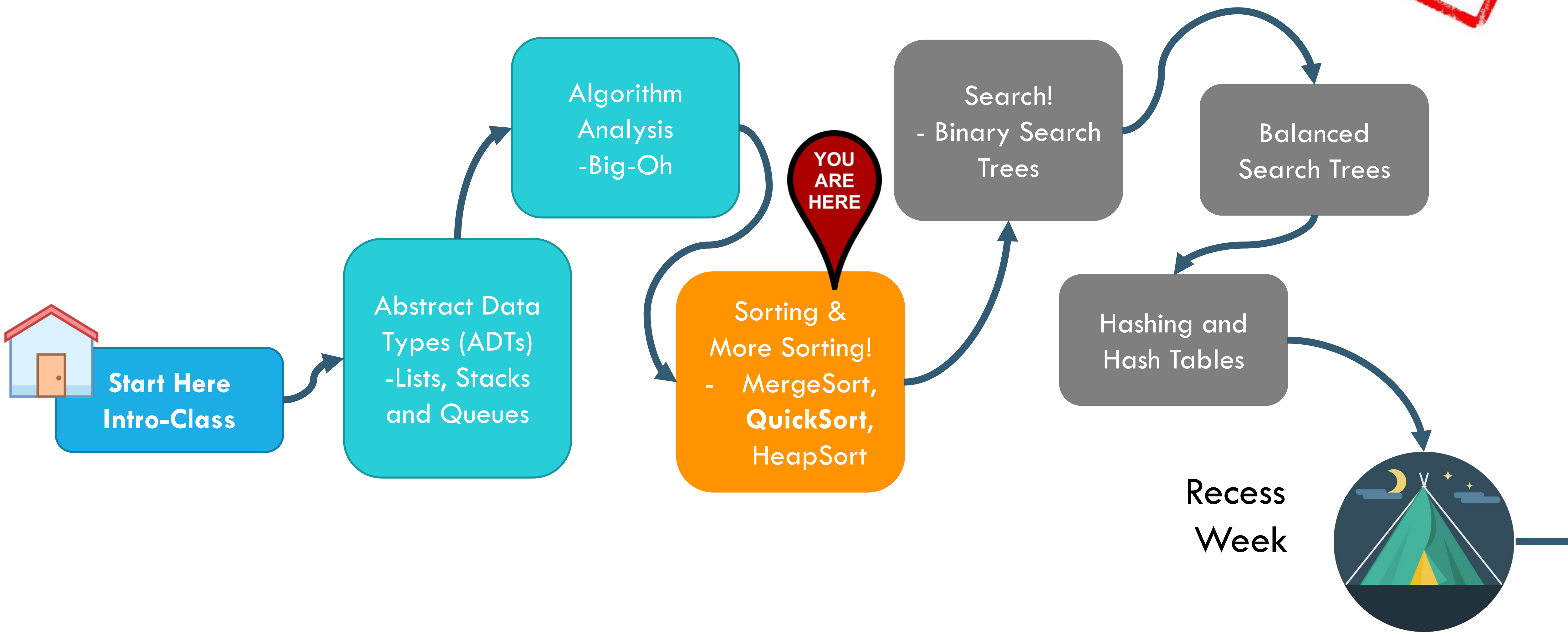
LECTURE 5: QUICKSORT

Harold Soh
harold@comp.nus.edu.sg

PATH TO MASTERY / COURSE STRUCTURE

DRAFT



QUIZ 1

On **Sept 4th**

Please don't be late.

Covers everything up to **Quicksort** (today's lecture).

COURSE FEEDBACK!

Your feedback is **important**!

- *Love the class?*
- *Hate the class?*
- *Meh the class?*

Tell us:

https://forms.gle/9XYMGsWr2d98CzwQ7

Only 5 questions.

QUESTIONS BEFORE WE GET STARTED?

LEARNING OUTCOMES

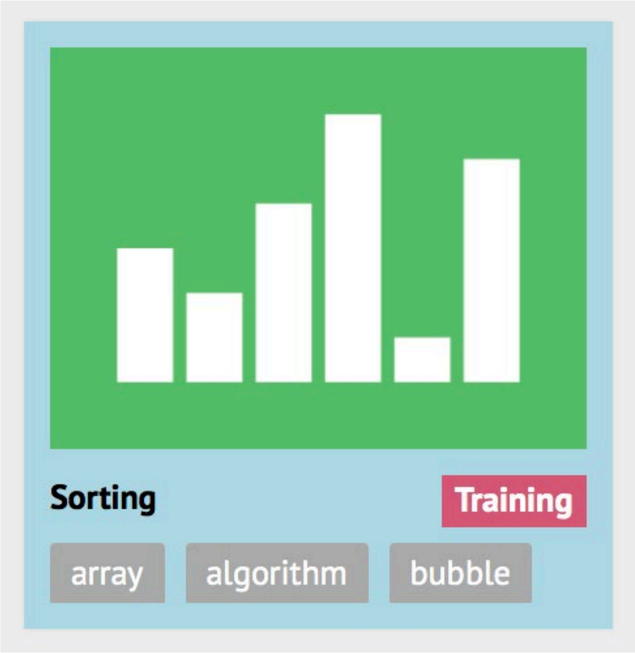
By the end of this session, you should be able to:

- **Describe** the **quicksort** algorithm and how it works.
- **Analyze the worst-case and average-case performance** of the quicksort algorithm

Poll Everywhere

https://bit.ly/2LvG9bq

DID YOU DO YOUR HOMEWORK?



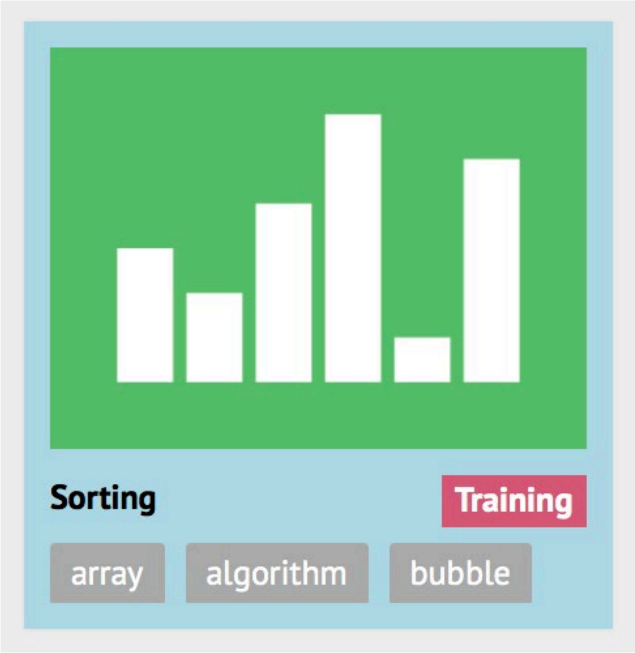
Did you revise the quicksort material on Visualgo?

A. Yes! I'm a champion!
B. No... I got a bit lazy.
C. umm... kind of half way...
D. there was homework?

Poll Everywhere

https://bit.ly/2LvG9bq

DID YOU DO YOUR HOMEWORK?



Did you revise the quicksort material on Visualgo?

A. **Yes! I'm a champion!**
B. No... I got a bit lazy.
C. umm... kind of half way...
D. there was homework?

PROBLEM: CUSTOMER LOYALTY REWARDS

Get a list of customers ordered by their purchasing spend. Reward the n who spent the most.

BOGOSORT

```
while items is not sorted
   permute(items)
```

INSERTION SORT

```
mark first element as sorted
for each unsorted element X
  'extract' the element X
  for j = lastSortedIndex down to 0
    if current element j > X
      move sorted element to the right by 1
    break loop and insert X here
```

MERGESORT

```
function MergeSort(A, low, high)
    if low < high
        mid = (high + low)/2
        MergeSort(A, low, mid)
        MergeSort(A, mid+1, high)
        Merge(A, low, mid, high)
```

Company Mascot

MERGESORT

```
function MergeSort(A, low, high)
    if low < high
        mid = (high + low)/2
        MergeSort(A, low, mid)
        MergeSort(A, mid+1, high)
        Merge(A, low, mid, high)
```

invented by John von Neumann in 1945

QUOTES ABOUT JOHN VON NEUMANN

"*I have sometimes wondered whether a brain like von Neumann's does not indicate a species superior to that of man*"

– Hans Bethe (Nobel Laureate)

"*Keeping up with him was … impossible. The feeling was you were on a tricycle chasing a racing car.*"

– Israel Halperin (Mathematician)

"*von Neumann would carry on a conversation with my 3-year-old son, and the two of them would talk as equals, and I sometimes wondered if he used the same principle when he talked to the rest of us*"

– Edward Teller (Physicist)

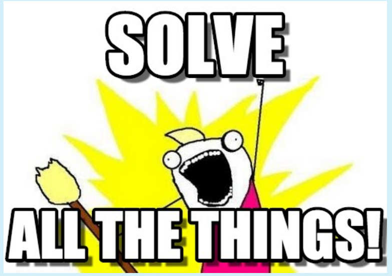
MERGESORT IS …

Mergesort is a nice example of which of the following strategies?

A

B

C

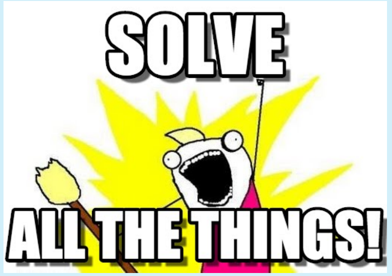


MERGESORT IS …

Mergesort is a nice example of which of the following strategies?

A 

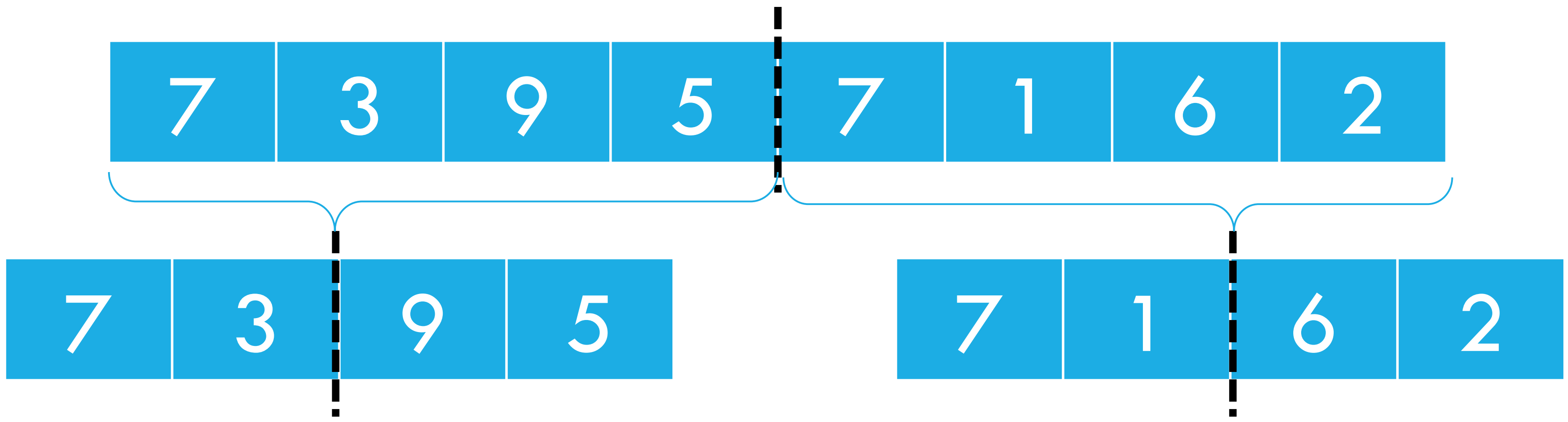
B 

C 

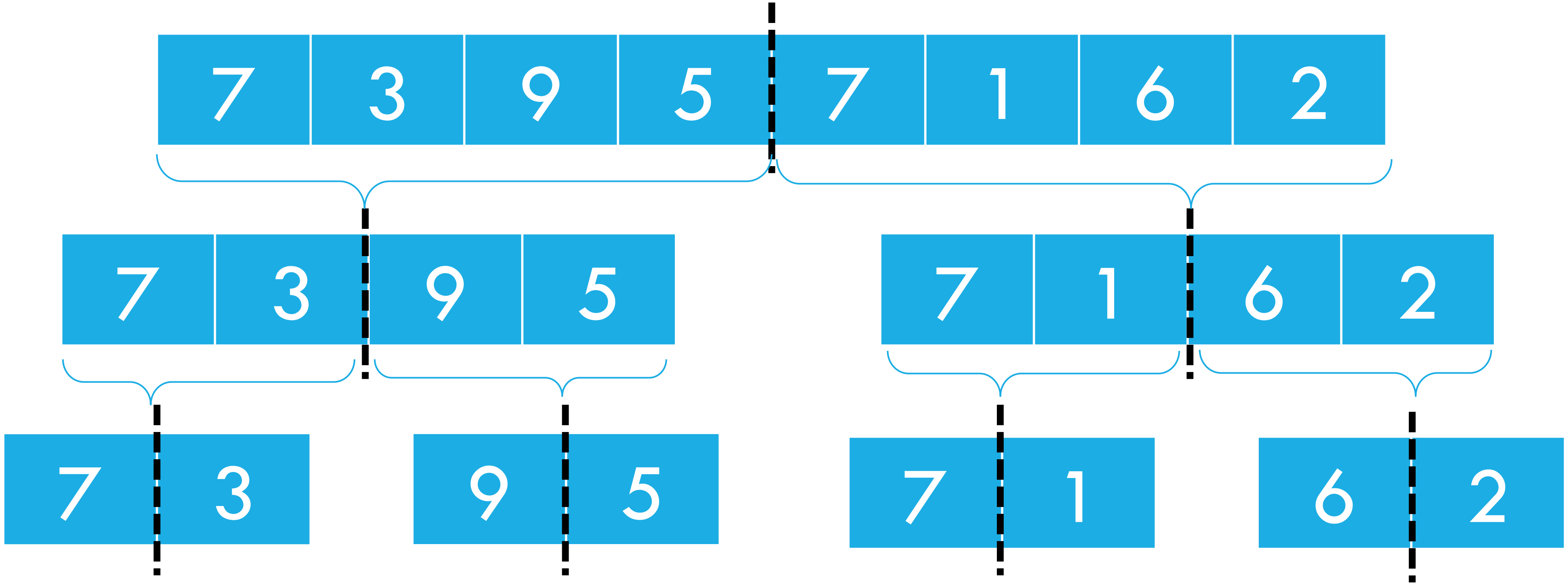
MERGESORT: RECURSE “DOWNWARDS”

7	3	9	5	7	1	6	2

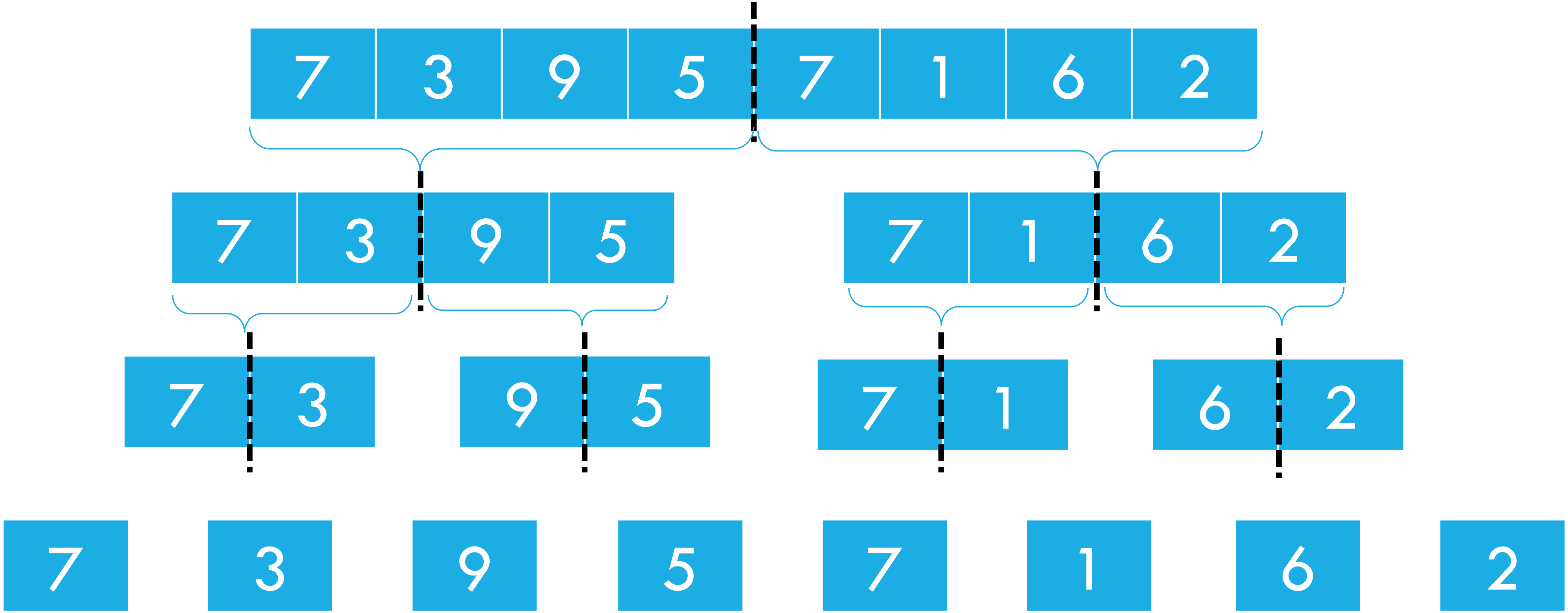
MERGESORT: RECURSE "DOWNWARDS"



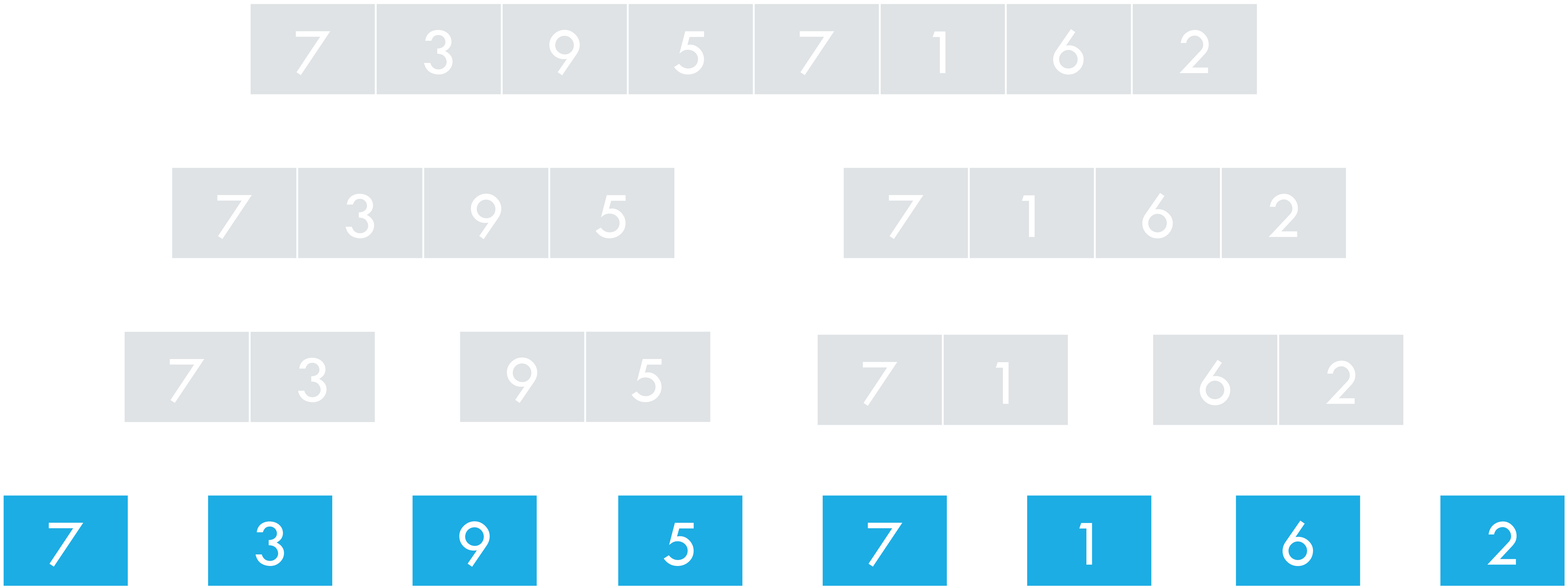
MERGESORT: RECURSE “DOWNWARDS”



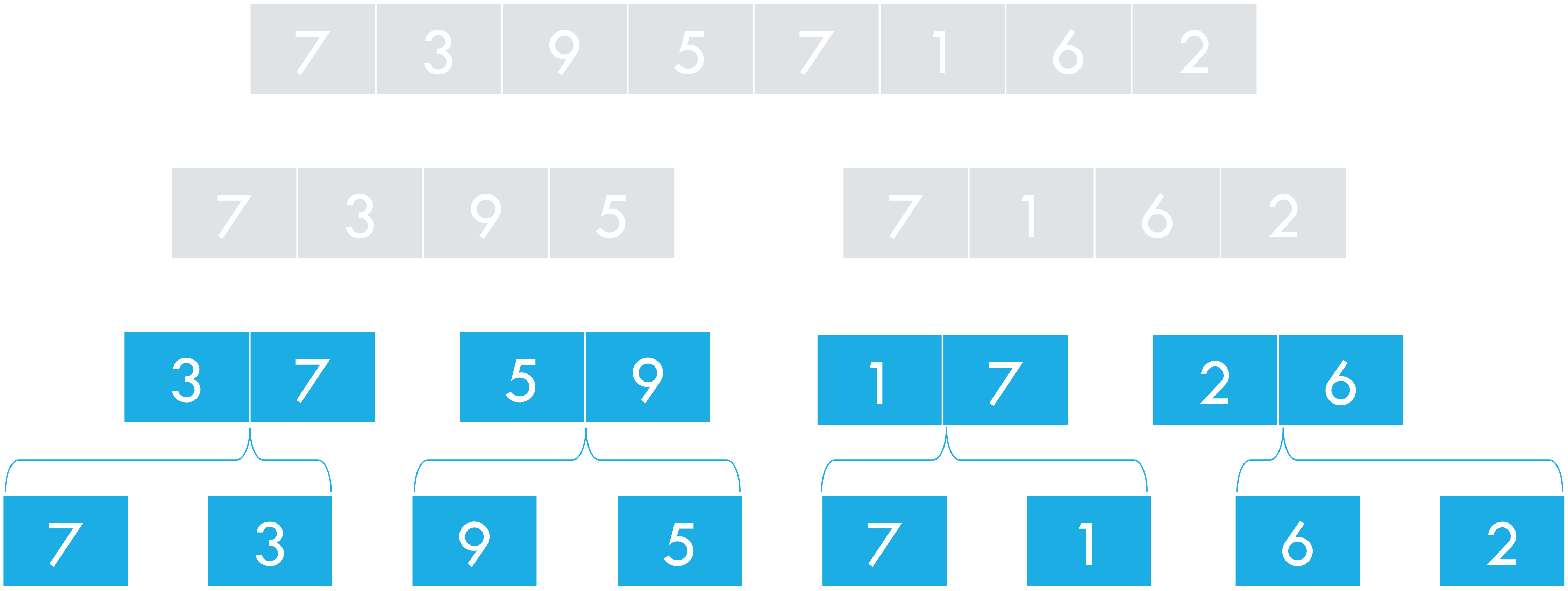
MERGESORT: RECURSE “DOWNWARDS”



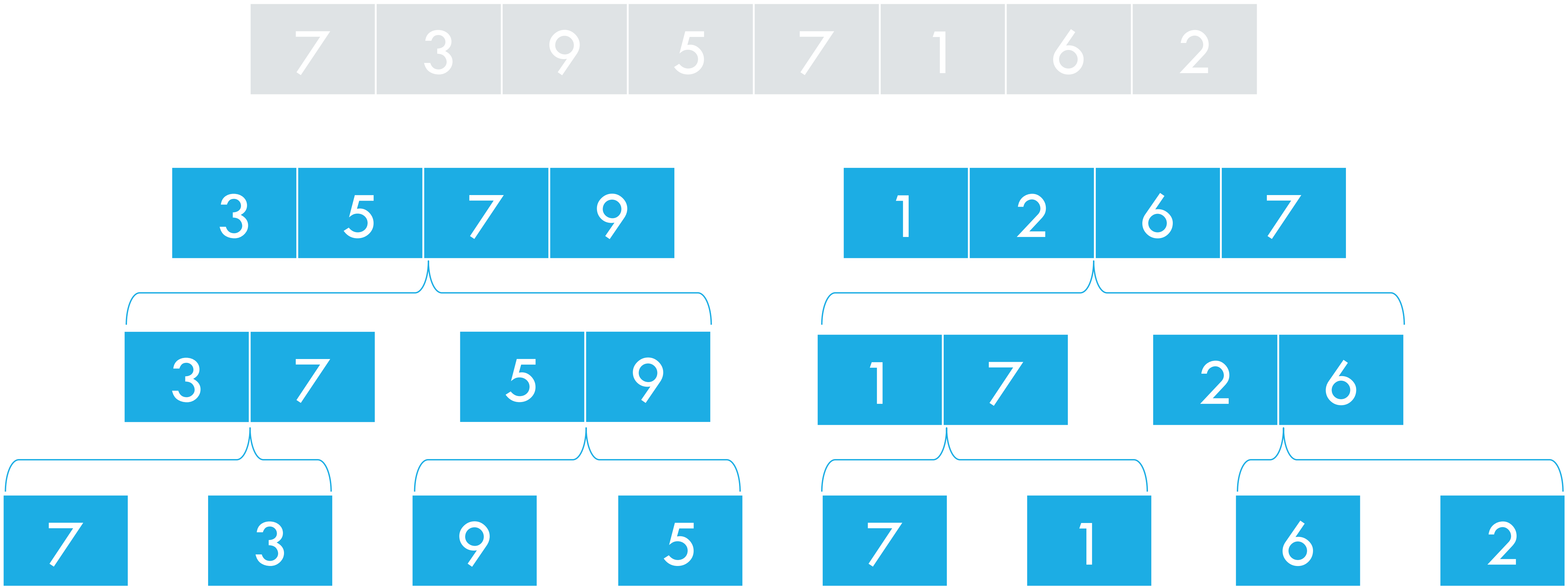
MERGESORT: MERGING “UPWARDS”



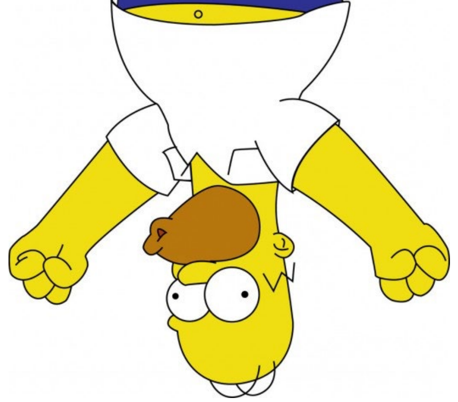
MERGESORT: MERGING "UPWARDS"

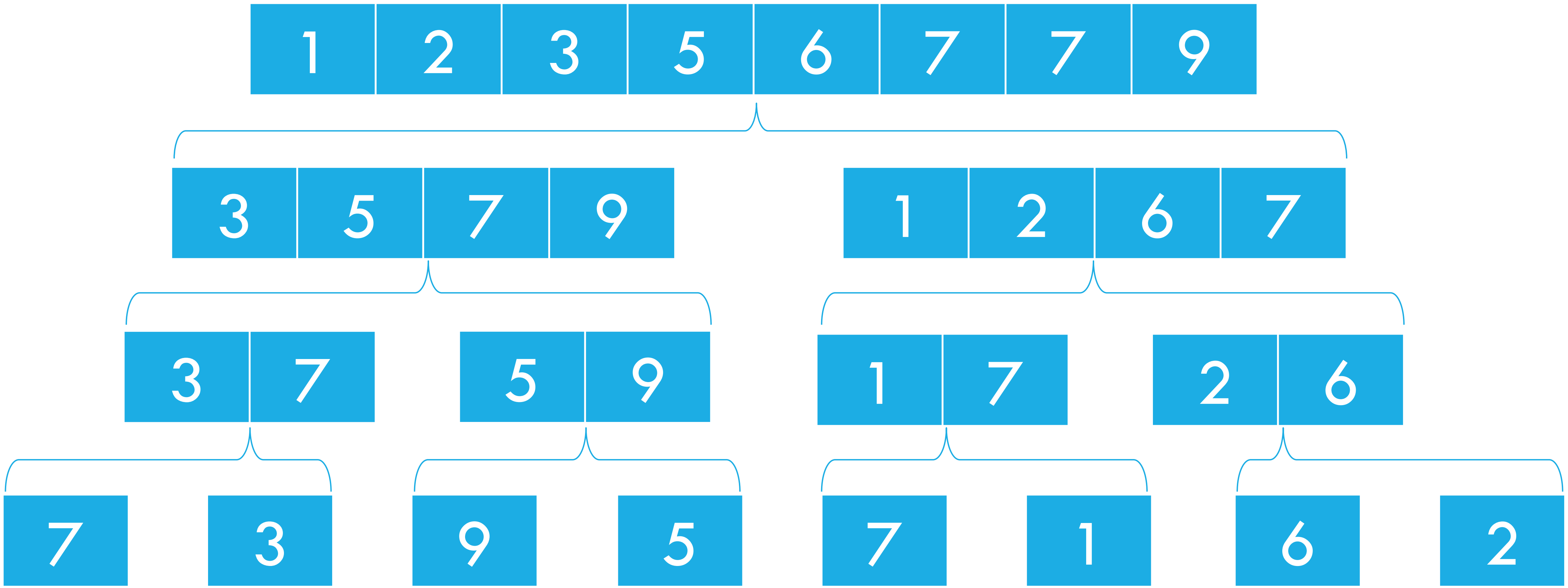


MERGESORT: MERGING "UPWARDS"



MERGESORT: MERGING “UPWARDS”





MERGESORT: SIMPLE SPLIT, CLEVER COMBINE

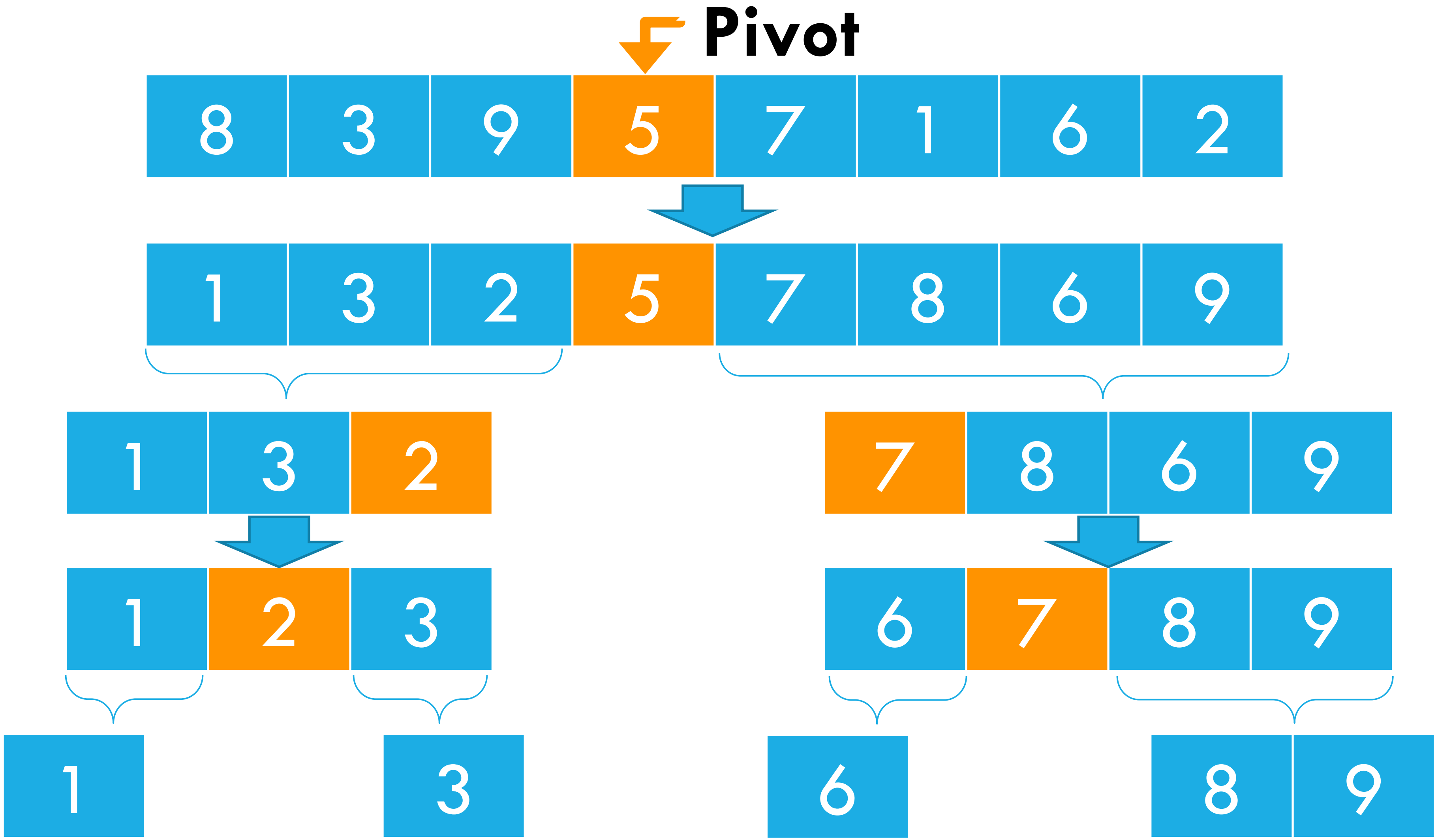
```
function MergeSort(A, low, high)
   if low < high
      mid = (high + low)/2
      MergeSort(A, low, mid)
      MergeSort(A, mid+1, high)
      Merge(A, low, mid, high)
```

Simple Split

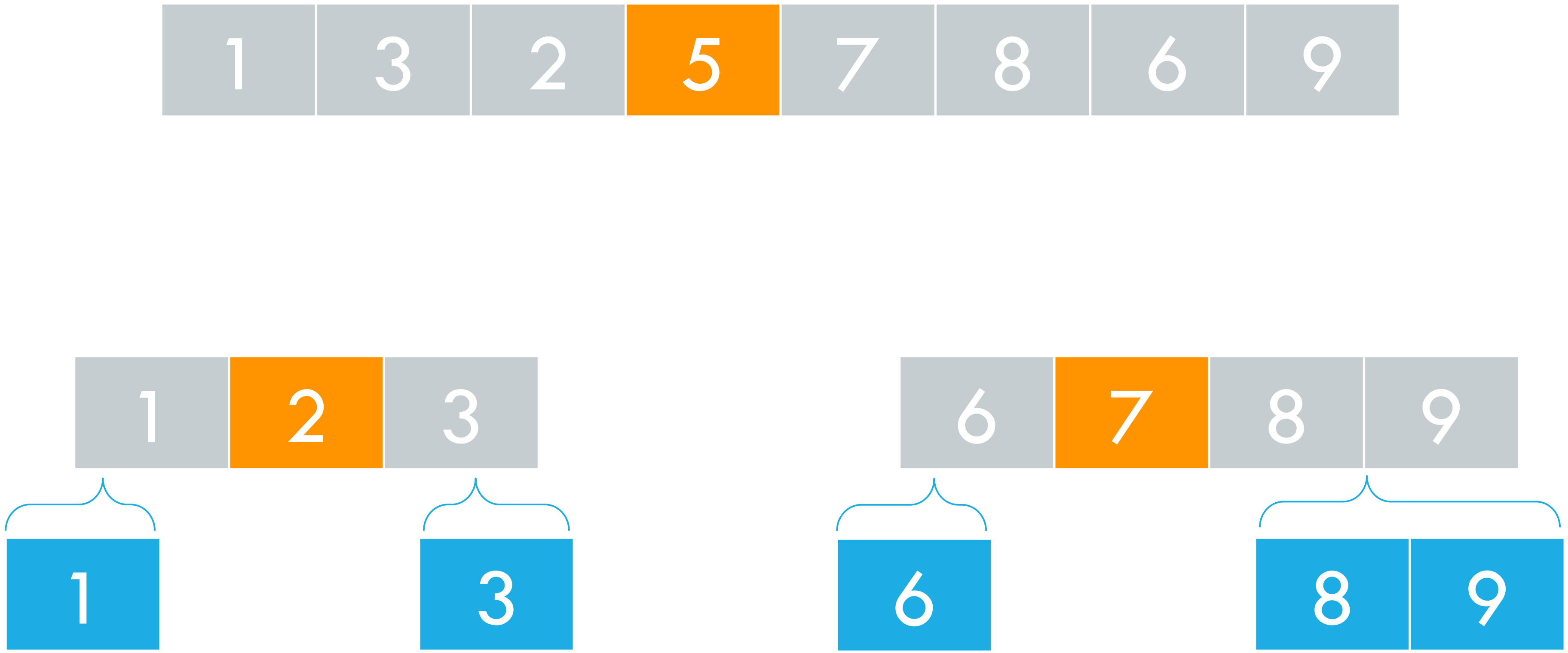
Recursive Solve

Clever Combine

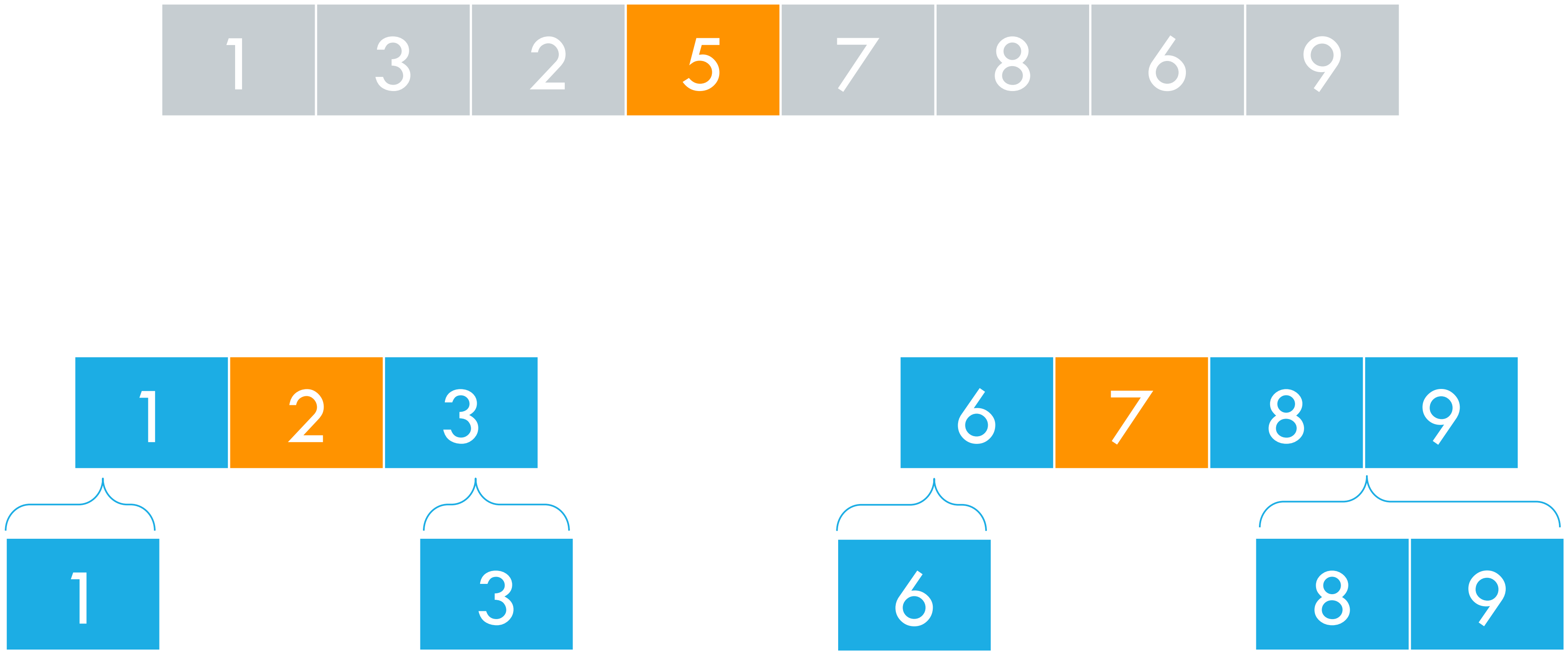
NEW IDEA: CLEVER SPLIT, SIMPLE COMBINE



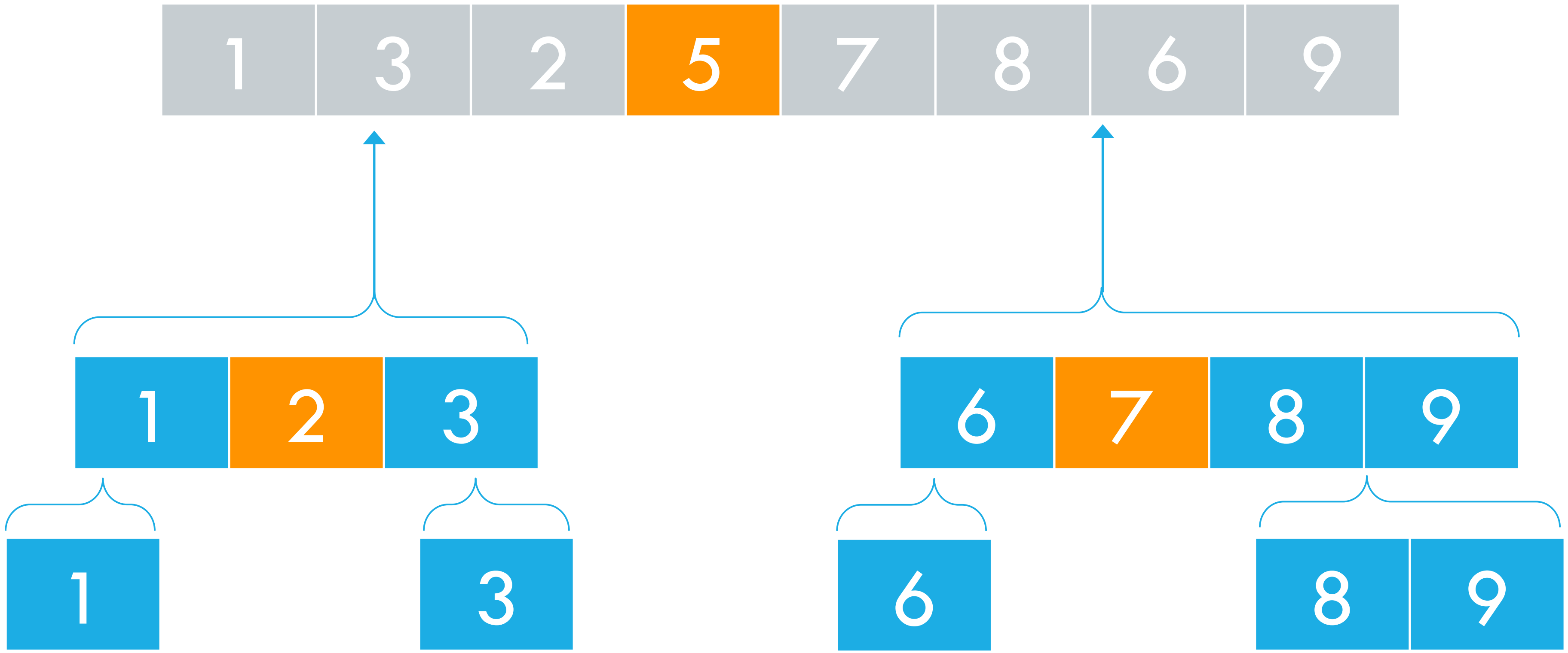
NEW IDEA: CLEVER SPLIT, SIMPLE COMBINE



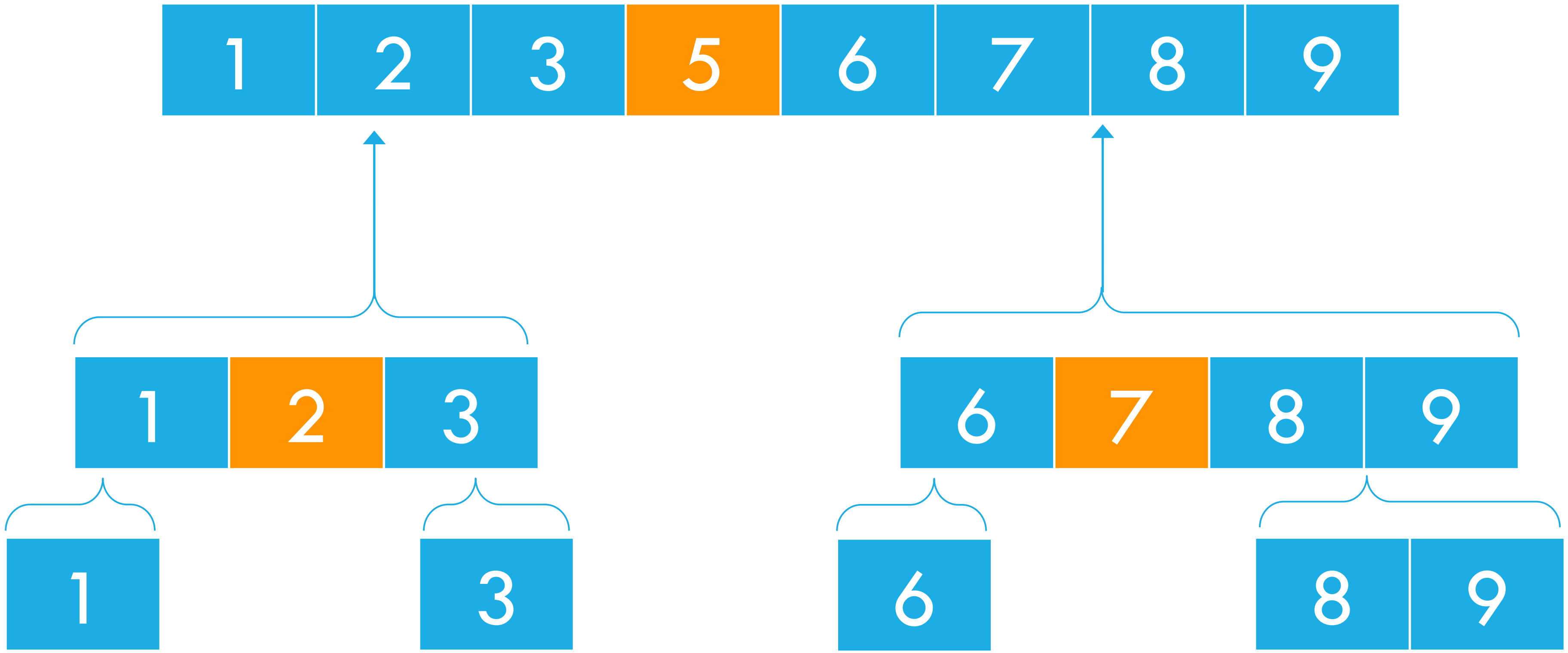
NEW IDEA: CLEVER SPLIT, SIMPLE COMBINE



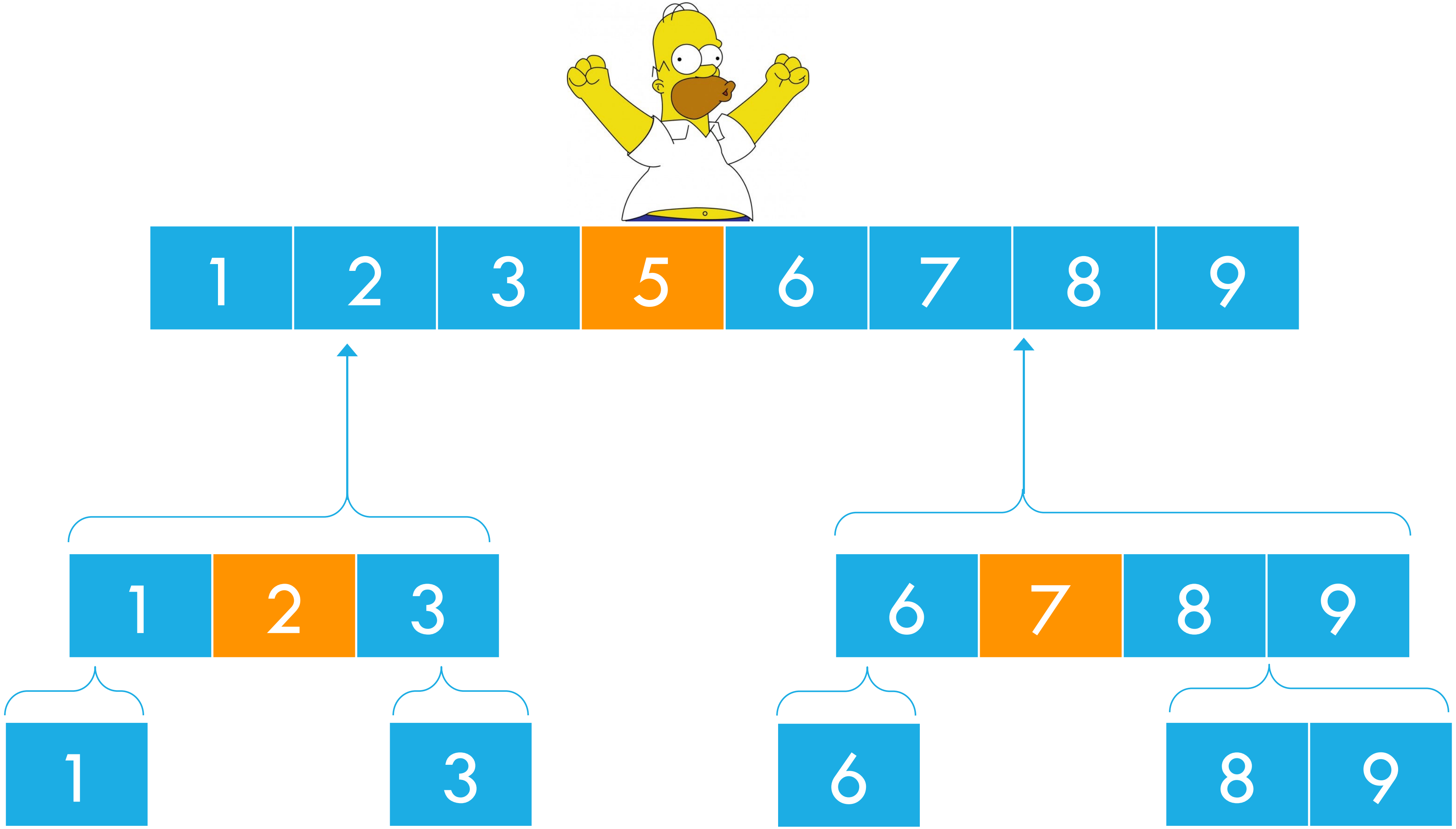
NEW IDEA: CLEVER SPLIT, SIMPLE COMBINE



NEW IDEA: CLEVER SPLIT, SIMPLE COMBINE



NEW IDEA: CLEVER SPLIT, SIMPLE COMBINE



QUICKSORT

sorts `A[low, ... high]` <u>inclusive</u>

call `Quicksort(A, 0, A.length – 1)` to start

```
function Quicksort(A, low, high)
   if low < high
      p = Partition(A, low, high)
      Quicksort (A, low, p-1)
      Quicksort (A, p+1, high)
```

Clever Split

Recursive Solve

Trivial Combine

QUICKSORT

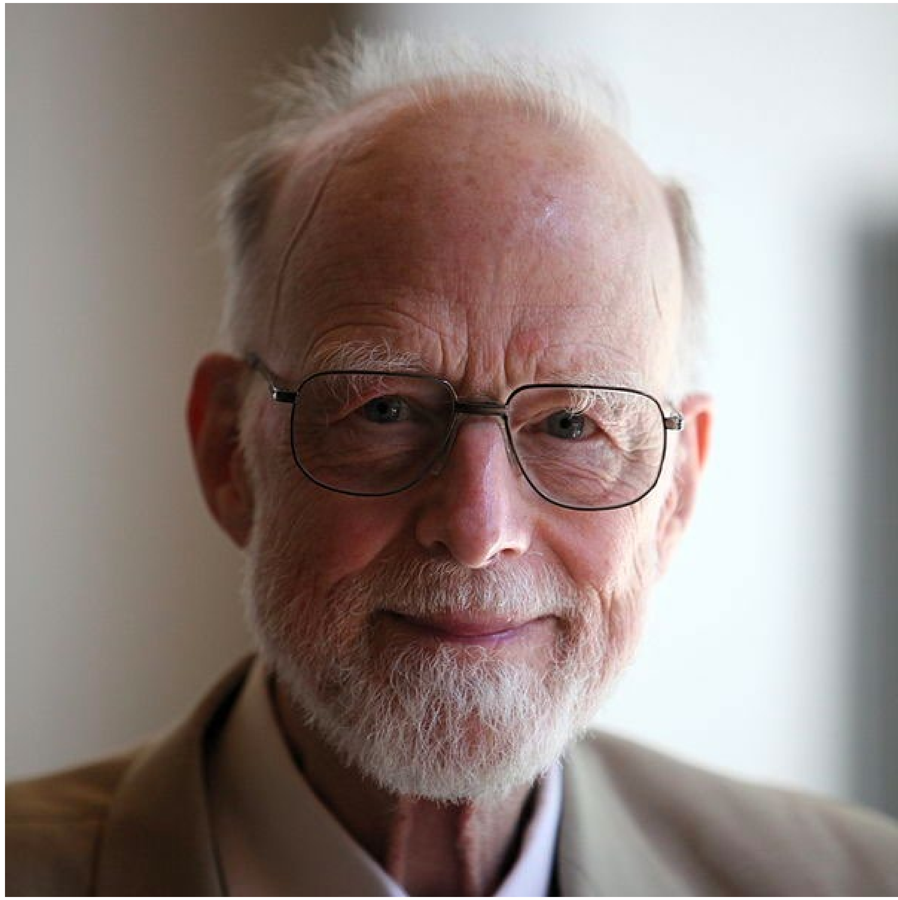
Invented by C.A.R. Hoare in 1959.

- Sir Charles Antony Richard Hoare
- **Turing award winner** (1980)

Visiting student at Moscow State University

Used for Machine Translation (English/Russian)

Quicksort is the de-facto sorting method in practice: **it can be 2-3x faster than mergesort!**



A QUOTE FROM HOARE

"There are two ways of constructing a software design:

One way is to make it **so simple** *that there are* **obviously no deficiencies** ...

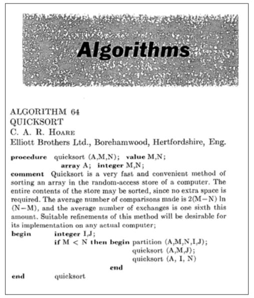
and the other way is to make it **so complicated** *that there are* **no obvious deficiencies.**

The first method is far more difficult."

\- Hoare

QUICKSORT TODAY

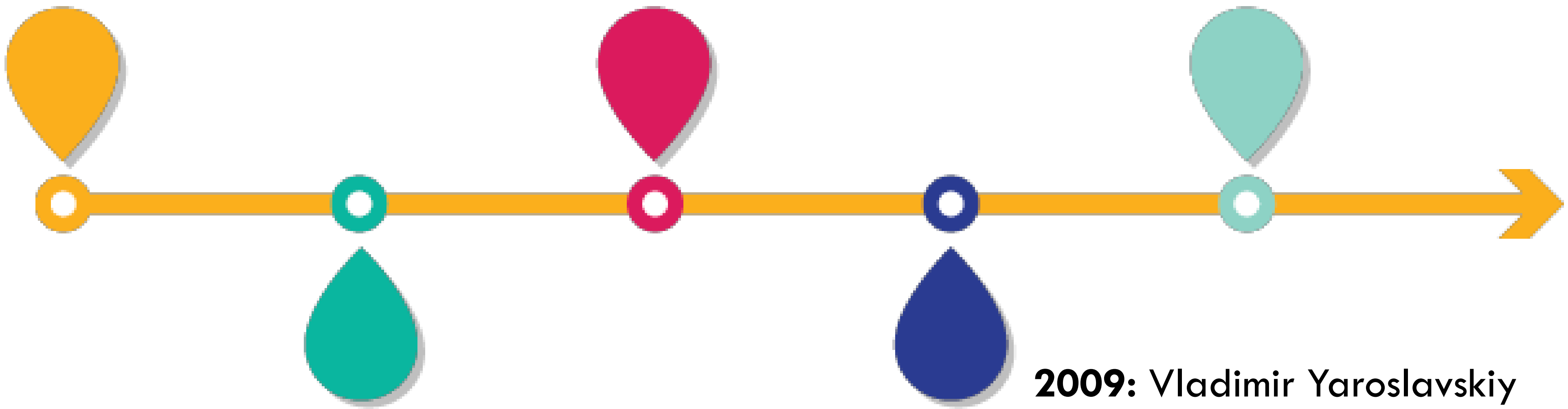
1959: Invented by Hoare



Algorithms

ALGORITHM 64
QUICKSORT
C. A. R. Hoare
Elliott Brothers Ltd., Borehamwood, Hertfordshire, Eng.

```
procedure  quicksort (A,M,N); value M,N;
             array A; integer M,N;
comment Quicksort is a very fast and convenient method of
sorting an array in the random-access store of a computer. The
entire contents of the store may be sorted, since no extra space is
required. The average number of comparisons made is 2(M−N) ln
(N−M), and the average number of exchanges is one sixth this
amount. Suitable refinements of this method will be desirable for
its implementation on any actual computer;
begin      integer I,J;
           if M < N then begin partition (A,M,N,I,J);
                               quicksort (A,M,J);
                               quicksort (A, I, N)
                         end
end        quicksort
```



1979: Adopted everywhere (e.g., Unix `qsort`)

1993: Bentley & McIlroy improvements

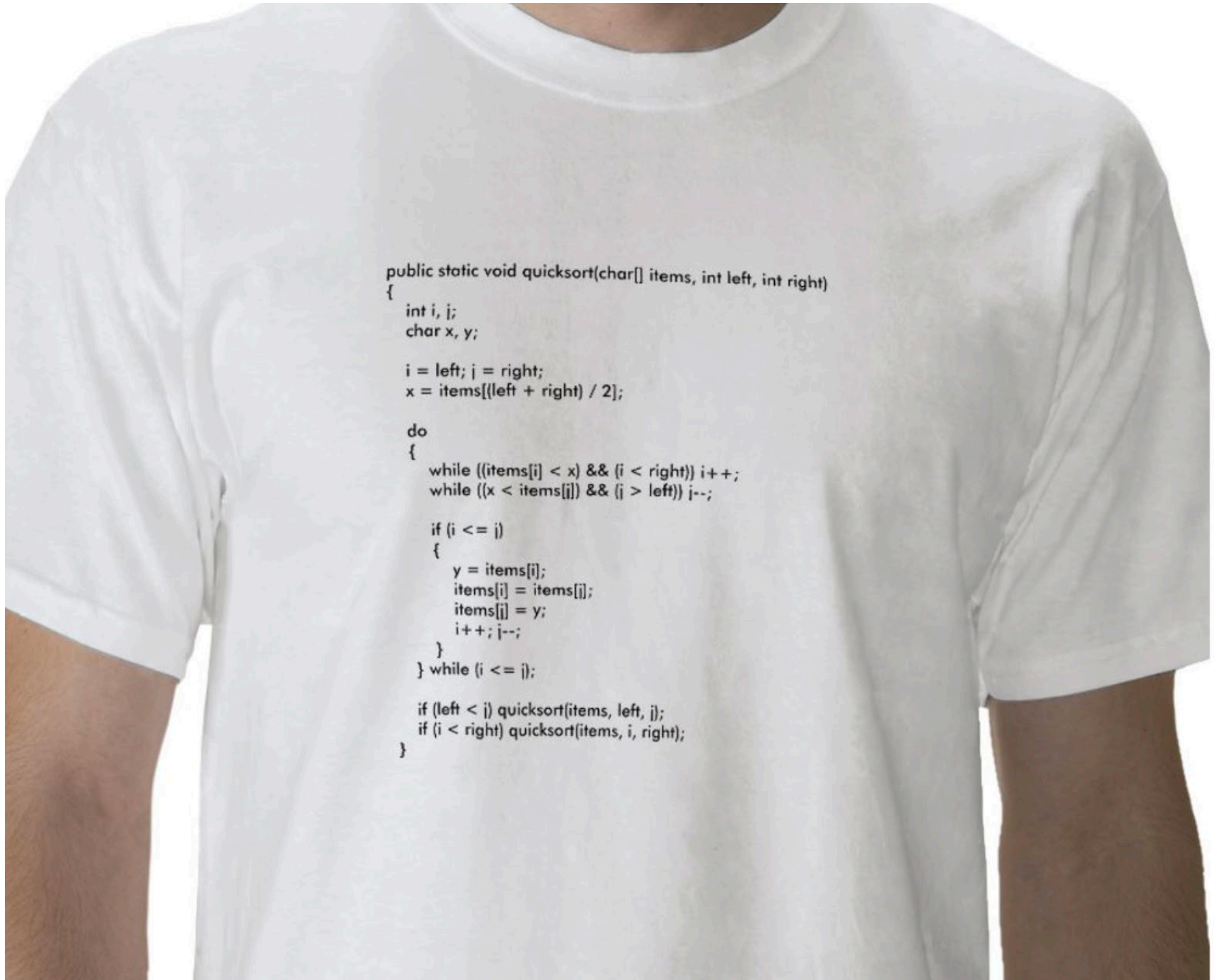
2009: Vladimir Yaroslavskiy
Dual-pivot Quicksort

- Standard sorting method in Java 7 and 8
- 10% Faster

2012: Sebastian Wild and Markus E. Nebel

- "Average Case Analysis of Java 7's Dual Pivot Quicksort"
- Best Paper Award at ESA 2012!

THERE'S EVEN A QUICKSORT T-SHIRT



From Sedgewick and Wayne's Princeton quicksort lecture slides: http://algs4.cs.princeton.edu/lectures/23Quicksort.pdf

QUICKSORT IN CLASS TODAY

Easy to understand

Moderately hard to implement correctly

Hard to analyze (randomization)

Challenging to optimize!

QUICKSORT

sorts `A[low, ... high]` <u>inclusive</u>

call `Quicksort(A, 0, A.length – 1)` to start

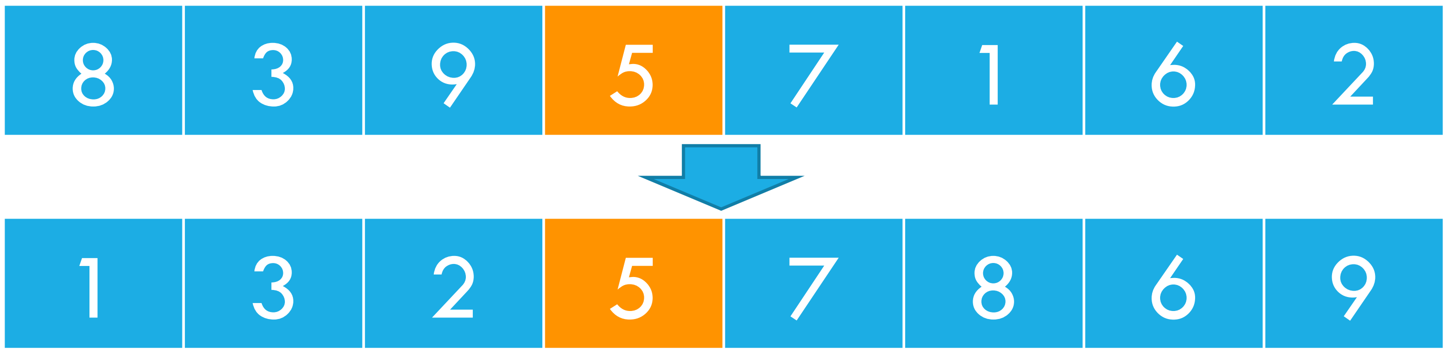
```
function Quicksort(A, low, high)
   if low < high
      p = Partition(A, low, high)
      Quicksort (A, low, p-1)
      Quicksort (A, p+1, high)
```

Clever Split

Recursive Solve

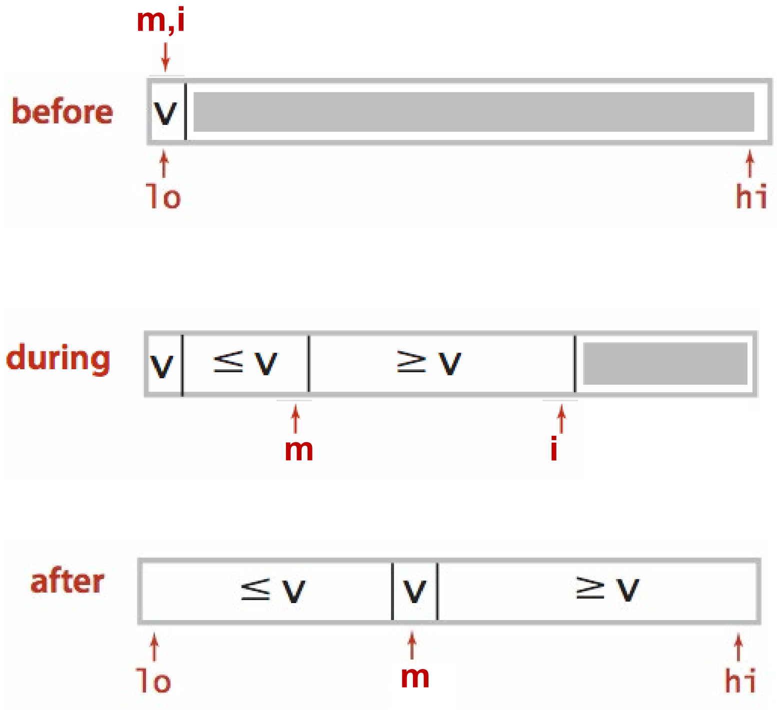
Trivial Combine

PARTITION FUNCTION

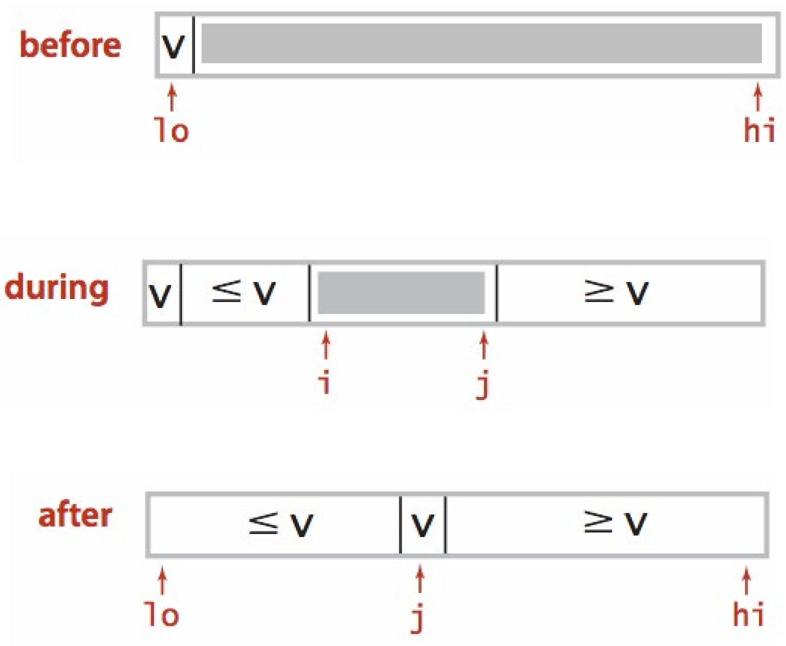


2 DIFFERENT PARTITIONING ALGS

Lumoto



Hoare

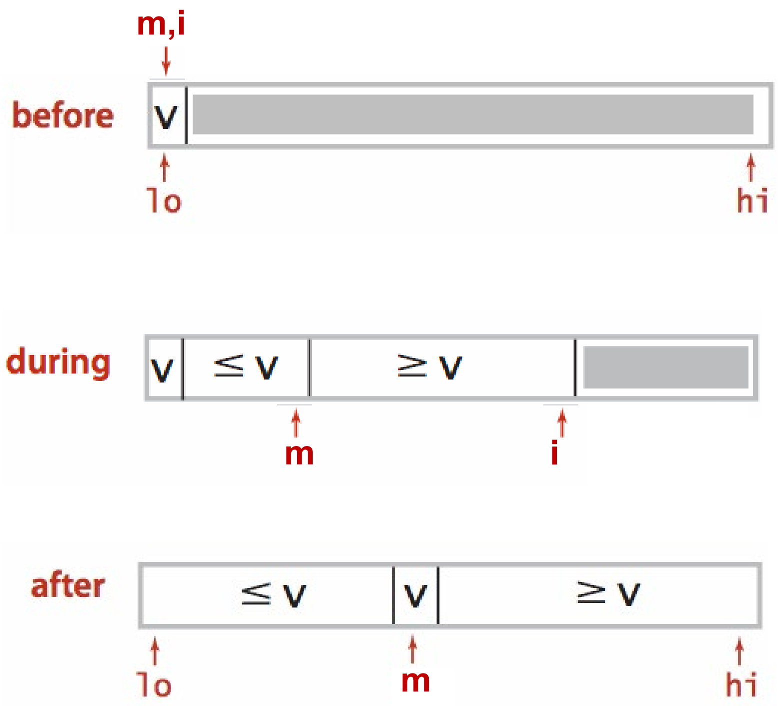


LUMOTO'S PARTITION ALGORITHM

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
```

Fill in the code

```
    swap(A[m], A[low])
    return m
```



LUMOTO'S PARTITION ALGORITHM: START

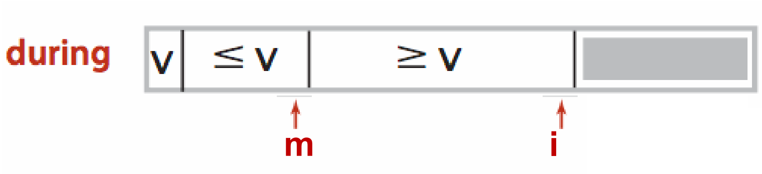
```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            Fill in the code
    swap(A[m], A[low])
    return m
```

LUMOTO'S PARTITION ALGORITHM: MIDDLE

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
```

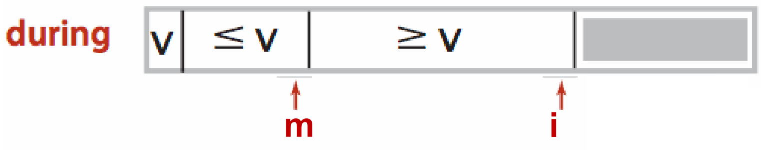
Fill in the code

```
    swap(A[m], A[low])
    return m
```



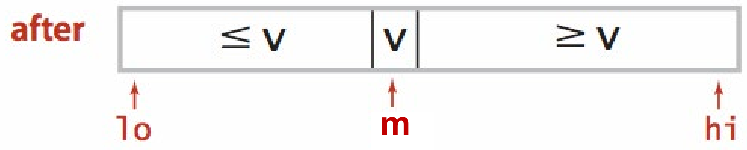
LUMOTO'S PARTITION ALGORITHM: MIDDLE

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```



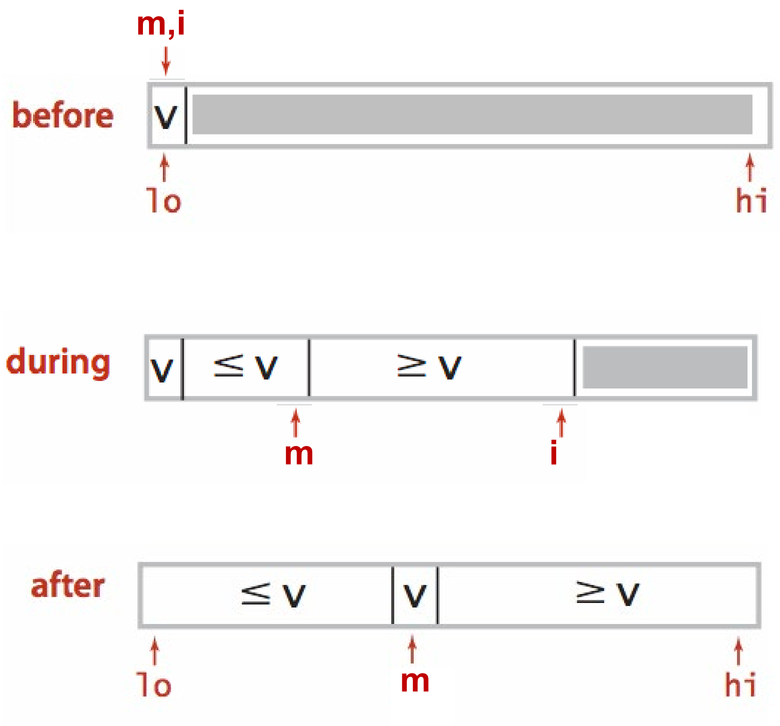
LUMOTO'S PARTITION ALGORITHM: END

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

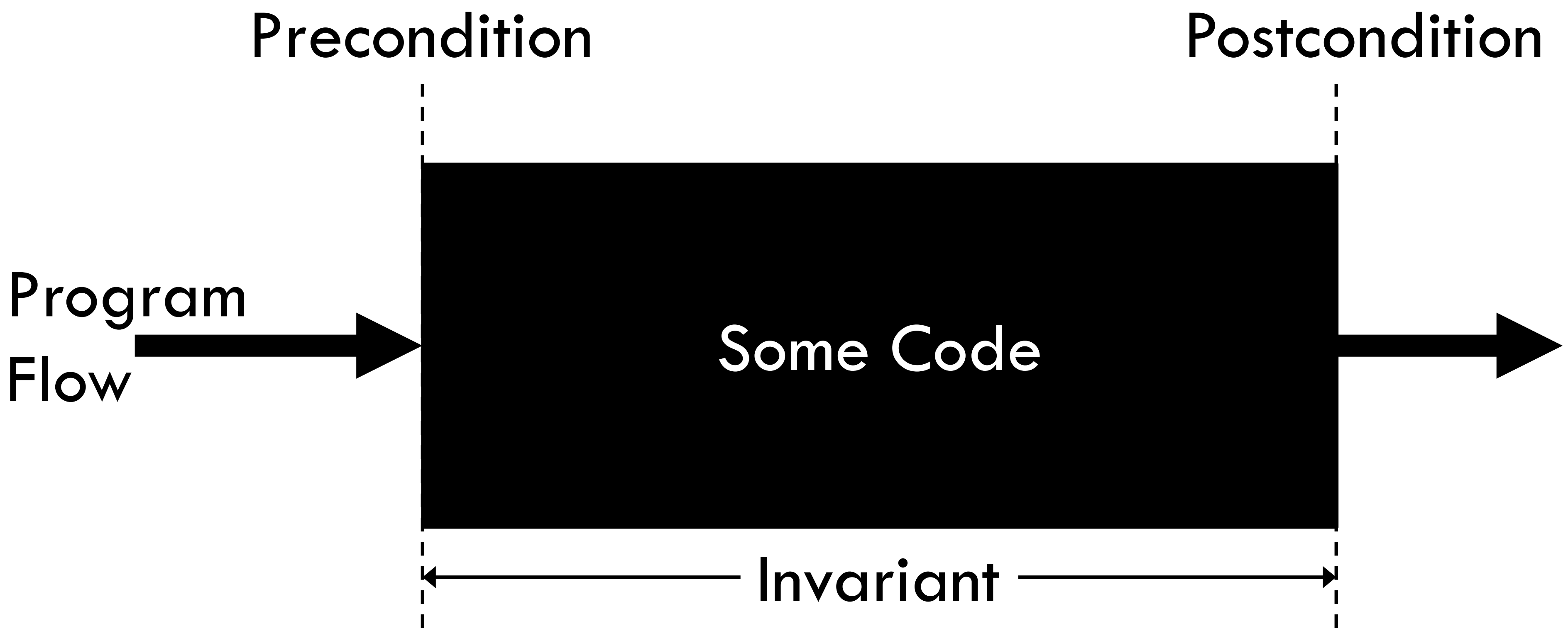


LUMOTO'S PARTITION ALGORITHM: CORRECT?

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```



IS LUMOTO'S CODE CORRECT?



LUMOTO'S PARTITION ALGORITHM

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

What are the pre-and-post conditions?

LUMOTO'S PARTITION ALGORITHM

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

Preconditions:

- **PRE-1:**
- **PRE-2:**
- **PRE-3:**

LUMOTO'S PARTITION ALGORITHM

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

Preconditions:

- **PRE-1**: A is an array
- **PRE-2:** $0 \leq low \leq high \leq A.length$
- **PRE-3:** A has no duplicates

LUMOTO'S PARTITION ALGORITHM

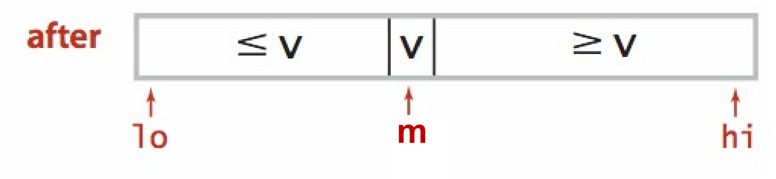
```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

Preconditions:

- **PRE-1**: A is an array
- **PRE-2:** $0 \leq$ low $\leq$ high $\leq$ A.length
- **PRE-3:** A has no duplicates

Postconditions:

- **POST-1:**
- **POST-2:**
- **POST-3:**



LUMOTO'S PARTITION ALGORITHM

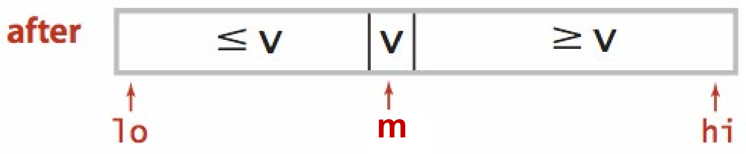
```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

Preconditions:

- **PRE-1**: A is an array
- **PRE-2:** $0 \le low \le high \le A.length$
- **PRE-3:** A has no duplicates

Postconditions:

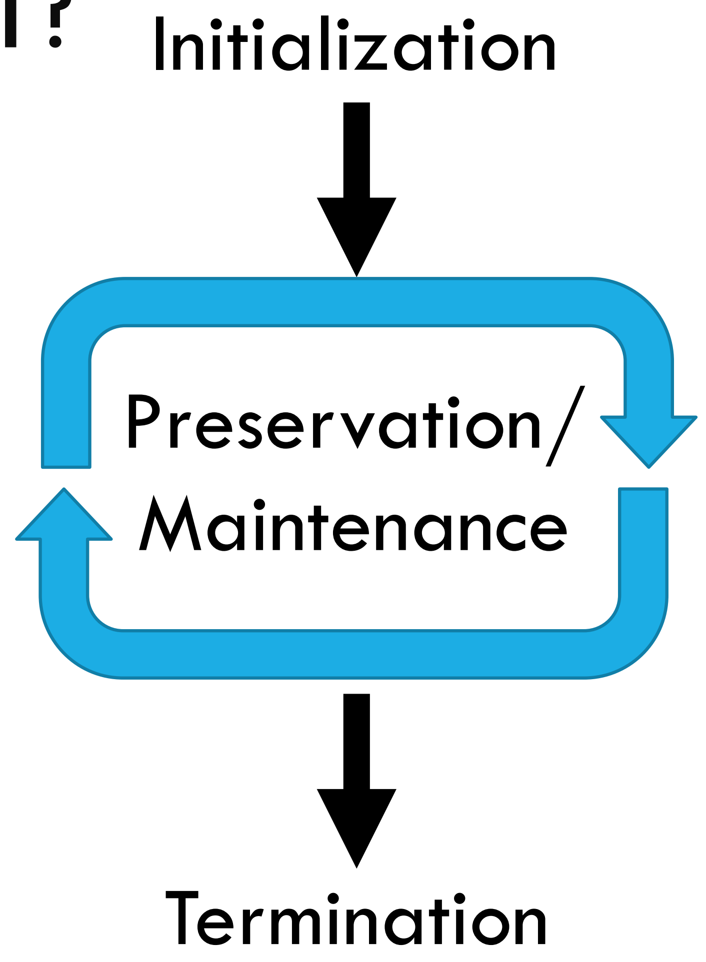
- **POST-1:** $A[m] = v$
- **POST-2:** if $low \le k \le m-1$, $A[k] < v$
- **POST-3:** if $m+1 \le k \le high$, $A[k] > v$



IS LUMOTO'S PARTITION CORRECT?

Strategy: Establish 3 properties for the main loop

- **Initialization:** we've set up our invariant
- **Preservation:** the invariant is true at *every* iteration
- **Termination:** when the loop terminates, the desired result is true



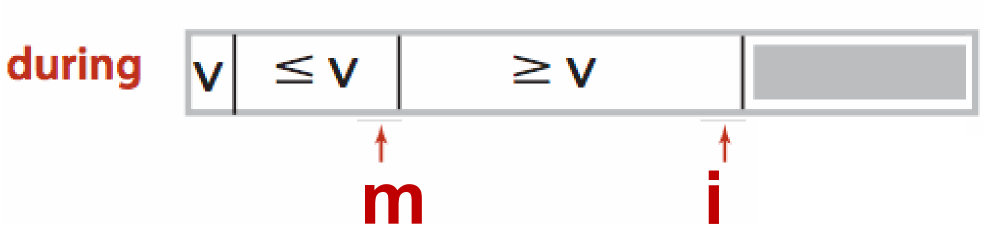
[Programming Pearls, Bentley]

LUMOTO'S PARTITION ALGORITHM

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

Loop Invariances:

- **I-1**: A[low] = **?**
- **I-2:** if low+1 $\leq$ k $\leq$ m, **?**
- **I-3:** if m+1 $\leq$ k $\leq$ i, **?**

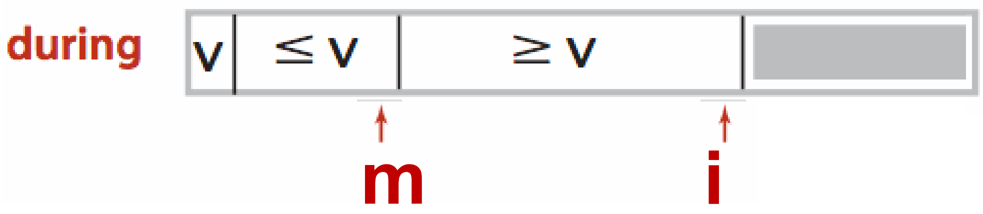


LUMOTO'S PARTITION ALGORITHM

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

Loop Invariances:

- **I-1**: A[low] = v
- **I-2:** if low+1 $\le$ k $\le$ m, A[k] < v
- **I-3:** if m+1 $\le$ k $\le$ i, A[k] > v



INITIALIZATION: BEFORE LOOP

Loop Invariances:

- **I-1**: A[low] = v
- **I-2:** if low+1 $\leq$ k $\leq$ m, A[k] < v
- **I-3:** if m+1 $\leq$ k $\leq$ i, A[k] > v

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

At **initialization,**

- I-1 holds, since v = A[low]
- I-2 holds (trivially) since there is no k that satisfies that range
- I-3 holds (trivially) since there is no k that satisfies that range either.

INITIALIZATION: DURING LOOP

Loop Invariances:

- **I-1**: A[low] = v
- **I-2:** if low+1 $\leq$ k $\leq$ m, A[k] < v
- **I-3:** if m+1 $\leq$ k $\leq$ i, A[k] > v

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

At the first loop,

- i = low+1
- Two cases.
- Case 1: A[i] < v
  - I-3 is broken!
  - Code fixes this by m++ and swap.
  - At the end of iteration:
    - I-1 holds
    - I-2 holds
    - I-3 is restored
- Case 2: A[i] > v
  - At the end of iteration:
    - All invariances still hold

INITIALIZATION: DURING LOOP

Loop Invariances:

- **I-1**: A[low] = v
- **I-2:** if low+1 ≤ k ≤ m, A[k] < v
- **I-3:** if m+1 ≤ k ≤ i, A[k] > v

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

At the next loop,

- i = low+2
- Two cases.
- Case 1: A[i] < v
  - I-3 is broken!
  - Code fixes this by m++ and swap.
  - At the end of iteration:
    - I-1 holds
    - I-2 holds
    - I-3 is restored
- Case 2: A[i] > v
  - At the end of iteration:
    - All invariances still hold

INITIALIZATION: DURING LOOP

Loop Invariances:

- **I-1**: A[low] = v
- **I-2:** if low+1 $\leq$ k $\leq$ m, A[k] < v
- **I-3:** if m+1 $\leq$ k $\leq$ i, A[k] > v

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

Assume true at k, check for k+1

- i = k+1
- Two cases.
- Case 1: A[i] < v
  - I-3 is broken!
  - Code fixes this by m++ and swap.
  - At the end of iteration:
    - I-1 holds
    - I-2 holds
    - I-3 is restored
- Case 2: A[i] > v
  - At the end of iteration:
    - All invariances still hold

LOOP TERMINATION

Loop Invariances:

- **I-1**: A[low] = v
- **I-2:** if low+1 $\leq$ k $\leq$ m, A[k] < v
- **I-3:** if m+1 $\leq$ k $\leq$ i, A[k] > v

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

At i = high

- Two cases.
- Case 1: A[i] < v
  - I-3 is broken!
  - Code fixes this by m++ and swap.
  - At the end of iteration:
    - I-1 holds
    - I-2 holds
    - I-3 is restored
- Case 2: A[i] > v
  - At the end of iteration:
    - All invariances still hold

FUNCTION TERMINATION: DO OUR POSTCONDITIONS HOLD?

Postconditions:

- **POST-1**: $A[m] = v$
- **POST-2**: if $low \leq k \leq m-1$, $A[k] < v$
- **POST-3**: if $m+1 \leq k \leq high$, $A[k] > v$

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

At the end of the loop:

- $A[low] = v$
- if $low+1 \leq k \leq m$, $A[k] < v$
- if $m+1 \leq k \leq i$, $A[k] > v$

Problem: ???

FUNCTION TERMINATION: DO OUR POSTCONDITIONS HOLD?

Postconditions:

- **POST-1**: A[m] = v
- **POST-2**: if low ≤ k ≤ m-1, A[k] < v
- **POST-3**: if m+1 ≤ k ≤ high, A[k] > v

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

At the end of the loop:

- A[low] = v
- if low+1 ≤ k ≤ m, A[k] < v
- if m+1 ≤ k ≤ i, A[k] > v

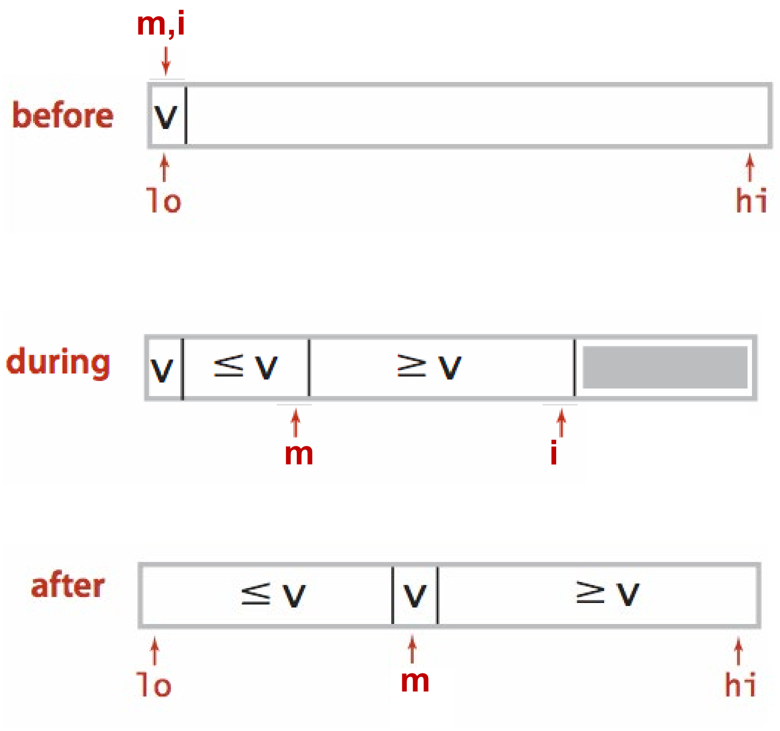
Problem: pivot is the wrong place.

Swap A[m] and A[low]

We are done! POST-1 to 3 all hold!

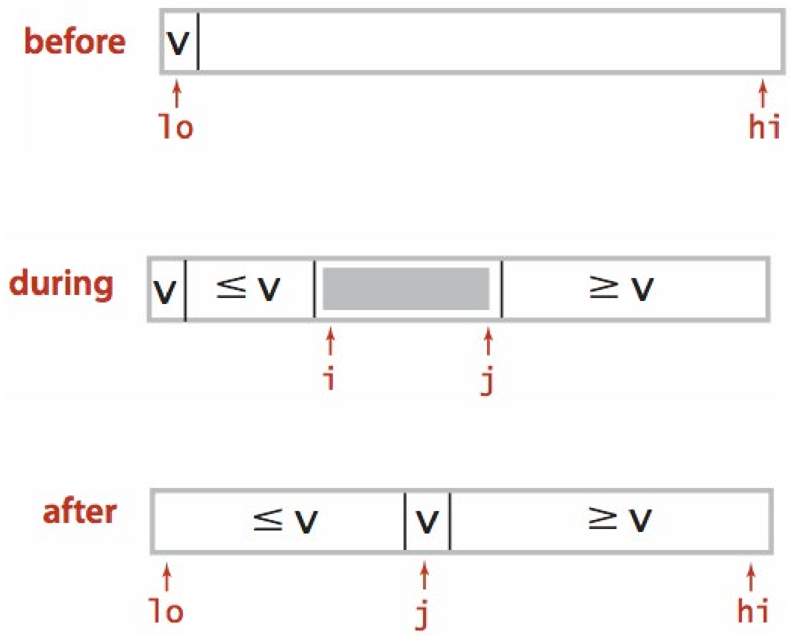
LUMOTO'S PARTITION IS CORRECT

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
            m++
            swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```



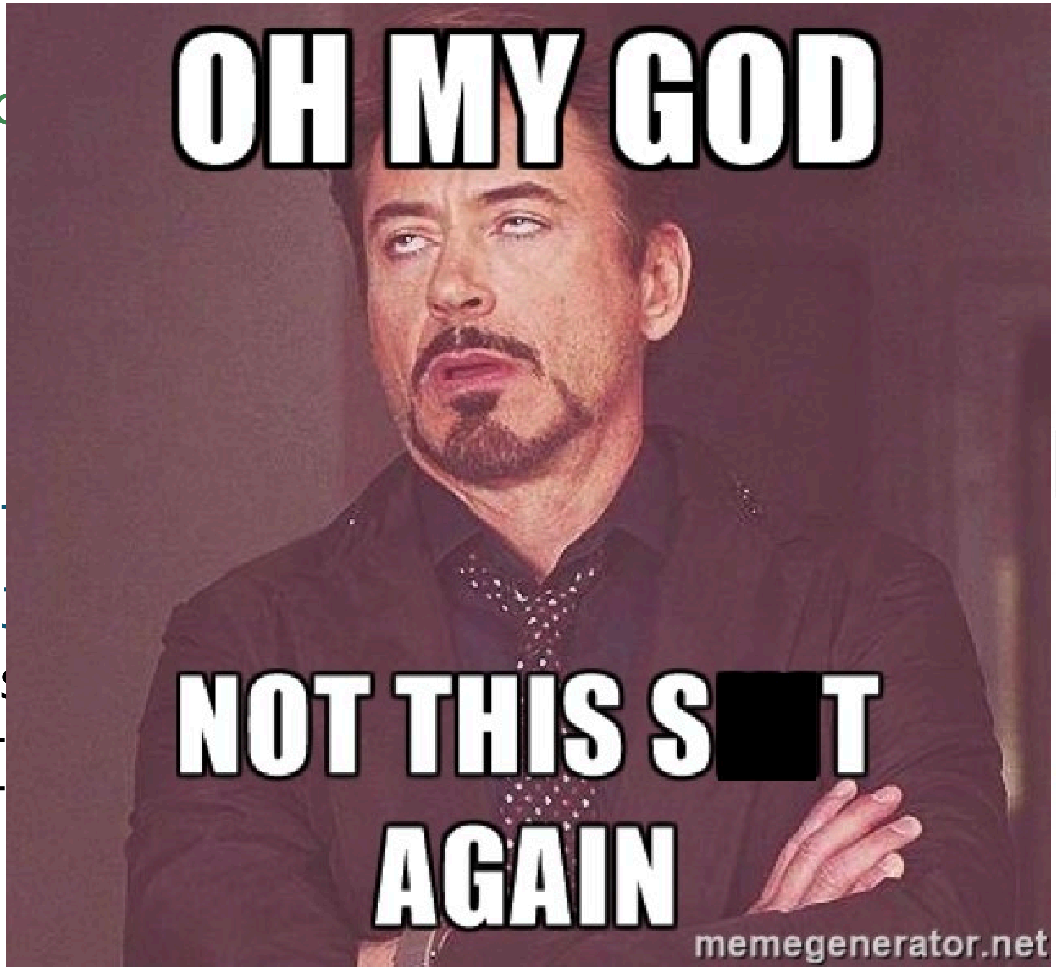
HOARE'S PARTITION ALGORITHM

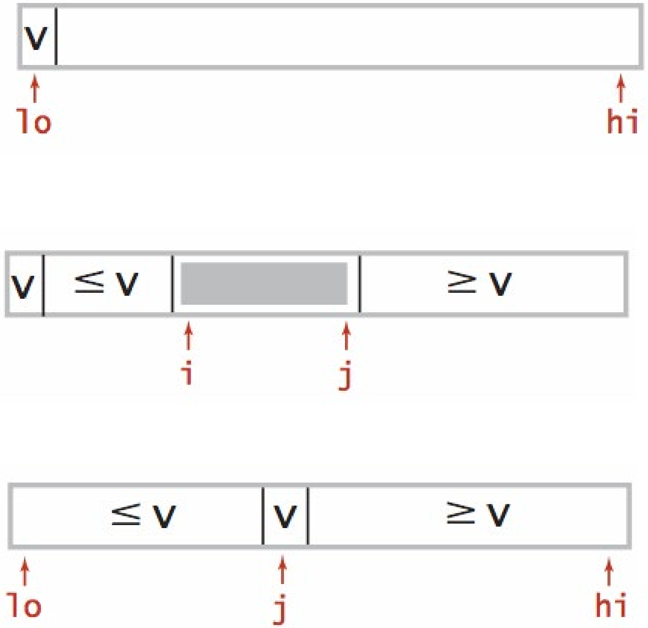
```
function Partition(A, low, high)
    v = A[low]
    i = low+1;
    j = high;
    while i < j
        while (A[i] < v) and (i <= high) i++
        while (A[j] > v) and (j >= low) j--
        if (i<j) swap(A[i], A[j])
    swap(A[j], A[low])
    return j
```



HOARE'S PARTITION ALGORITHM: CORRECT?

```
function Partiti
    v = A[low]
    i = low+1;
    j = high;
    while i < j
        while (A[
        while (A[
        if (i<j)
    swap(A[j], A[
    return i
```





ASIDE: CORRECTNESS & EFFICIENCY

	Inefficient	Efficient
Incorrect	Slow & Wrong	Very fast… but wrong (sometimes useful)
Correct	Correct but slow (sometimes useful)	**We want this!**

QUESTIONS?

BACK TO QUICKSORT

```
function Quicksort(A, low, high)
   if low < high
      p = Partition(A, low, high)
      Quicksort (A, low, p-1)
      Quicksort (A, p+1, high)
```

Clever Split

Recursive Solve

Trivial Combine

BACK TO QUICKSORT

Poll Everywhere
https://bit.ly/2LvG9bq

```
function Quicksort(A, low, high)
    if low < high
        p = Partition(A, low, high)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

What is the worst-case time complexity for Quicksort where pivot = A[lo]?

A. $O(n \log n)$
B. $O(n)$
C. $O(1)$
D. None of the above
E. All of the above
F. E doesn't make sense!

Poll Everywhere
https://bit.ly/2LvG9bq

BACK TO QUICKSORT

```
function Quicksort(A, low, high)
    if low < high
        p = Partition(A, low, high)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

Deterministic Quicksort is $O(n^2)$

What is the worst-case time complexity for Quicksort where pivot = A[lo]?

A. $O(n \log n)$
B. $O(n)$
C. $O(1)$
D. **None of the above**
E. All of the above
F. E doesn't make sense!

DETERMINISTIC QUICKSORT: COMPUTATIONAL COMPLEXITY

Always choose A[low] for the pivot.

Consider the following.

100	99	98	97	96	95	94	93	92

DETERMINISTIC QUICKSORT: COMPUTATIONAL COMPLEXITY

Always choose A[low] for the pivot.

Consider the following.

100	99	98	97	96	95	94	93	92

99	98	97	96	95	94	93	92	100

DETERMINISTIC QUICKSORT: COMPUTATIONAL COMPLEXITY

Always choose A[low] for the pivot.

Consider the following.

99	98	97	96	95	94	93	92	100

98	97	96	95	94	93	98	99	100

DETERMINISTIC QUICKSORT: COMPUTATIONAL COMPLEXITY

Always choose A[low] for the pivot.

Consider the following.

98	97	96	95	94	93	92	99	100

⇓

97	96	95	94	93	92	98	99	100

DETERMINISTIC QUICKSORT: COMPUTATIONAL COMPLEXITY

Always choose A[low] for the pivot.

Consider the following.

97	96	95	94	93	92	98	99	100

96	95	94	93	92	97	98	99	100

DETERMINISTIC QUICKSORT: COMPUTATIONAL COMPLEXITY

Always choose A[low] for the pivot.

Consider the following.

Uh oh… I see a (bad) pattern

96	95	94	93	92	97	98	99	100

95	94	93	92	96	97	98	99	100

DETERMINISTIC QUICKSORT

96	95	94	93	92	97	98	99	100

95	94	93	92	96	97	98	99	100

```
function Quicksort(A, low, high)
    if low < high
        p = Partition(A, low, high)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

Each call to partition sorts only 1 element!

DETERMINISTIC QUICKSORT

Poll Everywhere

https://bit.ly/2LvG9bq

```
function Quicksort(A, low, high)
    if low < high
        p = Partition(A, low, high)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

What recurrent relation best describes deterministic quicksort?

A. $T(n) = 2T\left(\frac{n}{2}\right) + cn$

B. $T(n) = 2T\left(\frac{n}{2}\right) + c$

C. $T(n) = T(n-1) + T(1) + cn$

D. $T(n) = T(n-1) + T(1) + c$

E. Hey! This one is tricky!

Poll Everywhere

https://bit.ly/2LvG9bq

DETERMINISTIC QUICKSORT

```
function Quicksort(A, low, high)
    if low < high
        p = Partition(A, low, high)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

What recurrent relation best describes deterministic quicksort?

A. $T(n) = 2T\left(\frac{n}{2}\right) + cn$

B. $T(n) = 2T\left(\frac{n}{2}\right) + c$

C. $\boldsymbol{T(n) = T(n-1) + T(1) + cn}$

D. $T(n) = T(n-1) + T(1) + c$

E. Hey! This one is tricky!

DETERMINISTIC QUICKSORT: RECURRENCE RELATION

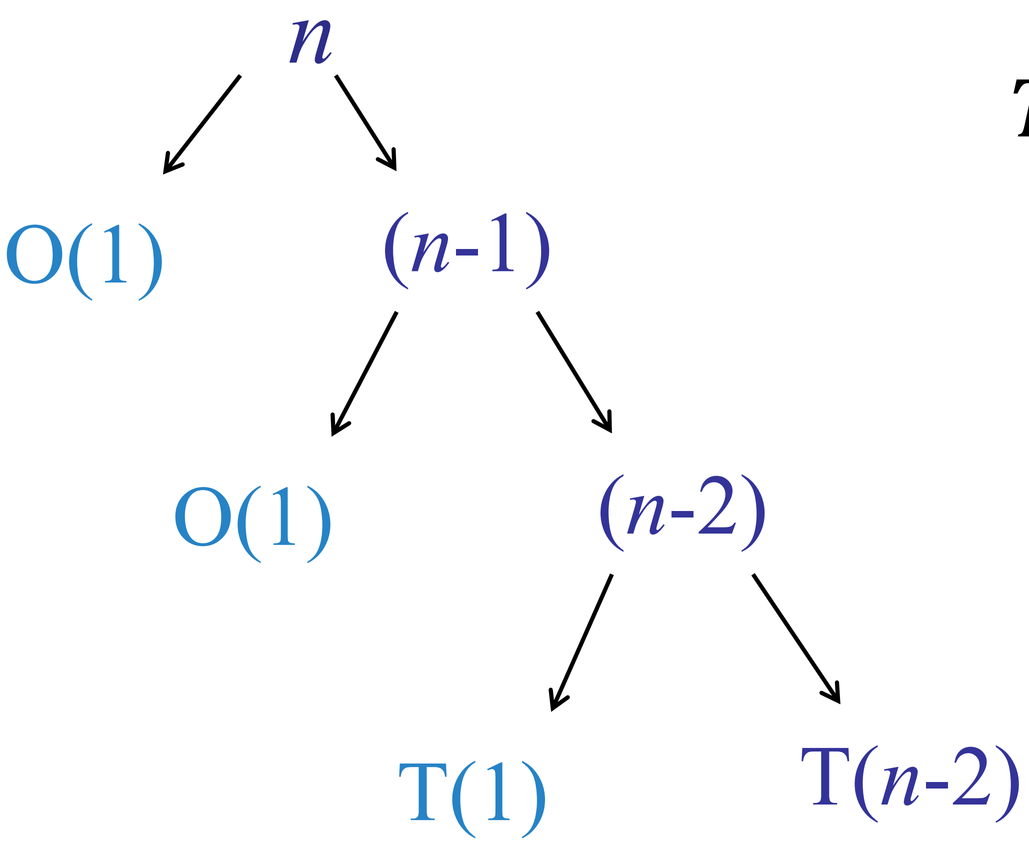
$$\overbrace{T(n)}^{\text{Cost of QuickSort on } n \text{ elements}} = \underbrace{T(n-1)}_{\text{Cost of QuickSort on } n-1 \text{ elements}} + \overbrace{T(1)}^{\text{Cost of QuickSort on } 1 \text{ element}} + \underbrace{cn}_{\text{Cost of partition on } n \text{ elements}}$$

$$\begin{array}{ccc} & n & \\ \swarrow & & \searrow \\ \mathrm{T}(1) & & \mathrm{T}(n\text{-}1) \end{array}$$

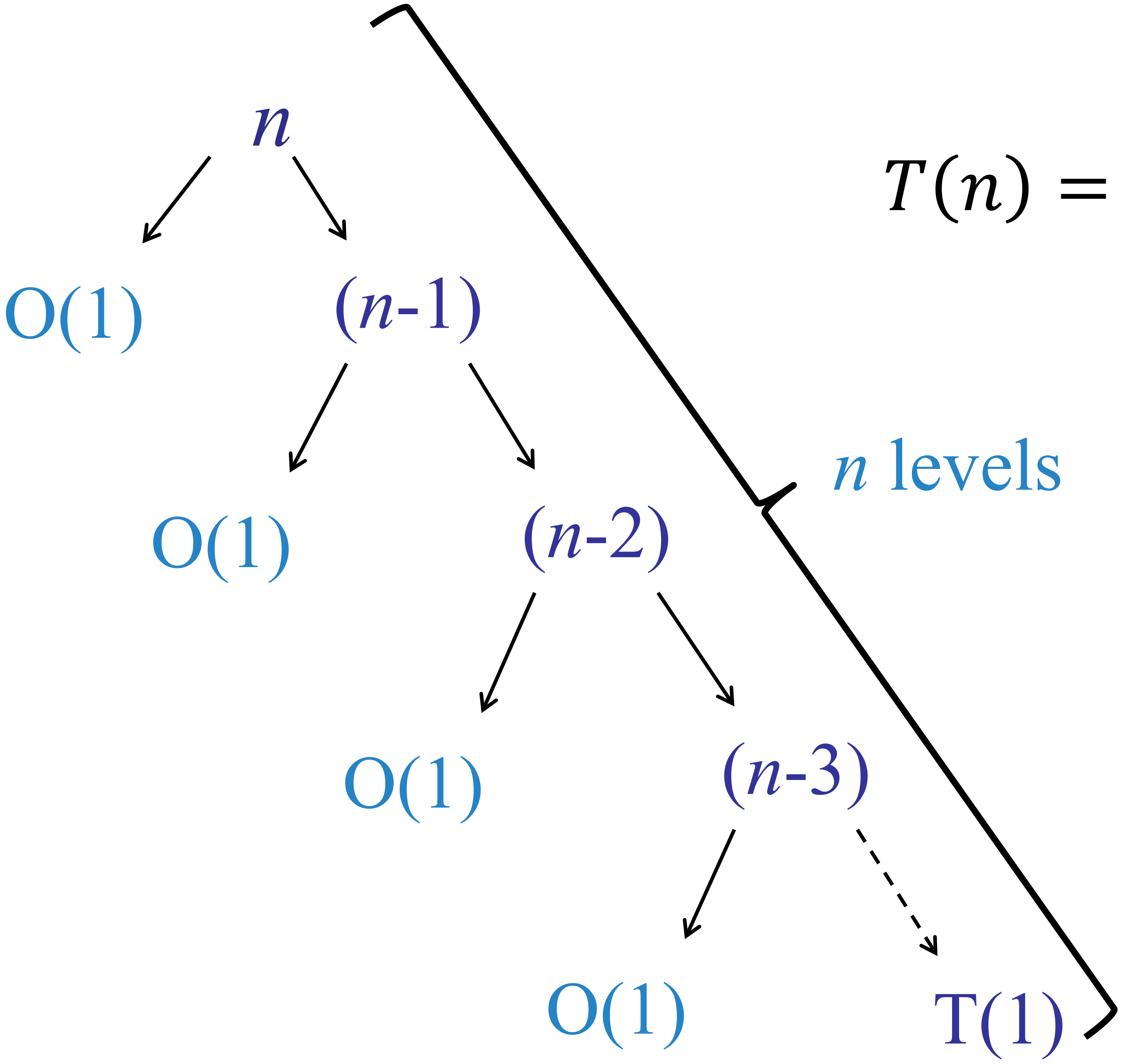
$$T(n) = T(n-1) + T(1) + cn$$

$$
\begin{tikzcd}
 & n \arrow[dl] \arrow[dr] & & \\
O(1) & & (n\text{-}1) \arrow[dl] \arrow[dr] & \\
 & T(1) & & T(n\text{-}2)
\end{tikzcd}
$$

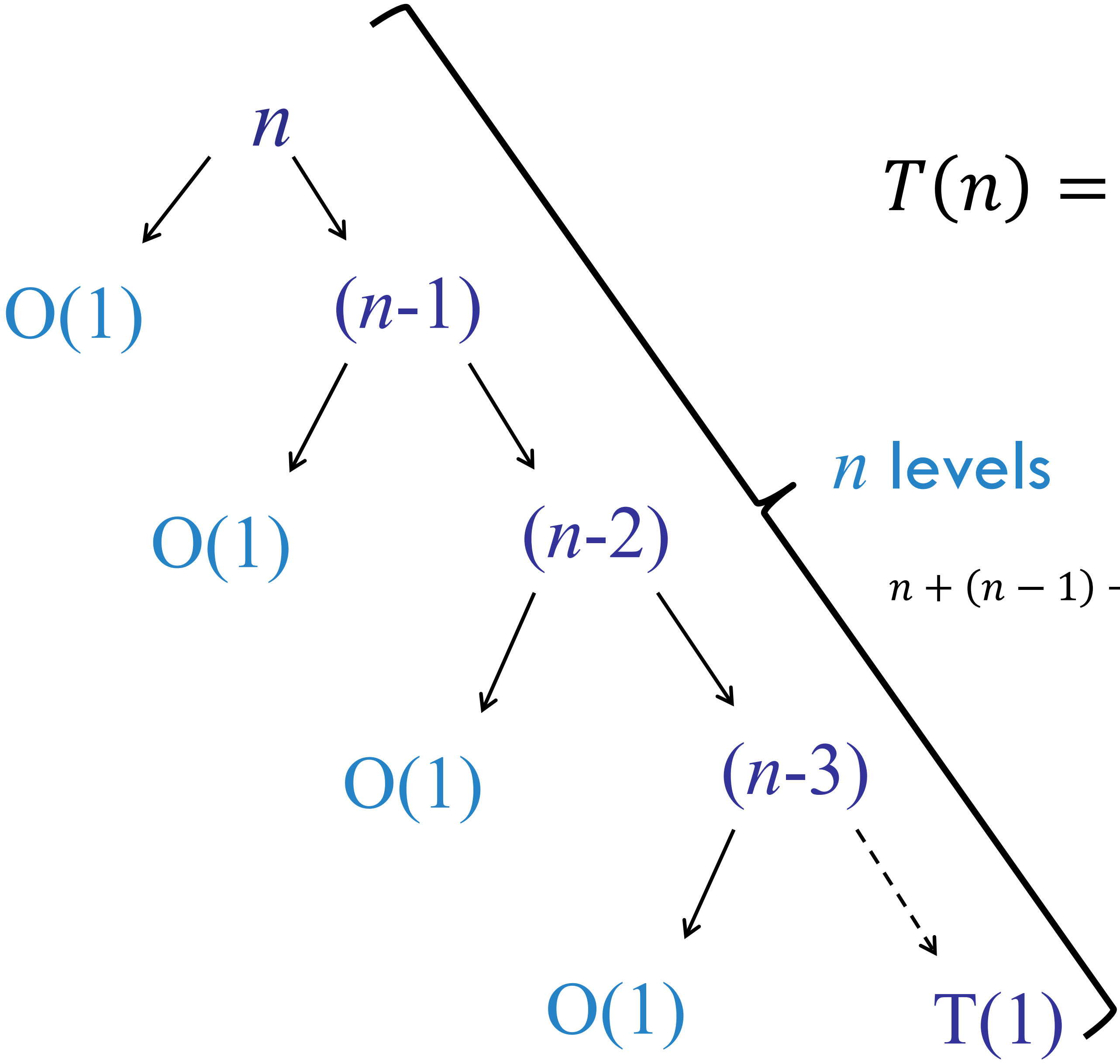
$$T(n) = T(n-1) + T(1) + cn$$



$$T(n) = T(n-1) + T(1) + cn$$



$$T(n) = T(n-1) + T(1) + cn$$



$$T(n) = T(n-1) + T(1) + cn$$

$$n + (n-1) + (n-2) + \ \ldots + 1 = \frac{n(n+1)}{2} = \ O(n^2)$$

CHANGE THE PIVOT CHOICE, MAYBE?

Always choose A[low] for the pivot. The problem is here!

Consider the following.

100	99	98	97	96	95	94	93	92

99	98	97	96	95	94	93	92	100

BETTER QUICKSORT: COMPUTATIONAL COMPLEXITY

Choose median in A[low, ..., high] for the pivot

```
function Quicksort(A, low, high)
    if low < high
        pidx = chooseMedian(A, p, low, high)
        p = Partition(A, pidx, low, high)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

Assume: chooseMedian runs in O(1) time

What recurrent relation best describes better quicksort?

A. $T(n) = 2T\left(\frac{n}{2}\right) + cn$

B. $T(n) = 2T\left(\frac{n}{2}\right) + c$

C. $T(n) = T(n-1) + T(1) + cn$

D. $T(n) = T(n-1) + T(1) + c$

E. Hey! This one is more tricky!

BETTER QUICKSORT: COMPUTATIONAL COMPLEXITY

Choose median in A[low, ..., high] for the pivot

```
function Quicksort(A, low, high)
    if low < high
        pidx = chooseMedian(A, p, low, high)
        p = Partition(A, pidx, low, high)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

Assume: chooseMedian runs in O(1) time

What recurrent relation best describes better quicksort?

A. $T(n) = 2T\left(\frac{n}{2}\right) + cn$
B. $T(n) = 2T\left(\frac{n}{2}\right) + c$
C. $T(n) = T(n-1) + T(1) + cn$
D. $T(n) = T(n-1) + T(1) + c$
E. Hey! This one is more tricky!

BETTER QUICKSORT: COMPUTATIONAL COMPLEXITY

Choose median for the pivot

100	99	98	97	96	95	94	93	92

BETTER QUICKSORT: COMPUTATIONAL COMPLEXITY

Choose median for the pivot

100	99	98	97	96	95	94	93	92

92	93	94	95	96	97	98	99	100

That looks nice! ☺

BETTER QUICKSORT: COMPUTATIONAL COMPLEXITY

Cost of QuickSort
on n elements

Cost of QuickSort on
n /2 "low" elements

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

Cost of QuickSort on
n /2 "high" elements

Cost of partition
on n elements

BETTER QUICKSORT: COMPUTATIONAL COMPLEXITY

Choose median in A[low, ..., high] for the pivot (assume constant operation)

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

Assume: chooseMedian runs in O(1) time

What is the computational complexity of better quicksort?

A. $O(n)$
B. $O(n^2)$
C. $O(n \log n)$
D. $O(2^{\log n})$
E. Naruto!

BETTER QUICKSORT: COMPUTATIONAL COMPLEXITY

Choose median in A[low, ..., high] for the pivot (assume constant operation)

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

Assume: chooseMedian runs in O(1) time

What is the computational complexity of better quicksort?

A. $O(n)$
B. $O(n^2)$
C. $\boldsymbol{O(n \log n)}$
D. $O(2^{\log n})$
E. Naruto!

QUICKSORT PIVOT CHOICES

Choose A[low] for the pivot:

- Bad worst-case performance $O(n^2)$ ☹

If we could choose the median element

- Good worst-case performance $O(n \log n)$ ☺
- **Problem:** choosing the median is not easy

QUICKSORT PIVOT CHOICES

Choose A[low] for the pivot:

- Bad worst-case performance $O(n^2)$ ☹

If we could choose the median element

- Good worst-case performance $O(n \log n)$ ☺
- **Problem:** choosing the median is not easy

What if the split is 10:90?

- pivot splits array into 10% small, 90% large (or vice versa)?

What is the computational complexity of the split is 10:90?

A. $O(1)$
B. $O(n \log n)$
C. $O(n^{9/10})$
D. None of the above

QUICKSORT PIVOT CHOICES

Choose A[low] for the pivot:

- Bad worst-case performance $O(n^2)$ ☹

If we could choose the median element

- Good worst-case performance $O(n \log n)$ ☺
- **Problem:** choosing the median is not easy

What if the split is 10:90?

- pivot splits array into 10% small, 90% large (or vice versa)?

What is the computational complexity of the split is 10:90?

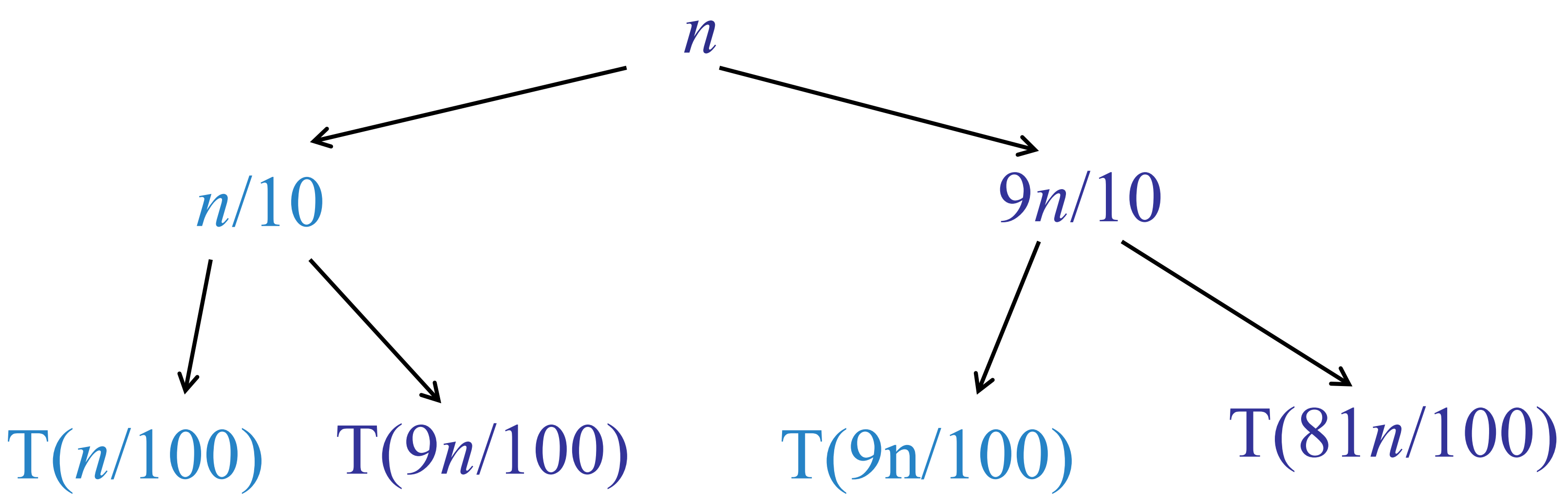
A. $O(1)$
B. $\boldsymbol{O(n \log n)}$
C. $O(n^{9/10})$
D. None of the above

Why?

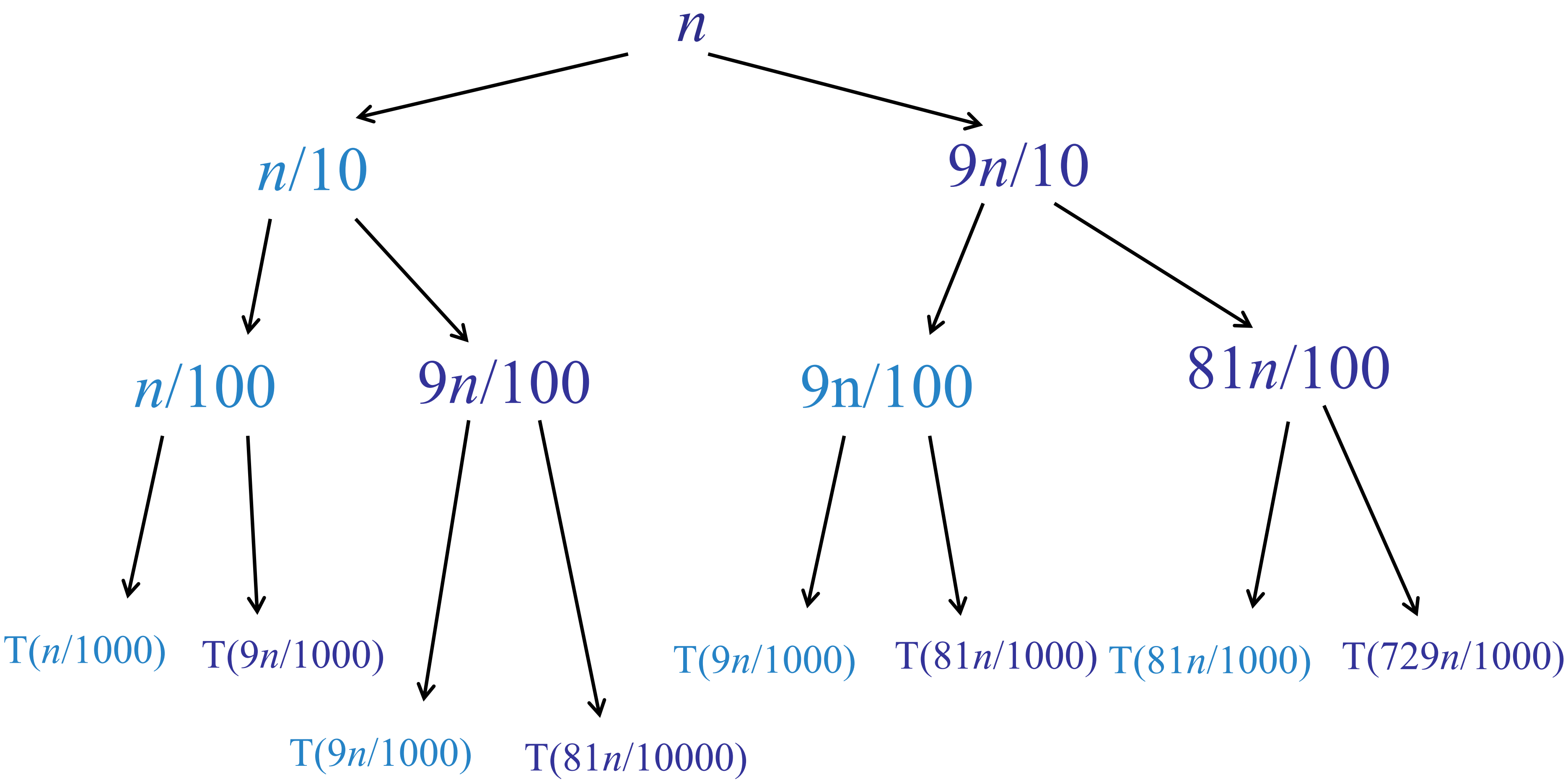
$$T(n) = T(n/10) + T(9n/10) + cn$$

$$
\begin{tikzcd}
 & n \arrow[dl] \arrow[dr] & \\
T(n/10) & & T(9n/10)
\end{tikzcd}
$$

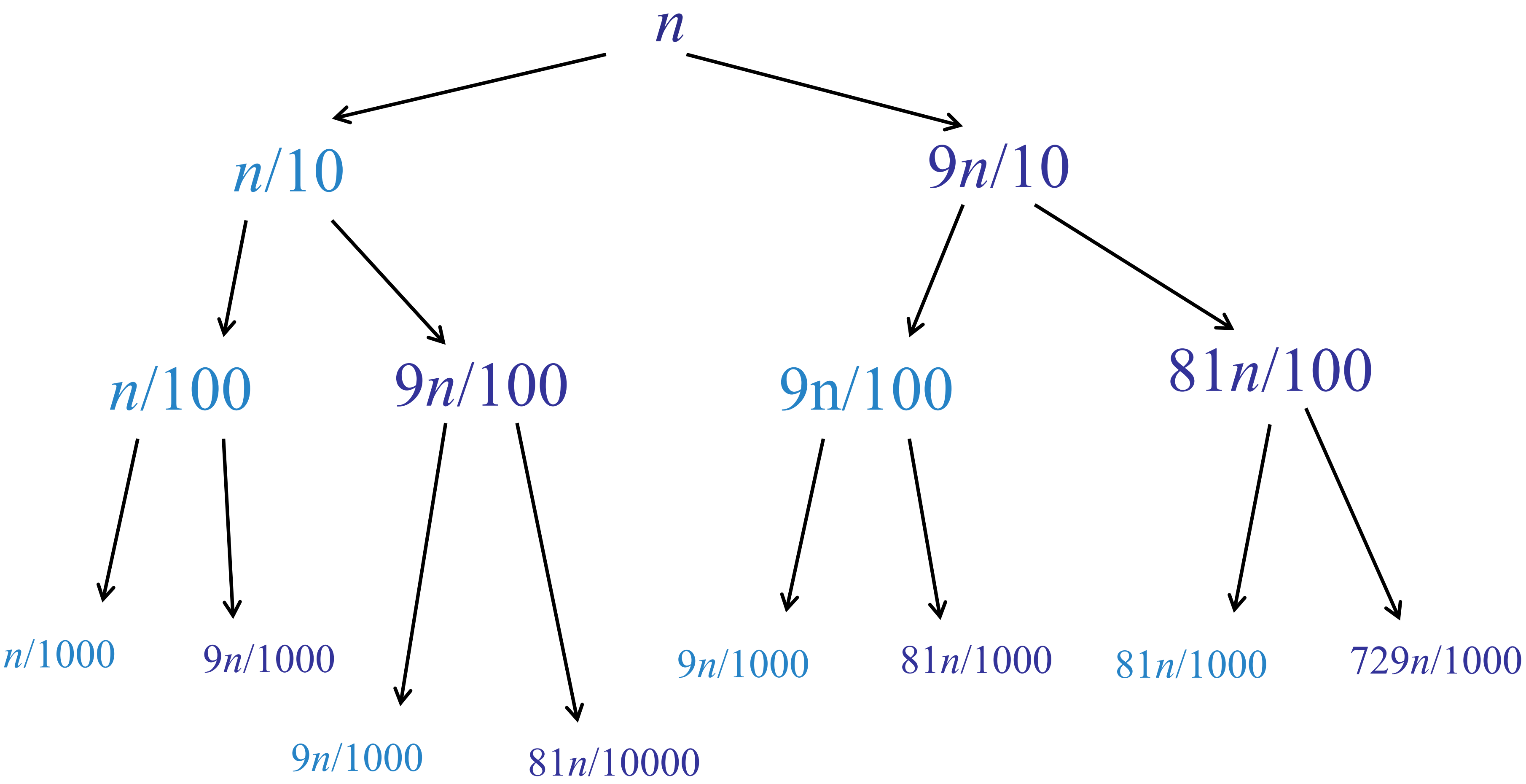
$$T(n) = T(n/10) + T(9n/10) + cn$$



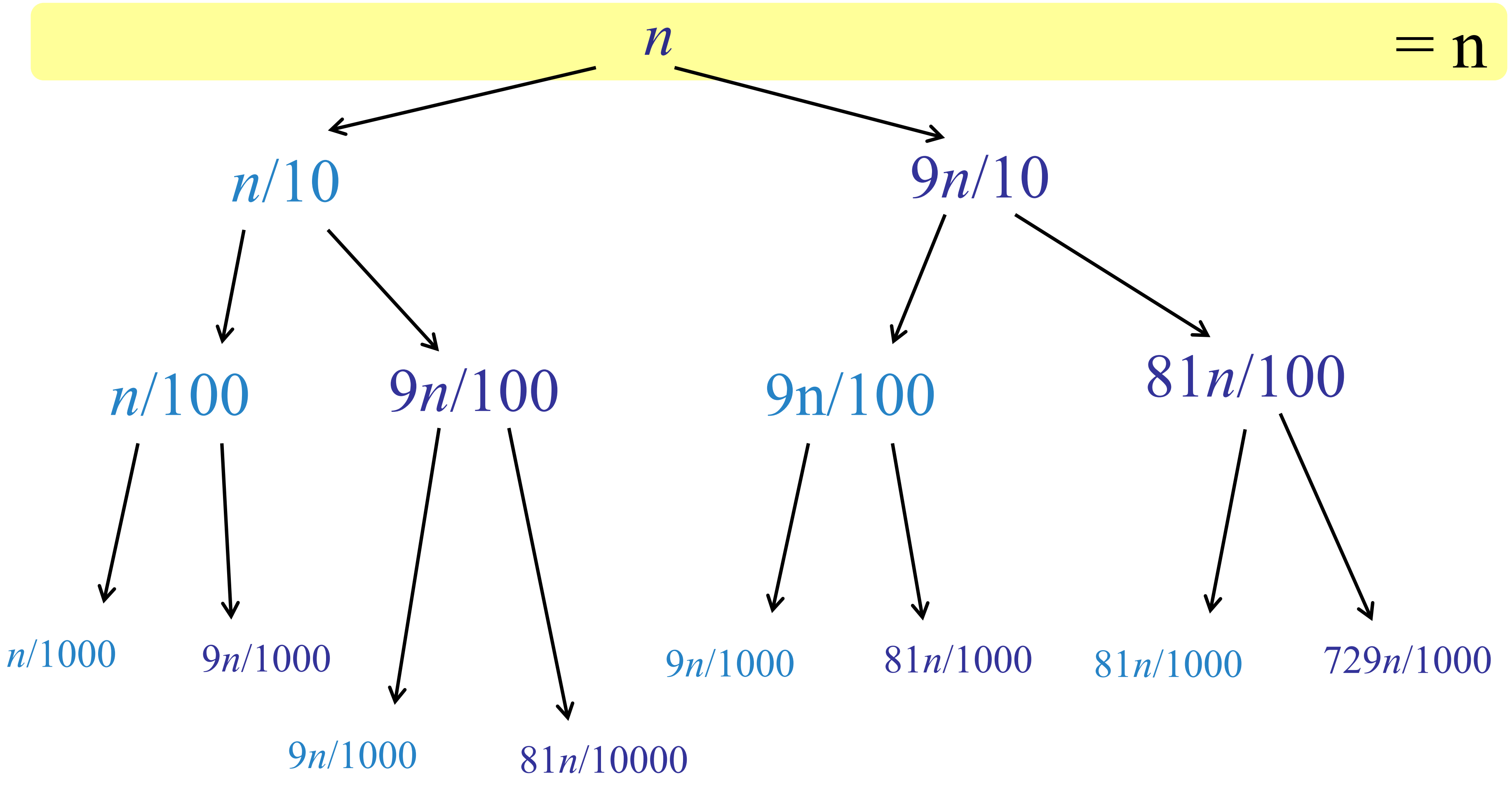
$$T(n) = T(n/10) + T(9n/10) + cn$$



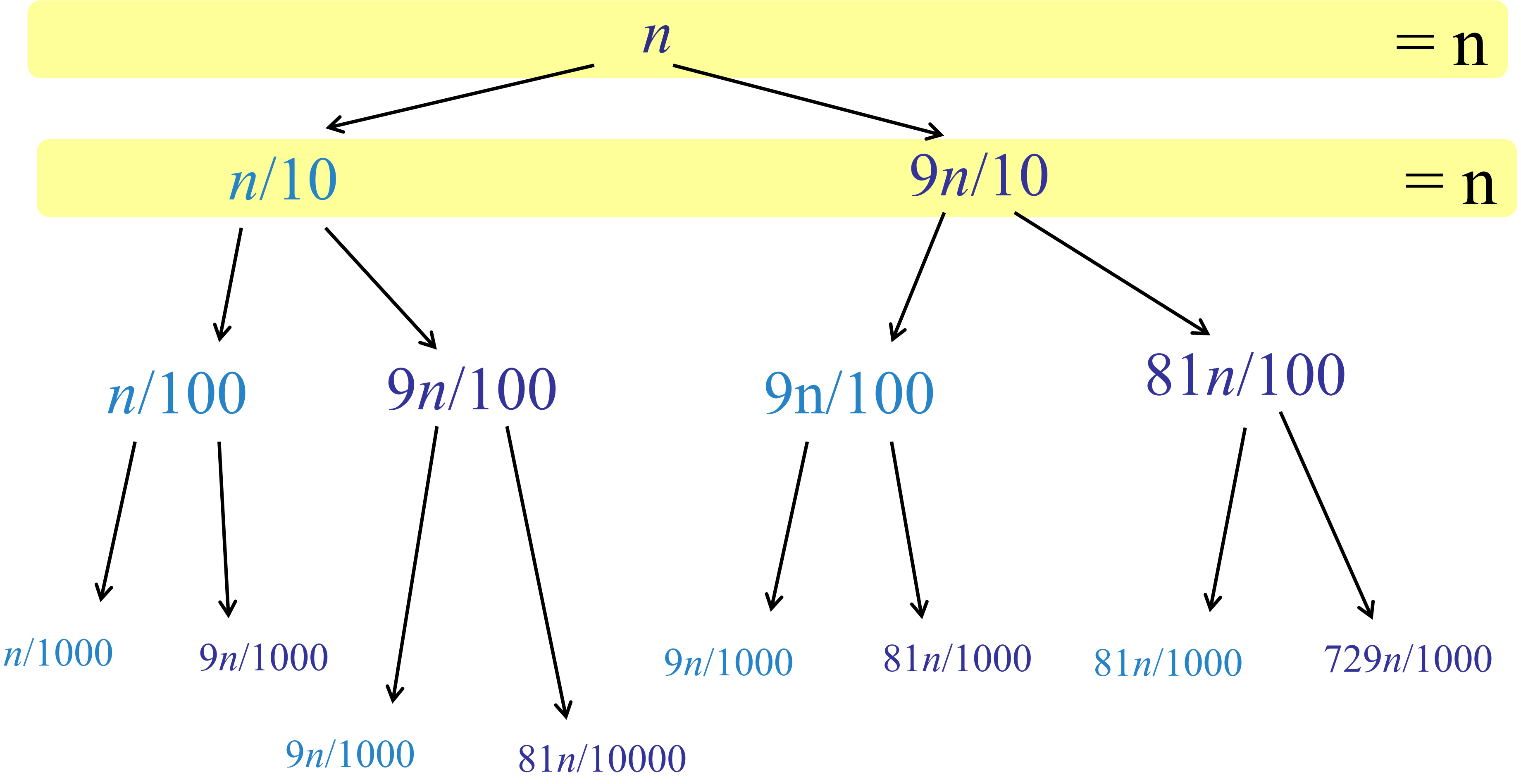
$$T(n) = T(n/10) + T(9n/10) + cn$$



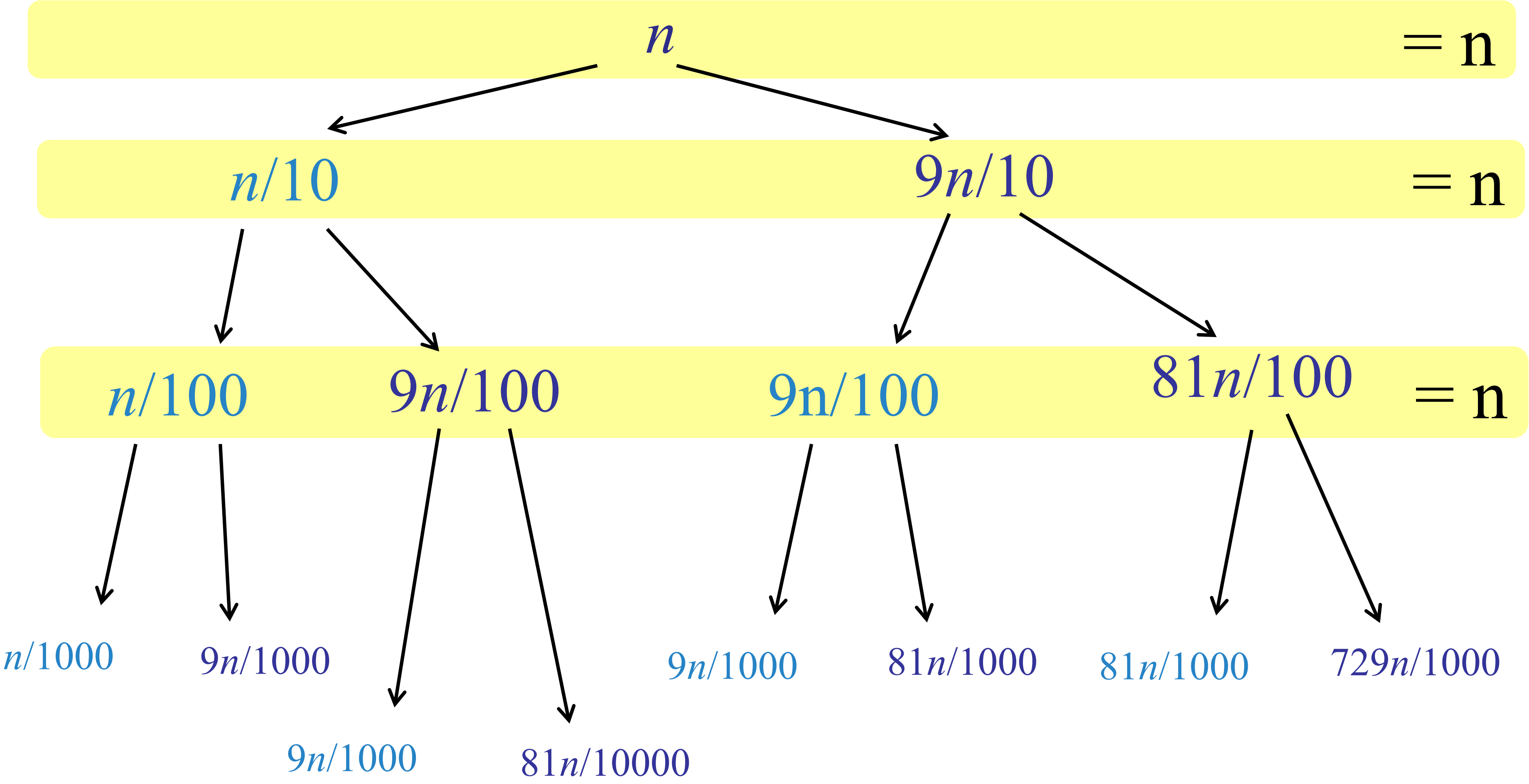
$$T(n) = T(n/10) + T(9n/10) + cn$$



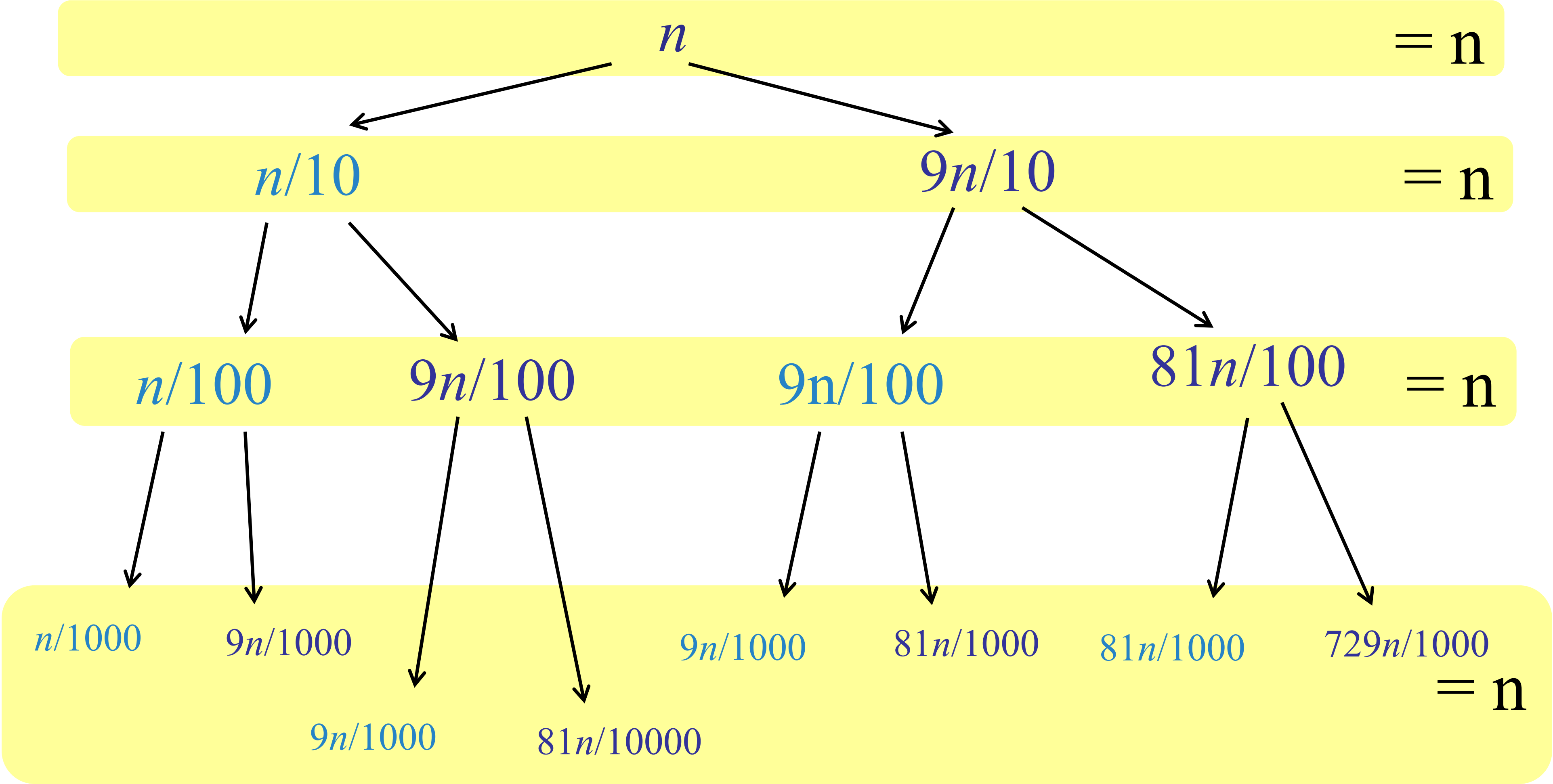
$$T(n) = T(n/10) + T(9n/10) + cn$$



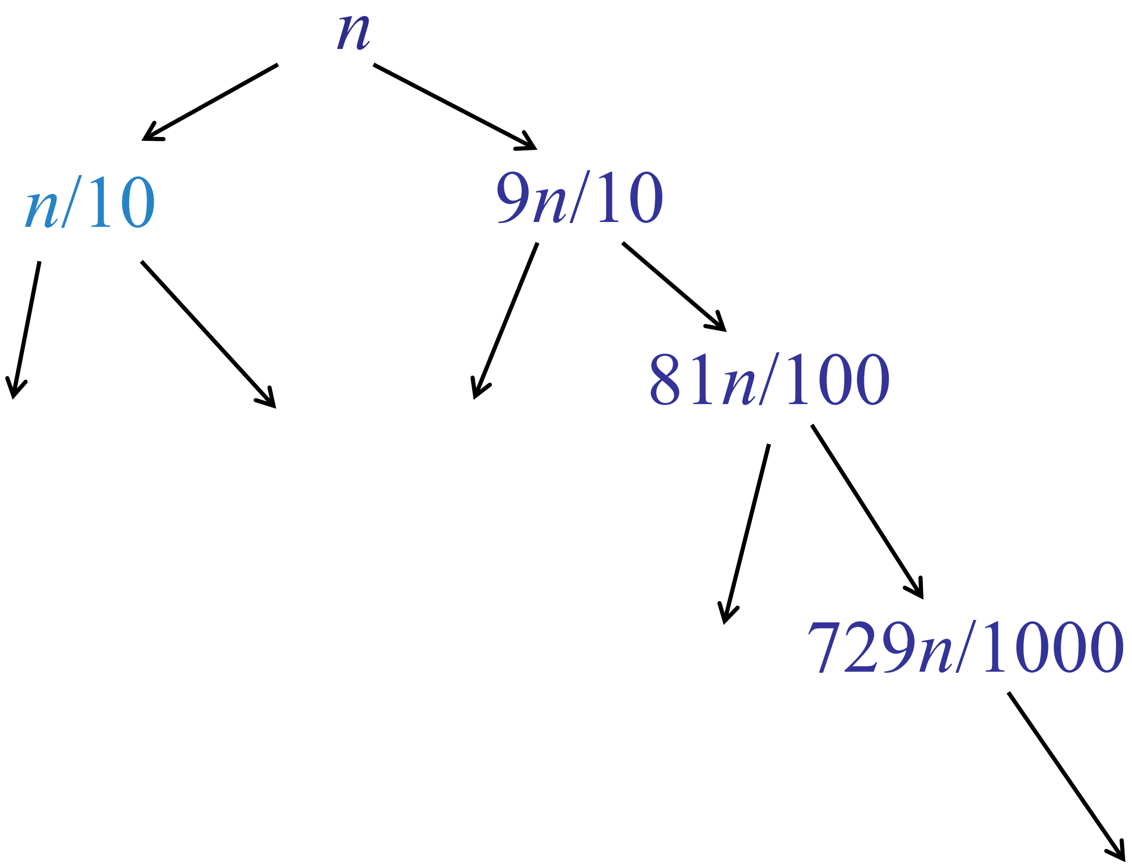
$$T(n) = T(n/10) + T(9n/10) + cn$$



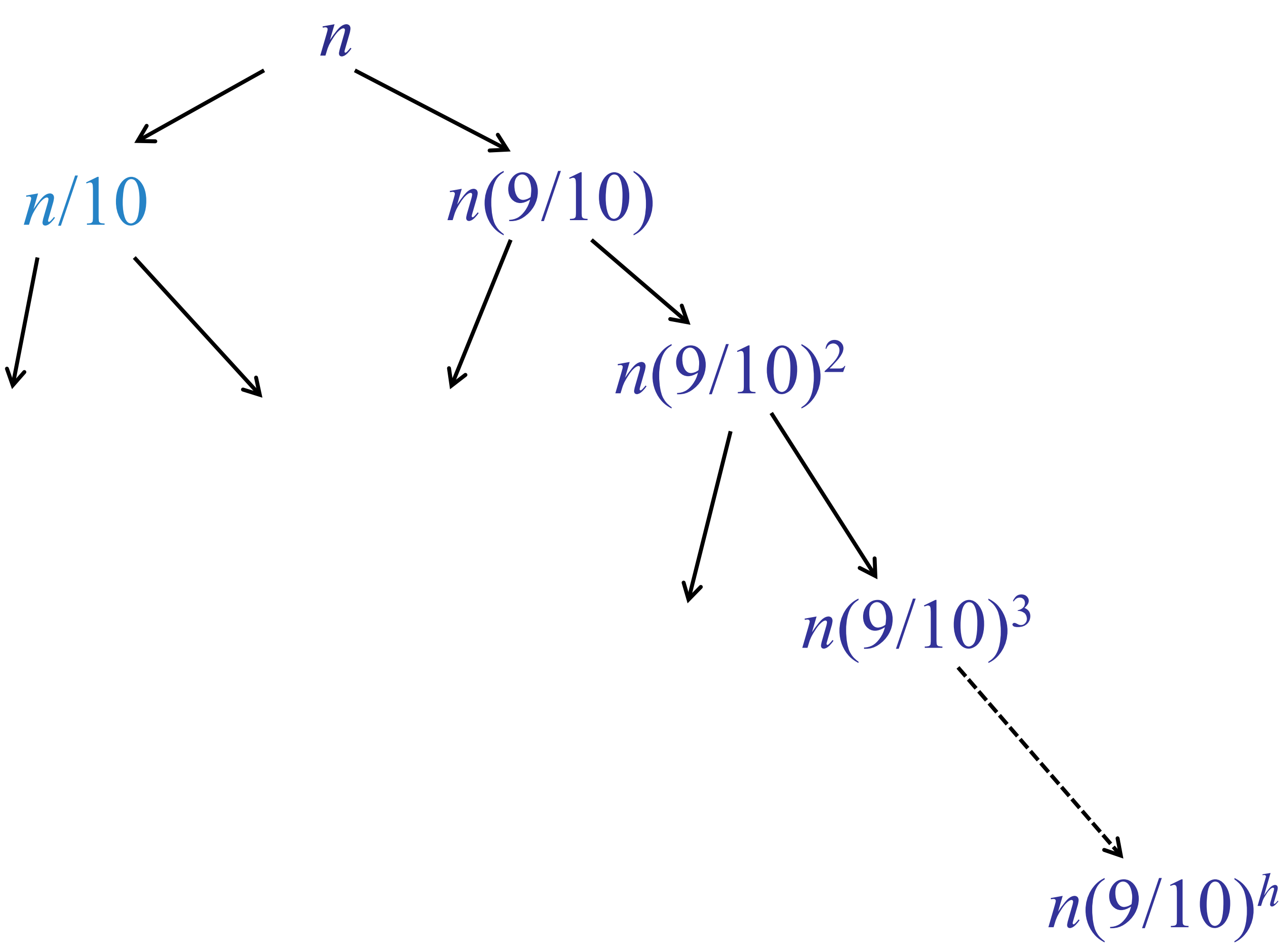
$$T(n) = T(n/10) + T(9n/10) + cn$$



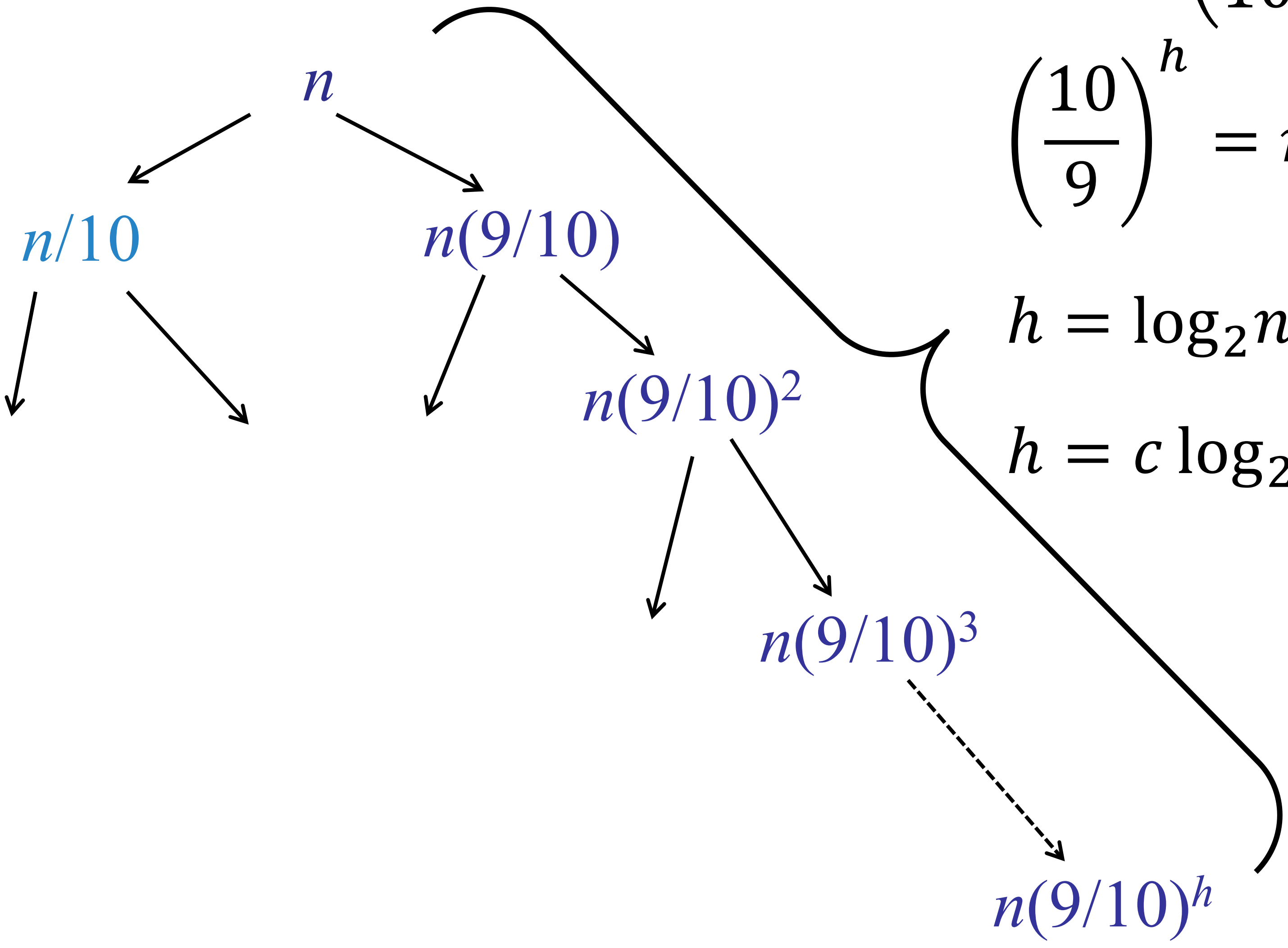
How many levels??



How many levels??



How many levels??



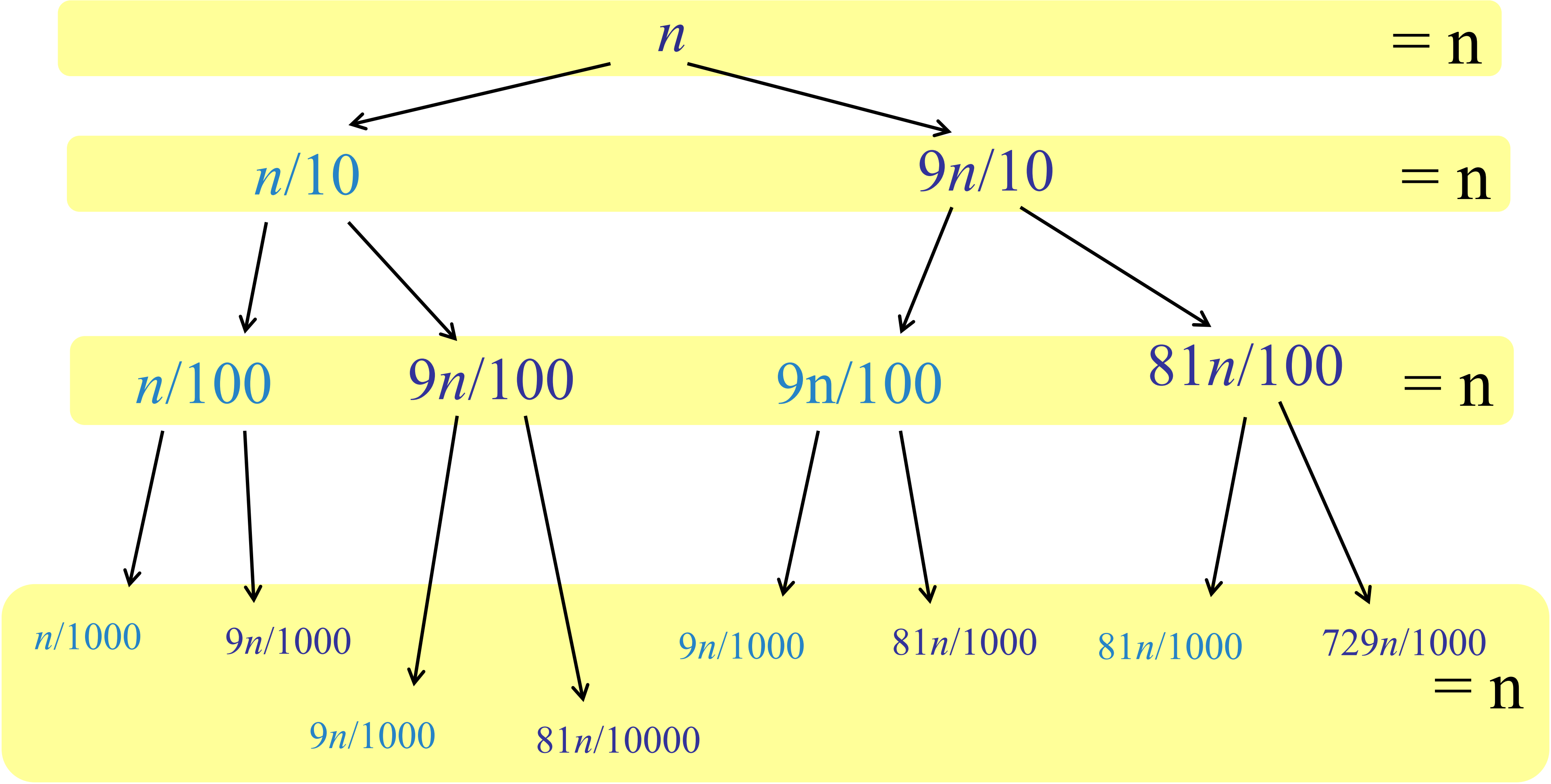
$$1 = n\left(\frac{9}{10}\right)^{h}$$

$$\left(\frac{10}{9}\right)^{h} = n$$

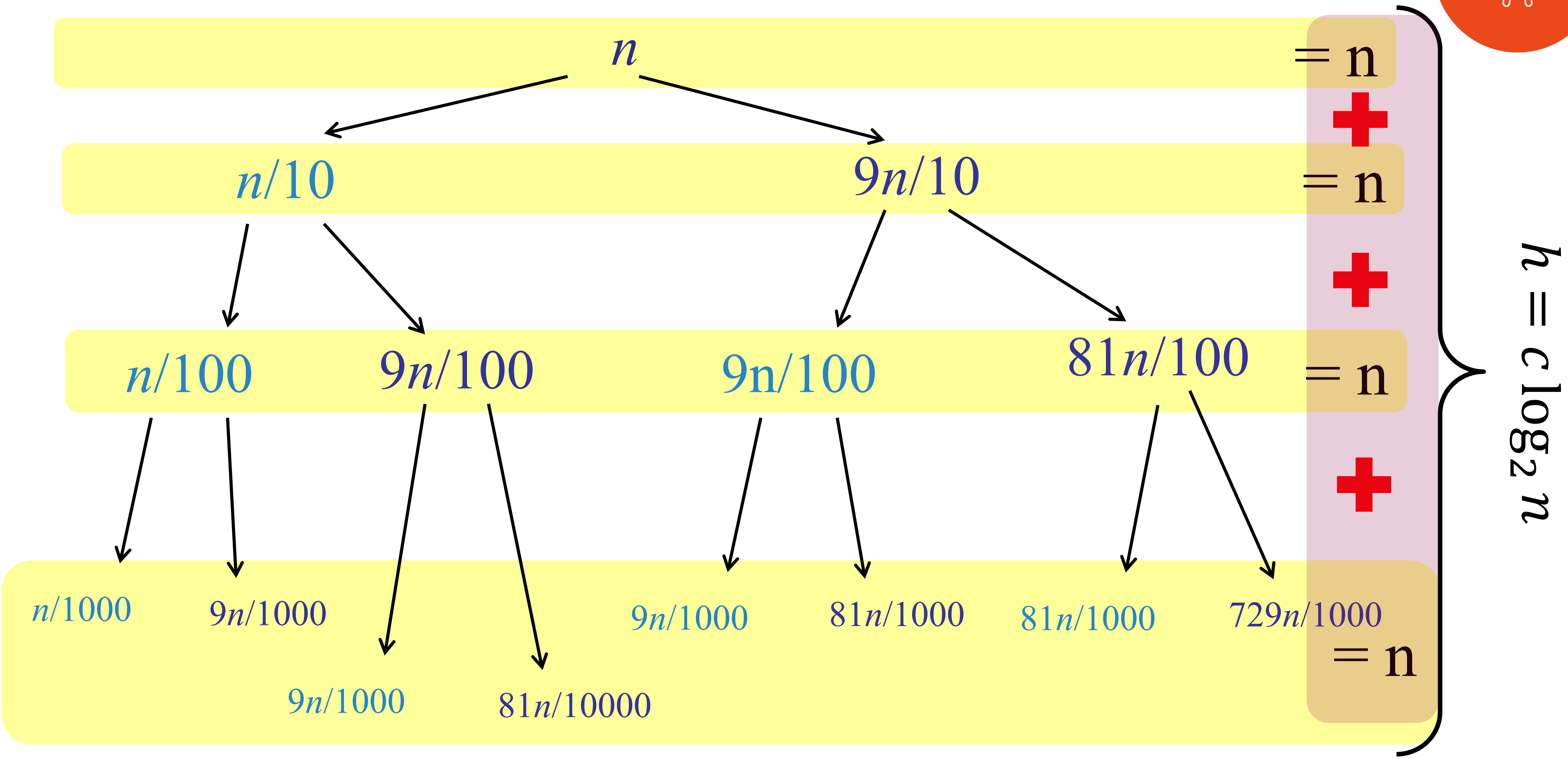
$$h = \log_2 n \Big/ \left(\log_2 \frac{10}{9}\right)$$

$$h = c \log_2 n$$

How many levels??



How many levels??



$$T(n) = cn \log n = O(\text{n}\log n)$$

QUICKSORT PIVOT CHOICES

Choose A[low] for the pivot:

- Bad worst-case performance $O(n^2)$ ☹

If we could choose the median element

- Good worst-case performance $O(n \log n)$ ☺
- **Problem:** choosing the median is not easy

What if the split is 10:90

- Good performance $O(n \log n)$ ☺
- **So, we don't need exact 50:50 splits!**

RANDOMIZED QUICKSORT

```
function RandomizedPartition(A, low, high)
    r = Random(A, low, high)
    swap(A[r], A[low])
    return Partition(A, low, high)

function Quicksort(A, low, high)
    if low < high
        p = RandomizedPartition(A, low, high)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

Just swap a random element with the first element.

Intuition: random element should result in balanced splits on average

PARANOID QUICKSORT (Easier to Analyze)

```
function Quicksort(A, low, high)
    if low < high
        do
              p = RandomizedPartition(A, low, high)
        while not (p > (1/10)n and p < (9/10)n)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

Claim: We only repeat this $O(1)$ times on average!

$$T(n) \leq T(n/10) + T(9n/10) + n * (\text{\# of repeats})$$

Poll Everywhere

https://bit.ly/2LvG9bq

WHAT'S THE PROBABILITY OF PICKING A GOOD PIVOT?

A good pivot if it divides the array into two pieces, each of which is of size at least $n/10$

1	2	3	4	5	6	7	8	9	10

Hint: Consider a specific array as above. Where are the bad pivots? Where are the good ones?

What is the probability of picking a good pivot?

A. 1/10
B. 2/10
C. 8/10
D. 1/n
E. What's the probability of me guessing the right answer?

Poll Everywhere
https://bit.ly/2LvG9bq

WHAT'S THE PROBABILITY OF PICKING A GOOD PIVOT?

A good pivot if it divides the array into two pieces, each of which is of size at least $n/10$

These are **good**!
How many of them are there?

What is the probability of picking a good pivot?

- A. 1/10
- B. 2/10
- **C. 8/10**
- D. 1/n
- E. What's the probability of me guessing the right answer?

Poll Everywhere

https://bit.ly/2LvG9bq

WHAT'S THE PROBABILITY OF PICKING A GOOD PIVOT?

A good pivot if it divides the array into two pieces, each of which is of size at least $n/10$

$$p = 8/10$$

What is the probability of picking a good pivot?

A. 1/10
B. 2/10
C. 8/10
D. 1/n
E. What's the probability of me guessing the right answer?

EXPECTED NUMBER OF REPEATS

If $p = 8/10$ then $E[\text{time to first hit }] = \frac{1}{p} = \frac{10}{8} < 2$

Expected time to first hit:

$$\begin{aligned} E[y] &= p + (1-p)(E[y]+1) \\ &= p + E[y] + 1 - pE[y] - p \\ &= (1-p)E[y] + 1 \end{aligned}$$

Rearranging and solving yields:

$$E[y] = 1/p$$

PARANOID QUICKSORT (Easier to Analyze)

```
function Quicksort(A, low, high)
    if low < high
        do
            p = RandomizedPartition(A, low, high)
        while not (p > (1/10)n and p < (9/10)n)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

Claim: We only repeat this $O(1)$ times on average!

$$T(n) \leq T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + n(\text{\# of repeats})$$

PARANOID QUICKSORT (Easier to Analyze)

```
function Quicksort(A, low, high)
    if low < high
        do
            p = RandomizedPartition(A, low, high)
        while not (p > (1/10)n and p < (9/10)n)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

Expected # of repeats < 2

$$T(n) \leq T\left(\frac{n}{10}\right) + T\left(\frac{9n}{10}\right) + 2n = O(n \log n)$$

SUMMARY

We don't need median splits!

Splitting 10:90 is good enough!

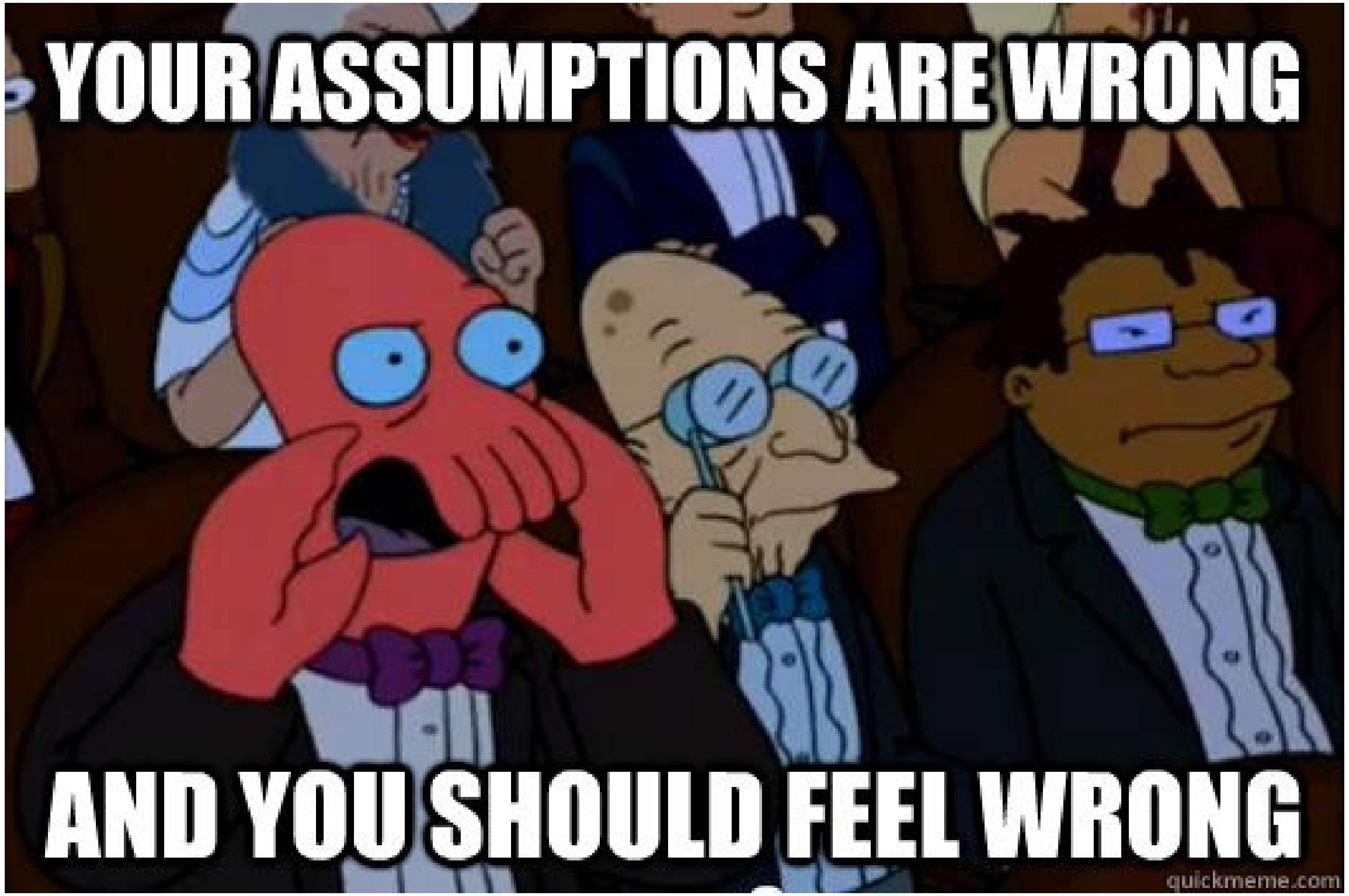
$$T(n) = O(n \log n)$$

QUICKSORT V.S. MERGESORT

	insertion sort (n^2)			mergesort (n log n)			quicksort (n log n)		
computer	thousand	million	billion	thousand	million	billion	thousand	million	billion
home	instant	2.8 hours	317 years	instant	1 second	18 min	instant	0.6 sec	12 min
super	instant	1 second	1 week	instant	instant	instant	instant	instant	instant

From Sedgewick and Wayne's Princeton quicksort lecture slides:
http://algs4.cs.princeton.edu/lectures/23Quicksort.pdf

TIDYING UP


YOUR ASSUMPTIONS ARE WRONG
AND YOU SHOULD FEEL WRONG
quickmeme.com

Poll Everywhere

https://bit.ly/2LvG9bq

QUICKSORT PRECONDITIONS

Preconditions:

- **PRE-1**: A is an array
- **PRE-2:** $0 \leq low \leq high \leq A.length$
- **PRE-3:** A has no duplicates

Which precondition is likely to be the most unreasonable?

A. PRE-1
B. PRE-2
C. PRE-3
D. All of the above!
E. None of the above!
F. What's a precondition???

Poll Everywhere
https://bit.ly/2LvG9bq

QUICKSORT PRECONDITIONS

Preconditions:

- **PRE-1**: A is an array
- **PRE-2:** $0 \leq low \leq high \leq A.length$
- **PRE-3:** A has no duplicates

Which precondition is likely to be the most unreasonable?

A. PRE-1
B. PRE-2
C. PRE-3
D. All of the above!
E. None of the above!
F. What's a precondition???

DUPLICATES ARE EVERYWHERE

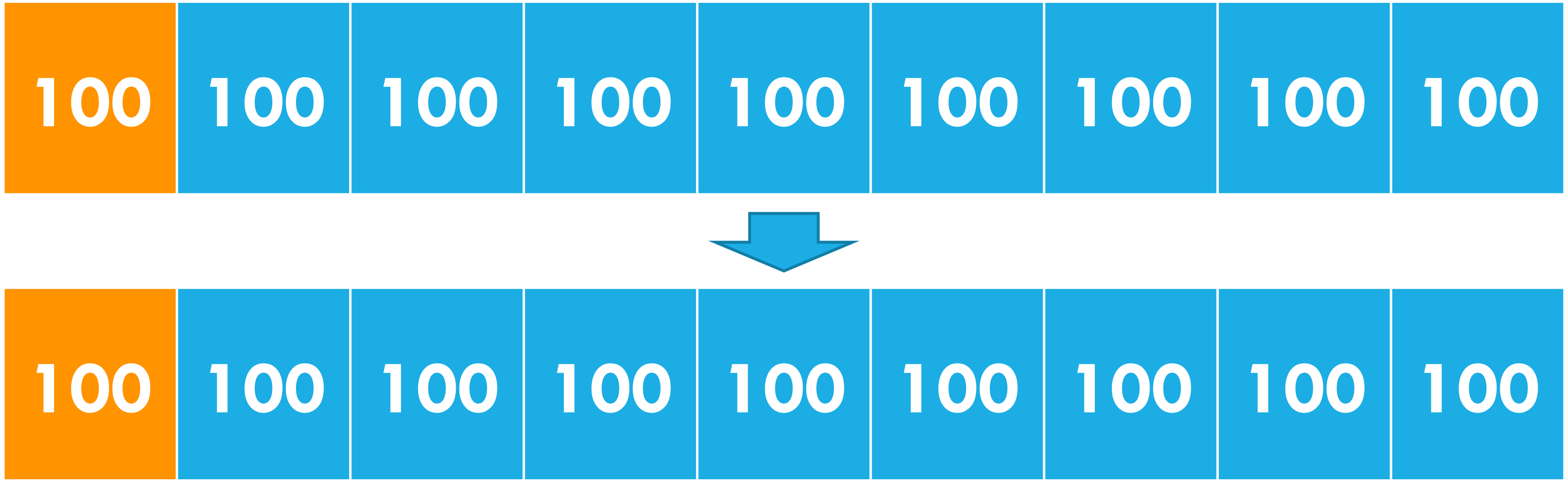
Sort

- Class by age
- Flights by departure city
- Products by Manufacturer
- etc.

The two-way partitioning algorithms handle duplicates (or can be slightly modified to do so) **BUT** …

DUPLICATES: EXTREME CASE

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
          m++
          swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```



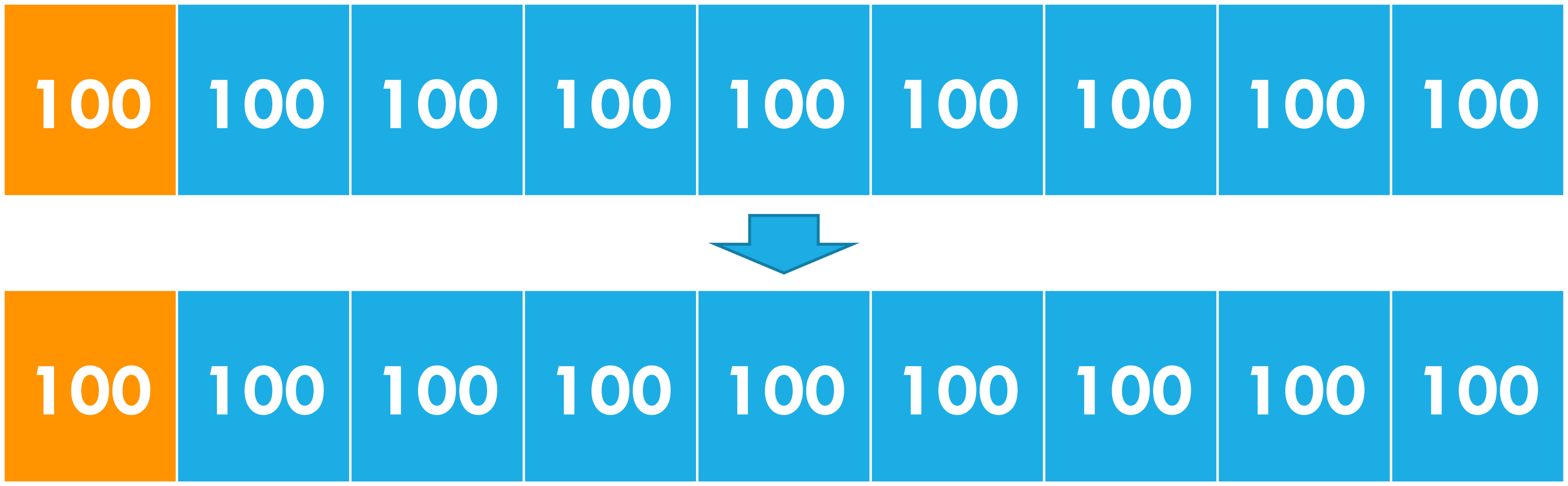
RANDOMIZED QUICKSORT

```
function Partition(A, low, high)
    v = A[low]
    m = low
    i = low
    for i = (low + 1) to high
        if A[i] < v
           m++
           swap(A[i], A[m])
    swap(A[m], A[low])
    return m
```

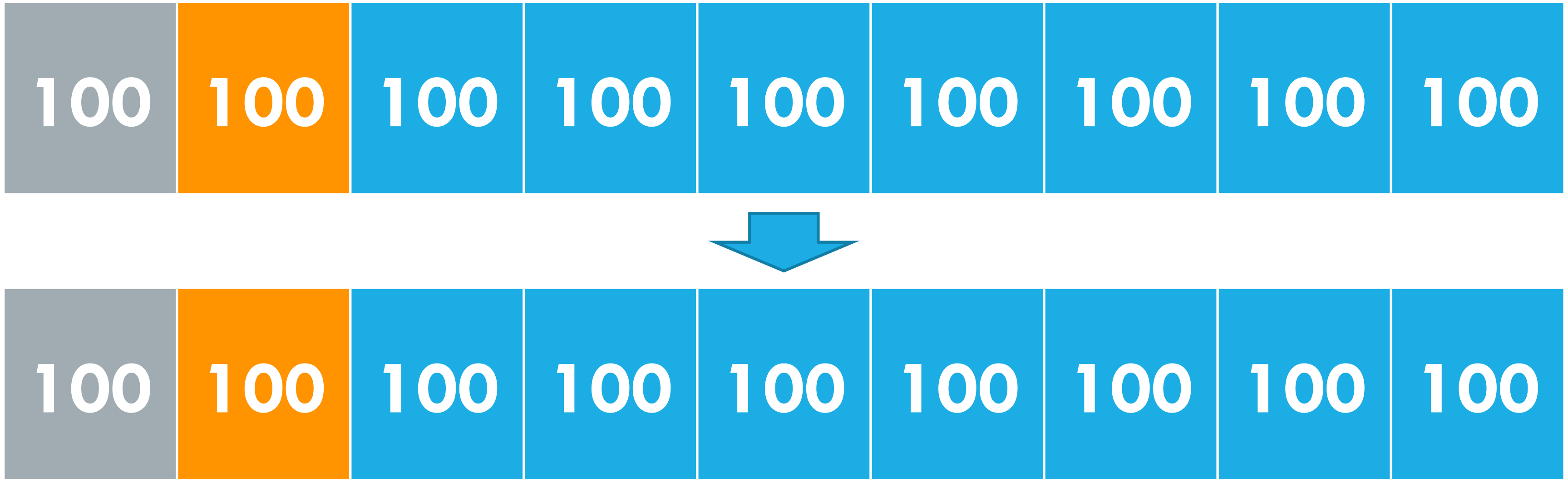
```
function RandomizedPartition(A, low, high)
    r = Random(A, low, high)
    swap(A[r], A[low])
    return Partition(A, low, high)

function Quicksort(A, low, high)
    if low < high
        p = RandomizedPartition(A, low, high)
        Quicksort (A, low, p-1)
        Quicksort (A, p+1, high)
```

100	100	100	100	100	100	100	100	100

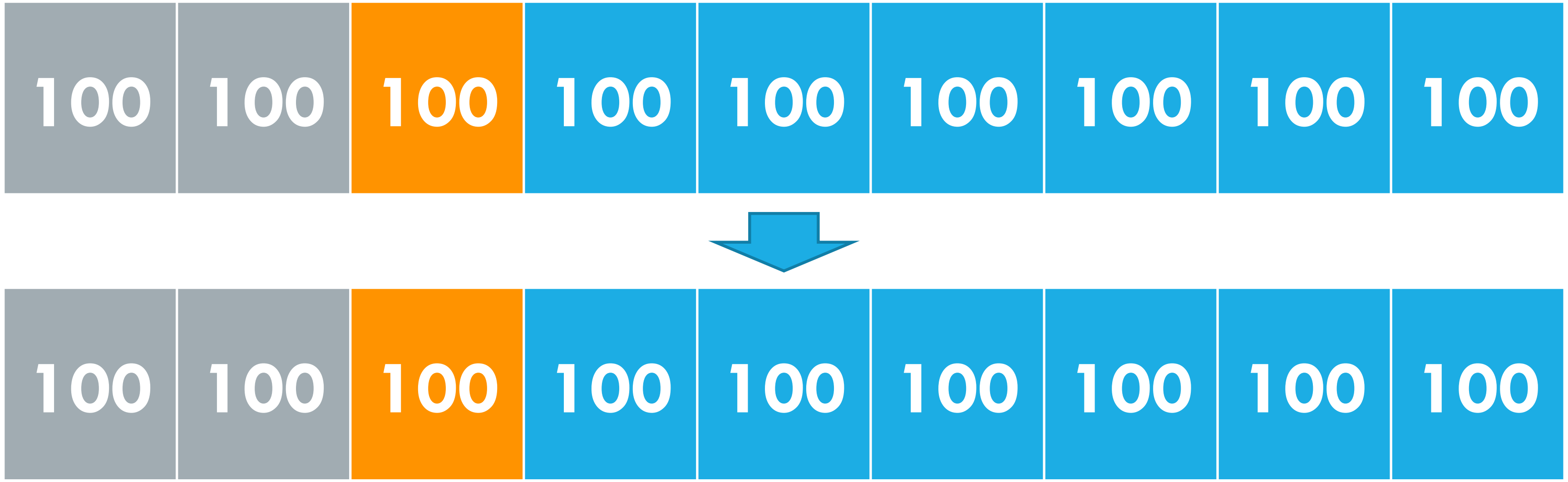
DUPLICATES: EXTREME CASE



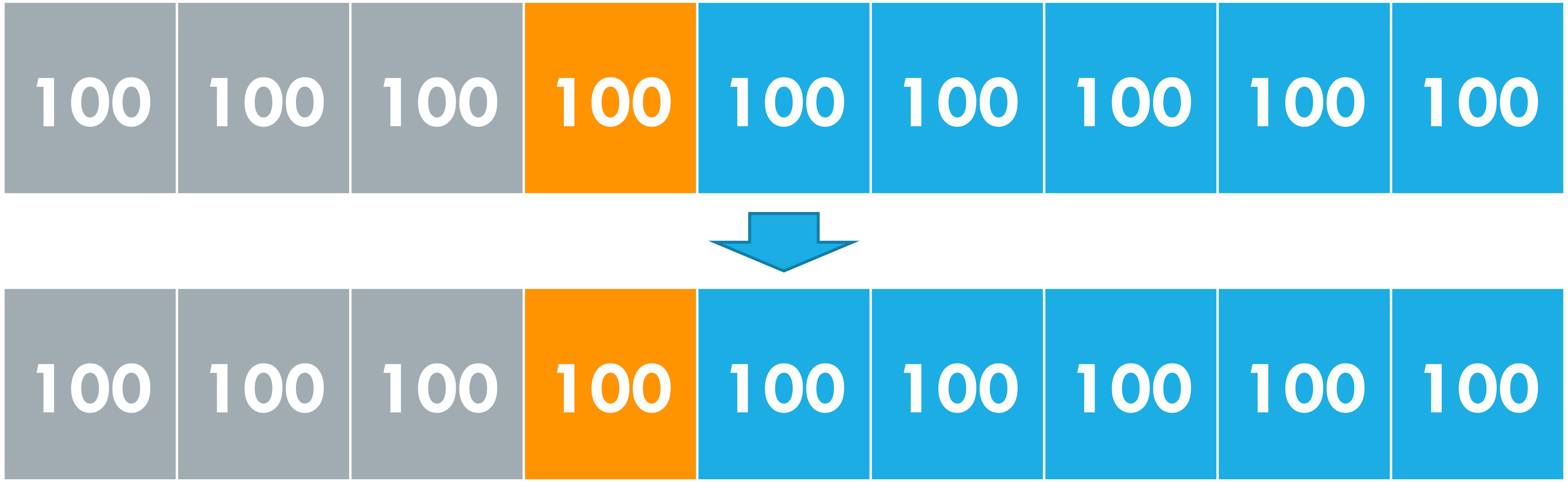
DUPLICATES: EXTREME CASE



DUPLICATES: EXTREME CASE



DUPLICATES: EXTREME CASE



DUPLICATES: EXTREME CASE

Uh oh... I see a (bad) pattern

100	100	100	100	100	100	100	100	100

100	100	100	100	100	100	100	100	100

A “BEAUTIFUL BUG REPORT”

```
We found that qsort is unbearably slow on "organ-pipe" inputs like "01233210":

main (int argc, char**argv) {
   int n = atoi(argv[1]), i, x[100000];
   for (i = 0; i < n; i++)
      x[i] = i;
   for ( ; i < 2*n; i++)
      x[i] = 2*n-i-1;
   qsort(x, 2*n, sizeof(int), intcmp);
}

Here are the timings on our machine:
$ time a.out 2000
real    5.85s
$ time a.out 4000
real   21.64s
$time a.out 8000
real   85.11s
```

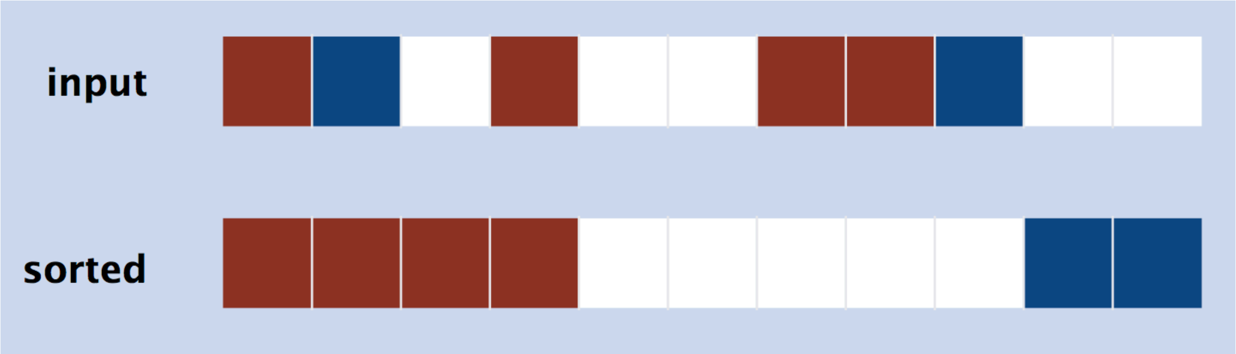
A beautiful bug report. [Allan Wilks and Rick Becker, 1991]

Quicksort was taking minutes instead of seconds because of duplicates.

[From Sedgewick and Wayne’s Princeton quicksort lecture slides: http://algs4.cs.princeton.edu/lectures/23Quicksort.pdf]

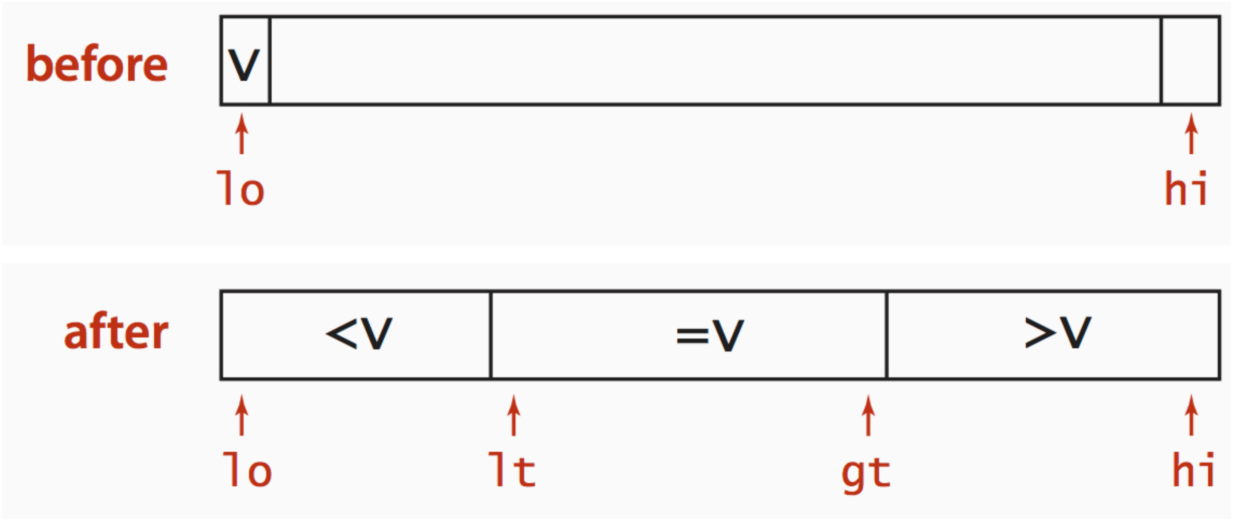
THE DUTCH NATIONAL FLAG PROBLEM

Courtesy of Dijkstra (who will will meet again after recess week)



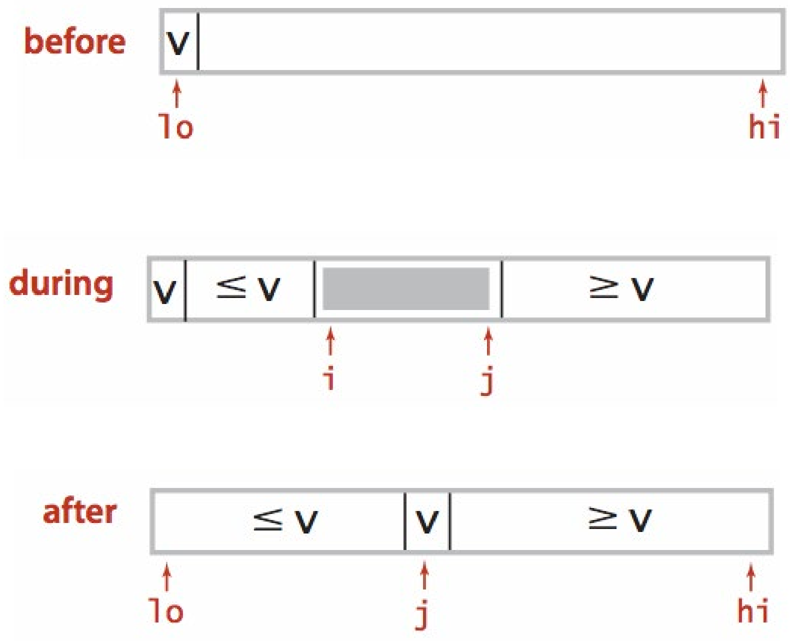
THE DUTCH NATIONAL FLAG PROBLEM

Courtesy of Dijkstra (who will will meet again after recess week)

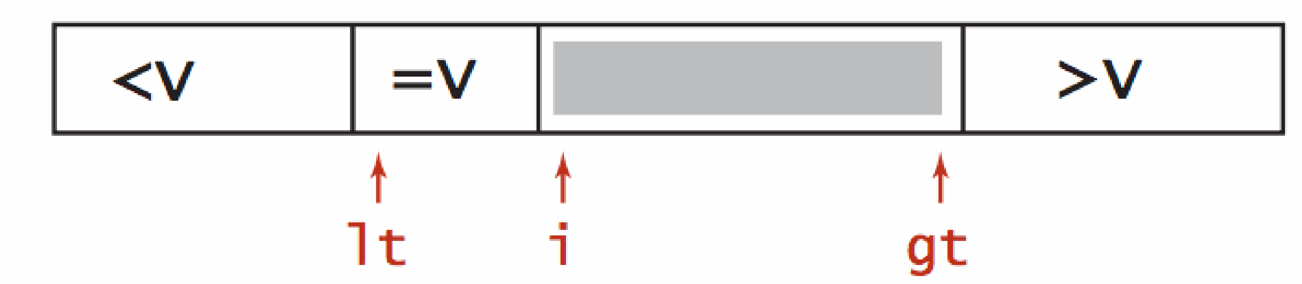


HOARE'S PARTITION ALGORITHM

```
function Partition(A, low, high)
    v = A[low]
    i = low+1;
    j = high;
    while i < j
        while (A[i] < v) and (i <= high) i++
        while (A[j] > v) and (j >= low) j--
        if (i<j) swap(A[i], A[j])
    swap(A[j], A[low])
    return i
```

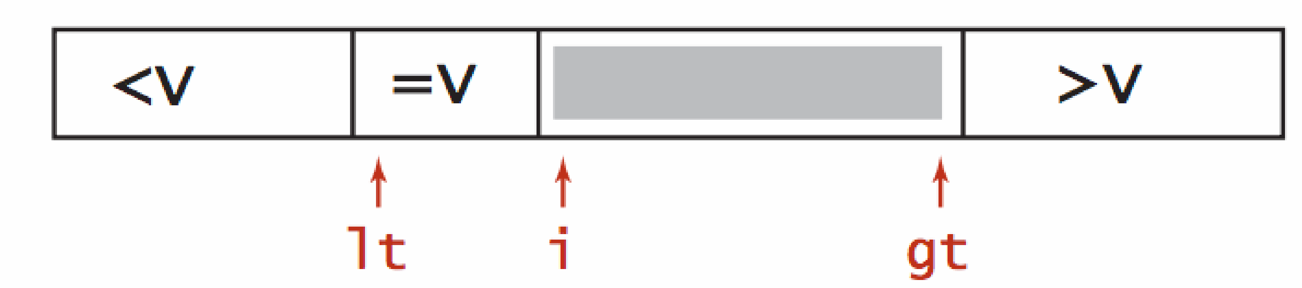


3 WAY PARTITION: THE IDEA



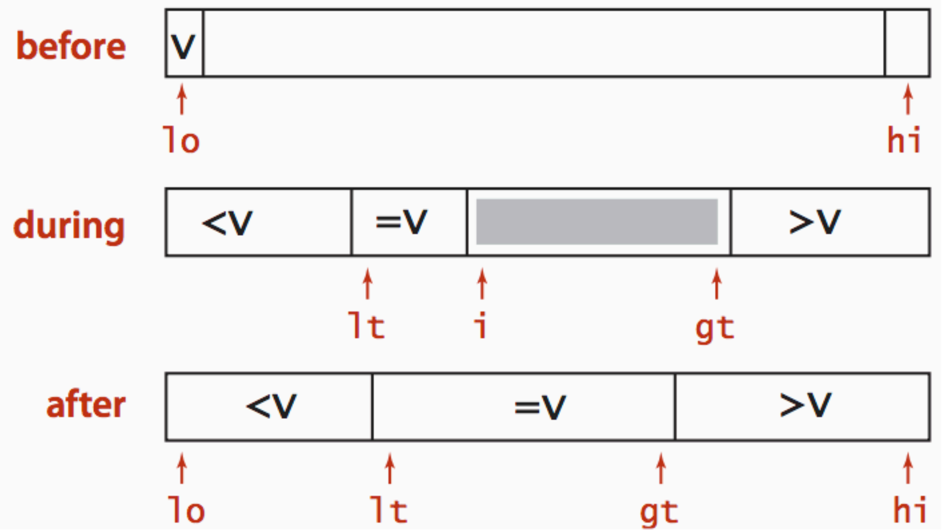
3 WAY PARTITION: THE IDEA

```
while (i <= gt) {
    if (A[i]< v) swap(A, lt++, i++);
    else if (A[i]> v) swap(A, i, gt--);
    else  i++;
}
```



3 WAY PARTITION + QUICKSORT

```
quicksort(int[] A, int lo, int hi) {
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    int v = A[lo];
    int i = lo + 1;
    while (i <= gt) {
        if      (A[i]< v) swap(A, lt++, i++);
        else if (A[i]> v) swap(A, i, gt--);
        else  i++;
    }

    quicksort(A, lo, lt-1);
    quicksort(A, gt+1, hi);
}
```



Modified from http://algs4.cs.princeton.edu/23quicksort/Quick3way.java.html

Poll Everywhere
https://bit.ly/2LvG9bq

QUICKSORT: STABLE?

A sorting algorithm is stable if two objects with equal keys appear in the same order in the output as in input.

- No changing places if you have the same key!

Why? Sort by lastname only:

input	stable sort	unstable sort
soh, aaron	**chan**, hazel	**chan**, hazel
soh, harold	**soh**, aaron	**soh**, harold
chan, hazel	**soh**, harold	**soh**, aaron

Is quicksort stable?

A. Yesssssss.
B. Noooooo
C. Who knows?
D. Stable as an ikea table.

Poll Everywhere

https://bit.ly/2LvG9bq

QUICKSORT: STABLE?

A sorting algorithm is stable if two objects with equal keys appear in the same order in the output as in input.

- No changing places if you have the same key!

Why? Sort by lastname only:

input	stable sort	unstable sort
soh, aaron	**chan**, hazel	**chan**, hazel
soh, harold	**soh**, aaron	**soh**, harold
chan, hazel	**soh**, harold	**soh**, aaron

Is quicksort stable?

A. Yesssssss.

B. Noooooo

C. Who knows?

D. Stable as an ikea table.

Consider sorting by last name:

[Cat D., Soh H, Bat M., Soh A., Ali G.]

Pivot

QUESTIONS?

LEARNING OUTCOMES

By the end of this session, you should be able to:

- **Describe** the **quicksort** algorithm and how it works.
- **Analyze the worst-case and average-case performance** of the quicksort algorithm

OTHER TAKE AWAYS

Divide & Conquer:

- Split
- Solve
- Combine

Set up your invariances to "match" your postconditions

Bad worst case does not mean bad algorithm!

QUESTIONS?

QUIZ 1

On **Sept 4th**

Please don't be late.

Covers everything up to **Quicksort** (today's lecture).

BEFORE NEXT LECTURE

Go to Visualgo.net and do the Binary Heap Module:

- https://visualgo.net/en/heap
- Review: 8 (Heapsort)