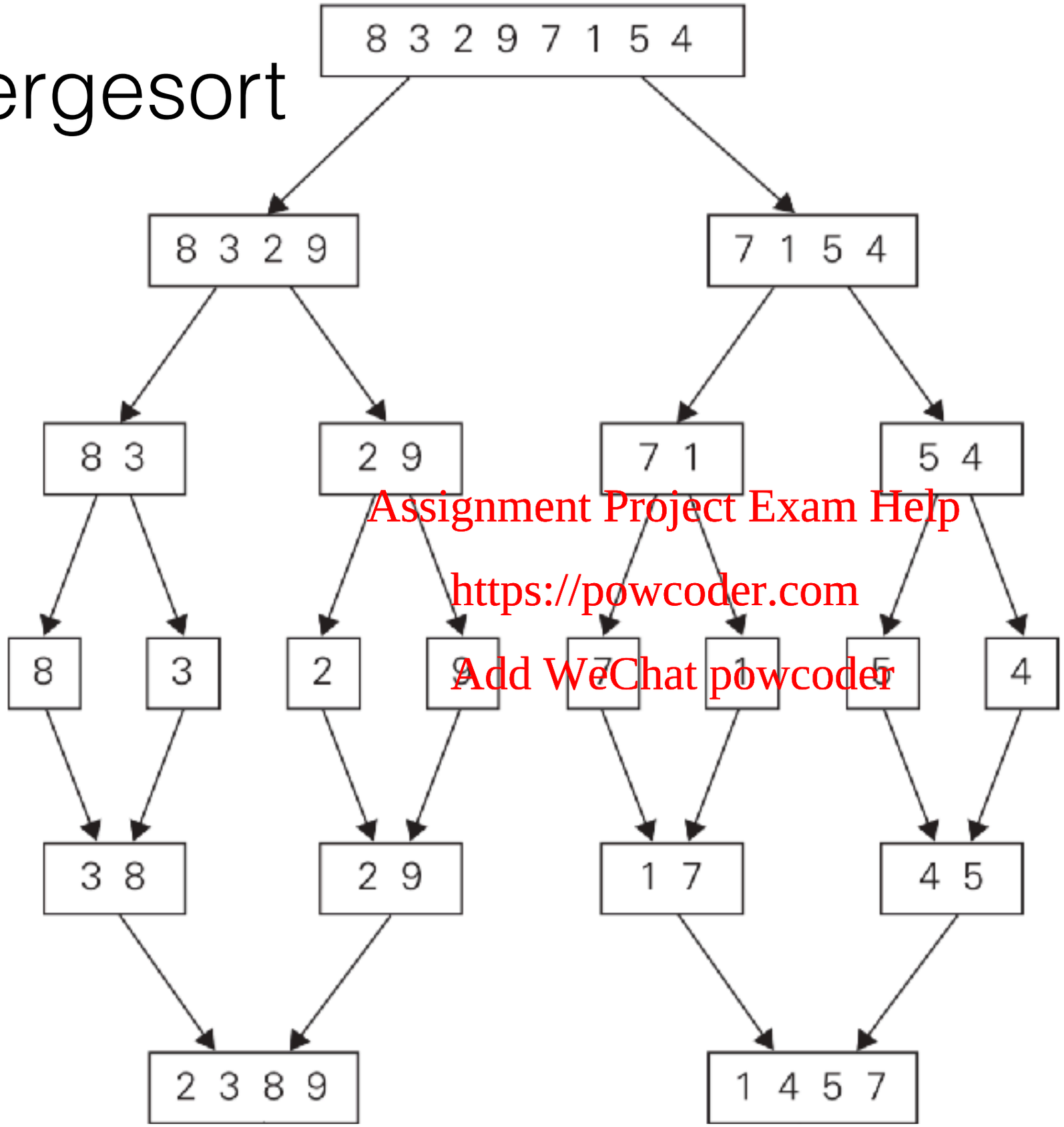
Mergesort



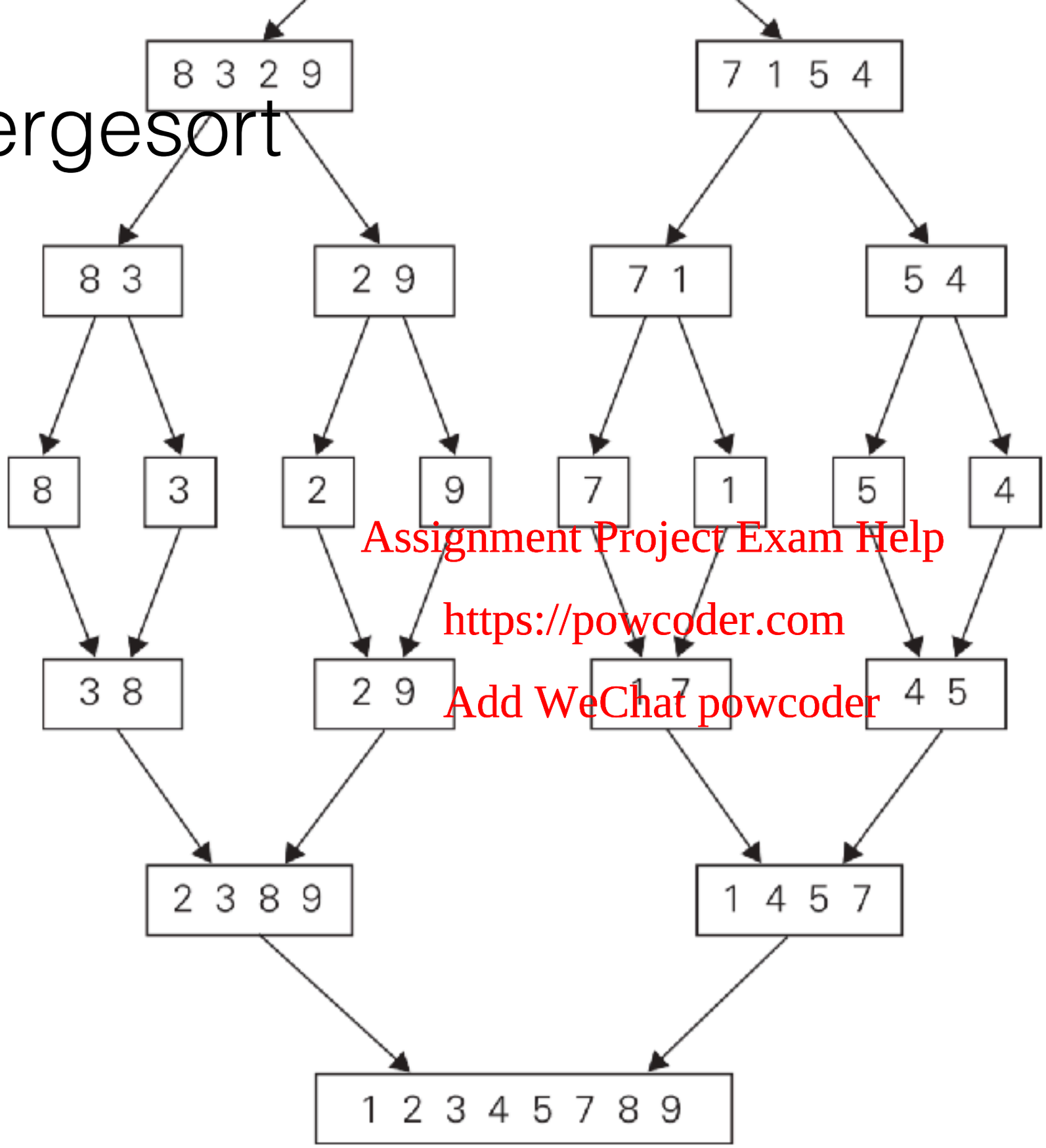
Mergesort



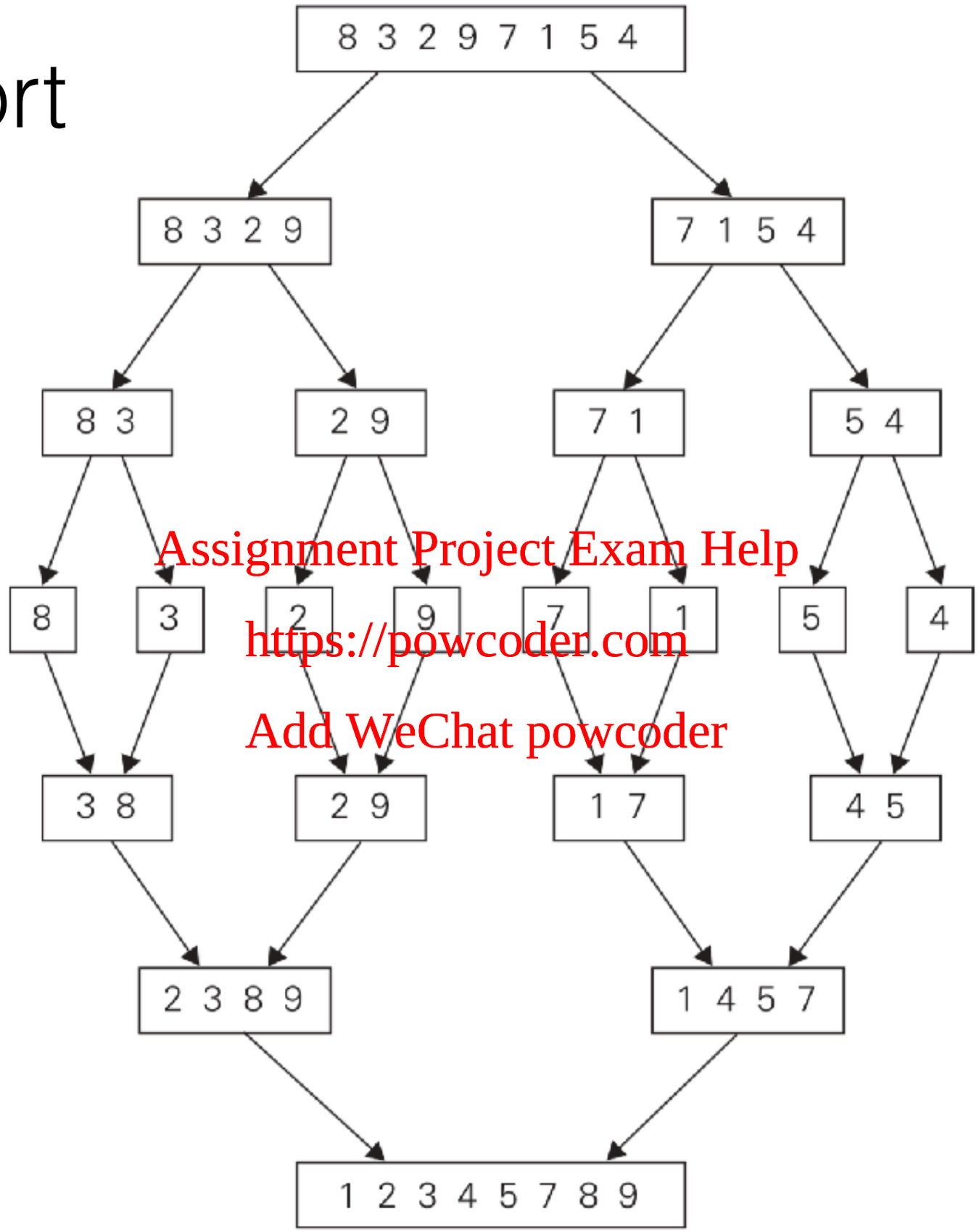
Mergesort



Mergesort



Mergesort



Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE

procedure MERGE($B[\cdot], p, C[\cdot], q, A[\cdot]$)

$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$

while $i < p$ and $j < q$ **do**

if $B[i] \leq C[j]$ **then**

$A[k] \leftarrow B[i]$

$i \leftarrow i + 1$

else

$A[k] \leftarrow C[j]$

$j \leftarrow j + 1$

$k \leftarrow k + 1$

if $i = p$ **then**

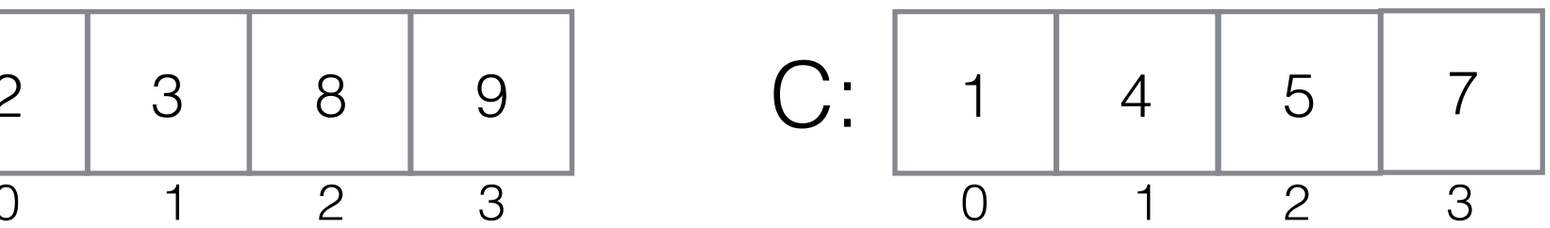
copy $C[j]..C[q-1]$ to $A[k]..A[p+q-1]$ ▷ (a for loop)

else

copy $B[i]..B[p-1]$ to $A[k]..A[p+q-1]$ ▷ (a for loop)

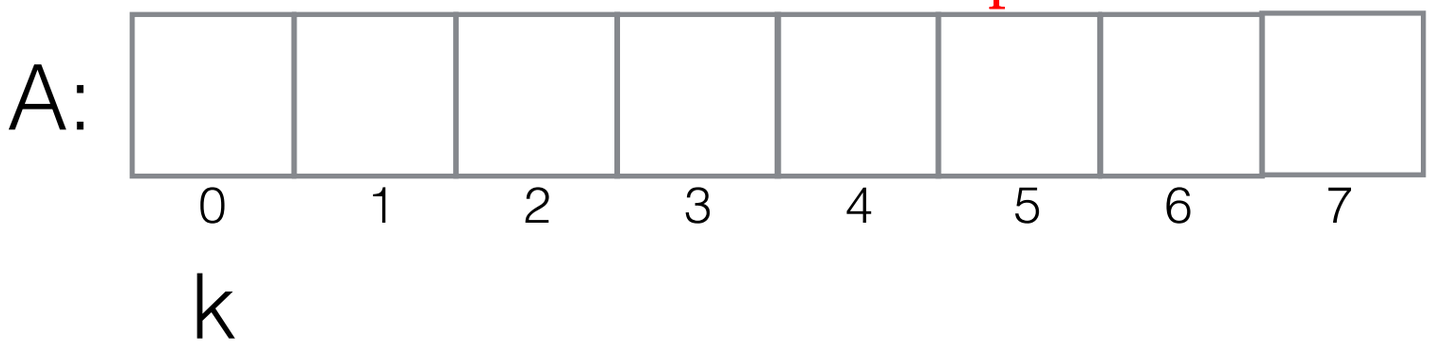
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



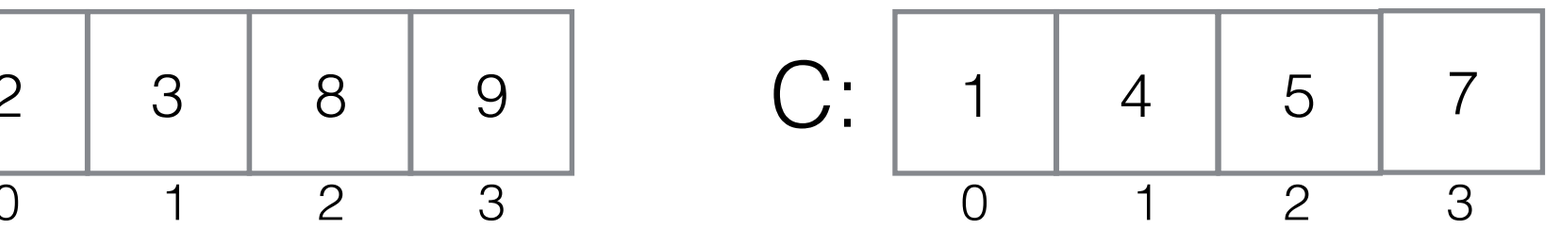
p: 4

q: 4



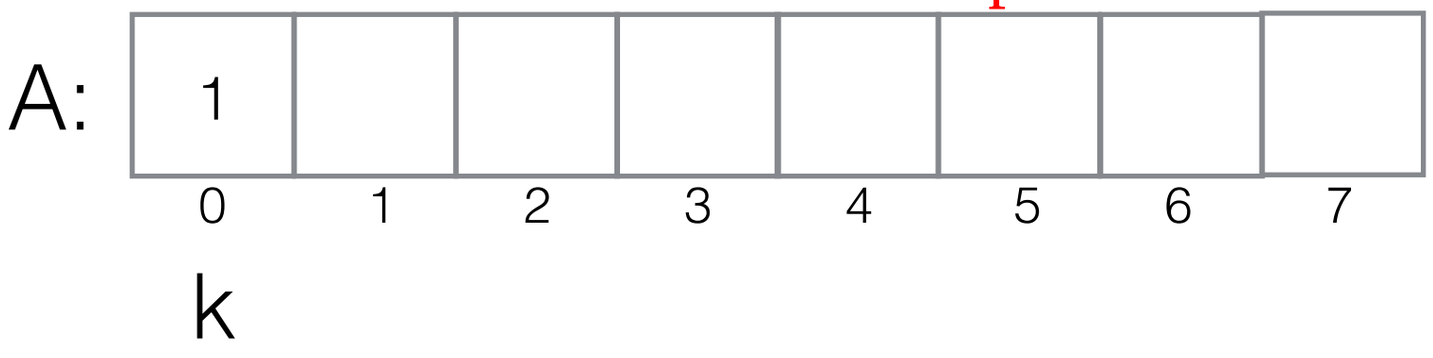
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



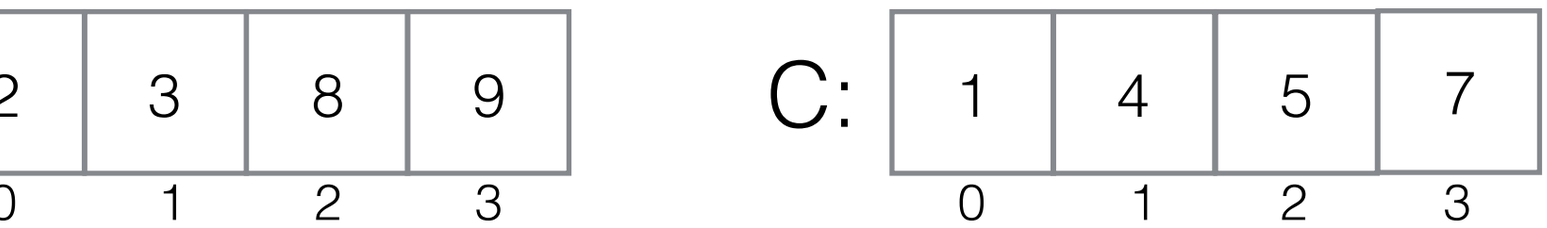
p: 4

q: 4



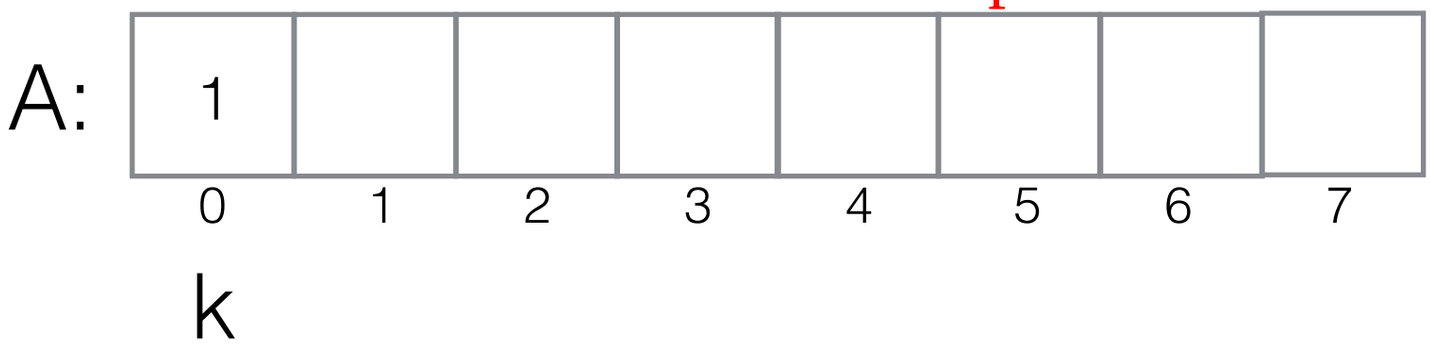
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



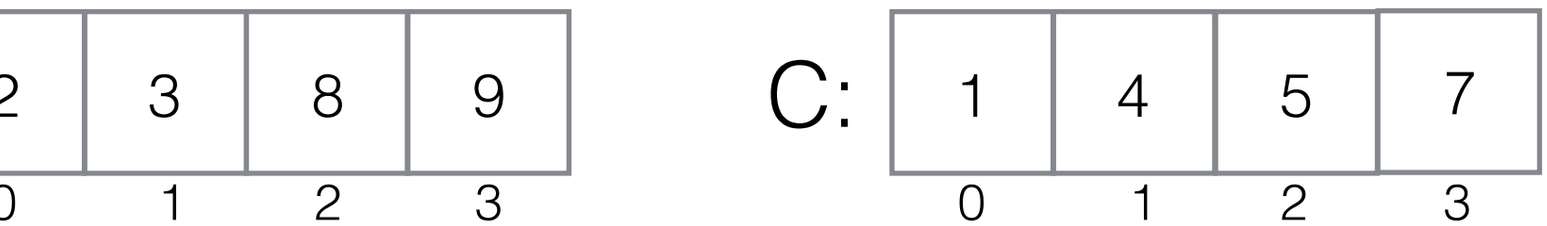
p: 4

q: 4



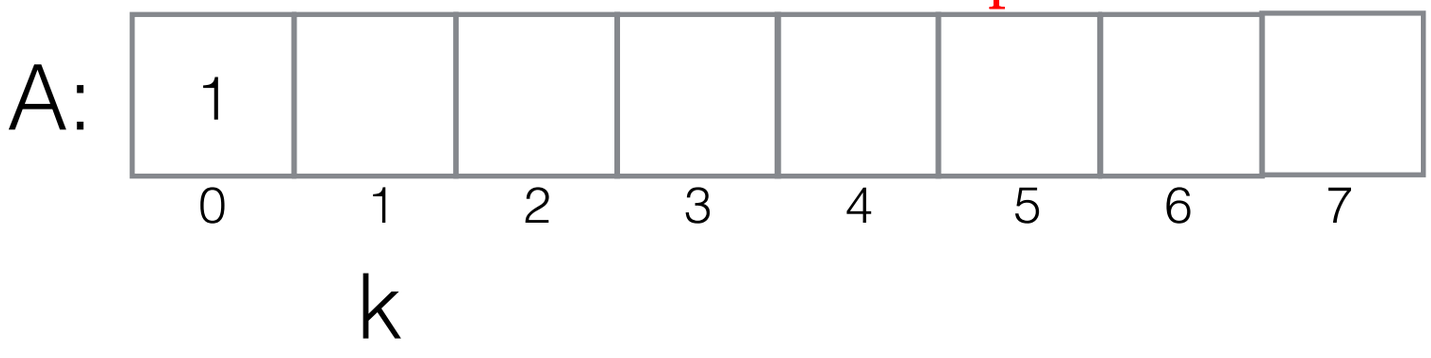
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



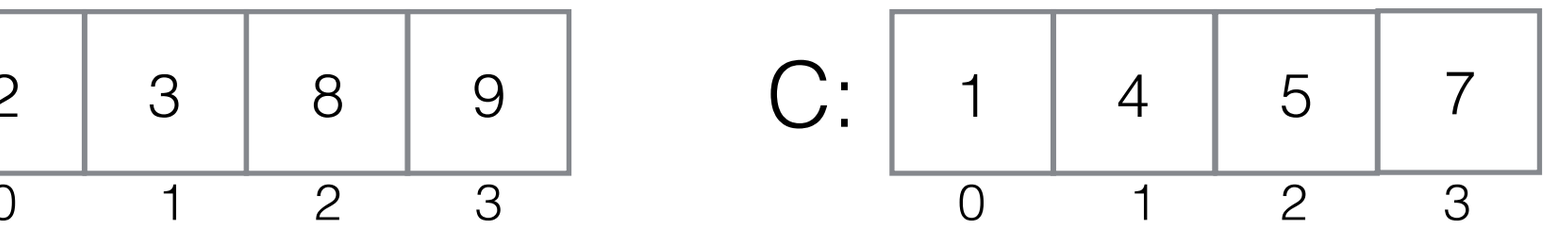
p: 4

q: 4



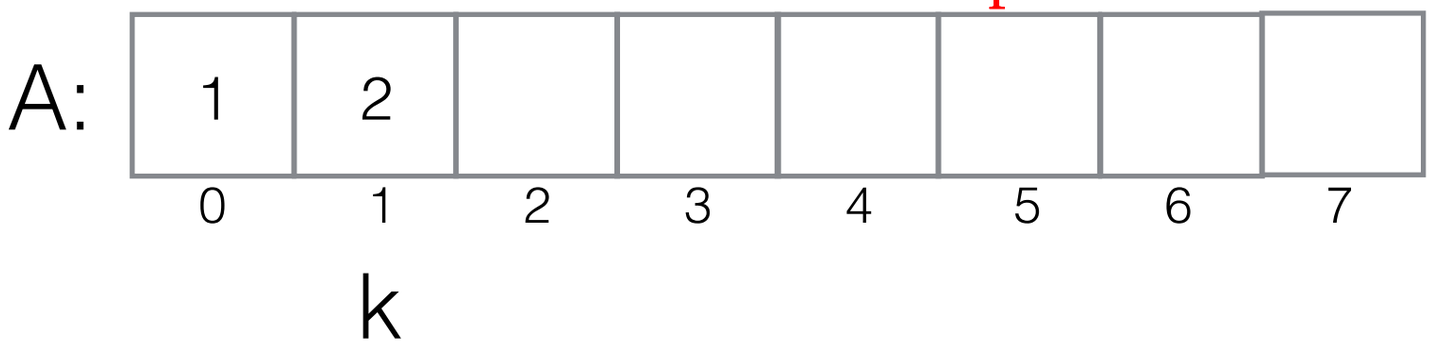
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



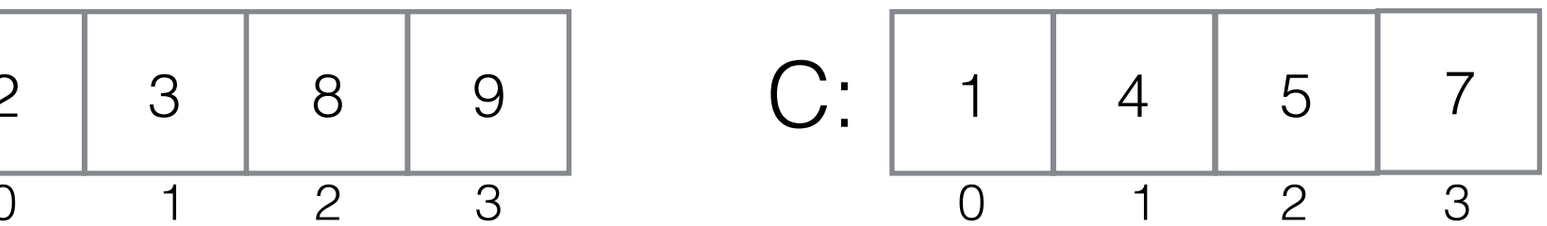
p: 4

q: 4



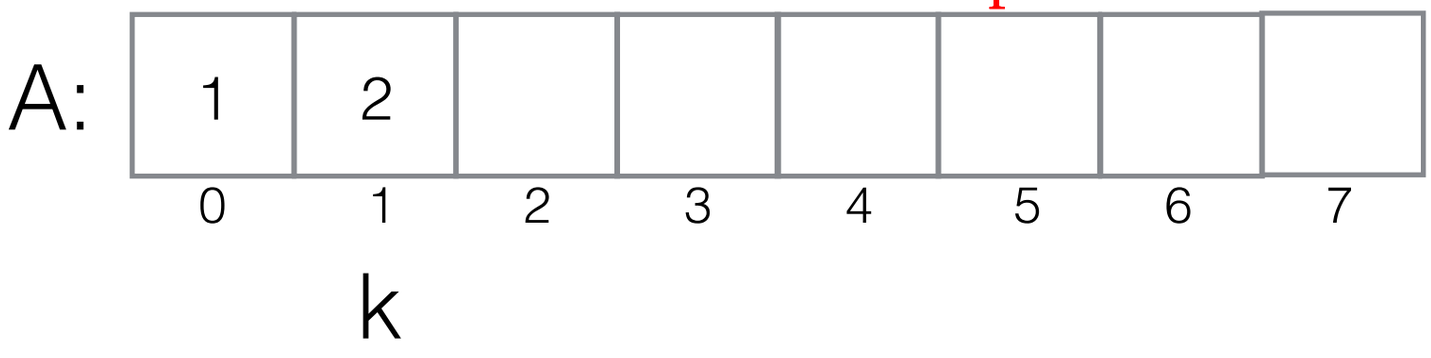
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



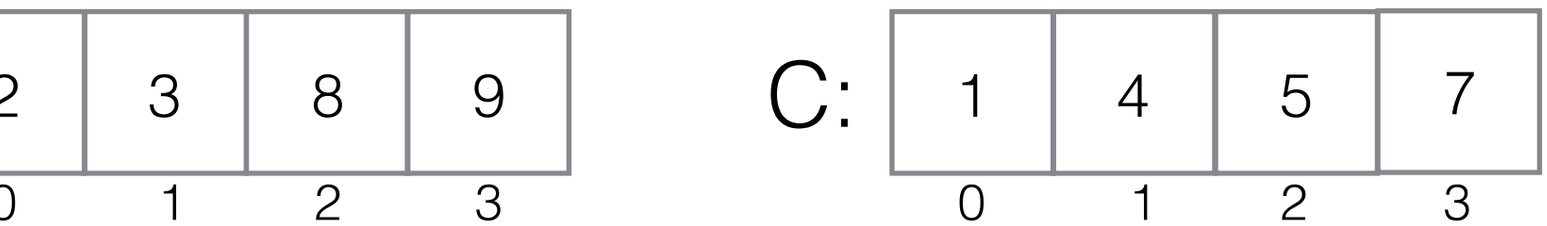
p: 4

q: 4



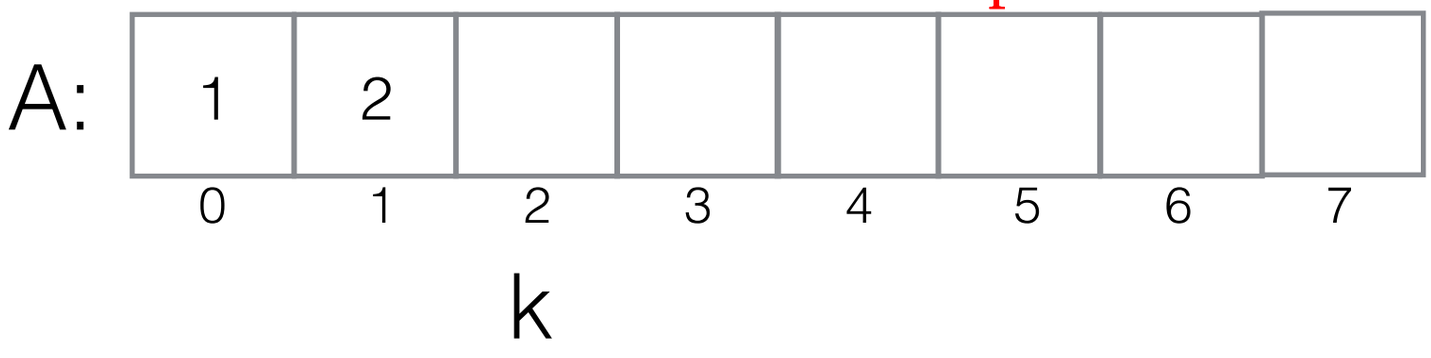
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



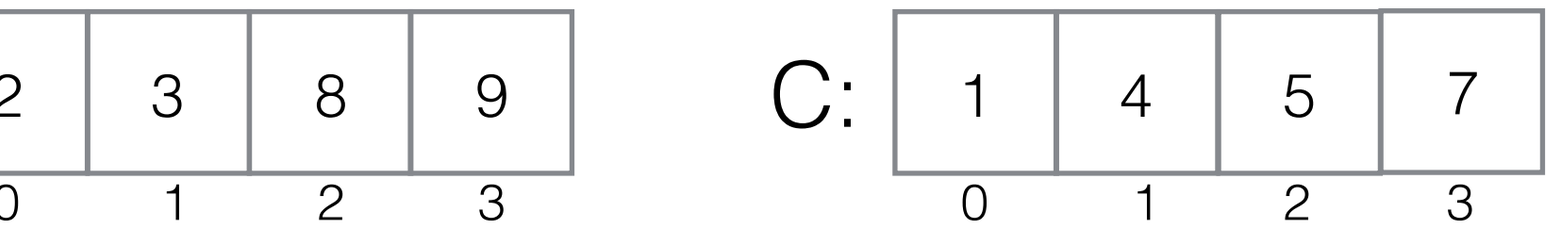
p: 4

q: 4



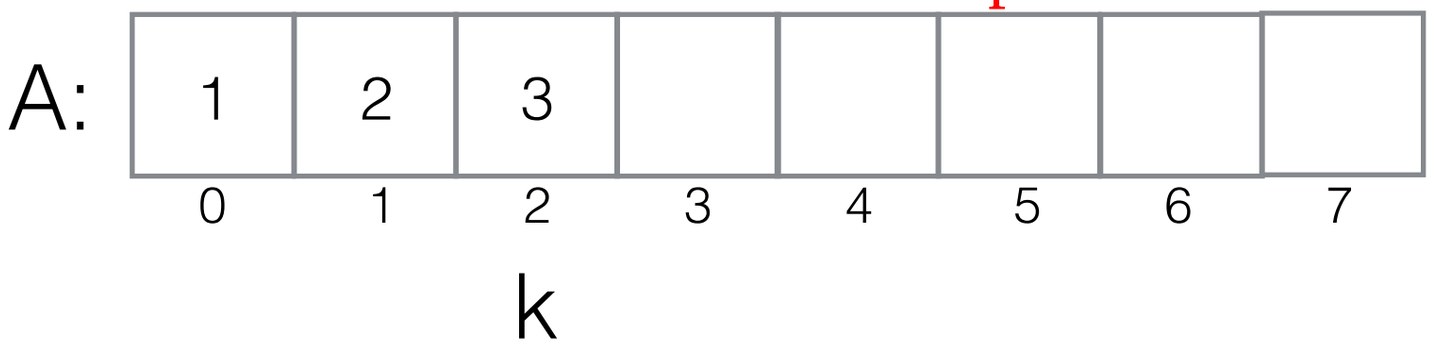
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



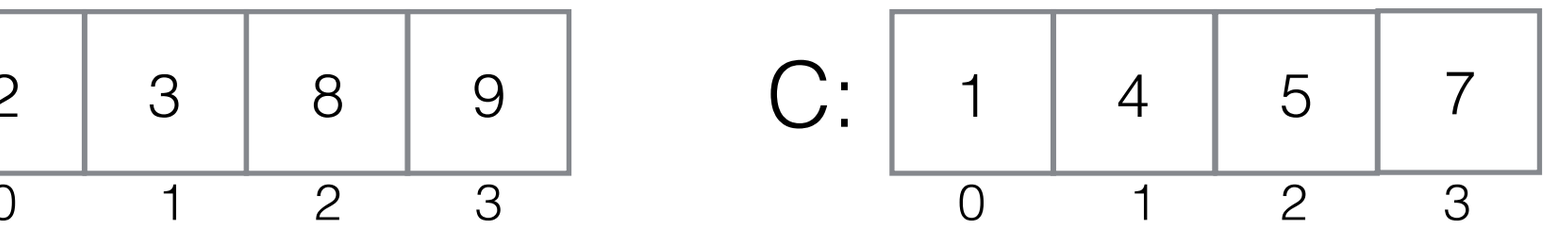
p: 4

q: 4



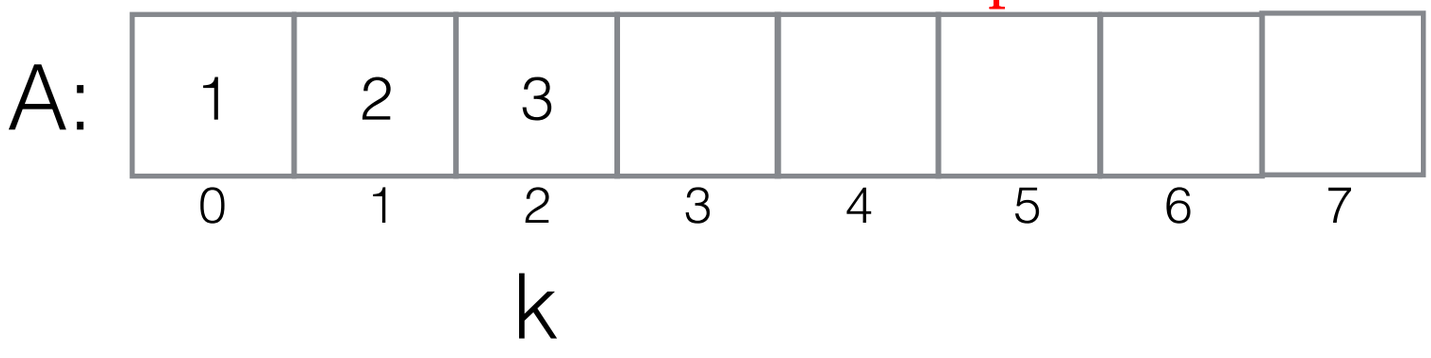
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



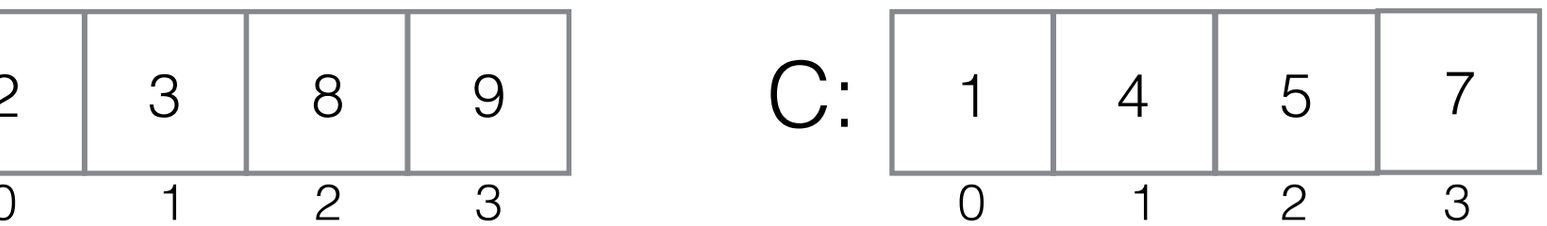
p: 4

q: 4



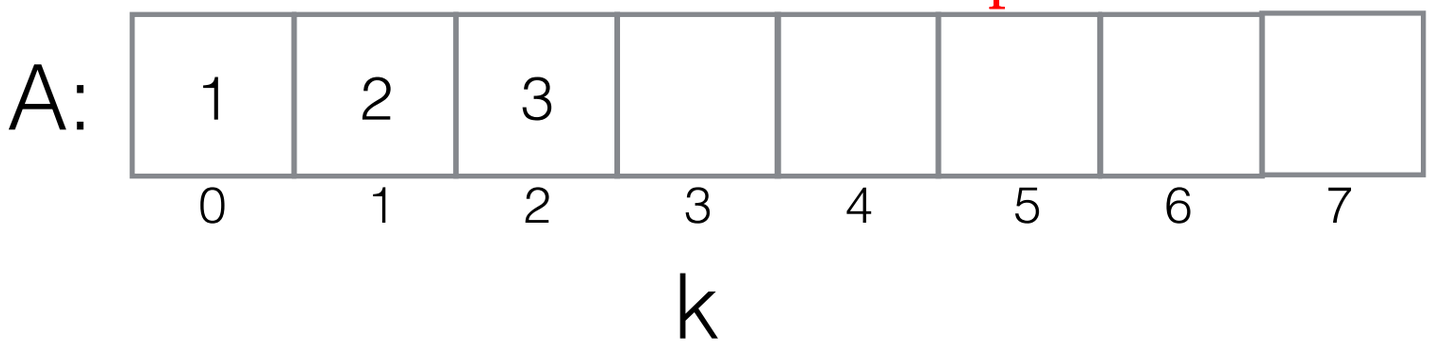
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



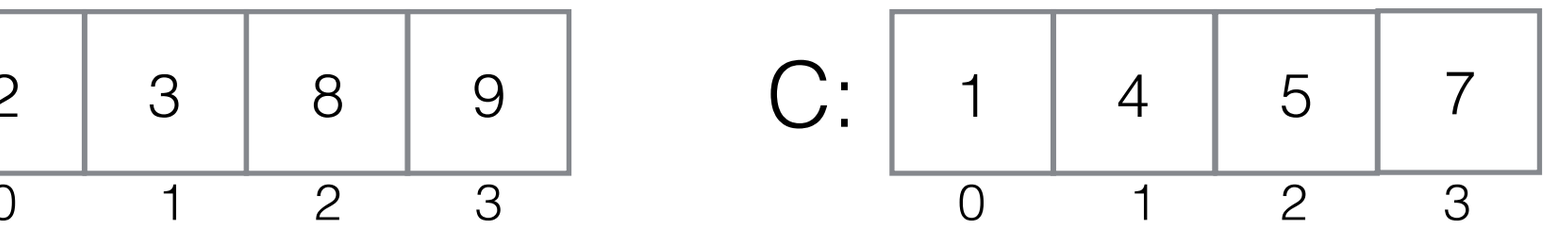
p: 4

q: 4



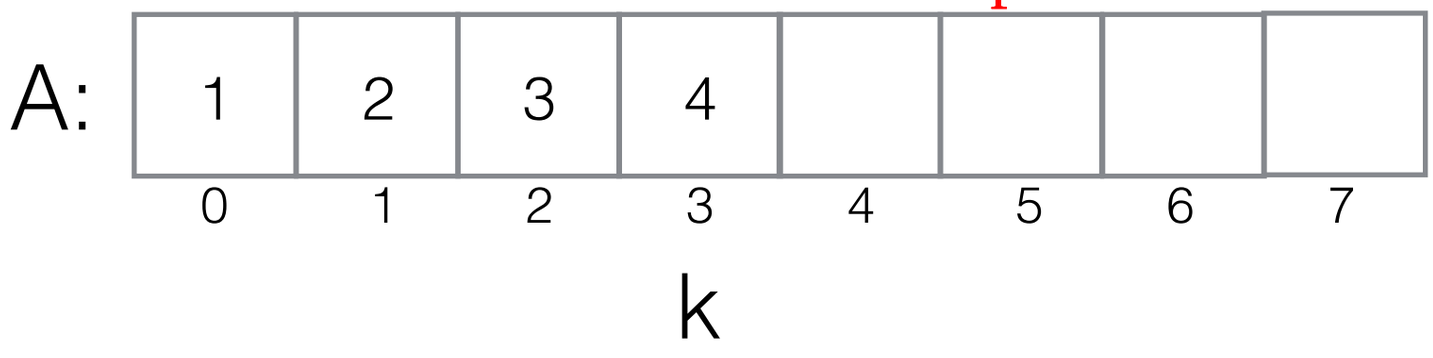
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



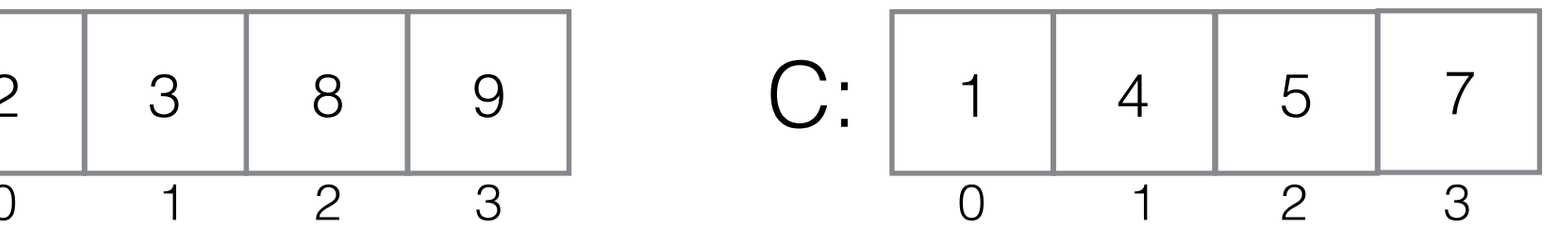
p: 4

q: 4



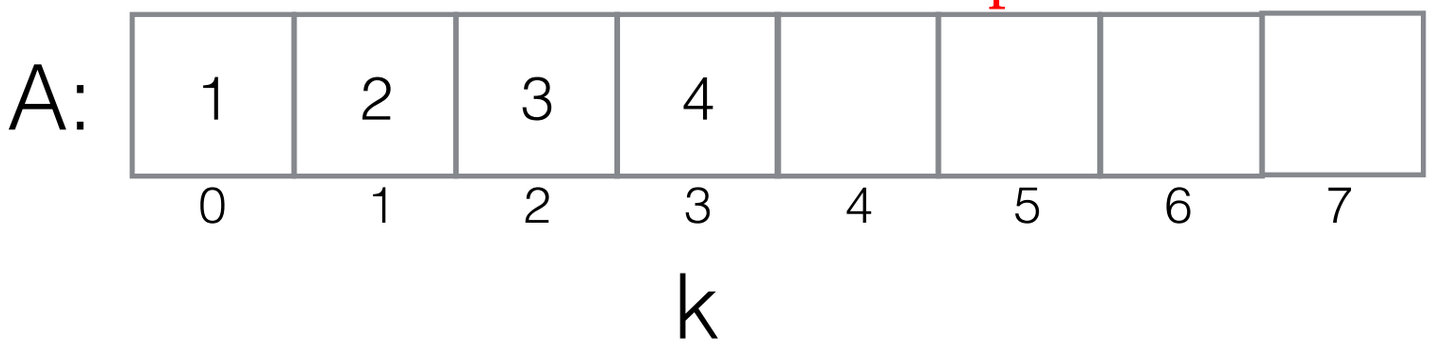
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



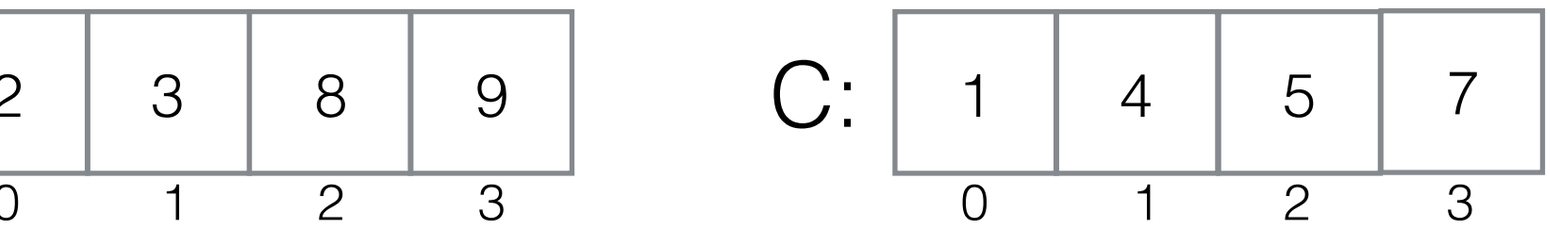
p: 4

q: 4



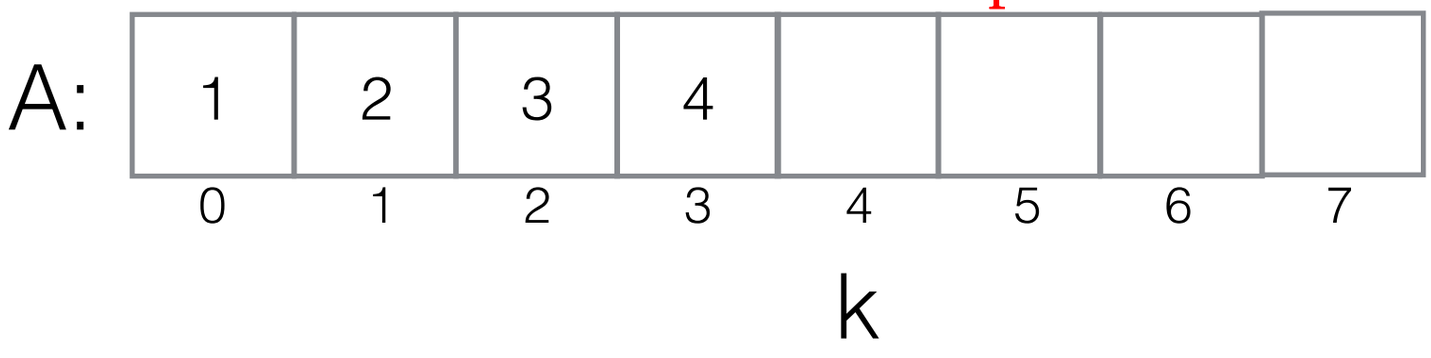
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



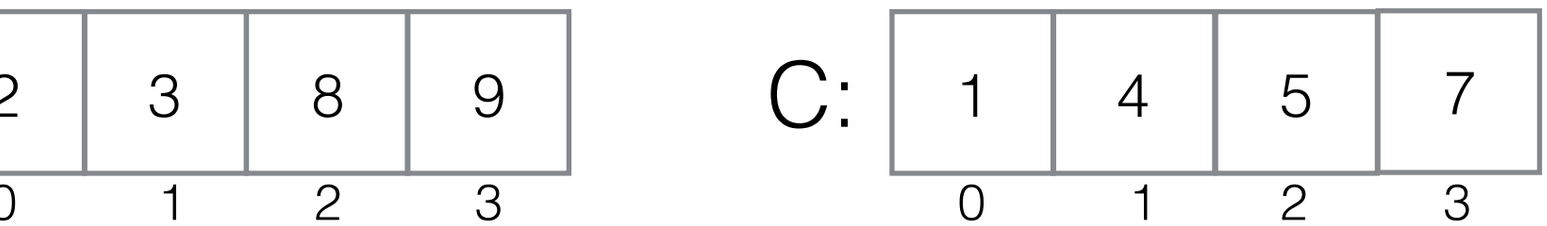
p: 4

q: 4



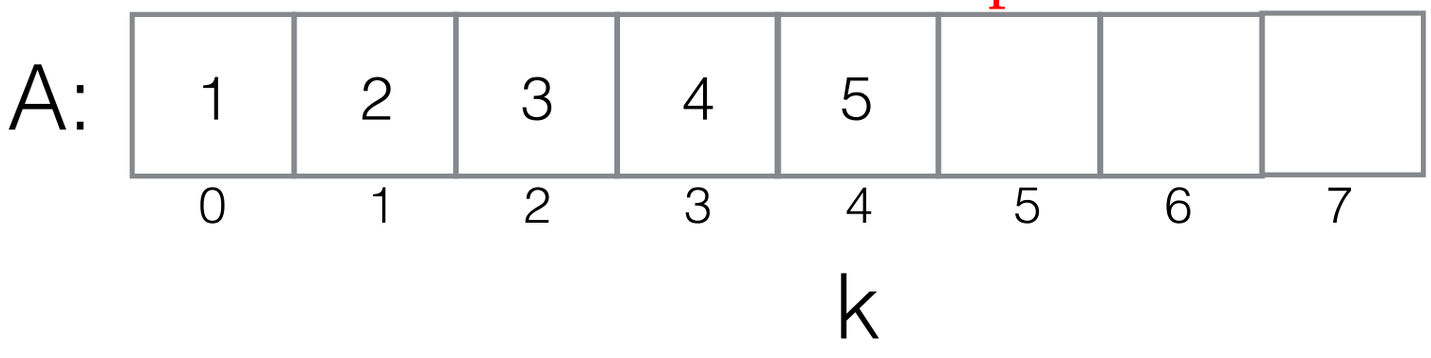
Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE



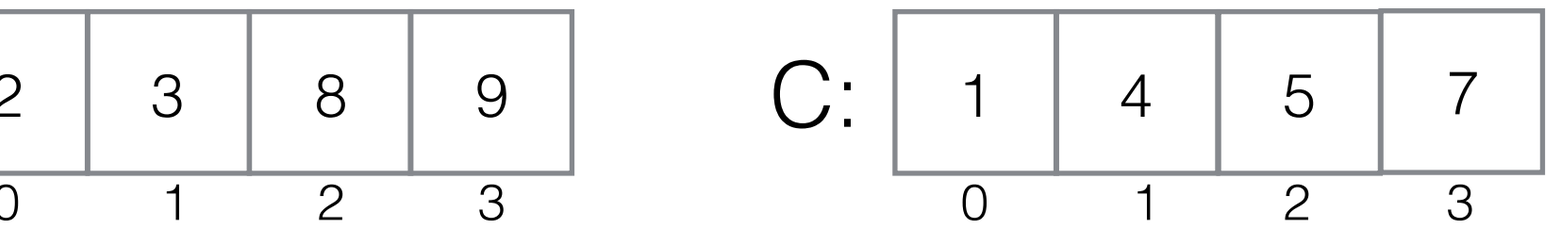
p: 4

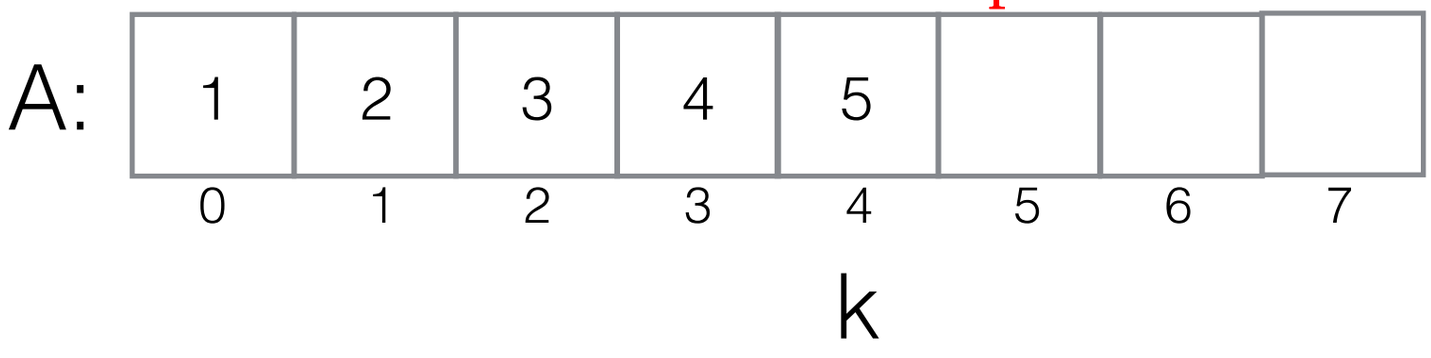
q: 4



Mergesort: Merging Arrays

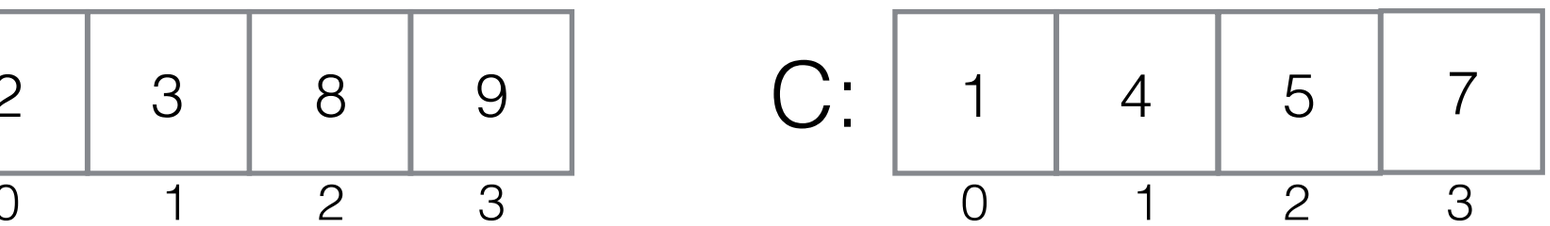
THE UNIVERSITY OF MELBOURNE

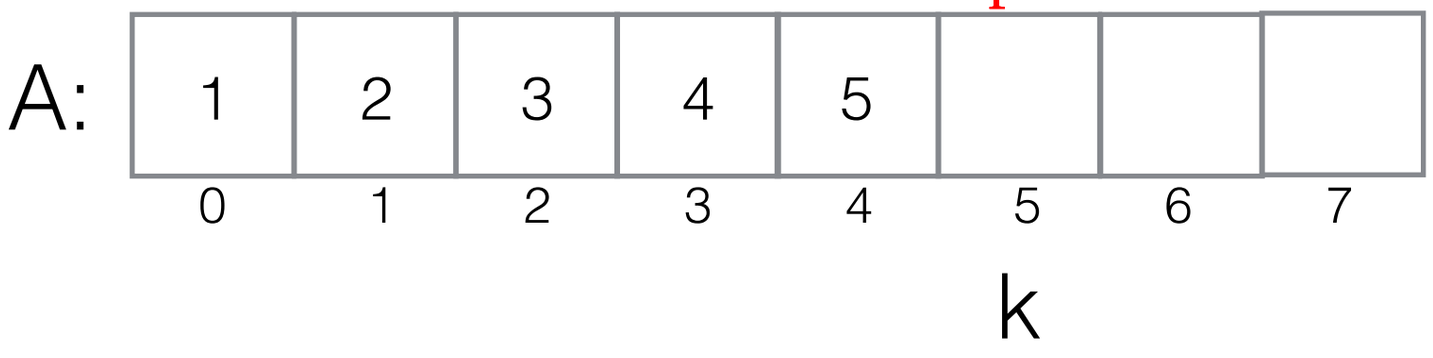




Mergesort: Merging Arrays

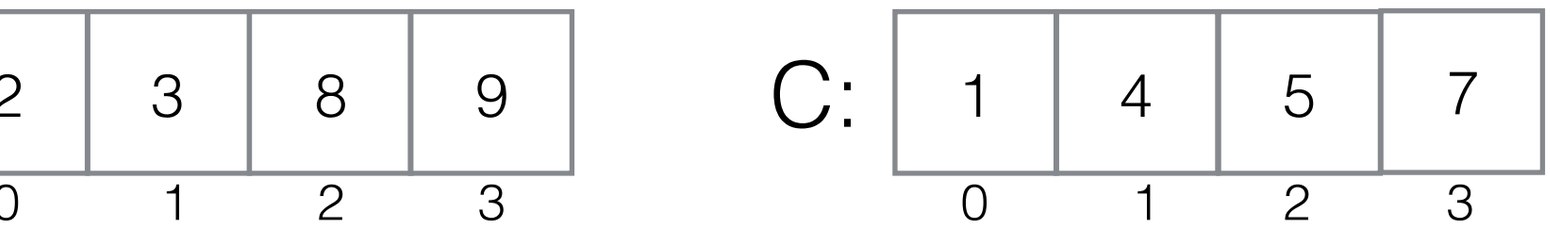
THE UNIVERSITY OF MELBOURNE

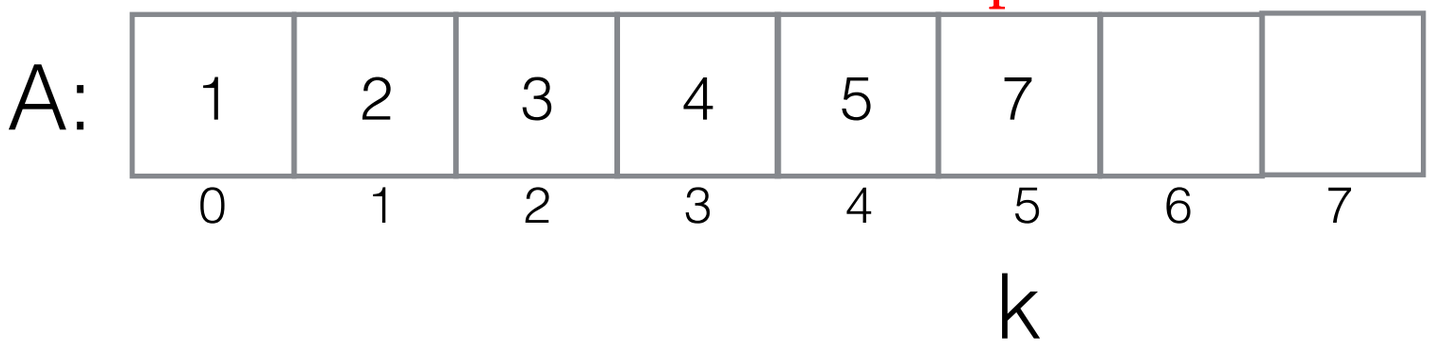




Mergesort: Merging Arrays

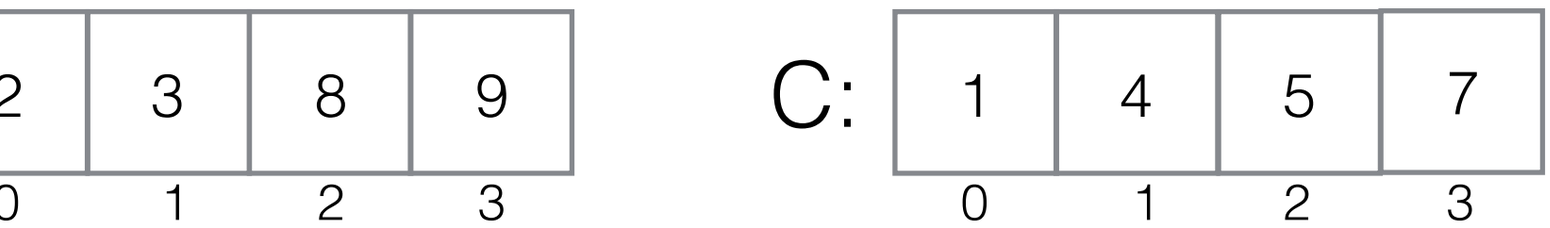
THE UNIVERSITY OF MELBOURNE

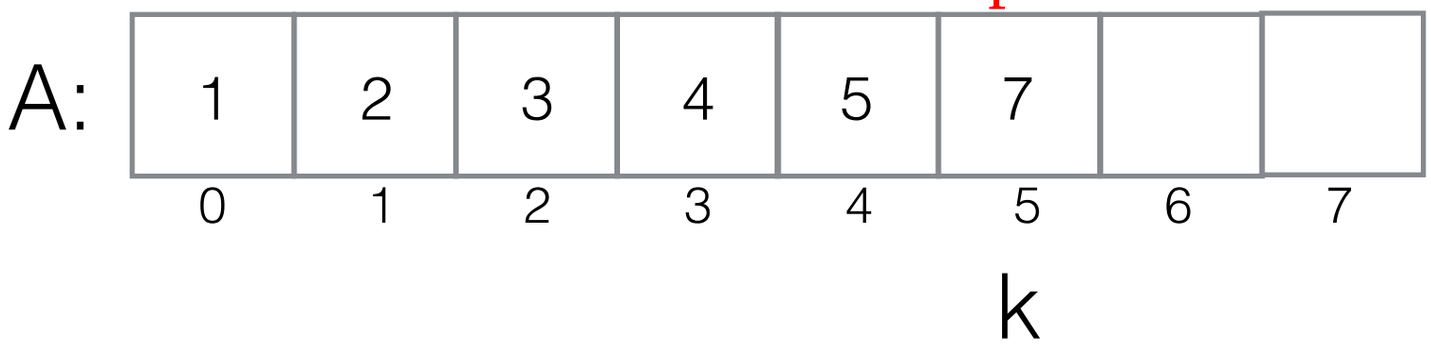




Mergesort: Merging Arrays

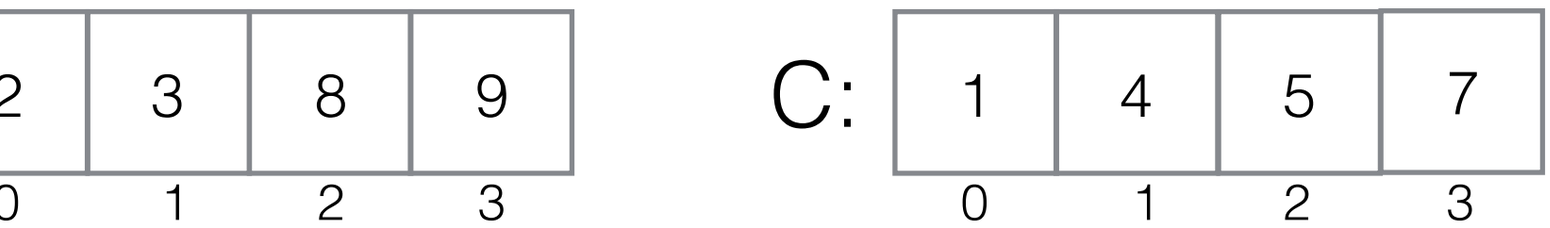
THE UNIVERSITY OF MELBOURNE

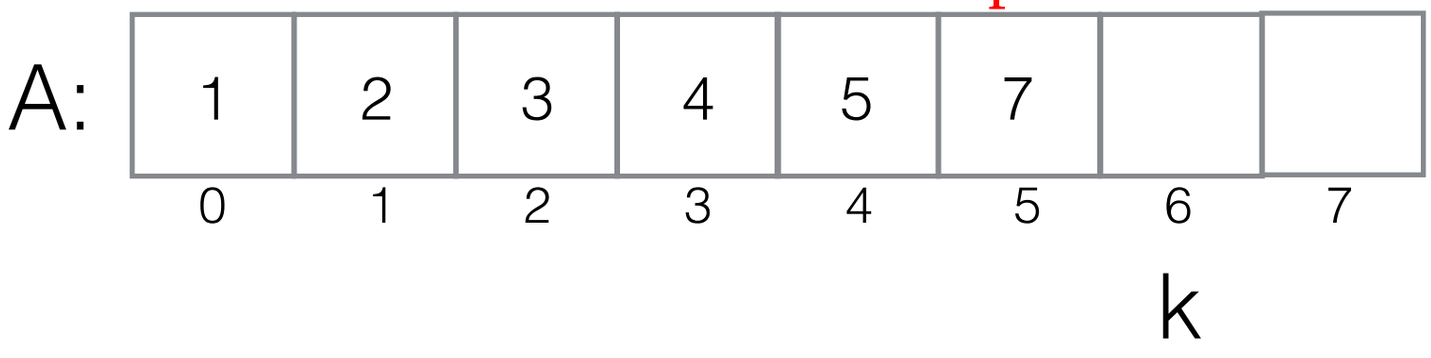




Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE





Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE

procedure MERGE($B[\cdot], p, C[\cdot], q, A[\cdot]$)

$i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$

while $i < p$ and $j < q$ **do**

if $B[i] \leq C[j]$ **then**

$A[k] \leftarrow B[i]$

$i \leftarrow i + 1$

else

$A[k] \leftarrow C[j]$

$j \leftarrow j + 1$

$k \leftarrow k + 1$

if $i = p$ **then**

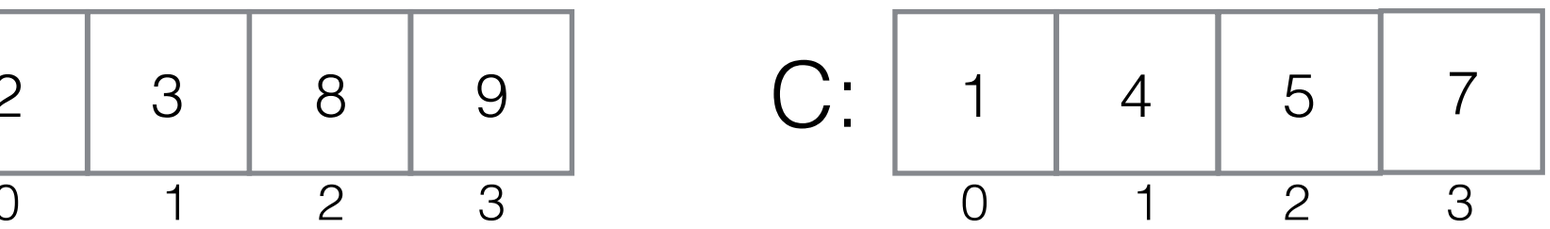
copy $C[j]..C[q-1]$ to $A[k]..A[p+q-1]$ ▷ (a for loop)

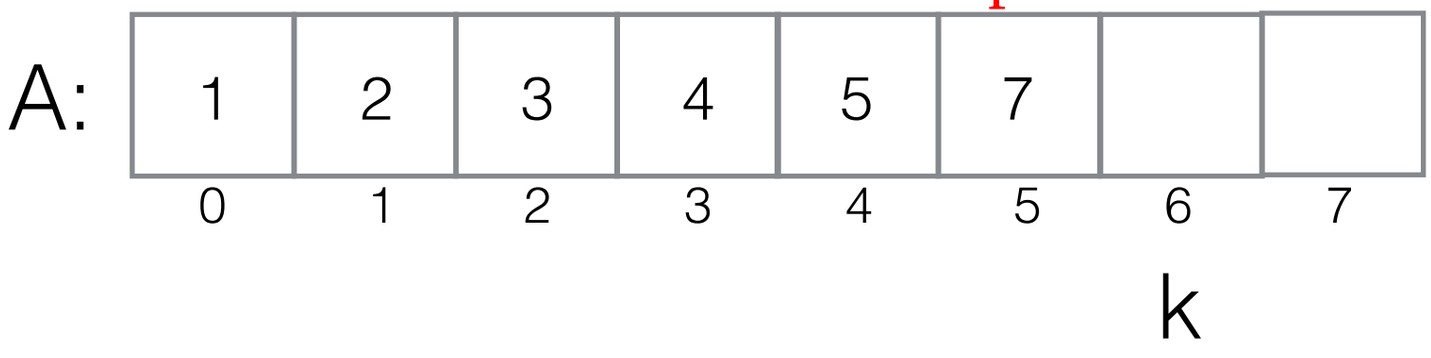
else

copy $B[i]..B[p-1]$ to $A[k]..A[p+q-1]$ ▷ (a for loop)

Mergesort: Merging Arrays

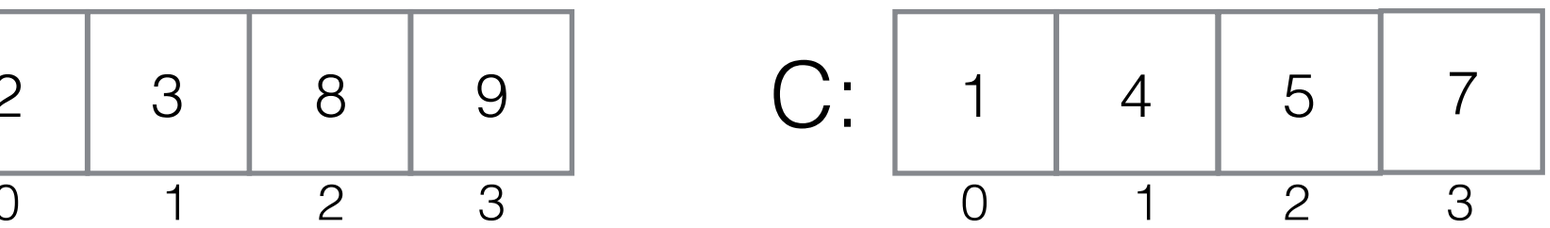
THE UNIVERSITY OF MELBOURNE

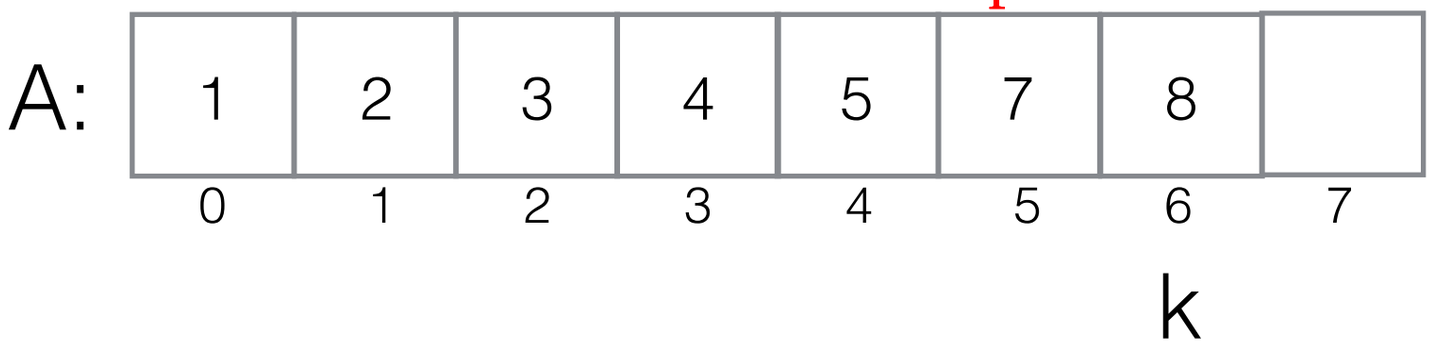




Mergesort: Merging Arrays

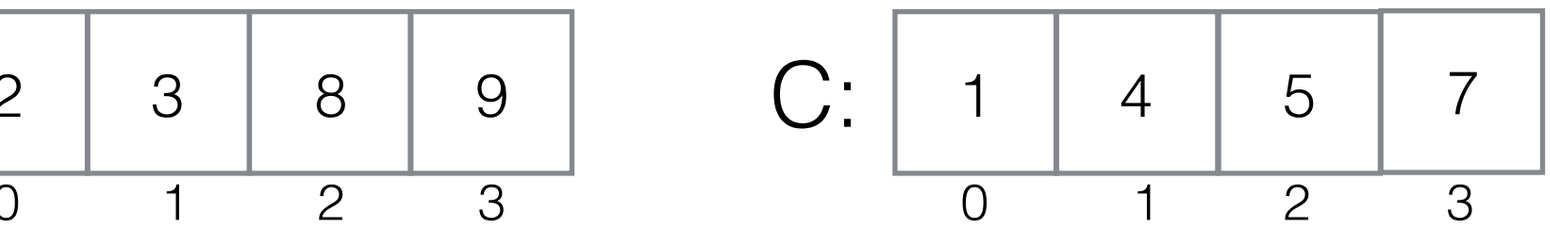
THE UNIVERSITY OF MELBOURNE

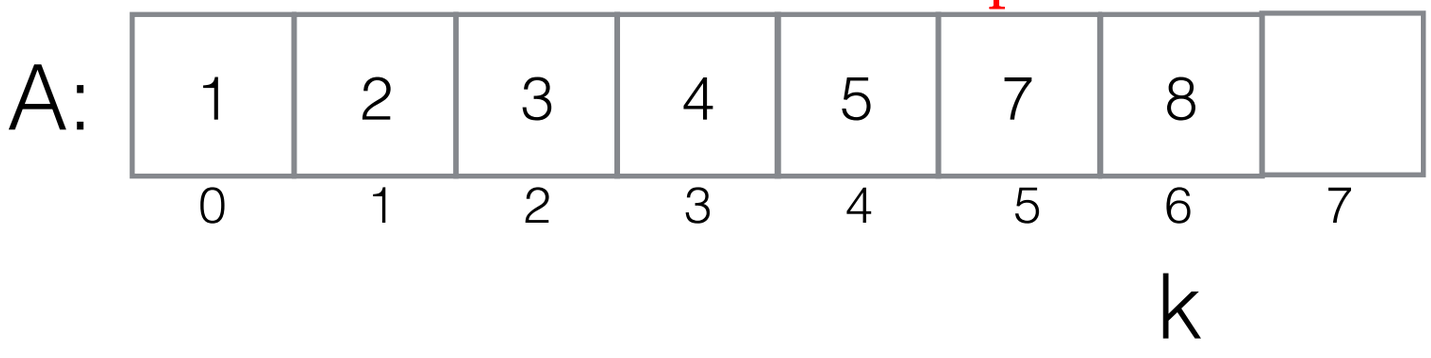




Mergesort: Merging Arrays

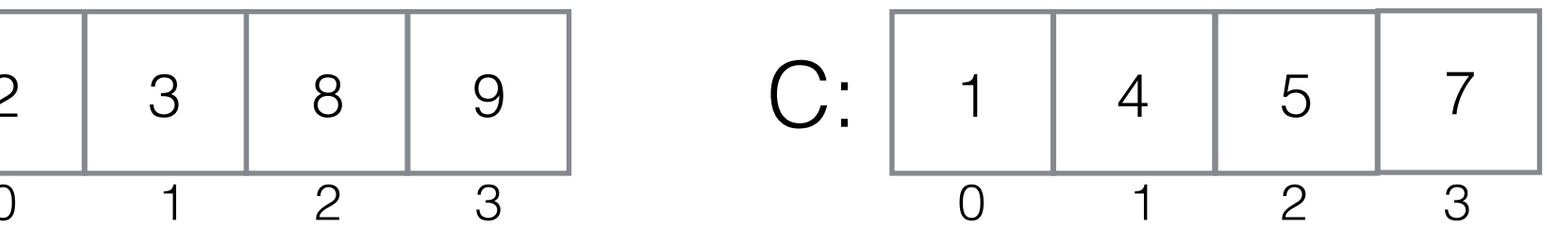
THE UNIVERSITY OF MELBOURNE





Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE

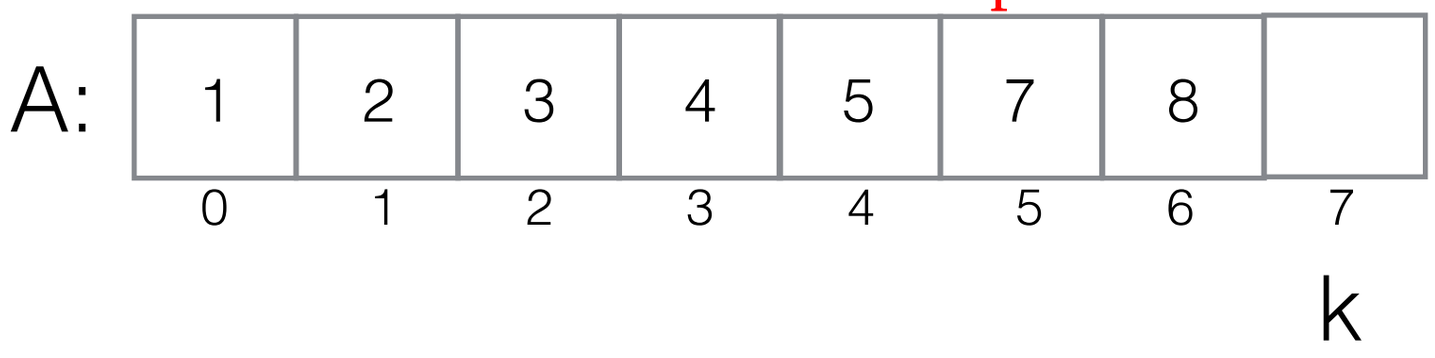


i

j

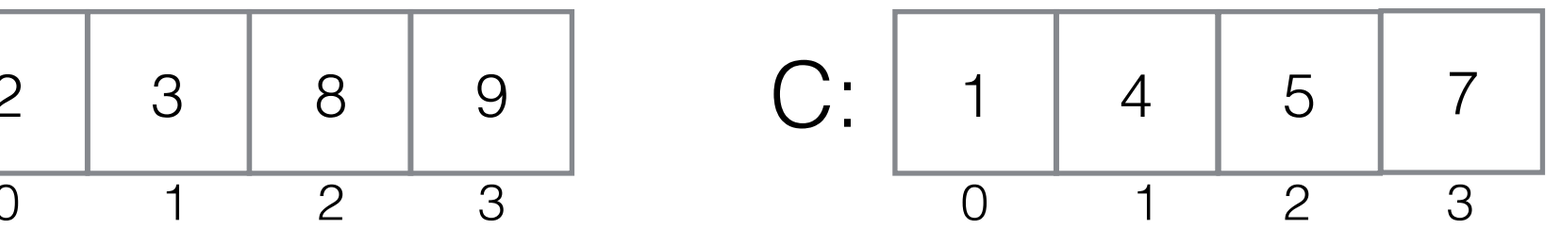
p: 4

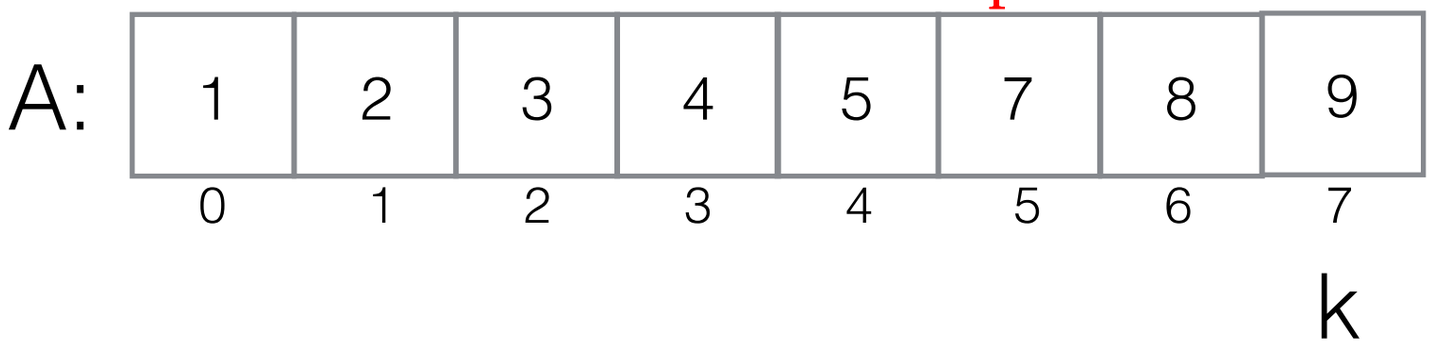
q: 4



Mergesort: Merging Arrays

THE UNIVERSITY OF MELBOURNE





Mergesort: Analysis

- How many comparisons will MERGE need to make in the worst case, when given arrays of size ⌊n/2⌋ and ⌈n/2⌉?

- If the largest and second-largest elements are in different arrays, then n – 1 comparisons. Hence the cost equation for Mergesort is

$$C(n) = \begin{cases} 0 & \text{if } n < 2 \\ 2C(n/2) + n - 1 & \text{otherwise} \end{cases}$$

- By the Master Theorem, C(n) ∈ Θ(n log n).

Mergesort: Properties

THE UNIVERSITY OF MELBOURNE

- For large n, the number of comparisons made tends to be around 75% of the worst-case scenario.

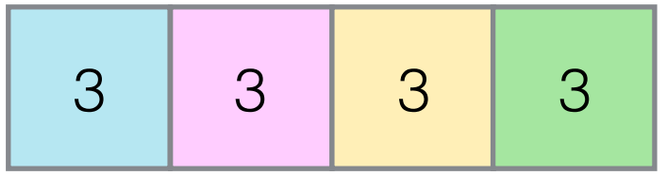
- Is mergesort stable? ?

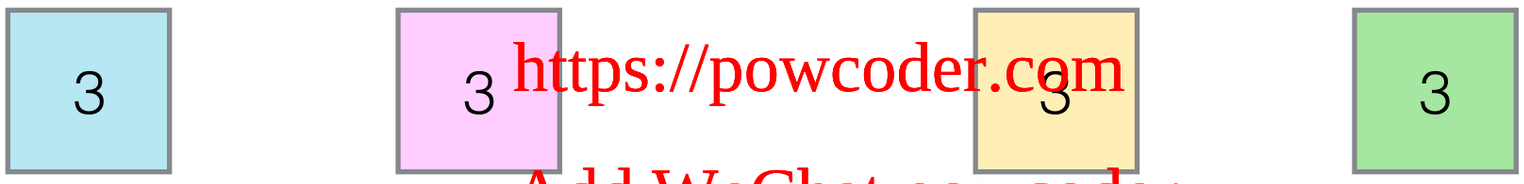
- Is mergesort in-place? no

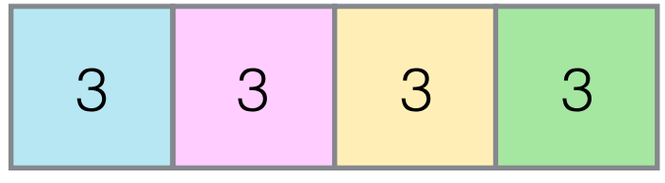
- If comparisons are fast, mergesort ranks between quicksort and heapsort (covered next week) for time, assuming random data.

- Mergesort is the method of choice for linked lists and for very large collections of data.

Mergesort: Stability







Mergesort: Properties

THE UNIVERSITY OF MELBOURNE

- For large n, the number of comparisons made tends to be around 75% of the worst-case scenario.

- Is mergesort stable? yes

- Is mergesort in-place? no

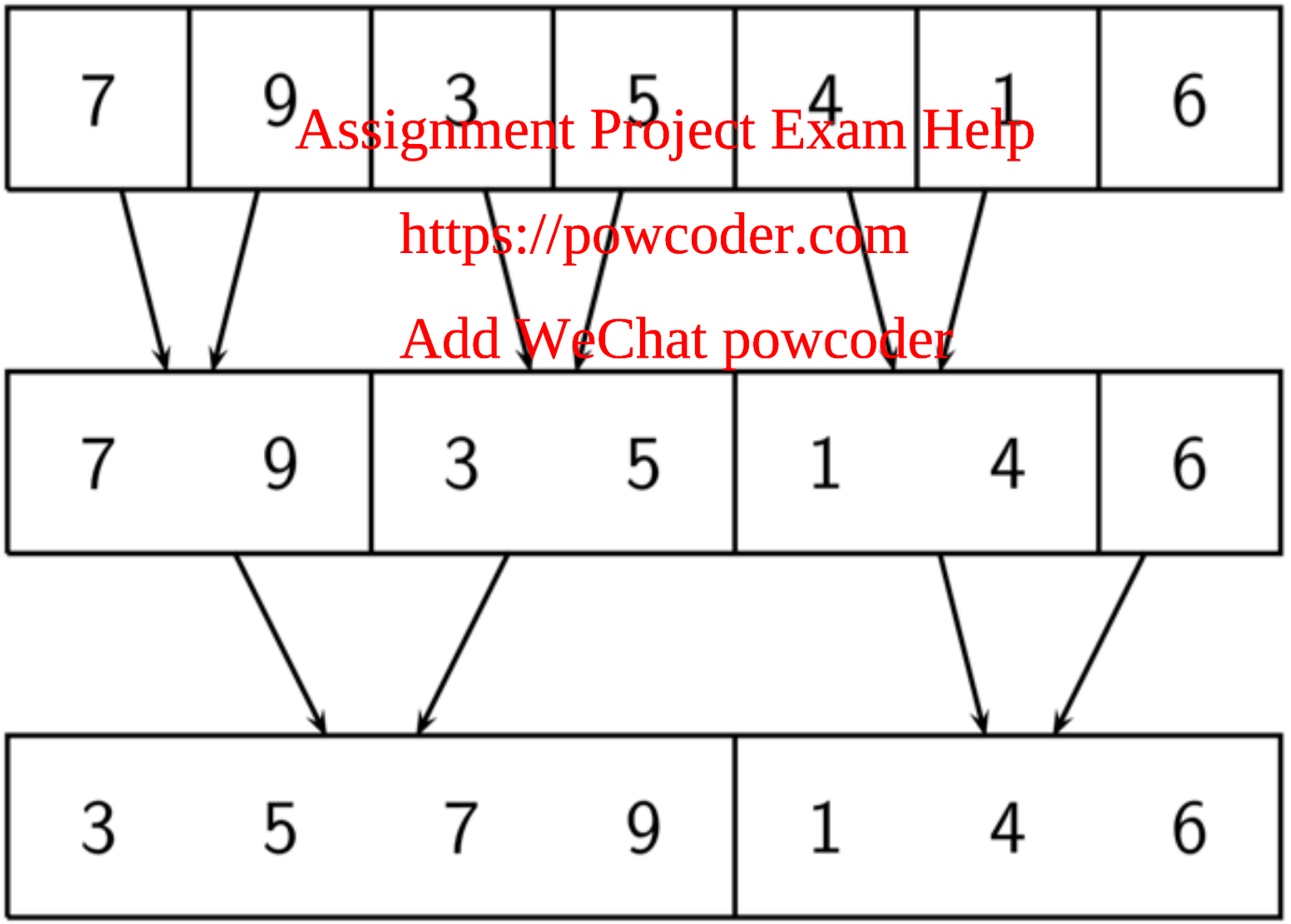
- If comparisons are fast, mergesort ranks between quicksort and heapsort (covered next week) for time, assuming random data.

- Mergesort is the method of choice for linked lists and for very large collections of data.

Bottom-Up Mergesort

- An alternative way of doing mergesort:

- Generate **runs** of length 2, then of length 4, and so on:



Quicksort

- Quicksort takes a divide-and-conquer approach that is different to mergesort's.

- It uses the **partitioning** idea from QuickSelect, picking a pivot element, and partitioning the array around that, so as to obtain this situation:

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are } \geq A[s]}$$

- The element A[s] will be in its final position (it is the (s + 1)th smallest element).

- All that then needs to be done is to sort the segment to the left, recursively, as well as the segment to the right.

Quicksort

- Very short and elegant:

procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s - 1$)
    QUICKSORT($A, s + 1, hi$)

- Initial call: Quicksort(A, 0, n – 1).

Quicksort: Example

THE UNIVERSITY OF MELBOURNE

procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s-1$)
    QUICKSORT($A, s+1, hi$)

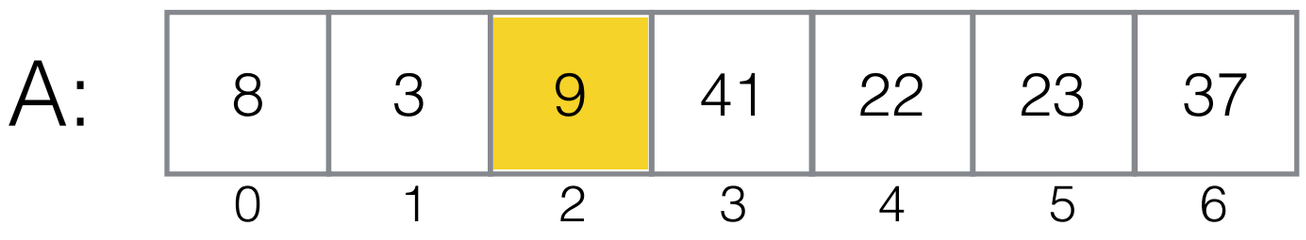
A:

9	23	8	41	22	3	37
0	1	2	3	4	5	6

Quicksort: Example

THE UNIVERSITY OF MELBOURNE

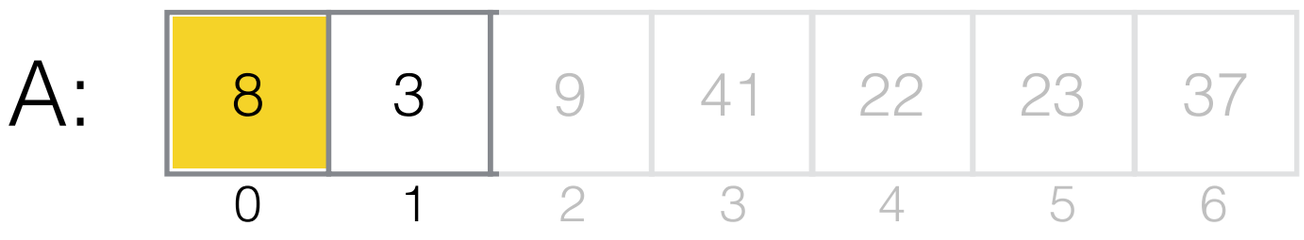
procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s-1$)
    QUICKSORT($A, s+1, hi$)



Quicksort: Example

THE UNIVERSITY OF MELBOURNE

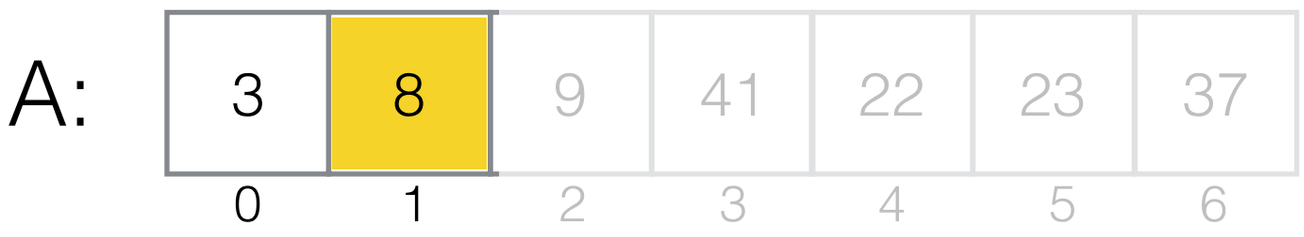
procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s-1$)
    QUICKSORT($A, s+1, hi$)



Quicksort: Example

THE UNIVERSITY OF MELBOURNE

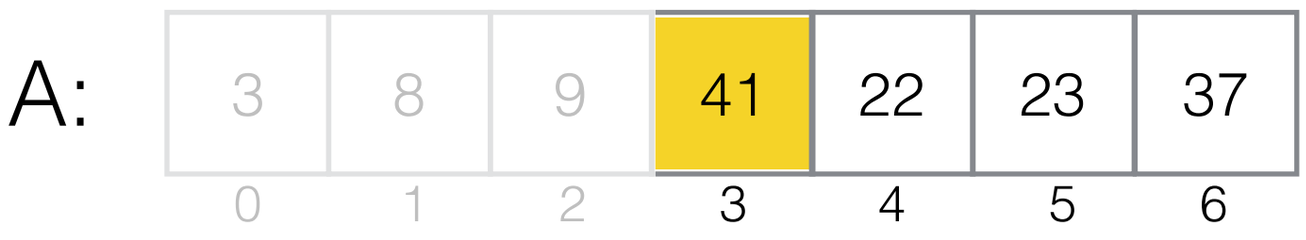
procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s-1$)
    QUICKSORT($A, s+1, hi$)



Quicksort: Example

THE UNIVERSITY OF MELBOURNE

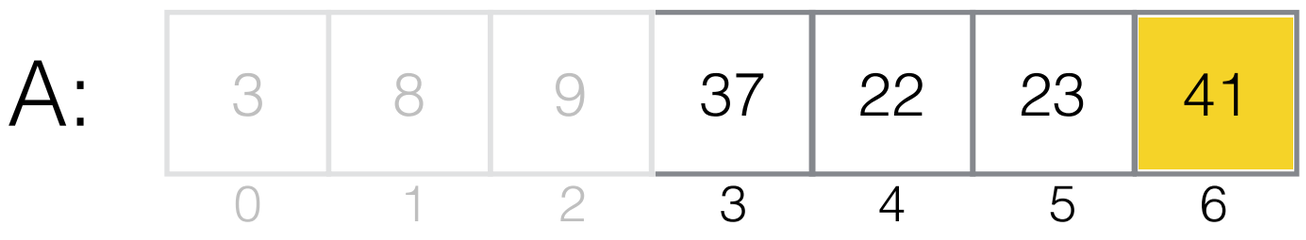
procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s-1$)
    QUICKSORT($A, s+1, hi$)



Quicksort: Example

THE UNIVERSITY OF MELBOURNE

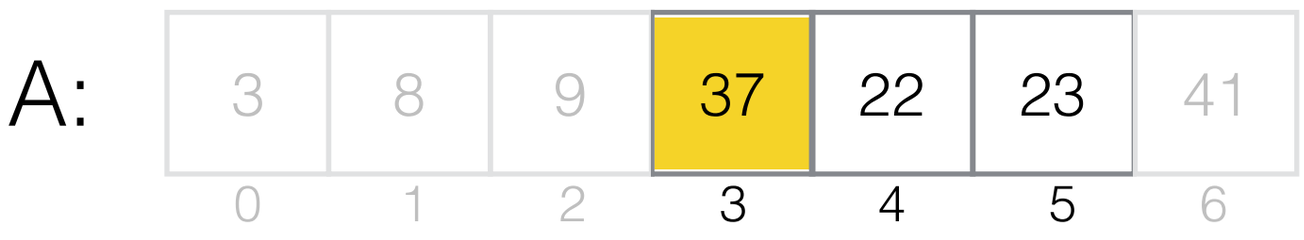
procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s-1$)
    QUICKSORT($A, s+1, hi$)



Quicksort: Example

THE UNIVERSITY OF MELBOURNE

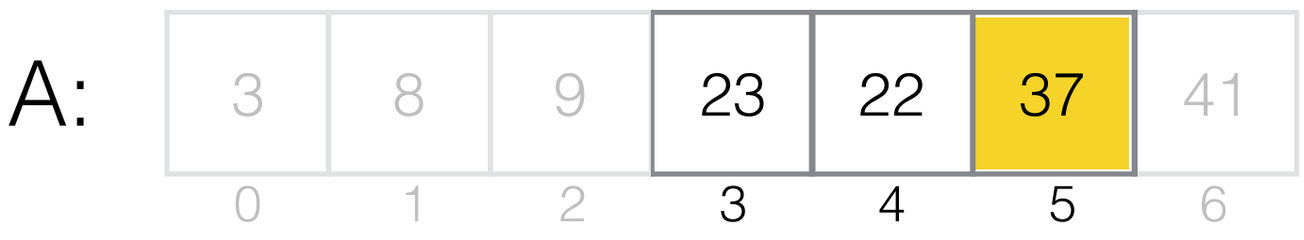
procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s-1$)
    QUICKSORT($A, s+1, hi$)



Quicksort: Example

THE UNIVERSITY OF MELBOURNE

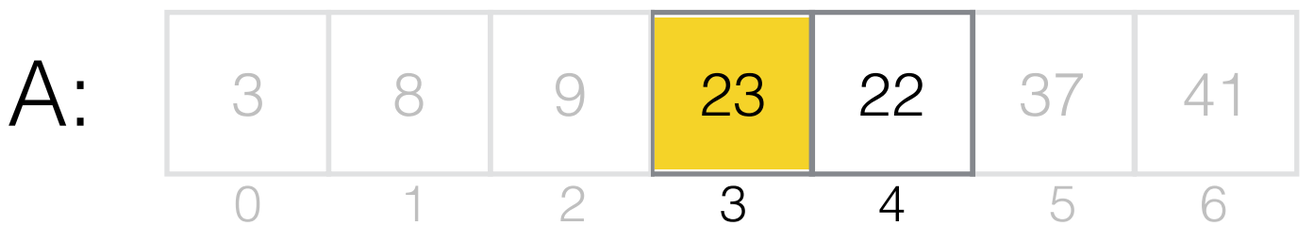
procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s-1$)
    QUICKSORT($A, s+1, hi$)



Quicksort: Example

THE UNIVERSITY OF MELBOURNE

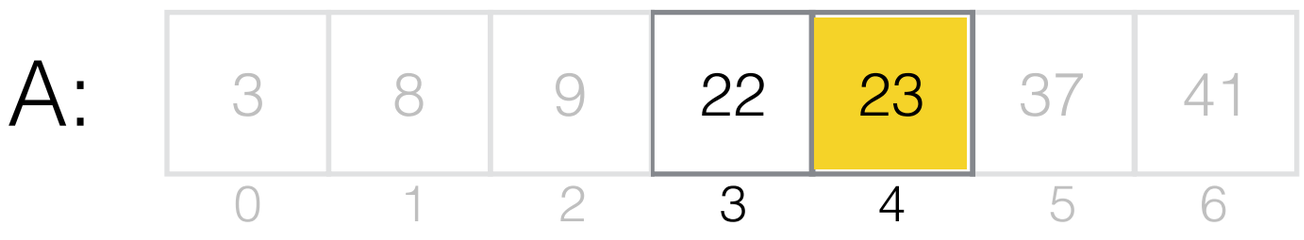
```
procedure QUICKSORT(A[·], lo, hi)
    if lo < hi then
        s ← PARTITION(A, lo, hi)
        QUICKSORT(A, lo, s − 1)
        QUICKSORT(A, s + 1, hi)
```

Quicksort: Example

THE UNIVERSITY OF MELBOURNE

procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s-1$)
    QUICKSORT($A, s+1, hi$)



Quicksort: Example

THE UNIVERSITY OF MELBOURNE

procedure QUICKSORT($A[\cdot], lo, hi$)
  if $lo < hi$ **then**
    $s \leftarrow$ PARTITION(A, lo, hi)
    QUICKSORT($A, lo, s-1$)
    QUICKSORT($A, s+1, hi$)

A:

3	8	9	22	23	37	41
0	1	2	3	4	5	6

Hoare Partitioning

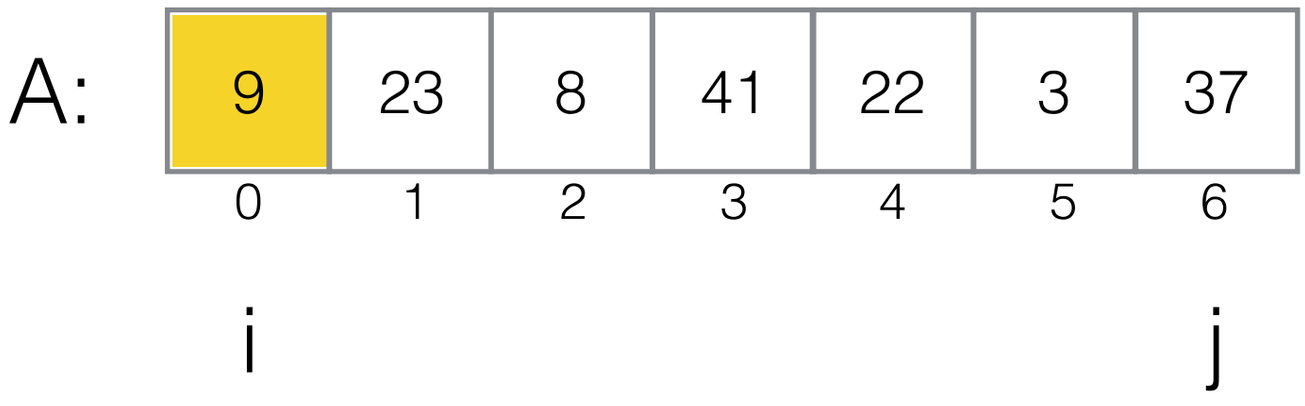
- The standard way of doing partitioning in Quicksort

```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] > p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])                    ▷ Undo the last swap
    swap(A[lo], A[j])                   ▷ Bring pivot to its correct position
    return j
```


Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

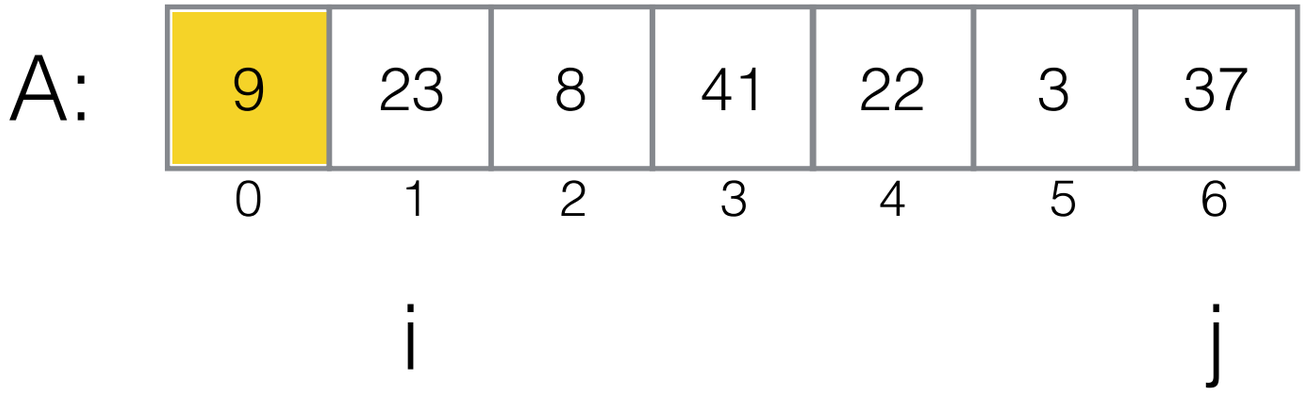
```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] ≥ p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

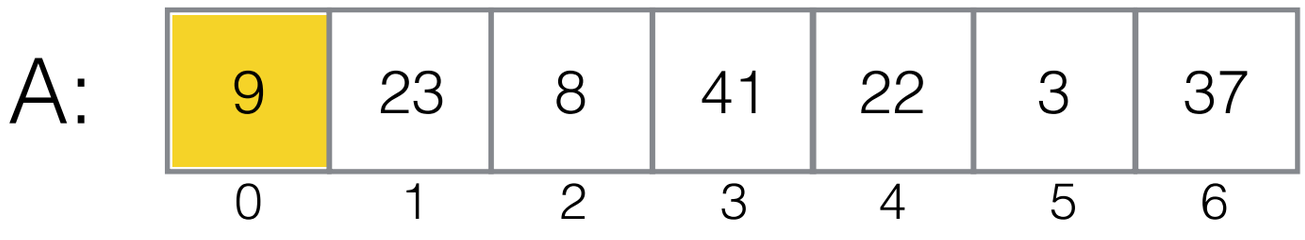
```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] ≥ p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] ≥ p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

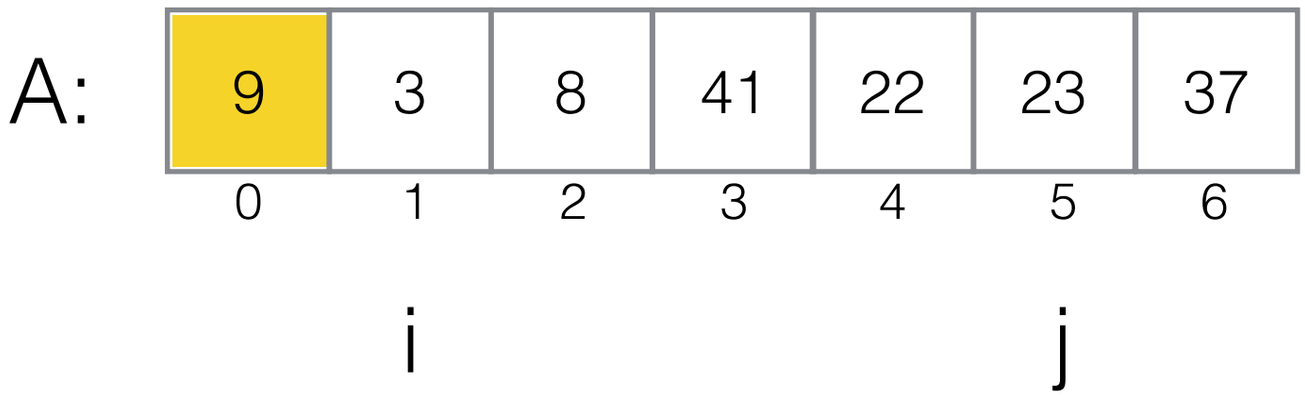



p: 9

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

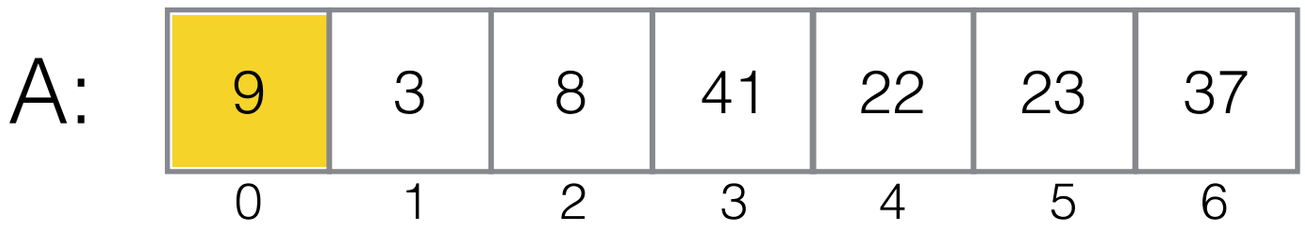
```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] > p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] ≥ p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

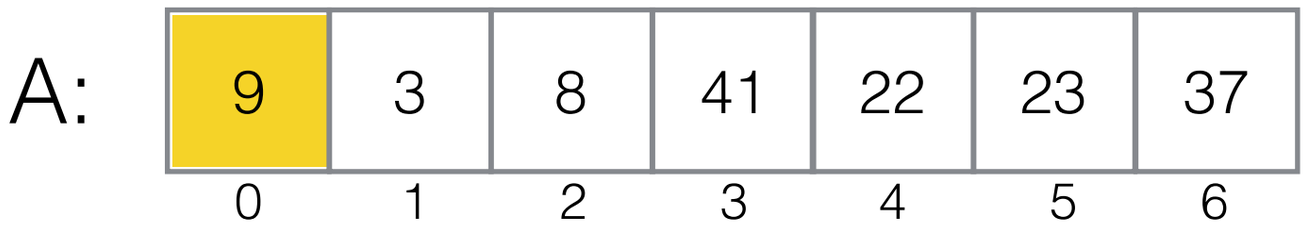



p: 9

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

function PARTITION($A[\cdot], lo, hi$)
  $p \leftarrow A[lo]$; $i \leftarrow lo$; $j \leftarrow hi$
  repeat
    while $i < hi$ and $A[i] \leq p$ **do** $i \leftarrow i + 1$
    while $j \geq lo$ and $A[j] > p$ **do** $j \leftarrow j - 1$
    $swap(A[i], A[j])$
  until $i \geq j$
  $swap(A[i], A[j])$
  $swap(A[lo], A[j])$
  return j

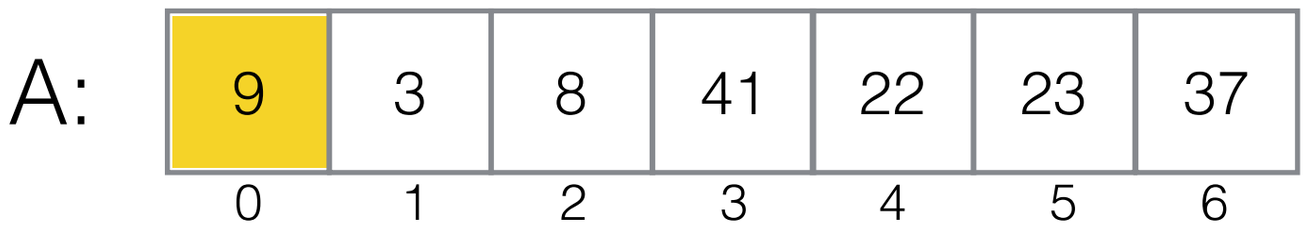


p: 9

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] > p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

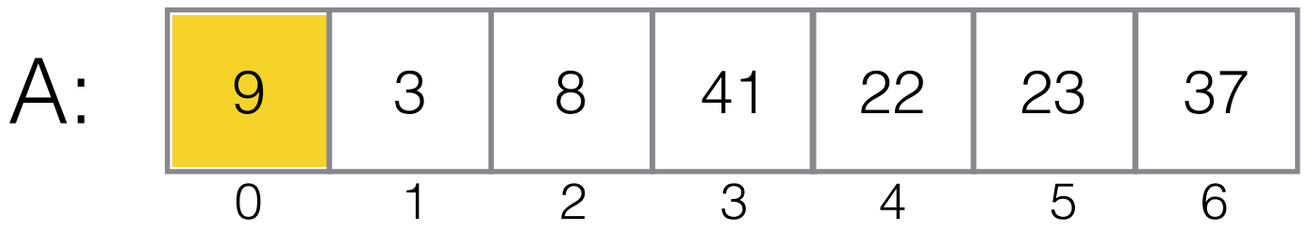



p: 9

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

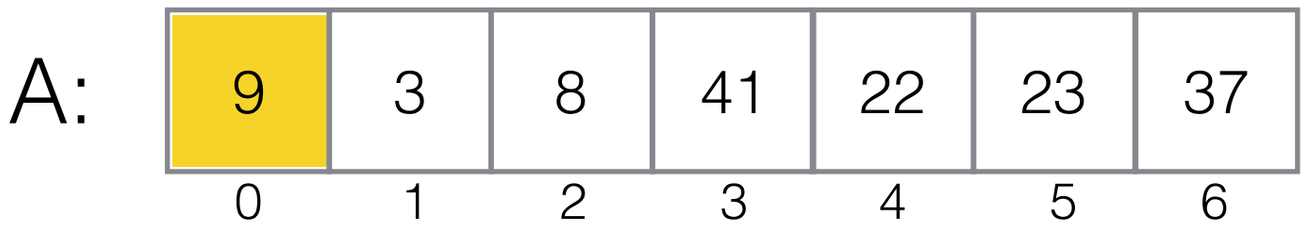
```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] ≥ p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

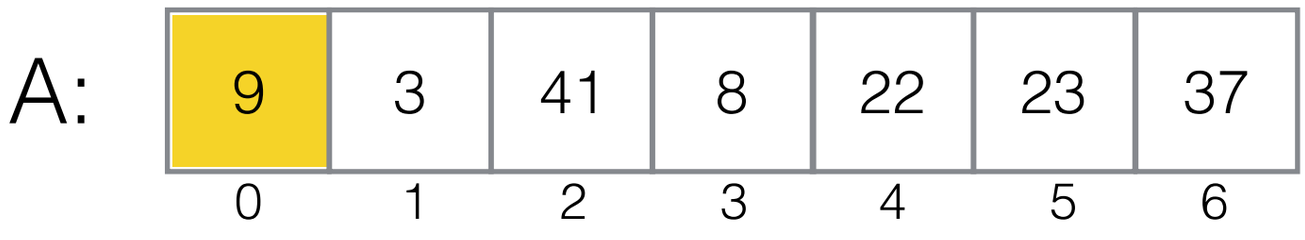
```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] > p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] ≥ p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

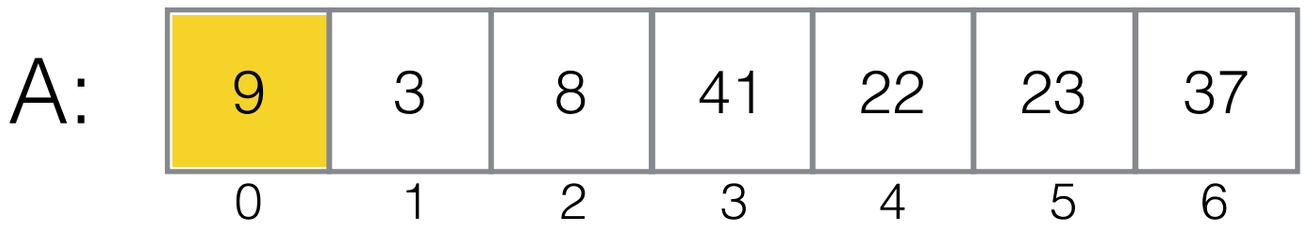



p: 9

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

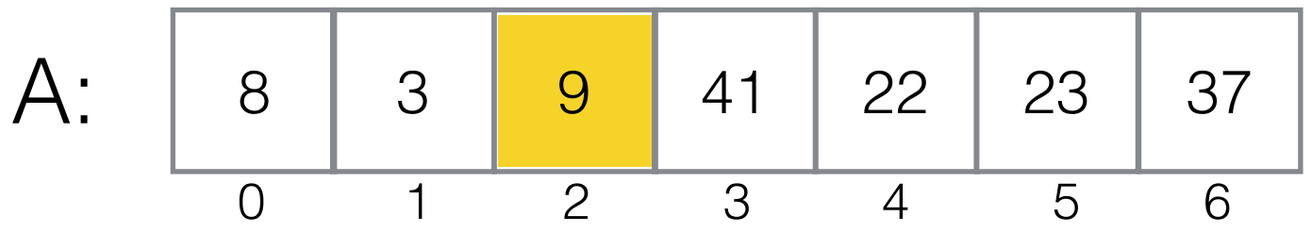
```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] ≥ p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

Hoare Partitioning

THE UNIVERSITY OF MELBOURNE

```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] ≥ p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])
    swap(A[lo], A[j])
    return j
```

Quicksort Analysis: Best Case Analysis

THE UNIVERSITY OF MELBOURNE

- The best case happens when the pivot is the median; that results in two sub-tasks of equal size.

$$C_{best}(n) = \begin{cases} 0 & \text{if } n < 2 \\ 2C_{best}(n/2) + n & \text{otherwise} \end{cases}$$

The 'n' is for the n key comparisons performed by Partition.

- By the Master Theorem, $C_{best}(n) \in \Theta(n \log n)$, just as for mergesort, so quicksort's best case is (asymptotically) no better than mergesort's worst case.

Quicksort Worst Case

THE UNIVERSITY OF MELBOURNE

A:

Quicksort Analysis: Worst Case Analysis

- The worst case happens if the array is already sorted.

- In that case, we don't really have divide-and-conquer, because each recursive call deals with a problem size that has only been decremented by 1:

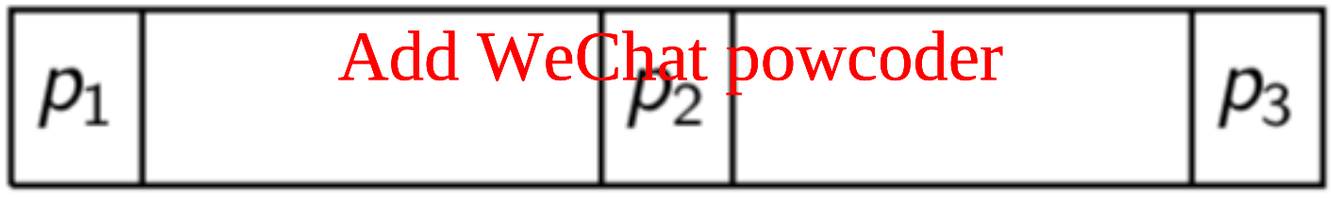
$$C_{worst}(n) = \begin{cases} 0 & \text{if } n < 2 \\ C_{worst}(n-1) + n & \text{otherwise} \end{cases}$$

- That is, $C_{worst}(n) = n + (n - 1) + \cdots + 3 + 2 \in \Theta(n^2)$.

Quicksort Improvements: Median-of-Three

- It would be better if the pivot was chosen randomly.

- A cheap and useful approximation to this is to take the median of three candidates, A[lo], A[hi], and A[⌊(lo + hi)/2⌋].



- Reorganise the three elements so that p_1 is the median, and p_3 is the largest of the three.

- Now run quicksort as before.

Quicksort Improvements: Median-of-Three

THE UNIVERSITY OF MELBOURNE

- In fact, with median-of-three, we can have a much faster version than before, simplifying tests in the innermost loops:

function PARTITION($A[\cdot], lo, hi$)
  $p \leftarrow A[lo]$; $i \leftarrow lo$; $j \leftarrow hi + 1$
  repeat
    ~~**while** $i < hi$ and $A[i] < p$ **do** $i \leftarrow i + 1$~~
    repeat $i \leftarrow i + 1$ **until** $A[i] \geq p$
    ~~**while** $j \geq lo$ and $A[j] > p$ **do** $j \leftarrow j - 1$~~
    repeat $j \leftarrow j - 1$ **until** $A[j] \leq p$
    $swap(A[i], A[j])$
  until $i \geq j$
  $swap(A[i], A[j])$
  $swap(A[lo], A[j])$
  return j

Quicksort Improvements: Early Cut-Off

- A second useful improvement is to stop quicksort early and switch to insertion sort. This is easily implemented:

procedure SORT($A[\cdot], n$)
    QUICKALMOSTSORT($A, 0, n-1$)
    INSERTIONSORT(A, n)

procedure QUICKALMOSTSORT($A[\cdot], lo, hi$)
    if $lo + 10 < hi$ **then**
        $s \leftarrow$ PARTITION(A, lo, hi)
        QUICKALMOSTSORT($A, lo, s-1$)
        QUICKALMOSTSORT($A, s+1, hi$)

Quicksort Properties

- With these (and other) improvements, quicksort is considered the best available sorting method for arrays of random data.

- A major reason for its speed is the very tight inner loop in PARTITION.

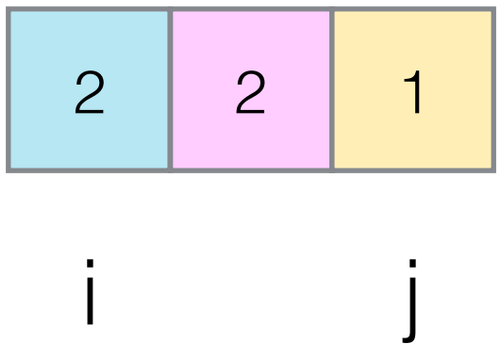
- Although mergesort has a better performance guarantee, quicksort is faster on average.

- In the best case, we get $\lceil \log_2 n \rceil$ recursive levels. It can be shown that on random data, the expected number is $2 \log_e n \approx 1.38 \log_2 n$. So quicksort's average behaviour is very close to the best-case behaviour.

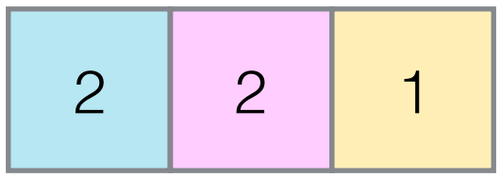
- Is quicksort stable? ?

- Is it in-place? yes

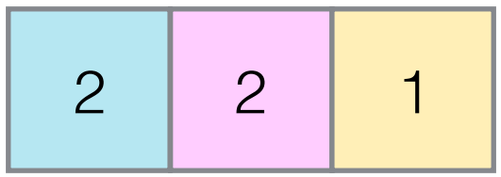
Quicksort Stability



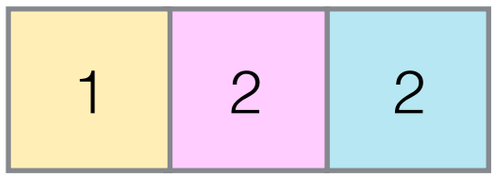
Quicksort Stability



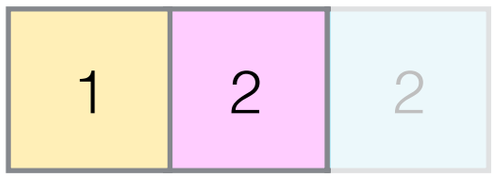
Quicksort Stability



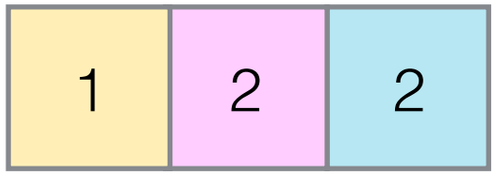
Quicksort Stability



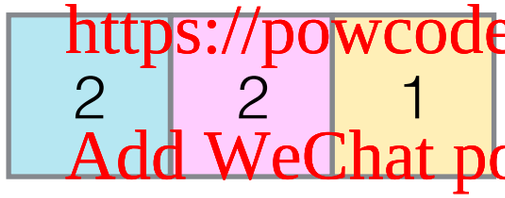
Quicksort Stability



Quicksort Stability



This is where we finished



This is where we started

Not stable

Quicksort Properties

THE UNIVERSITY OF MELBOURNE

- With these (and other) improvements, quicksort is considered the best available sorting method for arrays of random data.

- A major reason for its speed is the very tight inner loop in PARTITION.

- Although mergesort has a better performance guarantee, quicksort is faster on average.

- In the best case, we get $\lceil \log_2 n \rceil$ recursive levels. It can be shown that on random data, the expected number is $2 \log_e n \approx 1.38 \log_2 n$. So quicksort's average behaviour is very close to the best-case behaviour.

- Is quicksort stable? no

- Is it in-place? yes

Next up

- Tree traversal methods, plus we apply the divide-and-conquer technique to the closest-pair problem.

