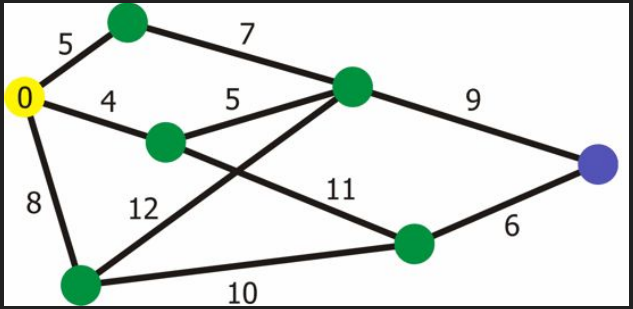
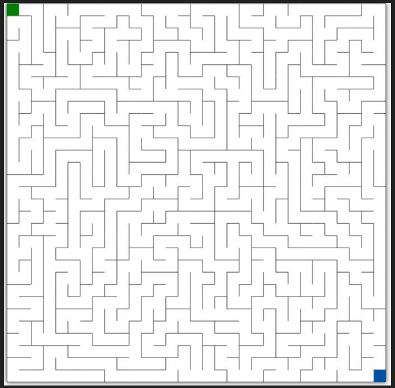




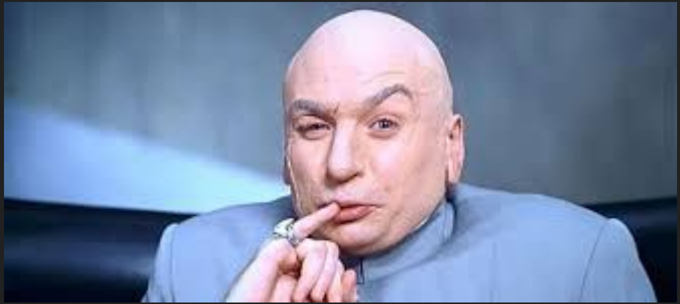
CS 5008/5009

Data Structures, Algorithms, and Their Applications Within Computer Systems

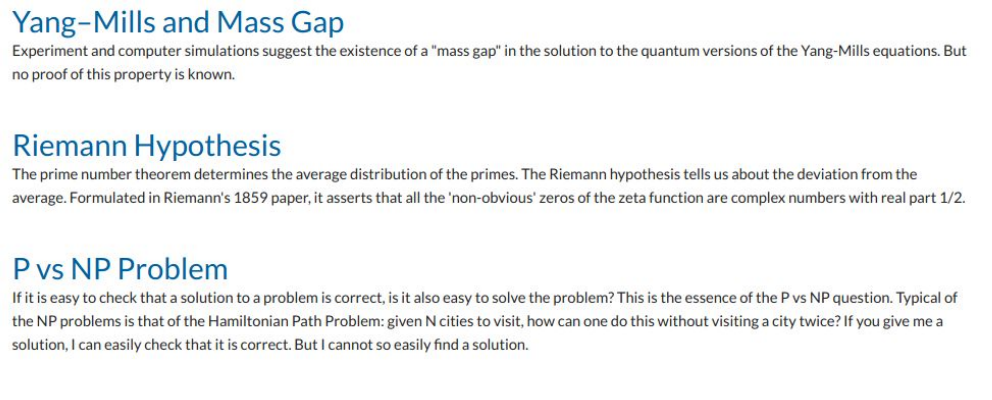
Dr. Mike Shah

Pre-Class Warmup

- Want one million dollars?
  - http://www.claymath.org/millennium-problems
- P vs NP is a problem you will learn about more in CS 5800.
  - The idea has to do with proving or disproving that there exists polynomial solutions to intractable problems.
  - The motivation is that can earn you a million dollars if you solve it!



“1 million dollars--muahaha”, Dr. Evil (Attempting to blackmail the government)
Movie Reference: Austin Powers [Youtube Clip]



Yang–Mills and Mass Gap

Experiment and computer simulations suggest the existence of a "mass gap" in the solution to the quantum versions of the Yang-Mills equations. But no proof of this property is known.

Riemann Hypothesis

The prime number theorem determines the average distribution of the primes. The Riemann hypothesis tells us about the deviation from the average. Formulated in Riemann's 1859 paper, it asserts that all the 'non-obvious' zeros of the zeta function are complex numbers with real part 1/2.

P vs NP Problem

If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

REC

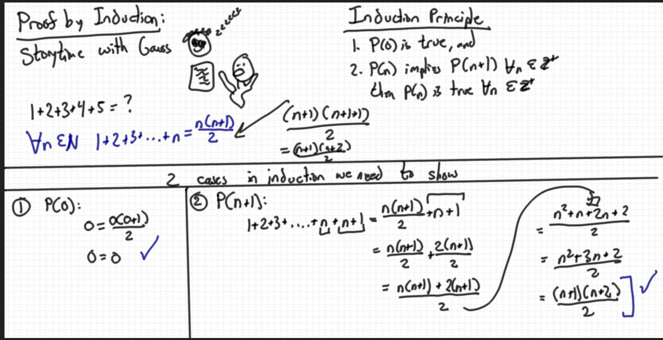
Note to self: Start audio recording of lecture :)
(Someone remind me if I forget!)

Course Logistics

- HW9 due next Friday anywhere on earth
- Lab 9 in your repos and due next week
- Make sure you are running your code on the servers whenever possible
  - ssh username@login.khoury.neu.edu

Last Time

- Primer on Proofs
    - What is a proof?
        - A convincing argument
    - Proof by case
    - Proof by induction
    - Invariant Proof
- Big-O
    - Asymptotic complexity
    - Proving Bounds



Proof by Induction:

Storytime with Gauss

$1+2+3+4+5=?$

$\forall n \in N \quad 1+2+3+\ldots+n=\frac{n(n+1)}{2}$

$\frac{(n+1)(n+1+1)}{2}$

$=\frac{(n+1)(n+2)}{2}$

Induction Principle

1. $P(0)$ is true, and
2. $P(n)$ implies $P(n+1)$ $\forall n \in \mathbb{Z}^+$ then $P(n)$ is true $\forall n \in \mathbb{Z}^+$

2 cases in induction we need to show

① $P(0)$:

$0=\frac{0(0+1)}{2}$

$0=0$ ✓

② $P(n+1)$:

$1+2+3+\ldots+\underbrace{n}+\underbrace{n+1}=\frac{n(n+1)}{2}+\overbrace{n+1}$

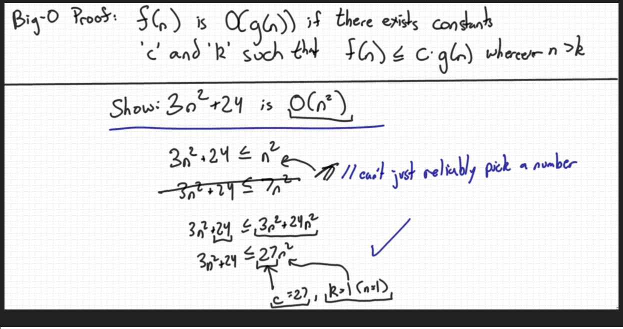
$=\frac{n(n+1)}{2}+\frac{2(n+1)}{2}$

$=\frac{n(n+1)+2(n+1)}{2}$

$=\frac{n^2+n+2n+2}{2}$

$=\frac{n^2+3n+2}{2}$

$=\frac{(n+1)(n+2)}{2}$ ✓



Big-O Proof: $f(n)$ is $O(g(n))$ if there exists constants 'c' and 'k' such that $f(n) \le c \cdot g(n)$ wherever $n > k$

Show: $3n^2+24$ is $\underbrace{O(n^2)}$

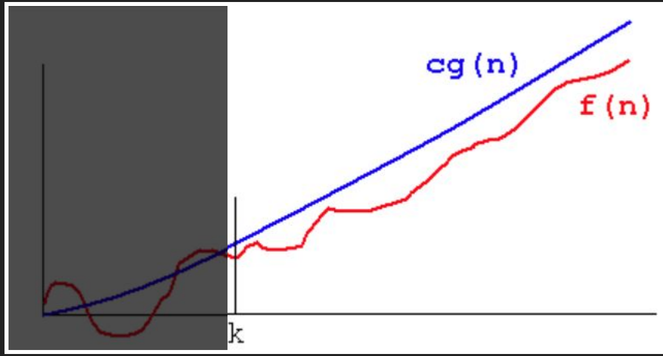
$3n^2+24 \le n^2$ // can't just reliably pick a number

~~$3n^2+24 \le 7n^2$~~

$3n^2+\underbrace{24} \le \underbrace{3n^2+24n^2}$ ✓

$3n^2+24 \le \underbrace{27}n^2$

$\underbrace{c=27}, \underbrace{k>1\ (n=1)}$

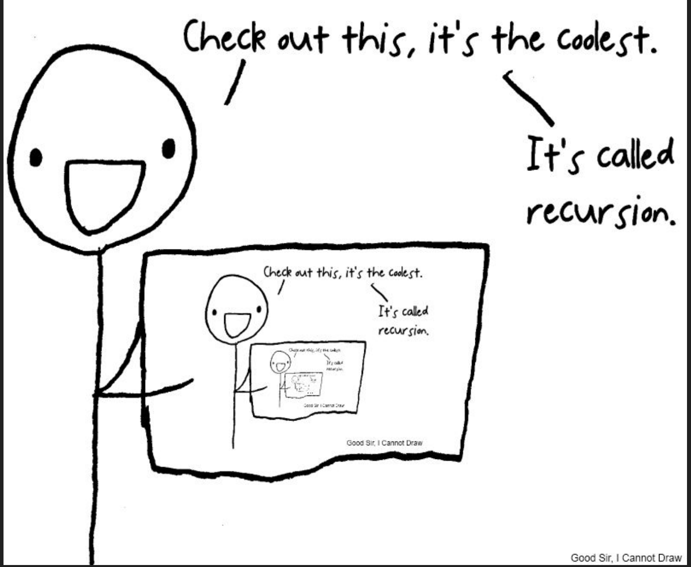


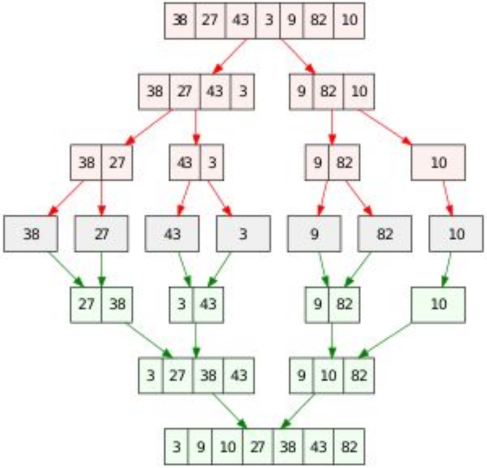
HW structure discussion

live drawing/coding/discussion

Today

- Recursion
    - The 'call stack'
- Merge Sort
- Randomized Algorithms -- Quicksort



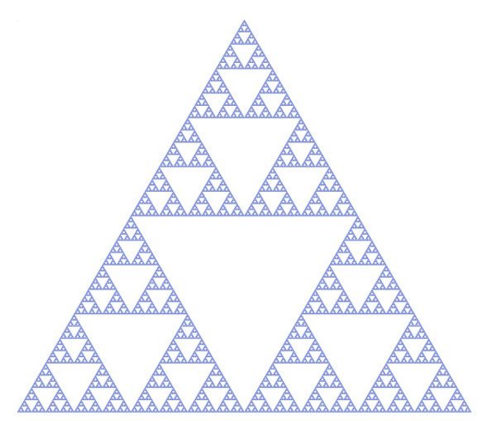


New (but old) concept in programming

Recursion

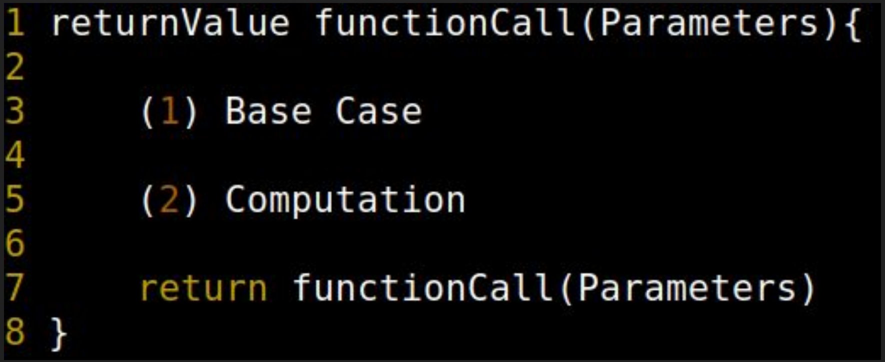
Recursion in ‘C’

- You have likely already seen recursion in other languages--but as a refresher.
- Recursion is: *The repeated application of a recursive procedure or definition.*
    - Recursion can often be an elegant way to solve problems, and simplify how we design algorithms.
    - Some examples of real world recursion are to the right



Structure of Recursive problem (1/2)

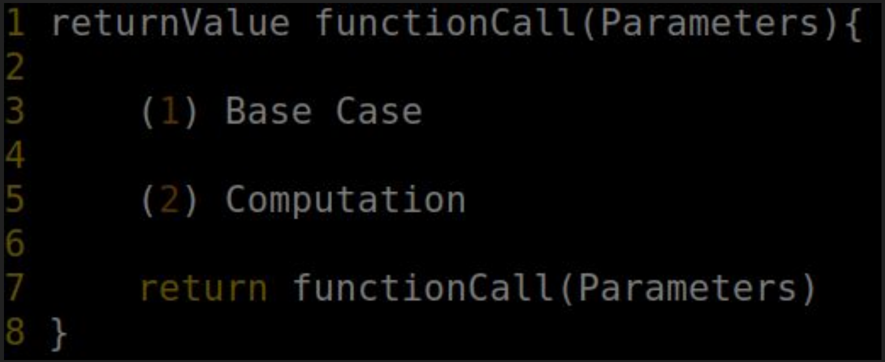
- The general structure to a recursive function in ‘C’ (any most every other language) is the following
  a. A base case that tells us when to terminate
     - (The smallest problem that can be solved)
  b. Some computation that is done
  c. And then a call to ‘ourselves’ that solves a smaller problem.



```
returnValue functionCall(Parameters){

    (1) Base Case

    (2) Computation

    return functionCall(Parameters)
}
```

Structure of Recursive problem (2/2)

- The general structure to a recursive function in 'C' (any most every other language) is the following
  a. A base case that tells us when to terminate
     - (The smallest problem that can be



```
returnValue functionCall(Parameters){

    (1) Base Case

    (2) Computation

    return functionCall(Parameters)
}
```

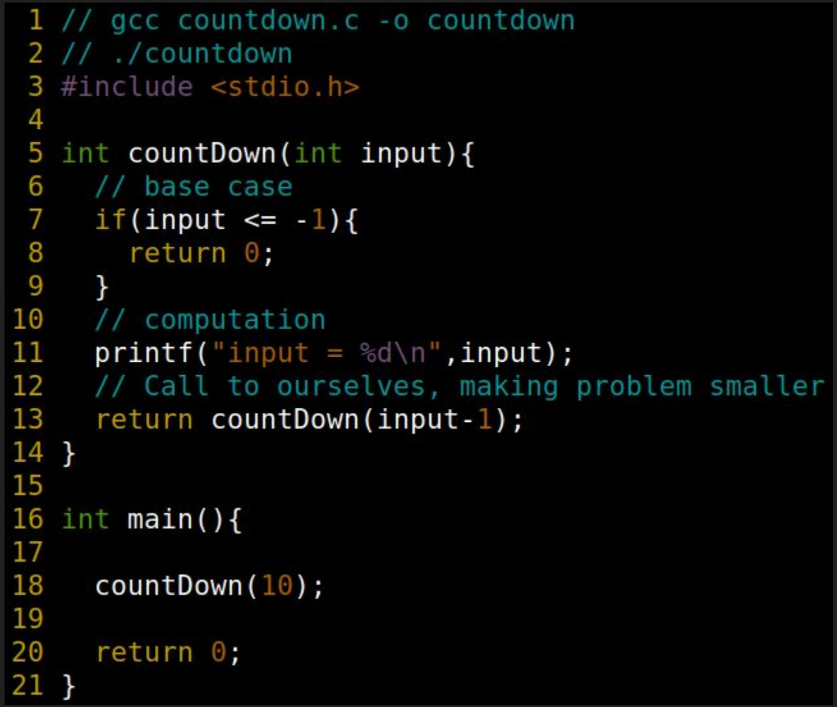
(Aside) Note that 'recursion' looks a lot like a proof by induction which we previously learned.

This is one of the powerful reasons to have a proof by induction. Because if we can proof something by induction, we *likely* can have a recursive solution when we actually implement in code the algorithm.

In general, recursion is just another useful tool in our toolbox for a way to thinking about how to implement a solution to a problem.

Example Recursion in C (1/5)

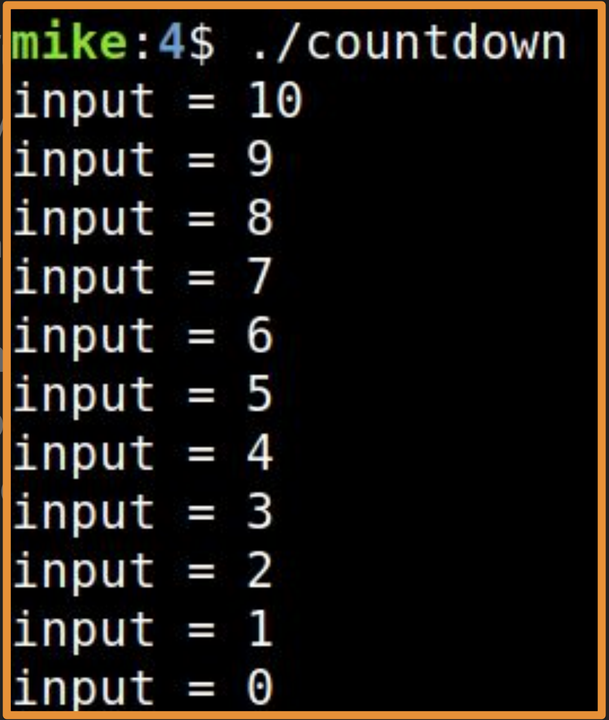
- Here is a function that we call recursively to move us slowly to a result of ‘0’
  - The function counts down until we reach ‘0’ from an integer.
- We can imagine how to do this with a ‘while loop’, but the recursive solution also ‘iterates’ in a way.



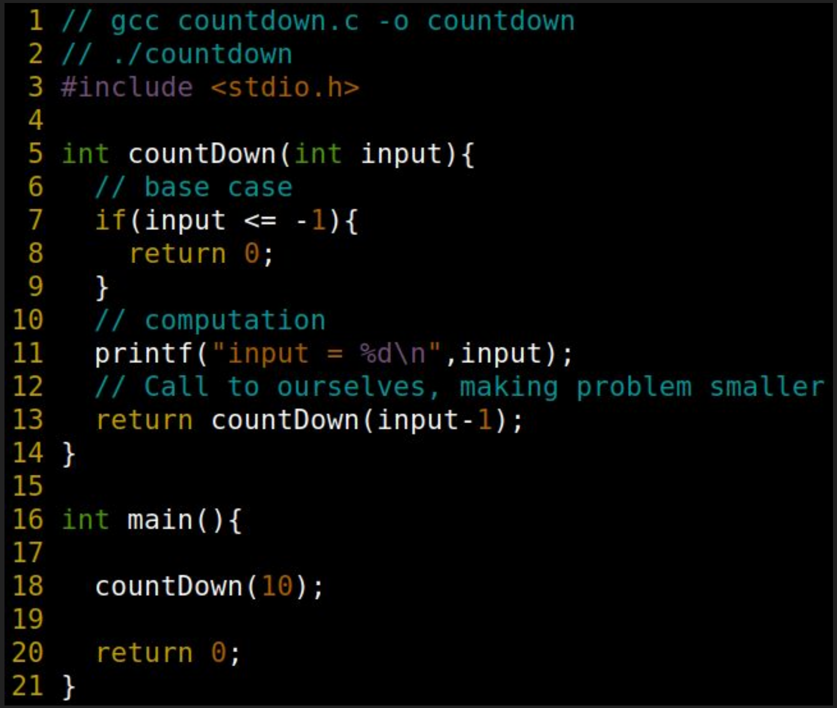
```c
// gcc countdown.c -o countdown
// ./countdown
#include <stdio.h>

int countDown(int input){
  // base case
  if(input <= -1){
    return 0;
  }
  // computation
  printf("input = %d\n",input);
  // Call to ourselves, making problem smaller
  return countDown(input-1);
}

int main(){

  countDown(10);

  return 0;
}
```

Example Recursion in C (2/5)

- Here is a f
  recursively
  result of ‘0
  - The fun
    ‘0’ from
- We can im
  ‘while loop
  also ‘iterat



```
mike:4$ ./countdown
input = 10
input = 9
input = 8
input = 7
input = 6
input = 5
input = 4
input = 3
input = 2
input = 1
input = 0
```

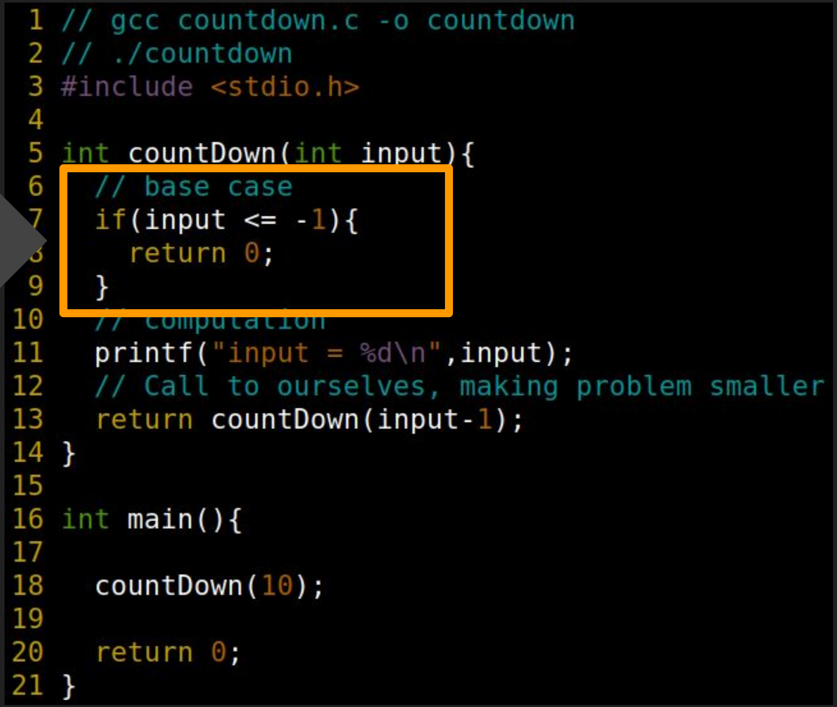


```c
// gcc countdown.c -o countdown
// ./countdown
#include <stdio.h>

int countDown(int input){
  // base case
  if(input <= -1){
    return 0;
  }
  // computation
  printf("input = %d\n",input);
  // Call to ourselves, making problem smaller
  return countDown(input-1);
}

int main(){

  countDown(10);

  return 0;
}
```

Example Recursion in C (3/5)

- Here is a function that we call recursively to move us slowly to a result of ‘0’
  - The funct ‘0’ from ar
- We can imagine how to do this with a ‘while loop’, but the recursive solution also ‘iterates’ in a way.

Our base case (i.e. when we terminate)

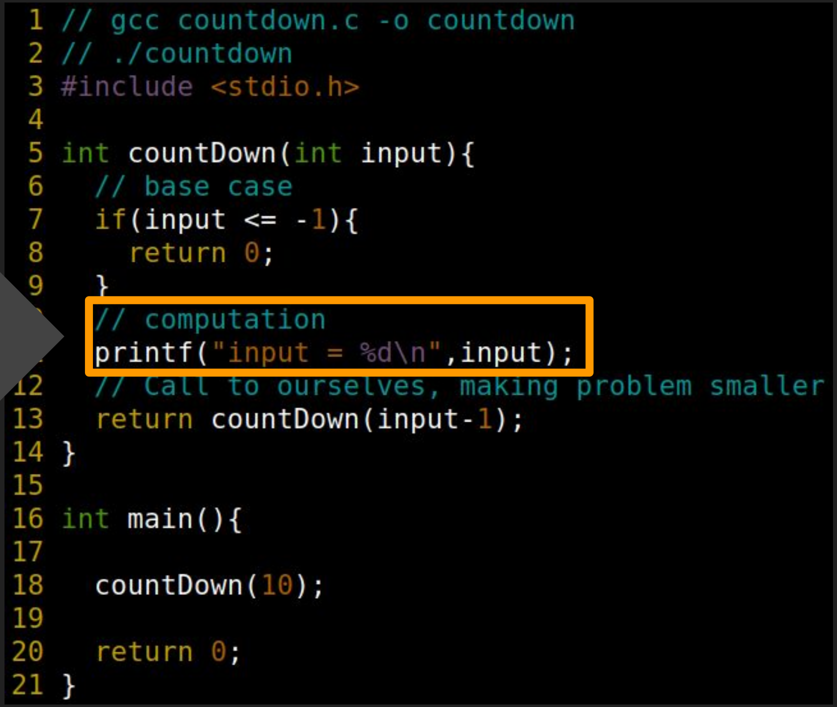


```c
// gcc countdown.c -o countdown
// ./countdown
#include <stdio.h>

int countDown(int input){
  // base case
  if(input <= -1){
    return 0;
  }
  // computation
  printf("input = %d\n",input);
  // Call to ourselves, making problem smaller
  return countDown(input-1);
}

int main(){

  countDown(10);

  return 0;
}
```

Example Recursion in C (4/5)

- Here is a function that we call recursively to move us slowly to a result of '0'
  - The function counts down until we reach '0' from an
- We can imag 'while loop', but the recursive solutio also 'iterates' in a way.

The work or computation we perform
In this case--printing a number



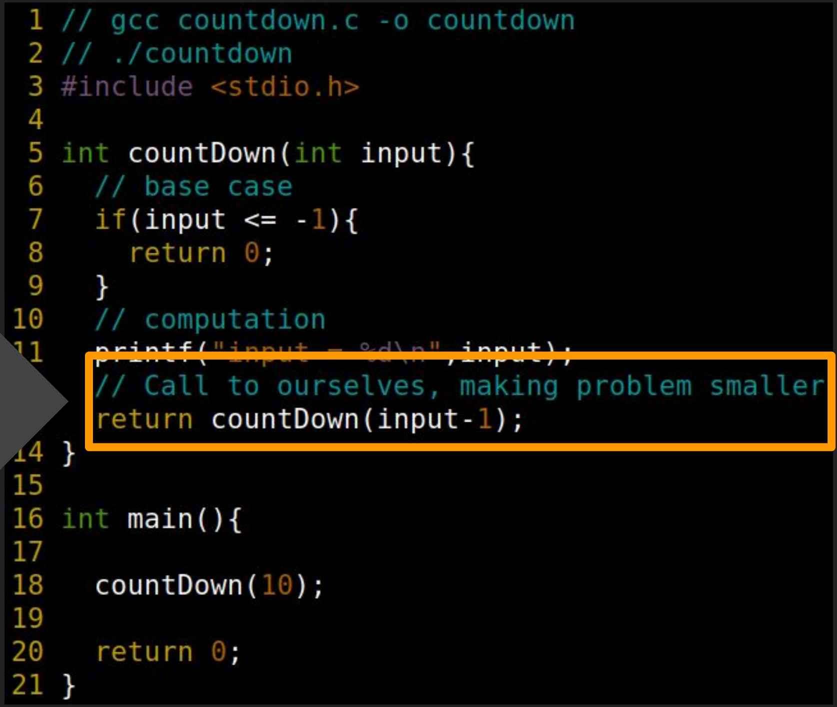
```
// gcc countdown.c -o countdown
// ./countdown
#include <stdio.h>

int countDown(int input){
  // base case
  if(input <= -1){
    return 0;
  }
  // computation
  printf("input = %d\n",input);
  // Call to ourselves, making problem smaller
  return countDown(input-1);
}

int main(){

  countDown(10);

  return 0;
}
```

Example Recursion in C (5/5)

- Here is a function that we call recursively to move us slowly to a result of ‘0’
  - The function counts down until we r h ‘0’ from an integer

Our recursive call.
Notice each recursive function call reduces the size of our problem (in the case of countdown, we get get closer to ‘0’ each call

- We
  ‘wh
  als



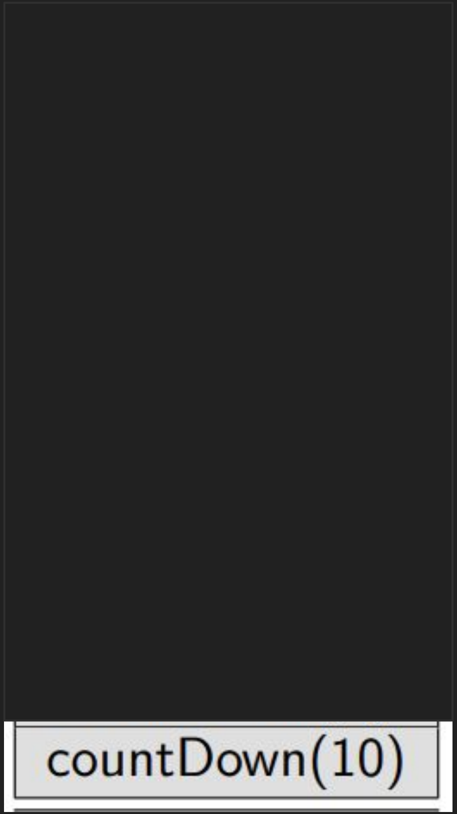
```
// gcc countdown.c -o countdown
// ./countdown
#include <stdio.h>

int countDown(int input){
  // base case
  if(input <= -1){
    return 0;
  }
  // computation
  printf("input = %d\n",input);
  // Call to ourselves, making problem smaller
  return countDown(input-1);
}

int main(){

  countDown(10);

  return 0;
}
```

Visualizing Recursion (1/13)

- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’

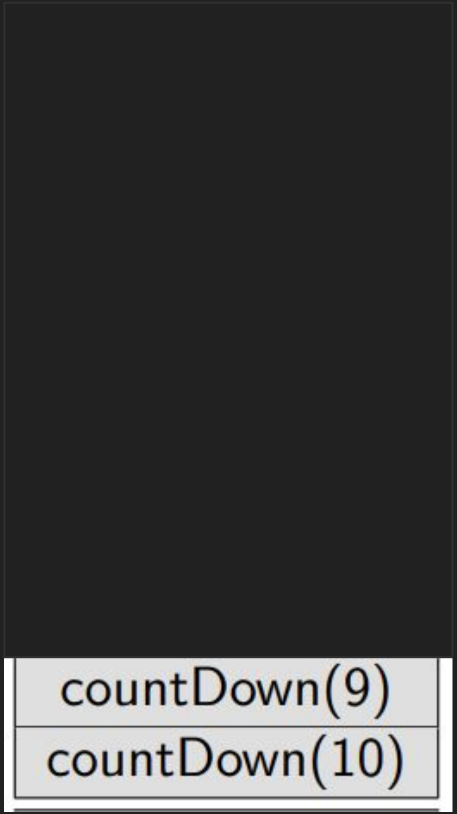
Visualizing Recursion (2/13)

- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’



Visualizing Recursion (3/13)

- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’

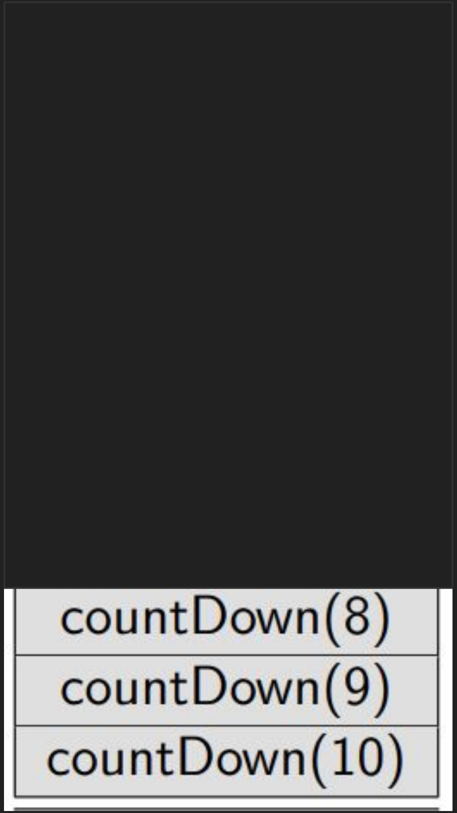


countDown(9)

countDown(10)

Visualizing Recursion (4/13)

- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’



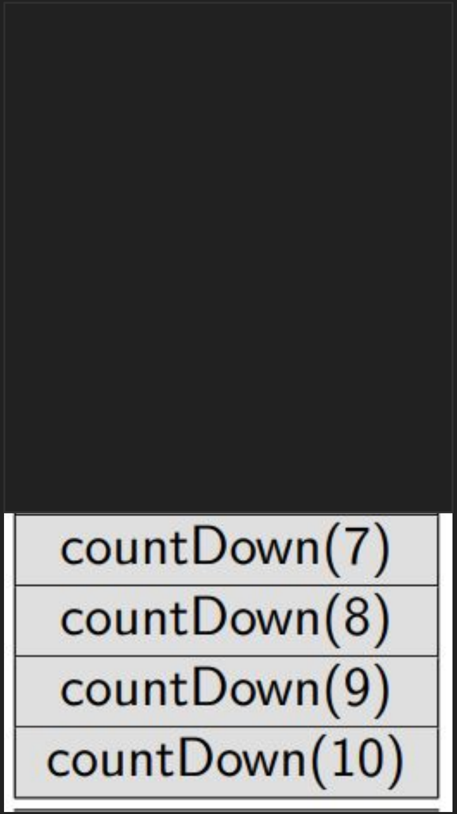
countDown(8)

countDown(9)

countDown(10)

Visualizing Recursion (5/13)

- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’



countDown(7)

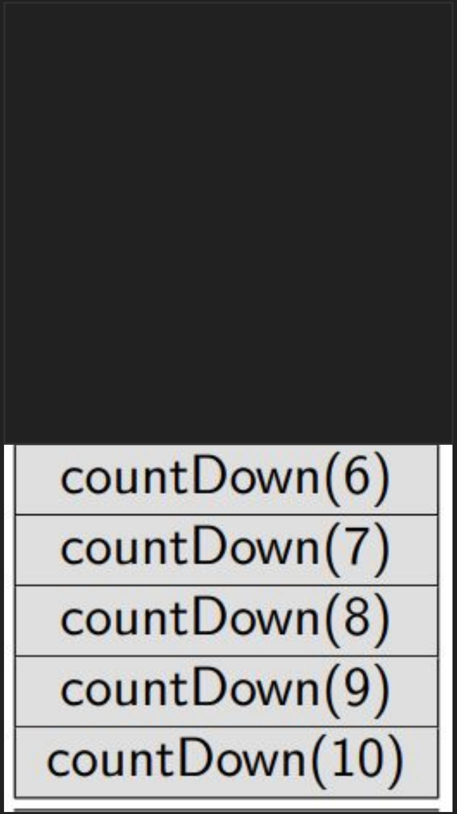
countDown(8)

countDown(9)

countDown(10)

Visualizing Recursion (6/13)

- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’



countDown(6)

countDown(7)

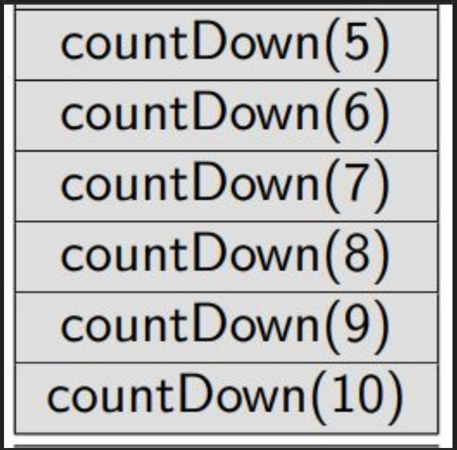
countDown(8)

countDown(9)

countDown(10)

Visualizing Recursion (7/13)

- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’



countDown(5)

countDown(6)

countDown(7)

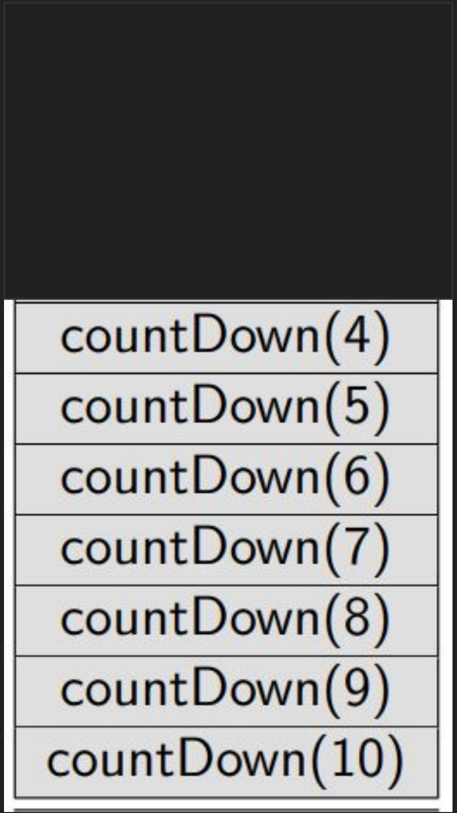
countDown(8)

countDown(9)

countDown(10)

Visualizing Recursion (8/13)

- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’



countDown(4)

countDown(5)

countDown(6)

countDown(7)

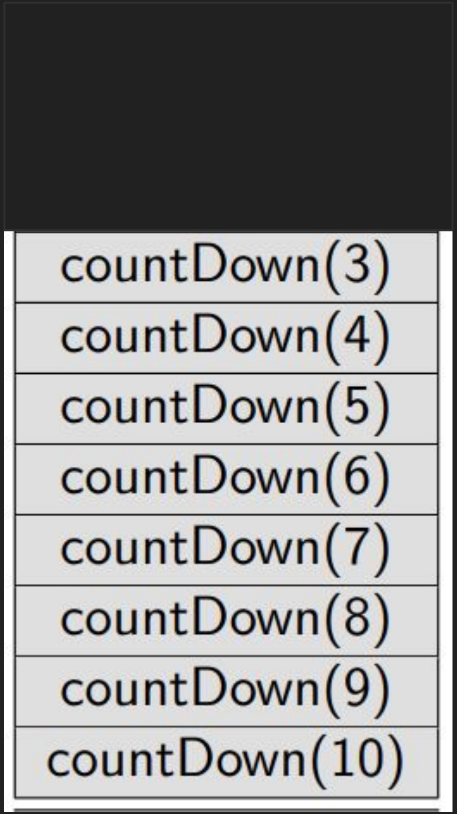
countDown(8)

countDown(9)

countDown(10)

Visualizing Recursion (9/13)

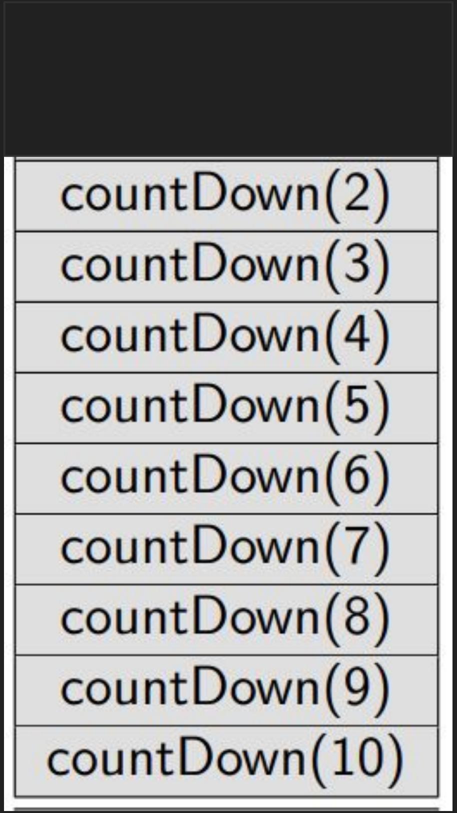
- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’



countDown(3)
countDown(4)
countDown(5)
countDown(6)
countDown(7)
countDown(8)
countDown(9)
countDown(10)

Visualizing Recursion (10/13)

- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’



countDown(2)

countDown(3)

countDown(4)

countDown(5)

countDown(6)

countDown(7)

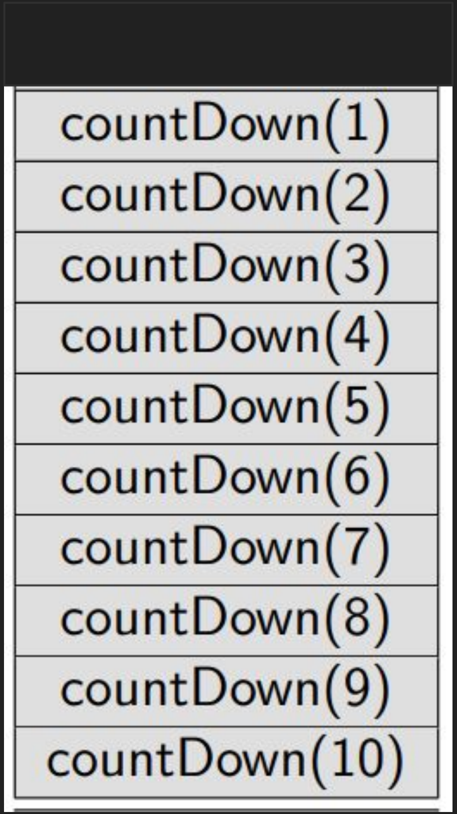
countDown(8)

countDown(9)

countDown(10)

Visualizing Recursion (11/13)

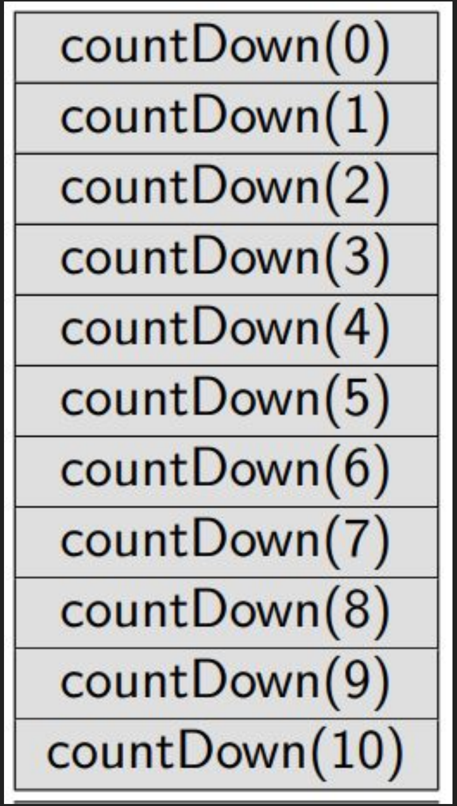
- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’



countDown(1)
countDown(2)
countDown(3)
countDown(4)
countDown(5)
countDown(6)
countDown(7)
countDown(8)
countDown(9)
countDown(10)

Visualizing Recursion (12/13)

- When we call a function over and over again, a ‘stack’ of function calls is created.
- Here is an example with ‘countDown’



countDown(0)
countDown(1)
countDown(2)
countDown(3)
countDown(4)
countDown(5)
countDown(6)
countDown(7)
countDown(8)
countDown(9)
countDown(10)

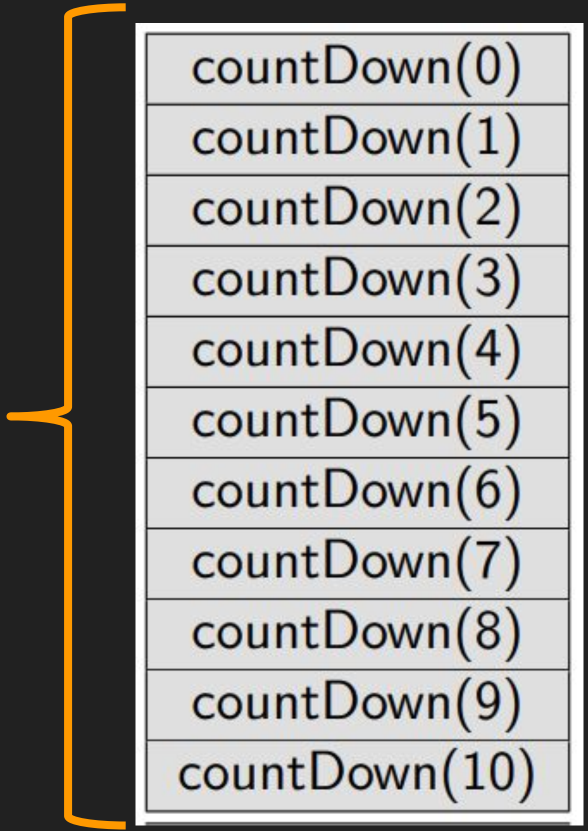
Visualizing Recursion (13/13)

- When we call a function over and over ag is creat
- Here is 'countD

This is called the 'call stack'.

A 'call stack' contains a Last-in-First-Out (LIFO) ordering of functions that have most recently been called appearing at the top.

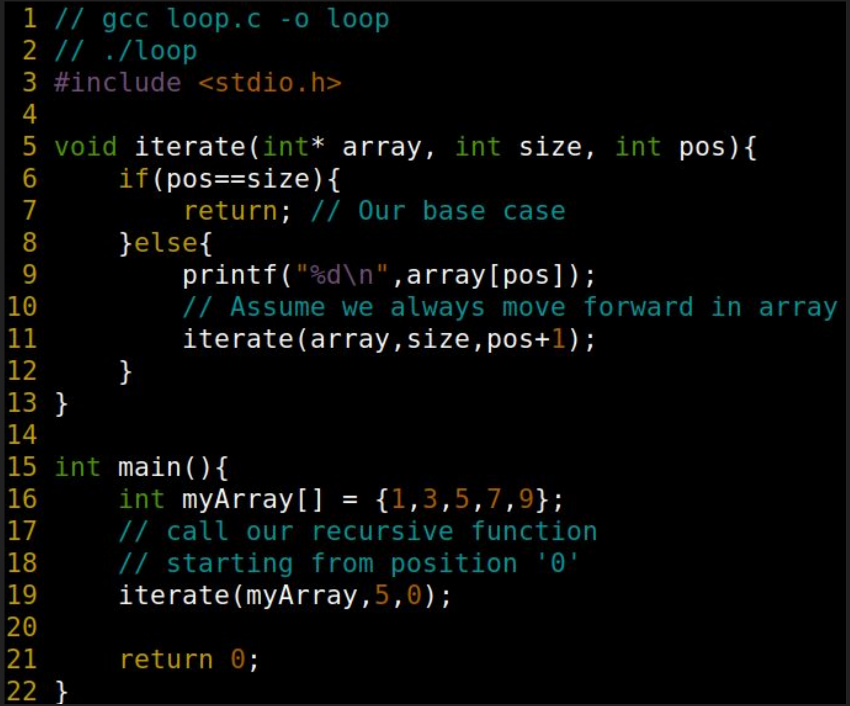
This 'call stack' is automatically maintained for us (by default, one call stack per single-threaded process)



countDown(0)
countDown(1)
countDown(2)
countDown(3)
countDown(4)
countDown(5)
countDown(6)
countDown(7)
countDown(8)
countDown(9)
countDown(10)

A second recursive example - array iteration (1/5)

- What is neat is, we can solve any problem recursively that we can with loops
  - (Some CS1 courses do not even introduce loops!)

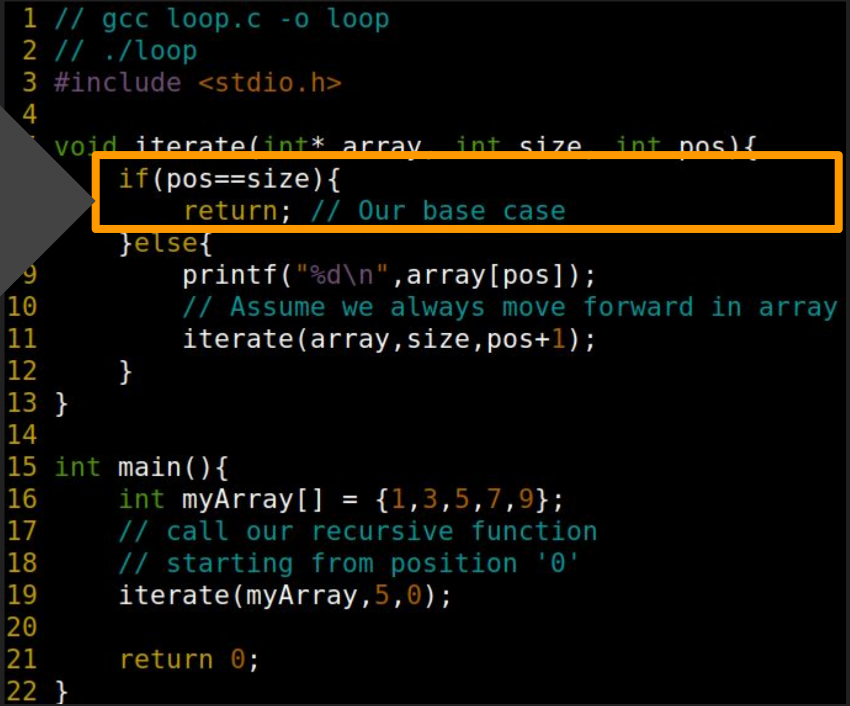


```
// gcc loop.c -o loop
// ./loop
#include <stdio.h>

void iterate(int* array, int size, int pos){
    if(pos==size){
        return; // Our base case
    }else{
        printf("%d\n",array[pos]);
        // Assume we always move forward in array
        iterate(array,size,pos+1);
    }
}

int main(){
    int myArray[] = {1,3,5,7,9};
    // call our recursive function
    // starting from position '0'
    iterate(myArray,5,0);

    return 0;
}
```

A second recursive example - array iteration (2/5)

- What is neat is, we can solve an
  pro
  wit
  ○

In our base case we iterate until we are at the end of our array.

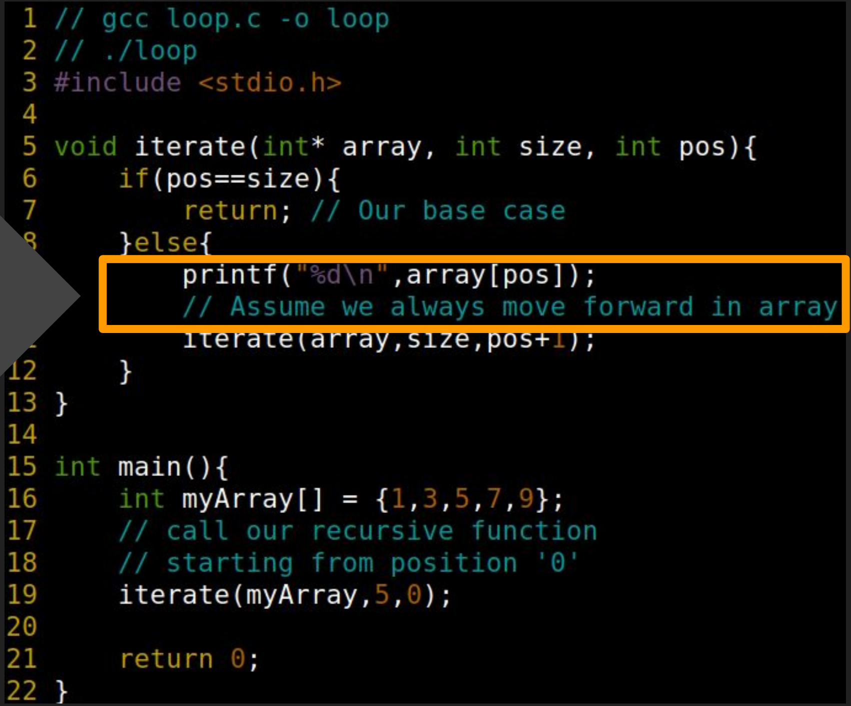


```
// gcc loop.c -o loop
// ./loop
#include <stdio.h>

void iterate(int* array, int size, int pos){
    if(pos==size){
        return; // Our base case
    }else{
        printf("%d\n",array[pos]);
        // Assume we always move forward in array
        iterate(array,size,pos+1);
    }
}

int main(){
    int myArray[] = {1,3,5,7,9};
    // call our recursive function
    // starting from position '0'
    iterate(myArray,5,0);

    return 0;
}
```

A second recursive example - array iteration (3/5)

- What is neat is, we can solve any problem recursively that we can with loops

The work we perform each step is to print out the array value

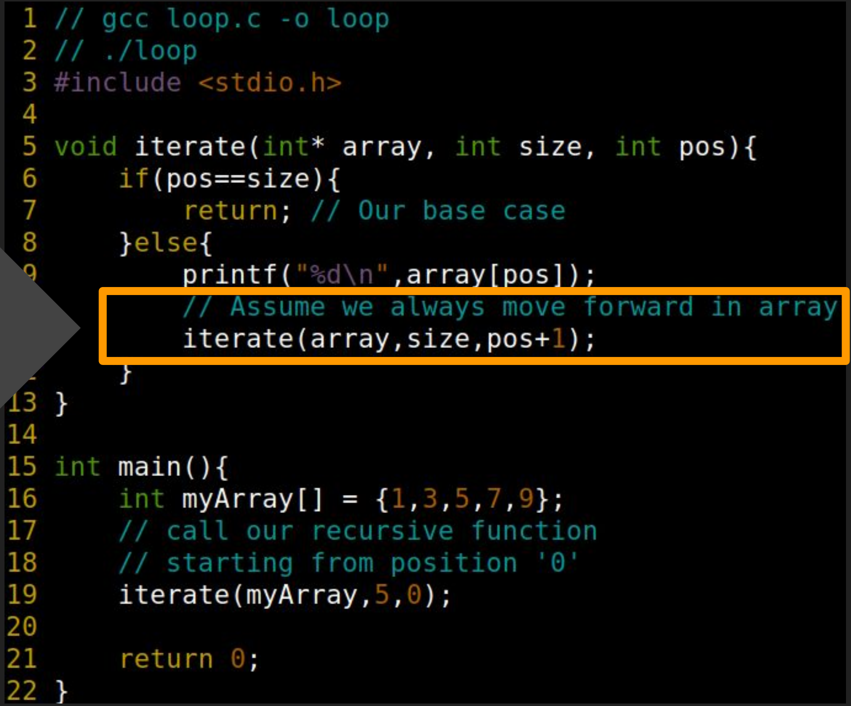


```
// gcc loop.c -o loop
// ./loop
#include <stdio.h>

void iterate(int* array, int size, int pos){
    if(pos==size){
        return; // Our base case
    }else{
        printf("%d\n",array[pos]);
        // Assume we always move forward in array
        iterate(array,size,pos+1);
    }
}

int main(){
    int myArray[] = {1,3,5,7,9};
    // call our recursive function
    // starting from position '0'
    iterate(myArray,5,0);

    return 0;
}
```

A second recursive example - array iteration (4/5)

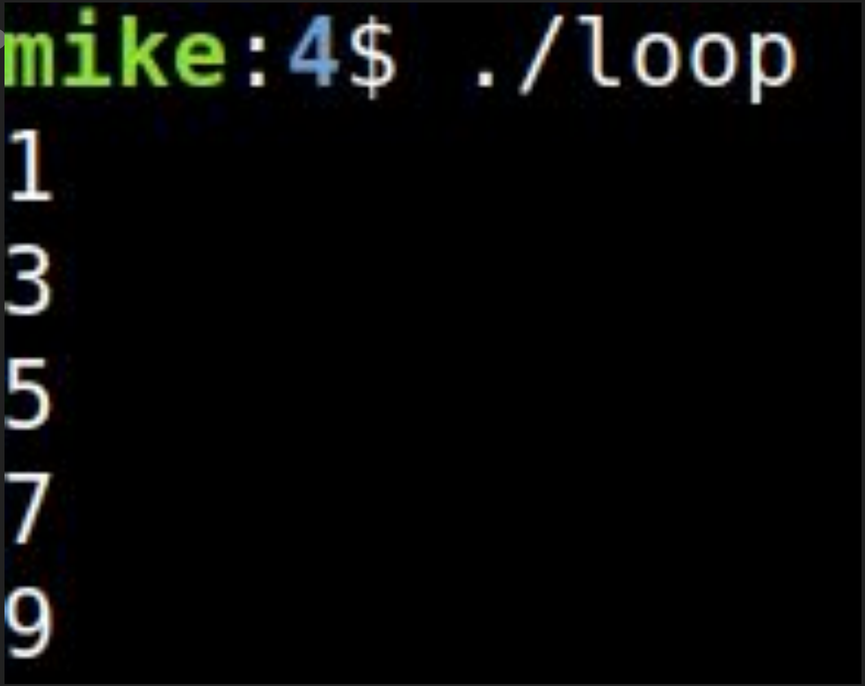
- What is neat is, we can solve any problem recursively that we can with loops

Our recursive call ‘iterates’ us 1 position forward in the array

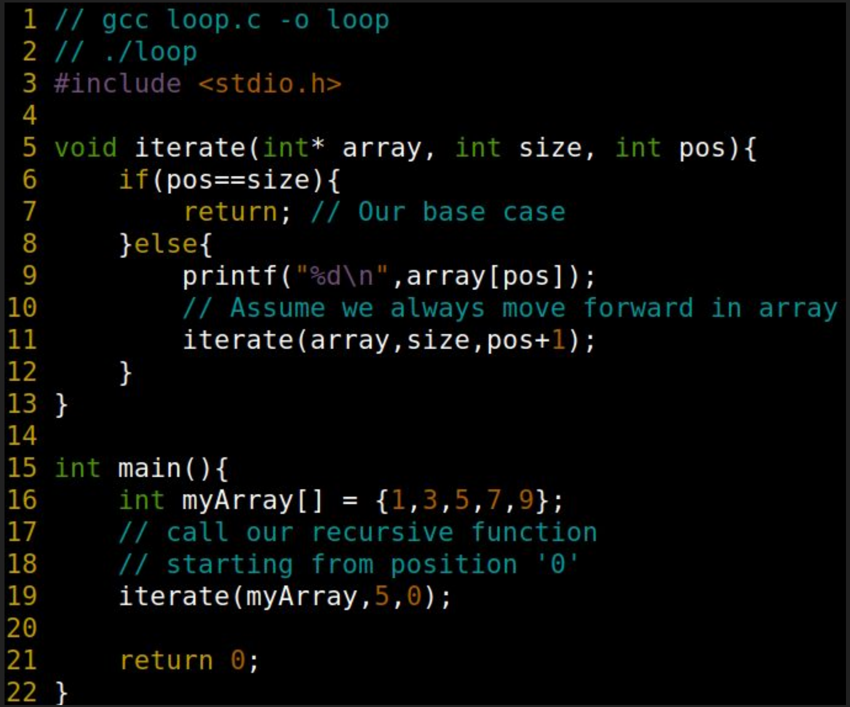


```
// gcc loop.c -o loop
// ./loop
#include <stdio.h>

void iterate(int* array, int size, int pos){
    if(pos==size){
        return; // Our base case
    }else{
        printf("%d\n",array[pos]);
        // Assume we always move forward in array
        iterate(array,size,pos+1);
    }
}

int main(){
    int myArray[] = {1,3,5,7,9};
    // call our recursive function
    // starting from position '0'
    iterate(myArray,5,0);

    return 0;
}
```

A second recursive example - array iteration (5/5)



```
mike:4$ ./loop
1
3
5
7
9
```



```c
// gcc loop.c -o loop
// ./loop
#include <stdio.h>

void iterate(int* array, int size, int pos){
    if(pos==size){
        return; // Our base case
    }else{
        printf("%d\n",array[pos]);
        // Assume we always move forward in array
        iterate(array,size,pos+1);
    }
}

int main(){
    int myArray[] = {1,3,5,7,9};
    // call our recursive function
    // starting from position '0'
    iterate(myArray,5,0);

    return 0;
}
```

A third example -- computing factorials [reference] (1/6)

- A factorial is a positive integer where we compute the product of all positive integers less than or equal to that number
    - An example is shown on the top right for 5!
- Factorials are useful, or typically used to compute how many ‘permutations’ or possible orderings there are of items.

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

A third example -- computing factorials [reference] (2/6)

- The more general form of a factorial is on the right
  - (The ‘Pi product notation’ is shown to the right as well)

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

$$n! = \prod_{i=1}^{n} i.$$

A third example -- computing factorials [reference] (3/6)

- The more general form of a factorial is on the right
  - (The ‘Pi product notation’ is shown to the right as well)

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

$$n! = \prod_{\boxed{i=1}}^{n} i.$$

If you want to think of this like a ‘for’ loop

A third example -- computing factorials [reference] (4/6)

- The more general form of a factorial is on the right
  - (The ‘Pi product notation’ is shown to the right as well)

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

$$n! = \prod_{\boxed{i=1}}^{n} i.$$

i is initialized to 1
e.g.
for(i=1; ... ; ++i)

A third example -- computing factorials [reference] (5/6)

- The more general form of a factorial is on the right
  - (The 'Pi product notation' is shown to the right as well)

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

$$n! = \prod_{i=1}^{\boxed{n}} i.$$

We iterate 'n' times
e.g.
for(i=1; i < n; ++i)

A third example -- computing factorials [reference] (6/6)

- The more general form of a factorial is on the right
  - (The ‘Pi product notation’ is shown to the right as well)

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

$$n! = \prod_{i=1}^{n} \boxed{i.}$$

The work we do e.g.

```
for(i=1; i < n; ++i){
  n_factorial *= i;
}
```

Recurrences

Recurrence within the factorial

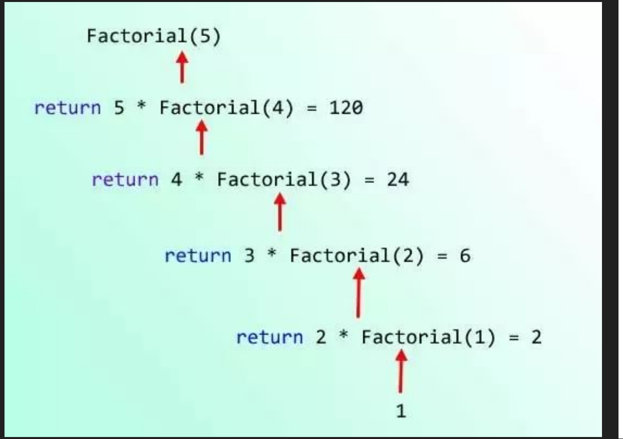
- Now from these formulas, you can derive the 'recurrence'
  - Identifying the 'recurrence' is helpful for 'creating a recursive' function
  - See in the example to the right
    - 5! is equivalent to 5 `* 4!`
  - 4! is the 'unsolved' part of the problem, and we can pull out the '5' and then try to solve a smaller sub-problem
    - (i.e. often a goal of recursion, solve smaller sub-problem then combine the results later)
  - This generalizes to: `n! = n * (n-1)!`

$$5! = 5 \cdot 4!$$

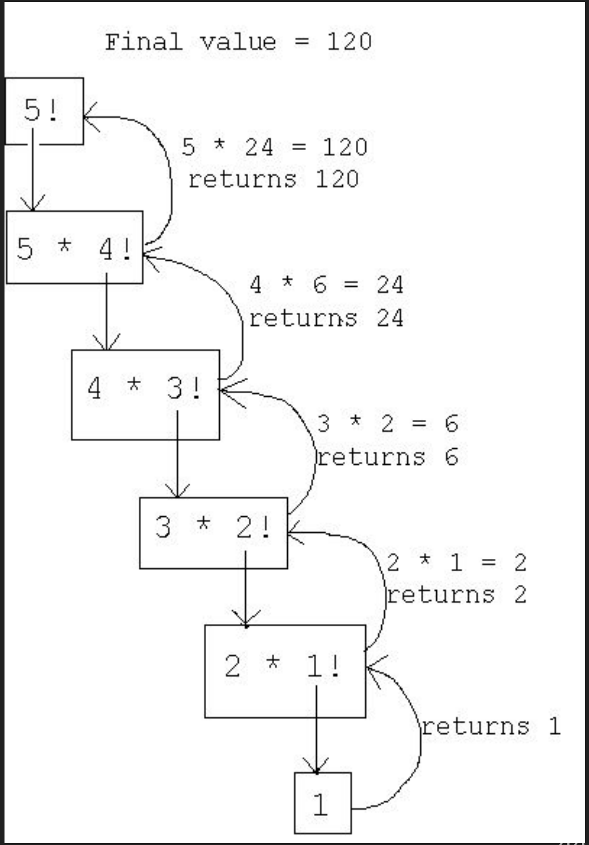
$$n! = n \cdot (n - 1)!.$$

Factorials Visualized

- Here are two visualizations for what you are computing at each step



```
    Factorial(5)
return 5 * Factorial(4) = 120
      return 4 * Factorial(3) = 24
            return 3 * Factorial(2) = 6
                  return 2 * Factorial(1) = 2
                                          1
```



Note: Pragmatically -- Can we recurse forever? (1/3)

- (Question to audience) If we recurse too many times (Say our input is n=1,000,000,000,000) and we have more computation to do, do you think most computers will be okay with that?

Note: Pragmatically -- Can we recurse forever? (2/3)

- (Question to audience) If we recurse too many times (Say our input is n=1,000,000,000,000) and we have more computation to do, do you think most computers will be okay with that?
  - Answer: We get an error known as a ‘stack overflow’!
    - (Yes--like the popular, very helpful programming site)
- This is because our ‘call stack’ can only get so large.
- So make sure, the problem we are computing is always getting smaller in the recursive case

Note: Pragmatically -- Can we recurse forever? (3/3)

- (Question to audience) If we recurse too many times (Say our input is n=1,000,000,000,000) and we have more computation to do, do you think most computers will be okay with that?
    - Answer: We get an error known as a ‘stack overflow’!
        - (Yes--like the popular, very helpful programming site)
- This is because our ‘call stack’ can only get so large.
- So make sure, the problem we are computing is always getting smaller in the recursive case
- For this course--we will see ‘recursion’ in algorithms.
- We are not constrained by physical cpu limits in this course *when talking about algorithms*--so we will see some recursion today--and how it can ‘simplify’ and make ‘elegant’ some algorithms.

“The more stuff you throw in a system, the more complicated it gets and the more likely it is not going to work properly” Barbara Liskov

“The more stuff you throw in a system, the more complicated it gets and the more likely it is not going to work properly” Barbara Liskov

“Sounds like recursion can be quite helpful then--even if just how we think about problems”

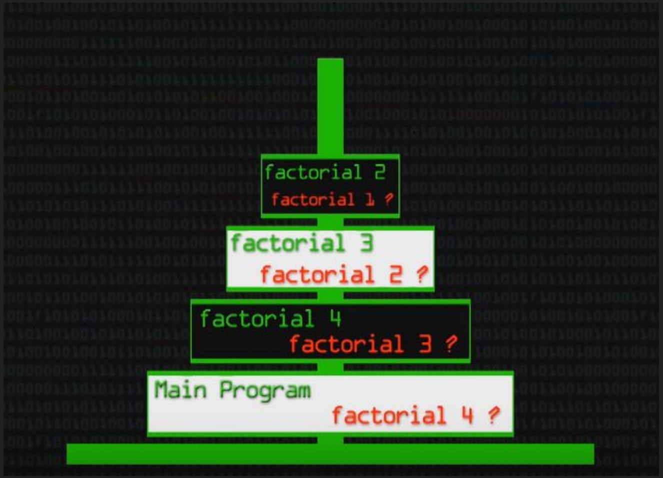
Who is Barbara Liskov? [wiki]

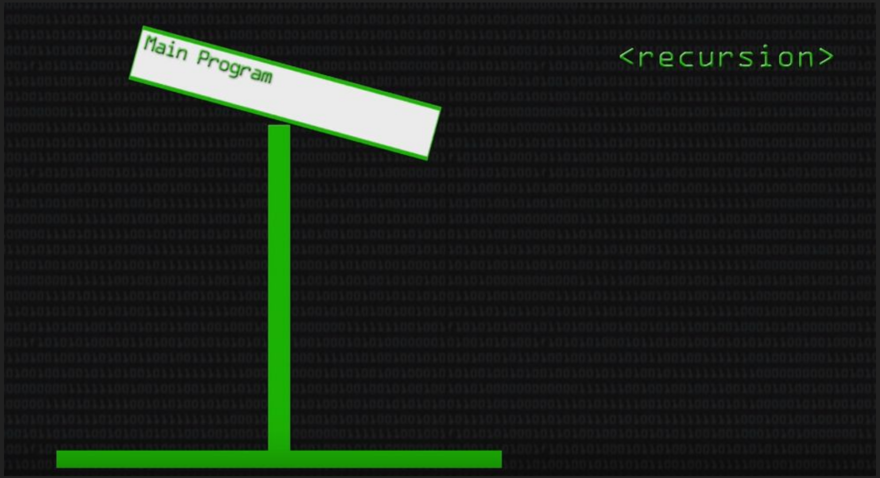
- MIT Professor
- One of the first woman in the US to earn a doctorate in Computer Science
- Turing Award Winner
- Well known in the Object-Oriented Programming world for ‘Liskov’s Substitution Principle’ for Type Systems
  - (Perhaps an inspiring figure for the OOD world?)

Computer Systems Feed

YOUR DAILY FEED

- (An article/image/video/thought injected in each class!)
- https://youtu.be/Mv9NEXX1VHc?t=343
- Here is a handy video (You can reference for your lab today)
  - (Computerphile is another great channel to subscribe too!)





Revisiting Time Complexity

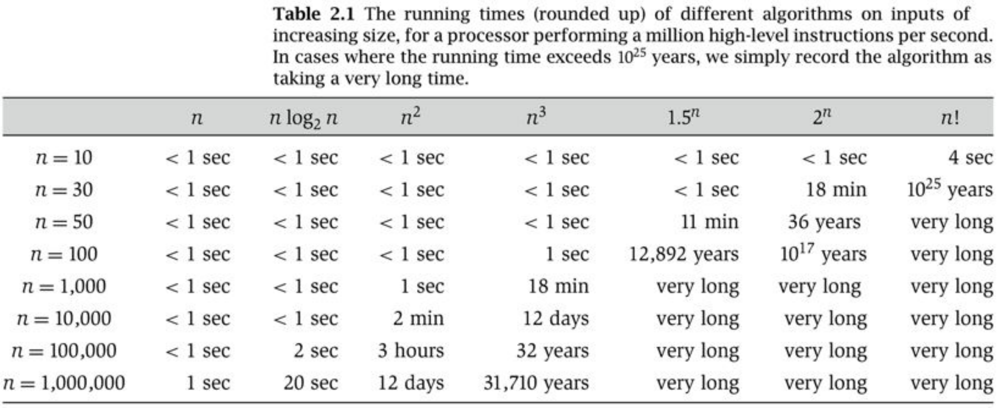


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Time Complexity (1/2)

- Previously we looked at Big-O notation.
- The takeaway is, that as we have more inputs ‘n’, then it takes our algorithms longer to compute a result
- If we choose a good algorithm, it can make quite a difference

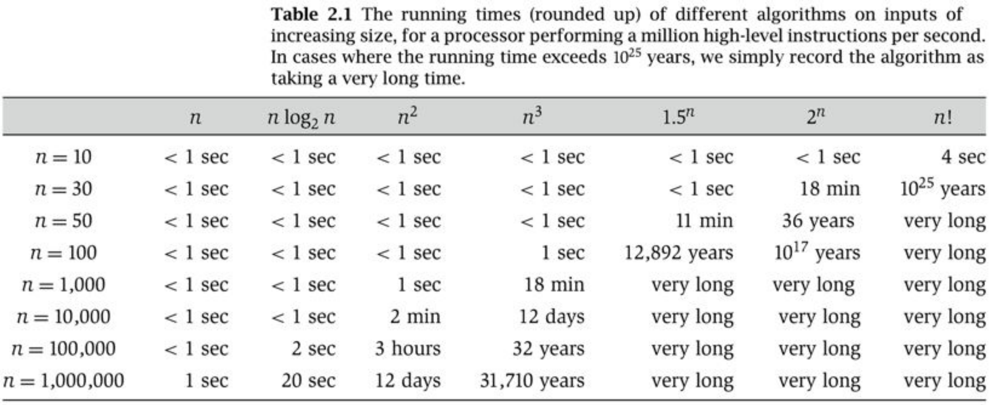


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Time Complexity (2/2)

- Last time we looked at Big-O notation.
- The takeaway is, that as we have more inputs ‘n’, then it takes our algorithms longer to compute a result
- If we choose a good algorithm, it can make quite a difference
  - e.g. compare n! vs n at n=30

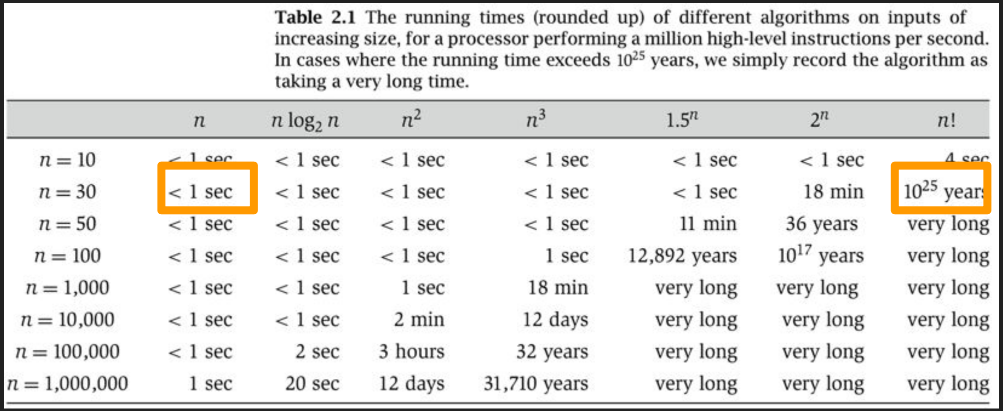


Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

Previously we looked at three sorts: (1/4)

1. bubble sort
2. selection sort
3. insertion sort

Question to the audience: What was common about their time complexity?

Previously we looked at three sorts: (2/4)

1. bubble sort
2. selection sort
3. insertion sort

Question to the audience: What was common about their time complexity?

- Answer: All n-squared (polynomial time--or more specifically quadratic) algorithms in the worse-case
  - Written as: O(n^2)

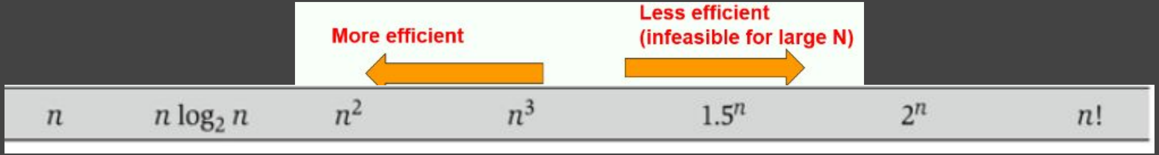
Previously we looked at three sorts: (3/4)

1. bubble sort
2. selectio
3. insertio

Question to

Answer: All

(Question) If we wanted to do better than O(n^2), how much better can we do according to this chart?



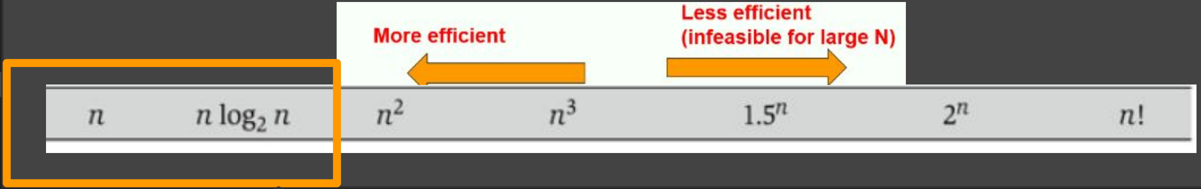
Previously we looked at three sorts: (4/4)

1. bubble sort
2. selectio
3. insertio

Question to

Answer: All r

(Question) If we wanted to do better than $O(n^2)$, how much better can we do according to this chart?



Need to think about sorting data in ‘n’ or ‘$n\log_2(n)$’

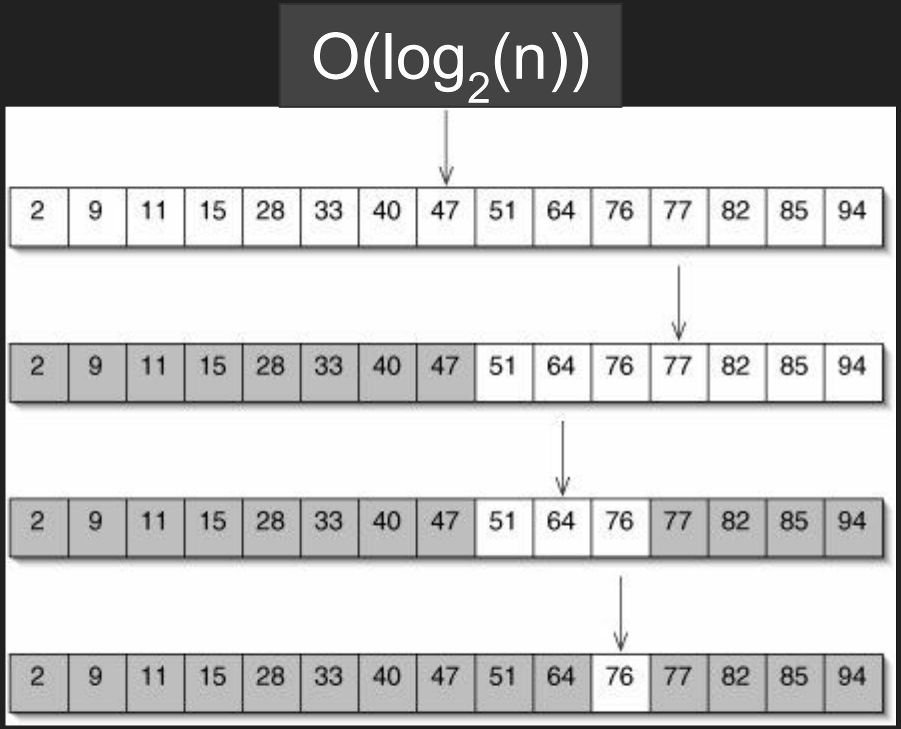
Let’s try $n\log_2(n)$

n*log$_2$n (1/2)

- Just looking at n*log$_2$(n), there are two parts I see.
- ‘n’ for the number of items, and then we also have‘* log$_2$(n)’

n*log$_2$n (2/2)

- Just looking at n*log$_2$(n), there are two parts I see.
- ‘n’ for the number of items, and then we also have‘* log$_2$(n)’
- If we remember from ‘binary search’ where we saw (O(log$_2$(n)) complexity, this had to deal with reducing our problem in half.
  - Hmm, so how can we do that with a sort?
  - That is, divide our problem in half?



$\log_2(n)$ and Divide and Conquer

Binary search is a ‘divide and conquer’ algorithm strategy (1/4)

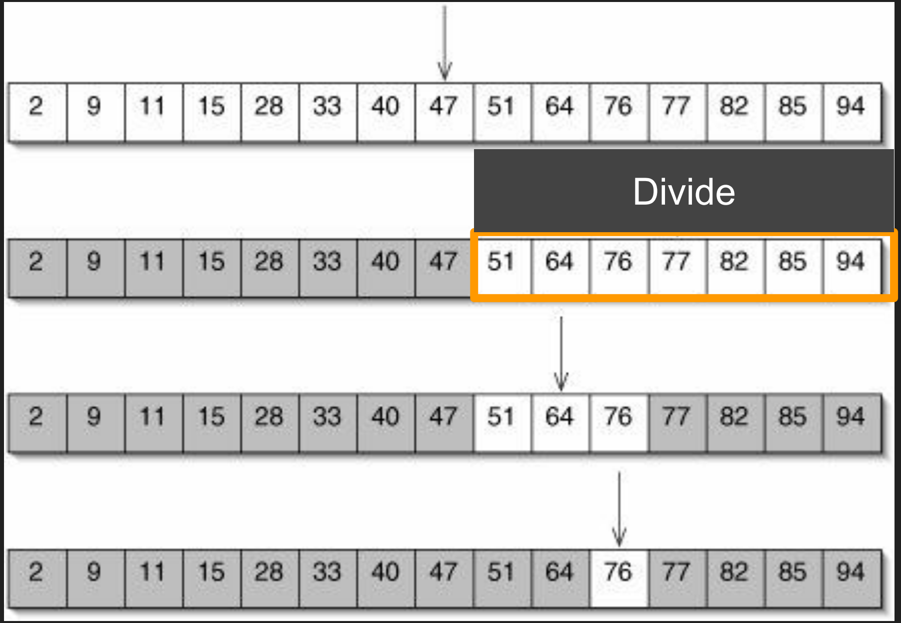
This is a general strategy where we have two parts

1. Divide: ‘array’ (or any data structure) into two smaller parts
2. Conquer: Search a smaller (i.e. half size array)

Binary search is a ‘divide and conquer’ algorithm strategy (2/4)

This is a general strategy where we have two parts

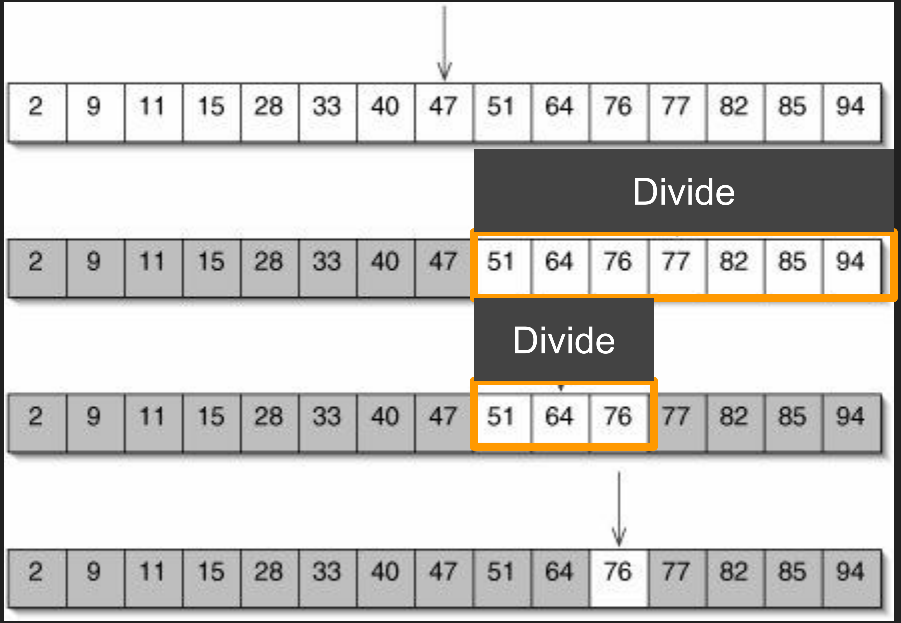
1. Divide: ‘array’ (or any data structure) into two smaller parts
2. Conquer: Search a smaller (i.e. half size array)



Binary search is a ‘divide and conquer’ algorithm strategy (3/4)

This is a general strategy where we have two parts

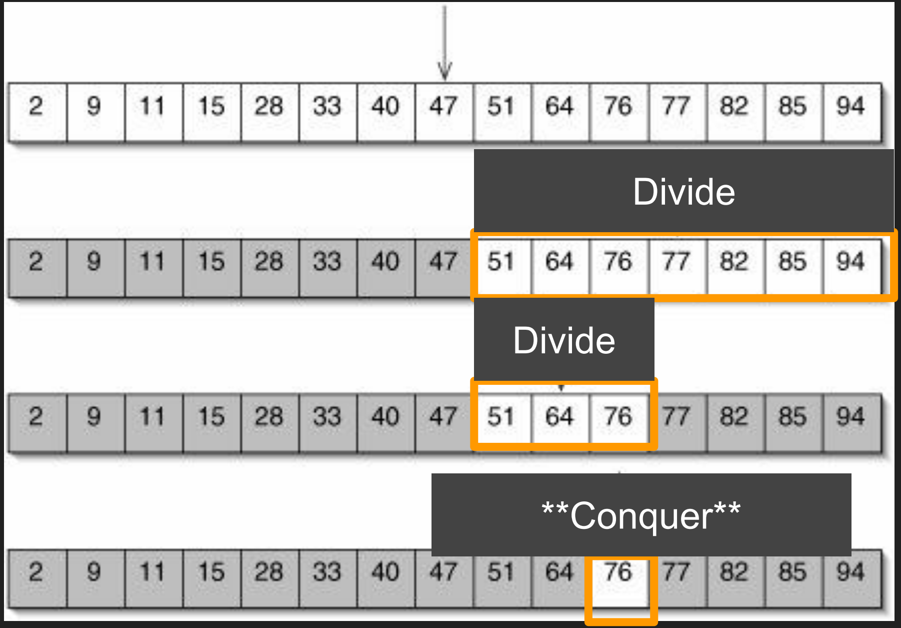
1. Divide: ‘array’ (or any data structure) into two smaller parts
2. Conquer: Search a smaller (i.e. half size array)



Binary search is a ‘divide and conquer’ algorithm strategy (4/4)

This is a general strategy where we have two parts

1. Divide: ‘array’ (or any data structure) into two smaller parts
2. Conquer: Search a smaller (i.e. half size array)



2	9	11	15	28	33	40	47	51	64	76	77	82	85	94

Divide

2	9	11	15	28	33	40	47	51	64	76	77	82	85	94

Divide

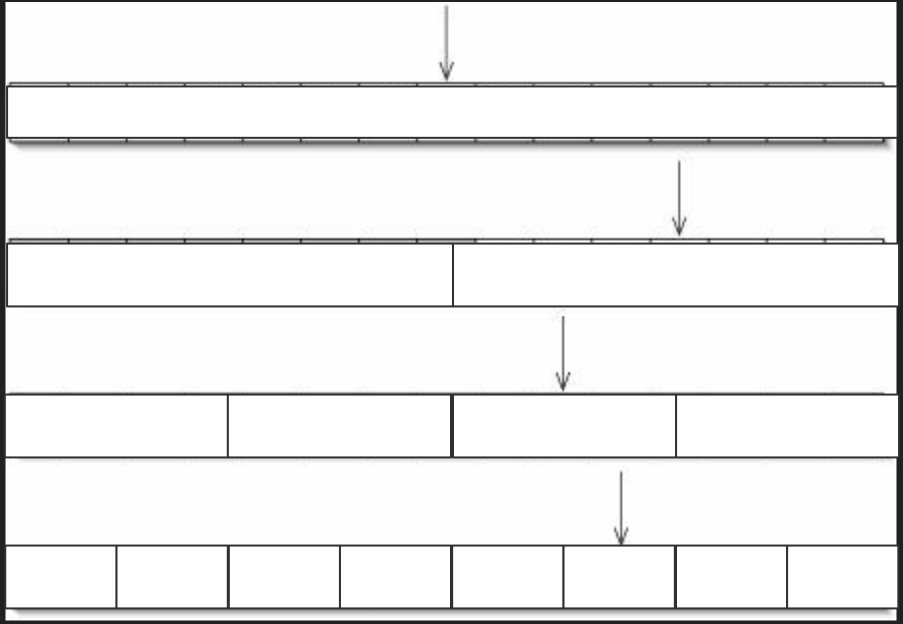
2	9	11	15	28	33	40	47	51	64	76	77	82	85	94

Conquer

2	9	11	15	28	33	40	47	51	64	76	77	82	85	94

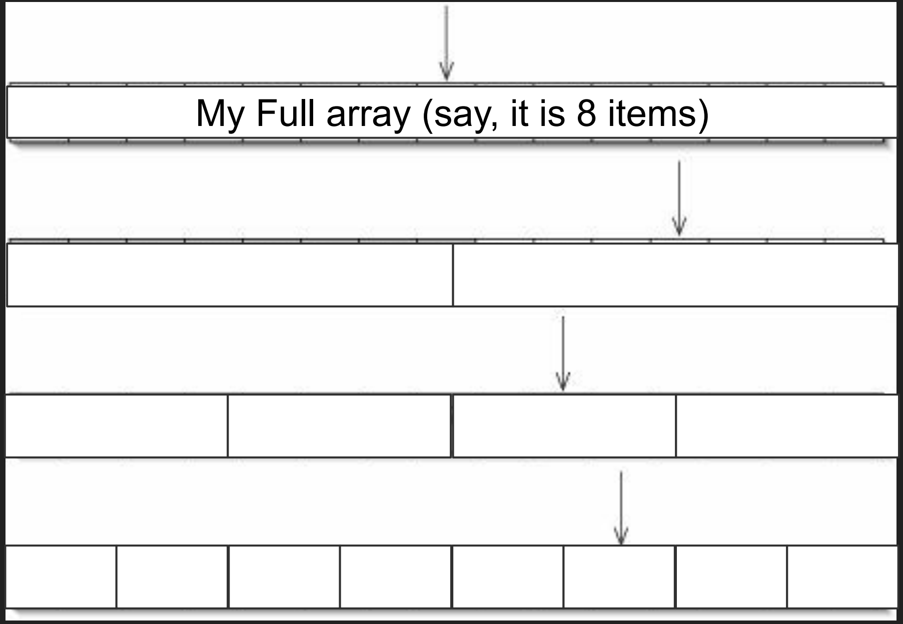
So if I transform this in a sorting problem (1/6)

- Generally, I just need to be able to divide our array into two smaller arrays
  - Then each step of the algorithm, I only have to work on half of the inputs
  - If each level I divide the problem in half again, this is ‘$\log_2(n)$’ complexity.



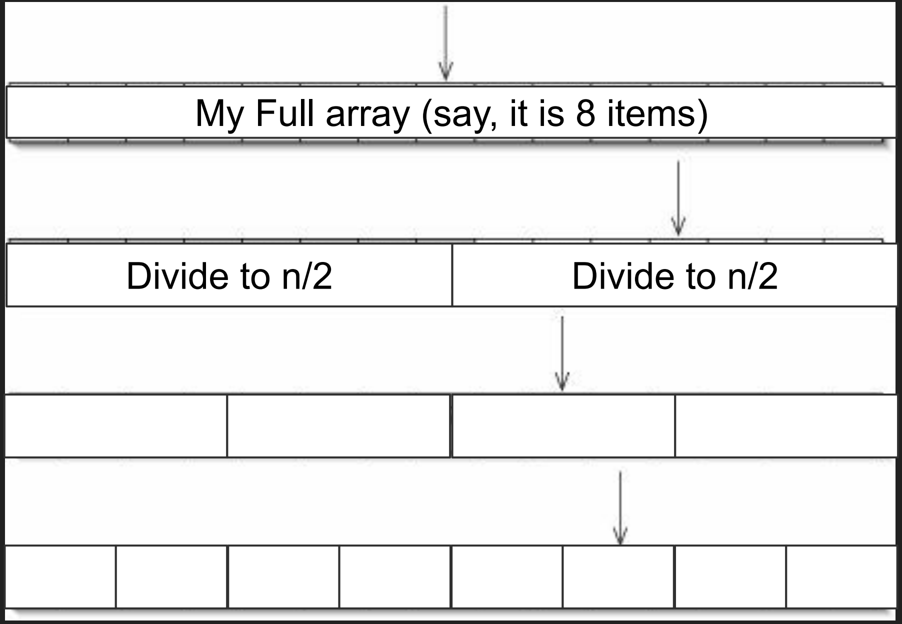
So if I transform this in a sorting problem (2/6)

- Generally, I just need to be able to divide our array into two smaller arrays
  - Then each step of the algorithm, I only have to work on half of the inputs
  - If each level I divide the problem in half again, this is ‘$\log_2(n)$’ complexity.



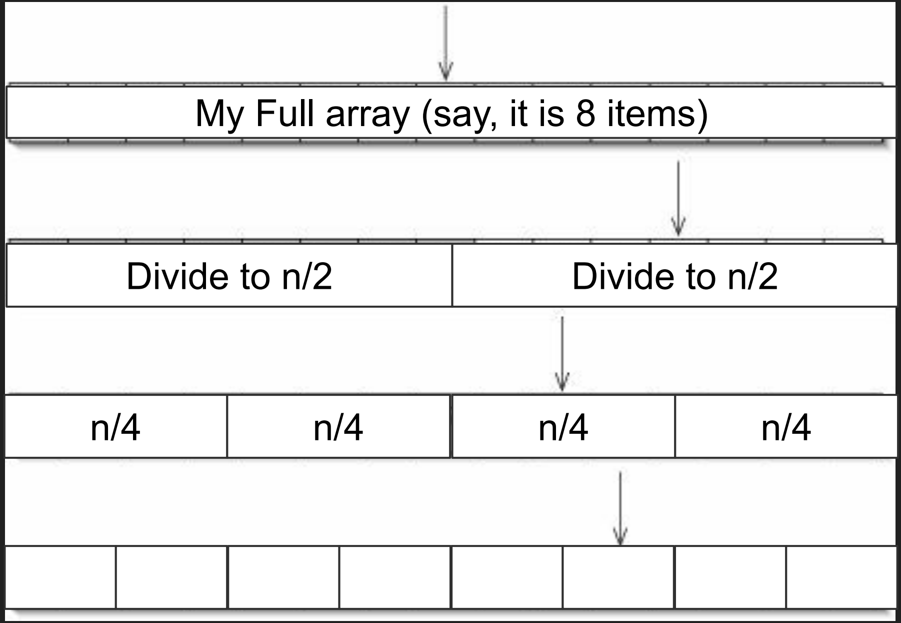
So if I transform this in a sorting problem (3/6)

- Generally, I just need to be able to divide our array into two smaller arrays
  - Then each step of the algorithm, I only have to work on half of the inputs
  - If each level I divide the problem in half again, this is ‘$\log_2$(n)’ complexity.



So if I transform this in a sorting problem (4/6)

- Generally, I just need to be able to divide our array into two smaller arrays
  - Then each step of the algorithm, I only have to work on half of the inputs
  - If each level I divide the problem in half again, this is ‘$\log_2$(n)’ complexity.



My Full array (say, it is 8 items)

Divide to n/2

Divide to n/2

n/4

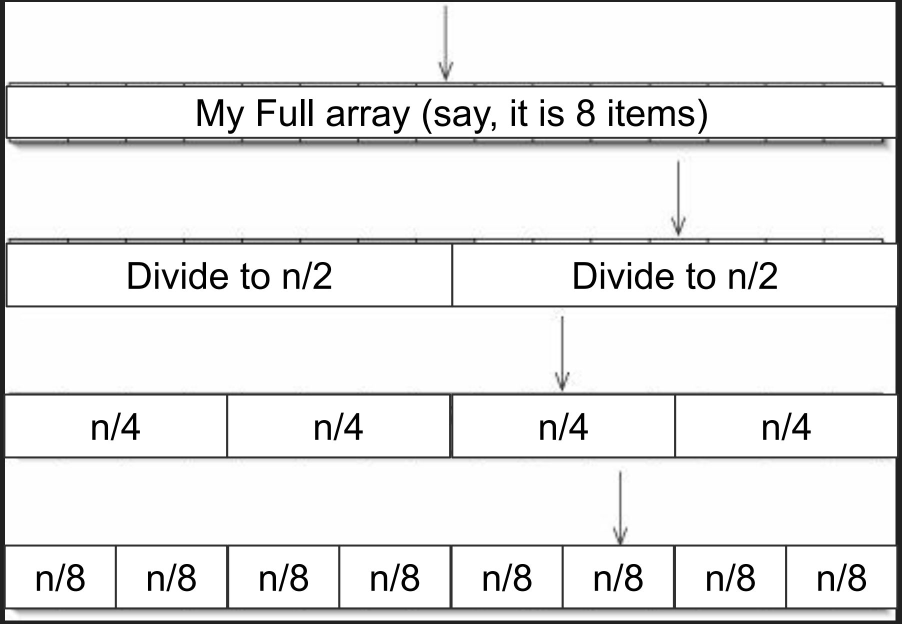
n/4

n/4

n/4

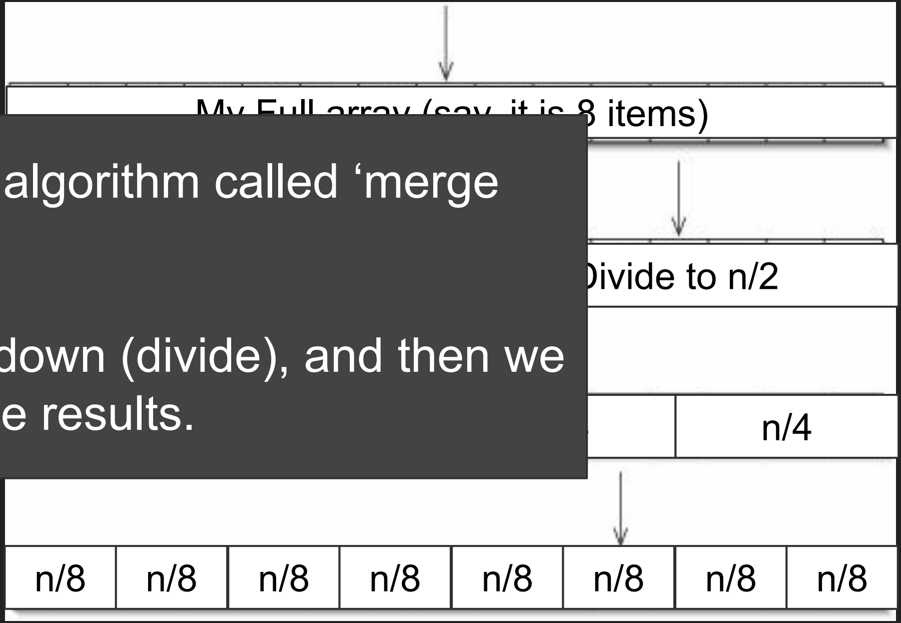
So if I transform this in a sorting problem (5/6)

- Generally, I just need to be able to divide our array into two smaller arrays
  - Then each step of the algorithm, I only have to work on half of the inputs
  - If each level I divide the problem in half again, this is ‘$\log_2(n)$’ complexity.



So if I transform this in a sorting problem (6/6)

- Generally, I just need to be able to divide our array into two smaller arrays
  - Then each ste
    have to work c
  - If each level I c
    again, this is ‘l



This is the idea of an algorithm called ‘merge sort’

We break a problem down (divide), and then we *combine* (conquer) the results.

Sorting Algorithm - Merge sort

Goal: Break problem down in half to get $n*\log_2(n)$ performance

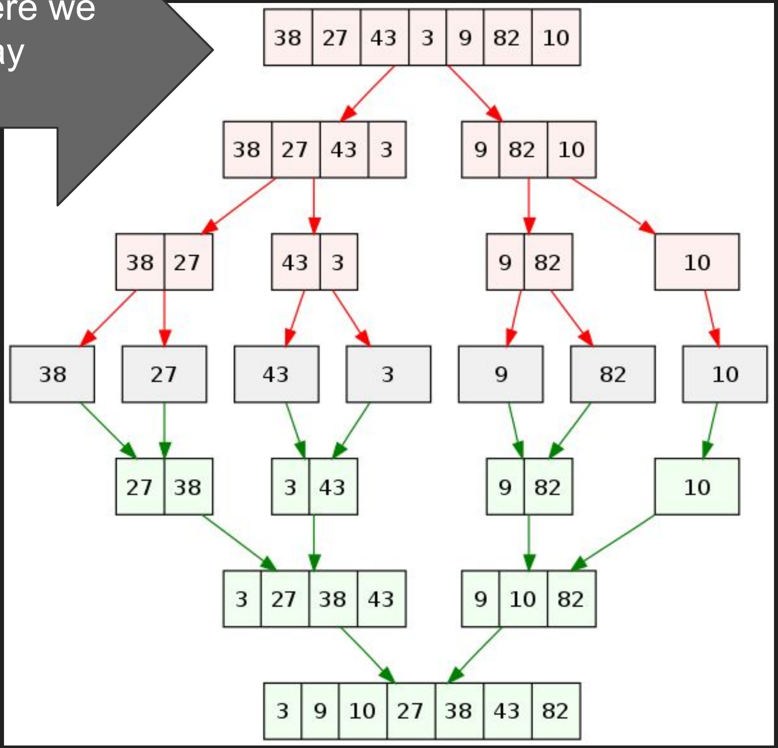
Merge Sort [reference] (1/2)

- Mergesort is a comparison based algorithm, where we divide, conquer, and then combine the results to build a sorted list.

Merge Sort [reference] (2/2)

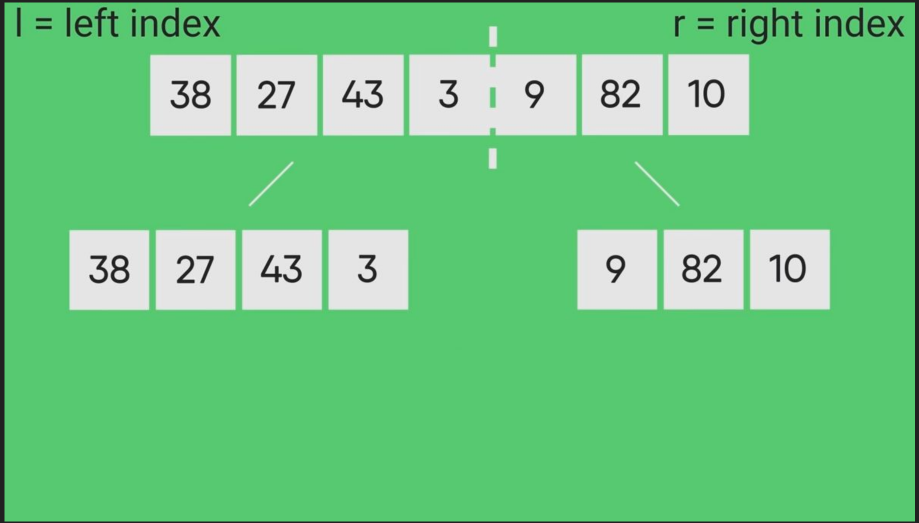
- Mergesort is a c algorithm, where we divide, conquer, and then combine the results to build a sorted list.

Take a moment to observe where we start, and how we ‘half’ our array every time



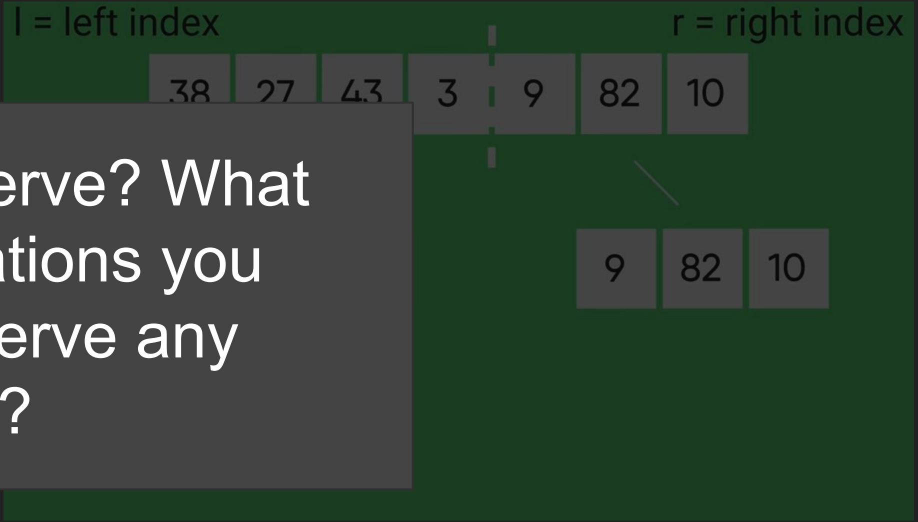
Merge Sort Visual (1/2)

- https://www.youtube.com/watch?v=JSceec-wEyw (1:15 min)



Merge Sort Visual (2/2)

- https://www.youtube.com/watch?v=JSceec-wE



What did you observe? What was the key operations you saw? Did you observe any loops or recursion?

Implementing Merge Sort

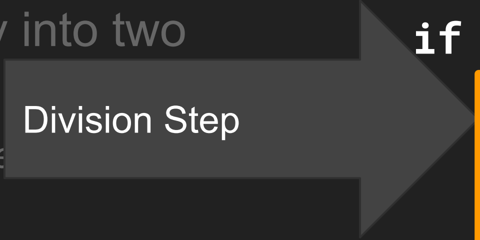
Merge sort - Pseudocode (1/4)

- MergeSort takes in an array, and then splits the array into two half-sized arrays:
  - One half from the *left* index until the middle index.
  - The other from the *right* index to the end

```
MergeSort(Array, left, right)
  if Array's size > 1
     Divide array Array in halves
     Call MergeSort on first half.
     Call MergeSort on second half.
     Merge two results (combine).
```

Merge sort - Pseudocode (2/4)

- MergeSort takes in an array, and then splits the array into two half-sized arrays:
    - One half from the *le* middle index.
    - The other from the *right* index to the end



Division Step

```
MergeSort(Array, left, right)
  if Array's size > 1
    Divide array Array in halves
    Call MergeSort on first half.
    Call MergeSort on second half.
    Merge two results (combine).
```

Merge sort - Pseudocode (3/4)

- MergeSort takes in an array, and then splits the array into two half-sized arrays:
    - One half from the *left* index until the middle index.
    - The other from the end

Conquer Step

```
MergeSort(Array, left, right)
  if Array's size > 1
     Divide array Array in halves
     Call MergeSort on first half.
     Call MergeSort on second half.
     Merge two results (combine).
```

Merge sort - Pseudocode (4/4)

- MergeSort takes in an array, and then splits the array into two half-sized arrays:
    - One half from the *left* index until the middle index.
    - The other from the *right* index to the end

```
MergeSort(Array, left, right)
  if Array's size > 1
     Divide array Array in halves
     Call MergeSort on first half.
     Call MergeSort on second half.
     Merge two results (combine).
```

How does merge work?

Merge

- Merge takes the 'divided' subarrays and 'selects' the smallest of the items and puts them into the leftmost index of a sorted array.
  - (You can think of this like selection sort, just selecting the smallest item
  - Our arrays are smaller, and we only need to iterate through '1' time, thus 'n' times total)

```
MergeSort(Array, left, right)
  if Array's size > 1
     Divide array Array in halves
     Call MergeSort on first half.
     Call MergeSort on second half.
     Merge two results (combine).
```

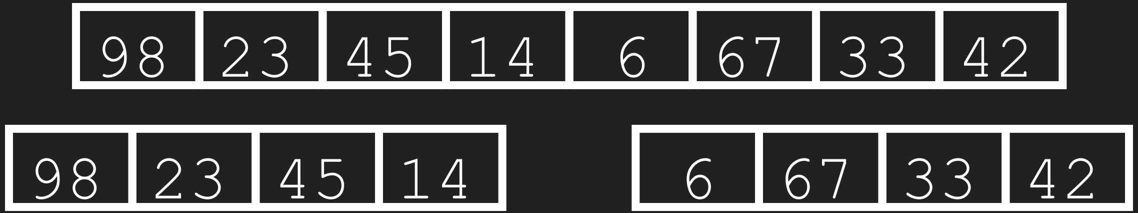
98	23	45	14	6	67	33	42

Let’s do an example

98	23	45	14	6	67	33	42

Narration: A given array of ‘unsorted’ numbers is given.

Note: Whether this is an array or ‘list’ the same ideas apply.

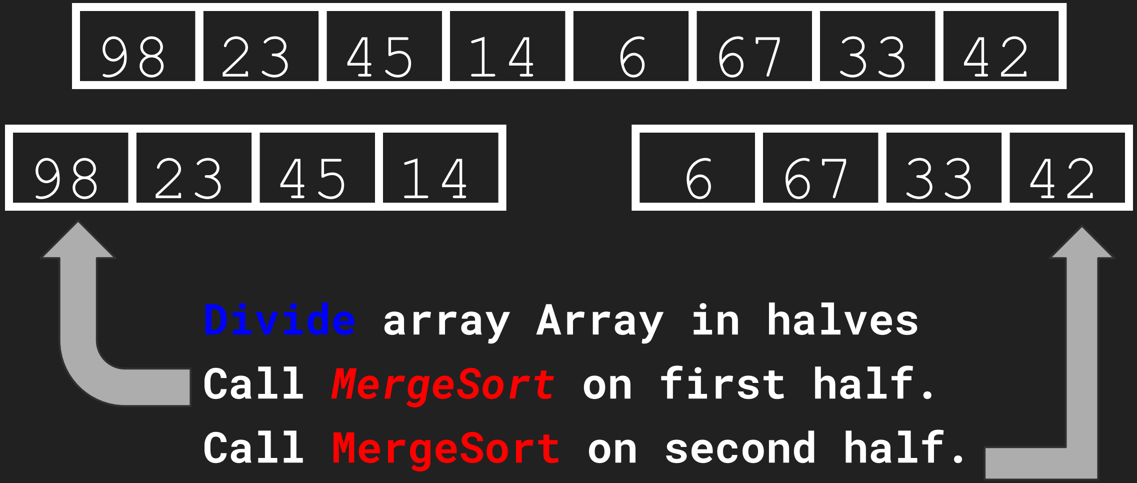


98	23	45	14	6	67	33	42

98	23	45	14

6	67	33	42

Narration: We split the array into two pieces.



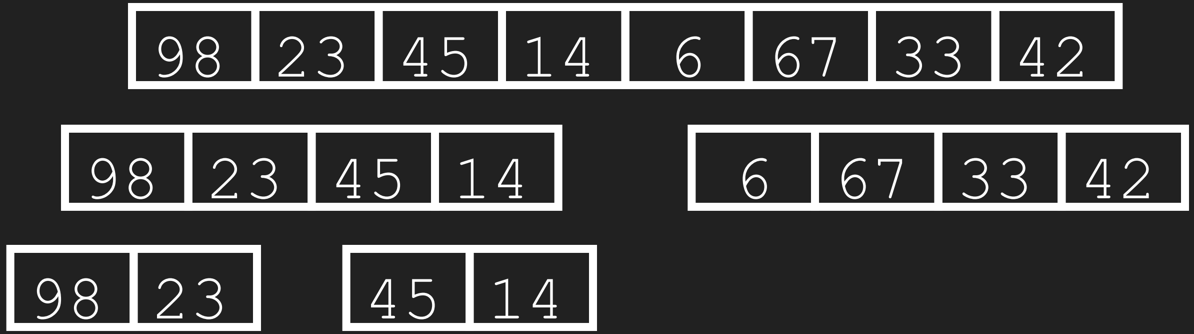
98	23	45	14	6	67	33	42

98	23	45	14

6	67	33	42

```
Divide array Array in halves
Call MergeSort on first half.
Call MergeSort on second half.
```

Narration: Call mergesort on first and second half of array



98 23 45 14 6 67 33 42

98 23 45 14

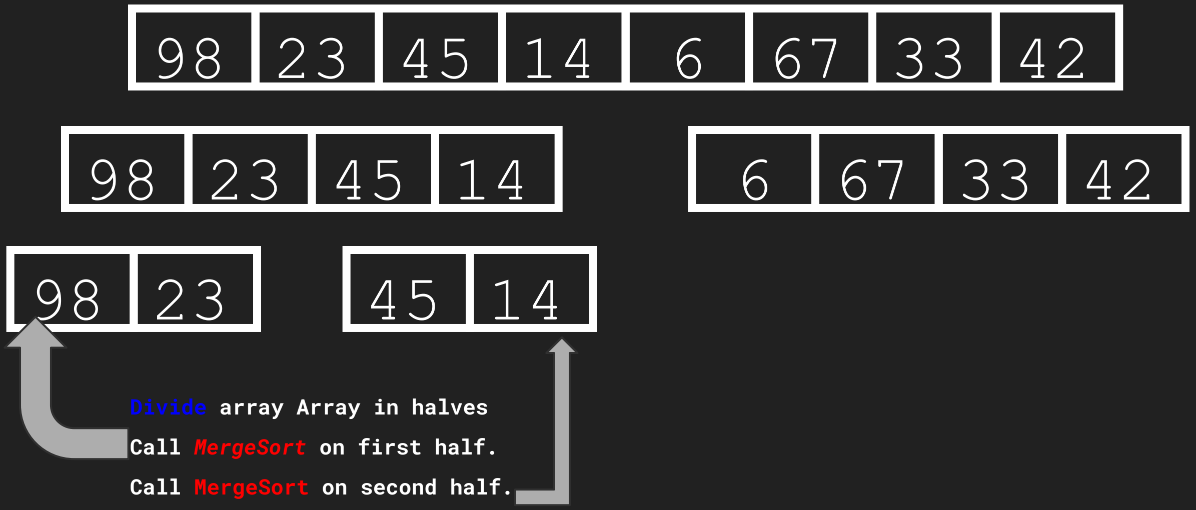
6 67 33 42

98 23

45 14

Narration: Repeat the mergesort part

Note however, we work on the ‘left’ side first, as that is the side we are recursing on first.



98	23	45	14	6	67	33	42

98	23	45	14

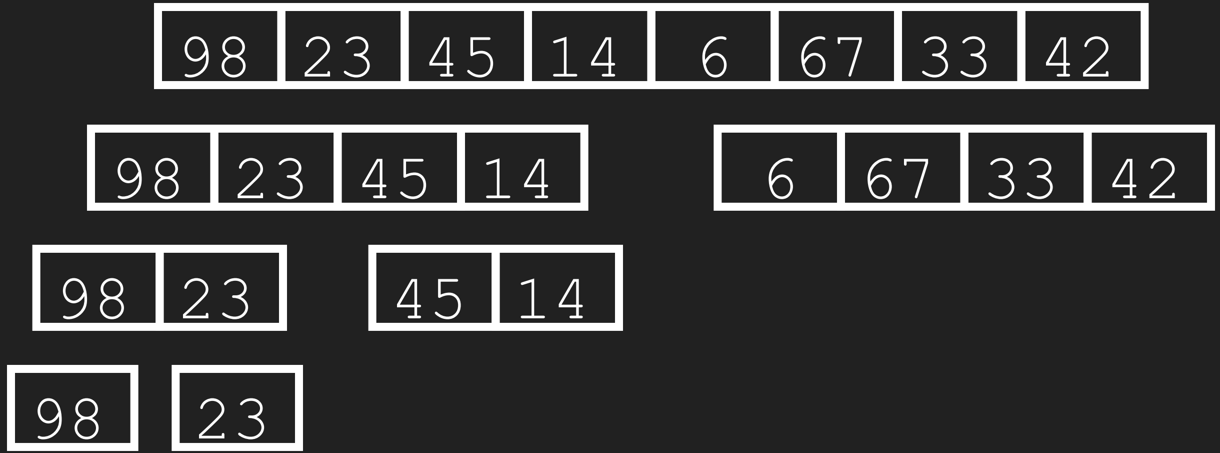
6	67	33	42

98	23

45	14

```
Divide array Array in halves
Call MergeSort on first half.
Call MergeSort on second half.
```

Narration: A given array of ‘unsorted’ numbers is given.



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

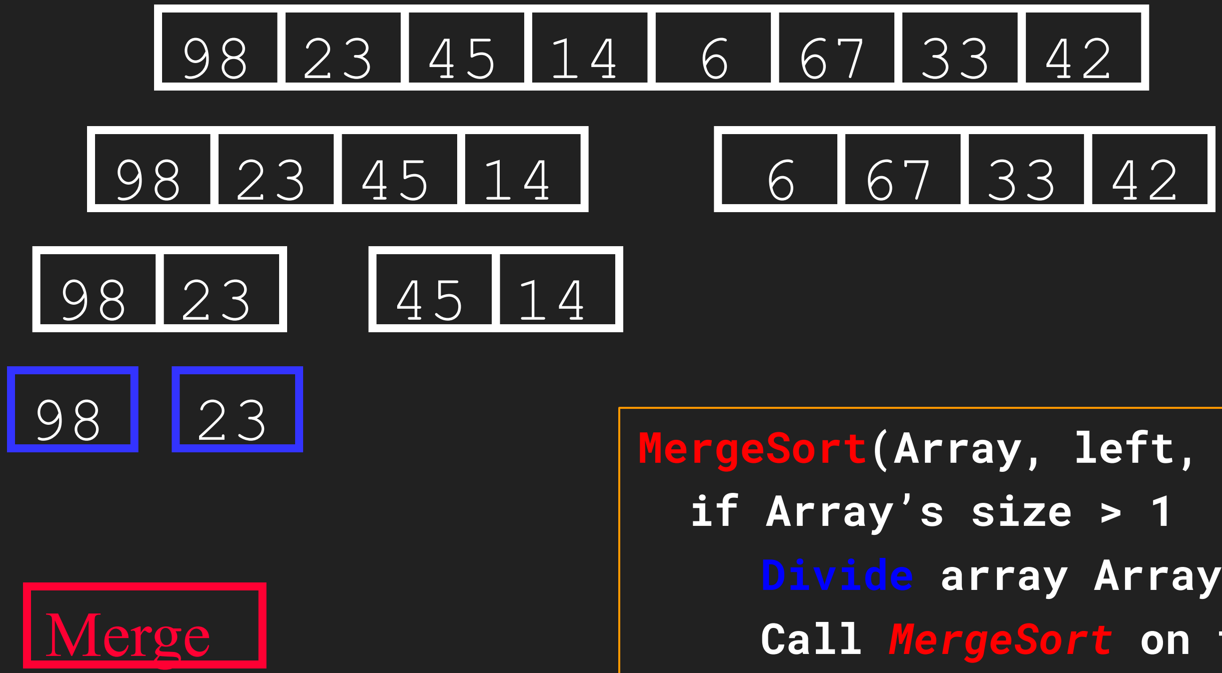
45 14

98

23

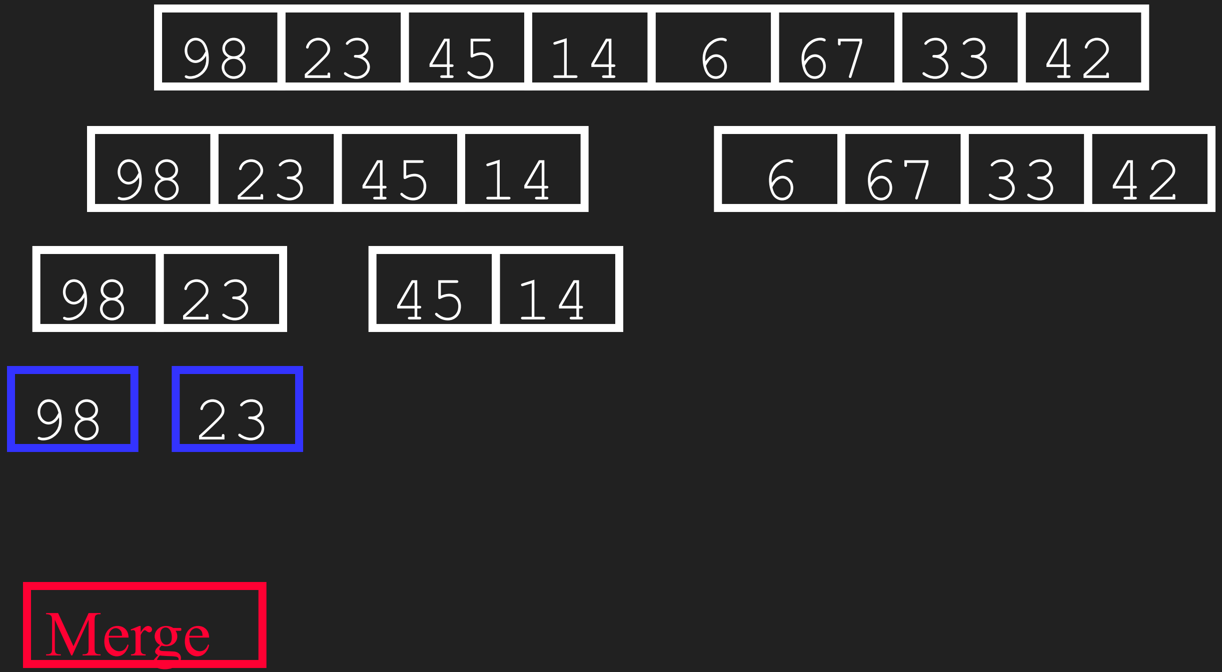
Narration: We would continue recursing, but we have met our base case!

Meaning, we cannot split an array of size 1 any further.



Narration: Combine 98 and 23

```
MergeSort(Array, left, right)
  if Array's size > 1
     Divide array Array in halves
     Call MergeSort on first half.
     Call MergeSort on second half.
     Merge two results (combine).
```



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

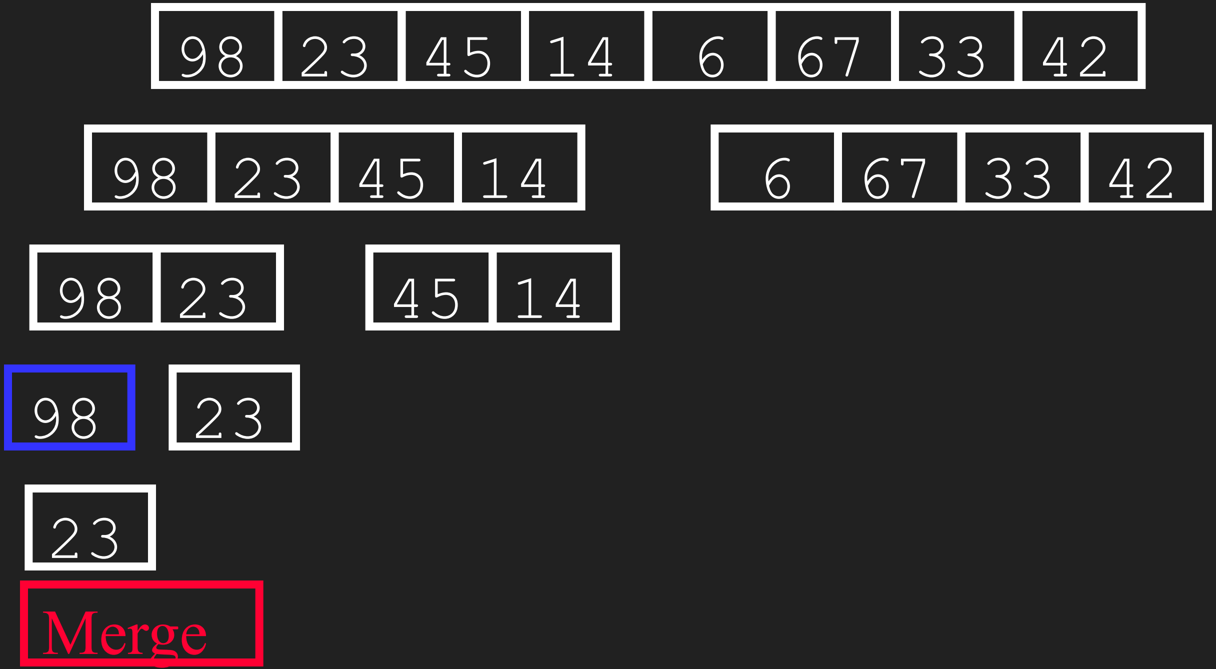
45 14

98

23

Merge

Narration: Executing Merge



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

98

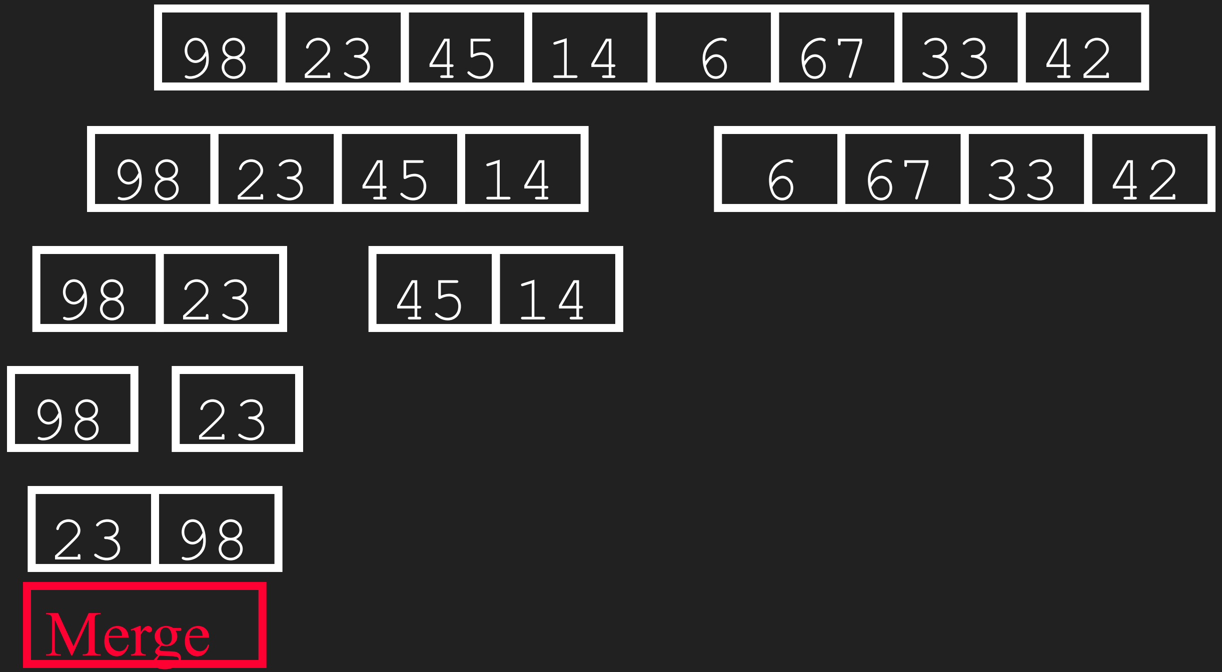
23

23

Merge

Narration: Merge ‘selects’ the smaller of the two items first.

Thus we will end with a sorted array of size two



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

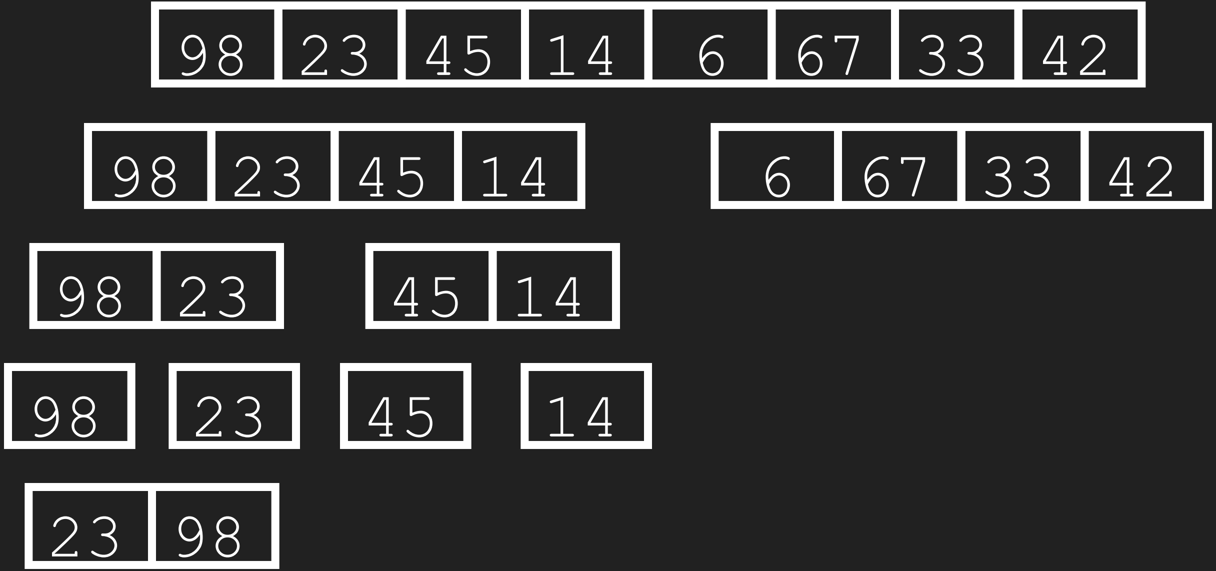
98 23   45 14

98   23

23 98

Merge

Narration: Sorted two elements



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

98

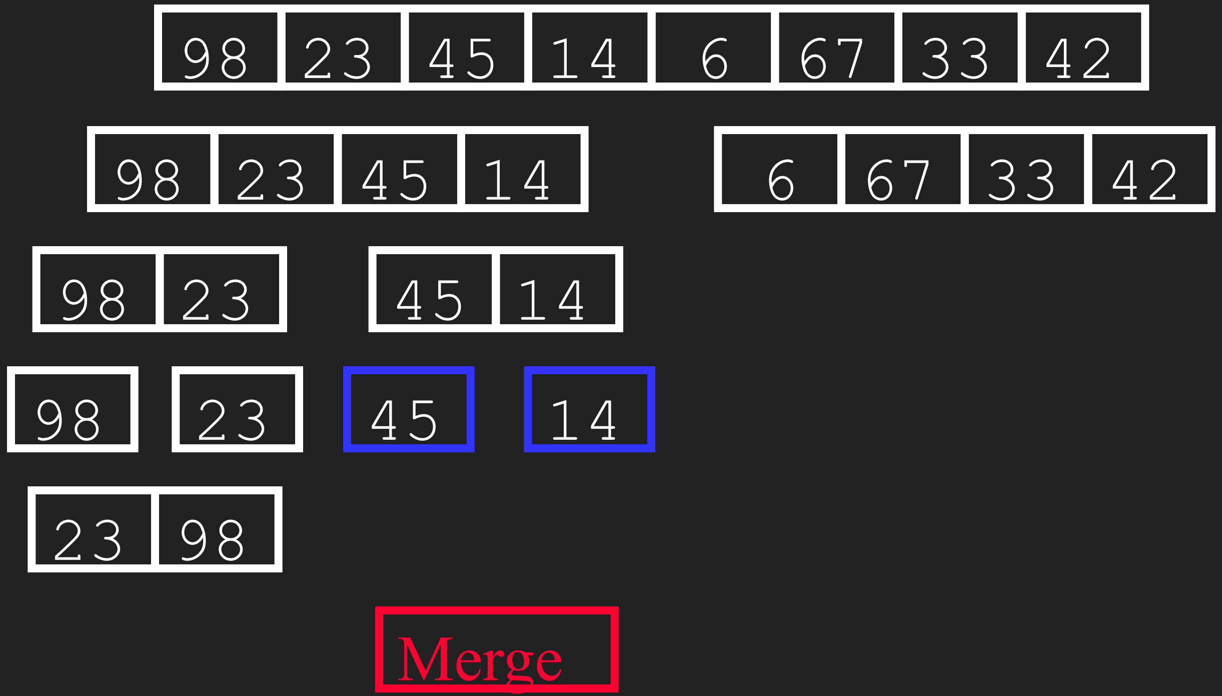
23

45

14

23 98

Narration: Repeat...



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

98

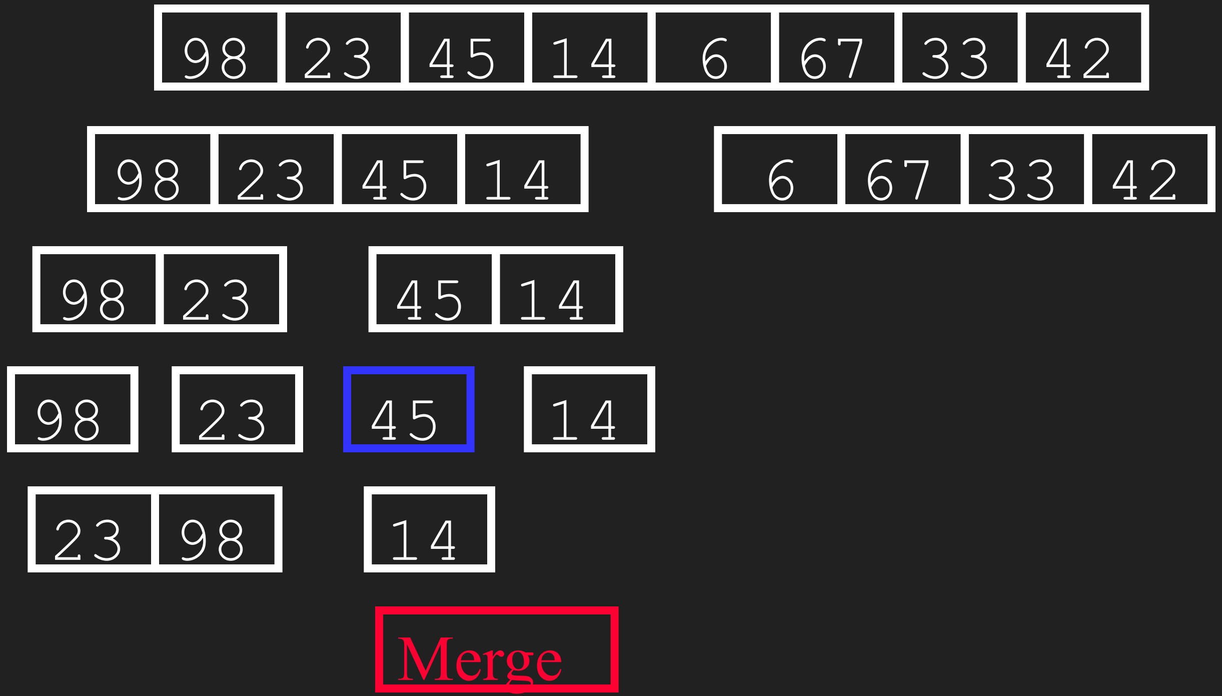
23

45

14

23 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

98

23

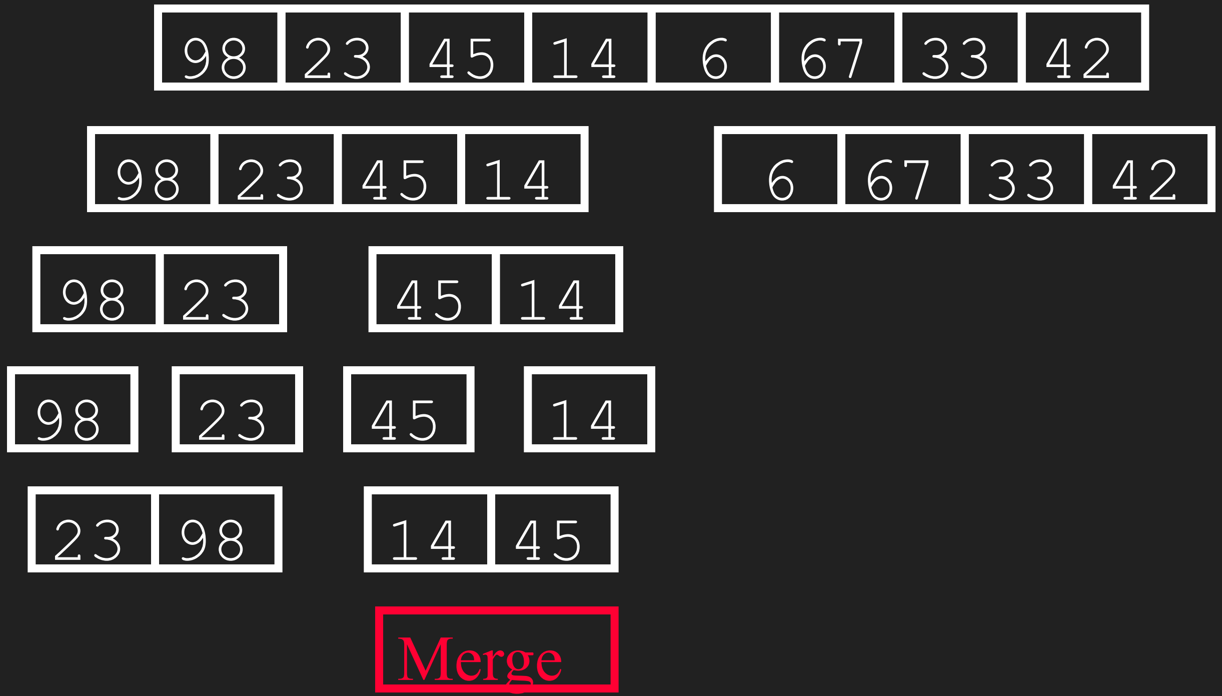
45

14

23 98

14

Merge



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

98

23

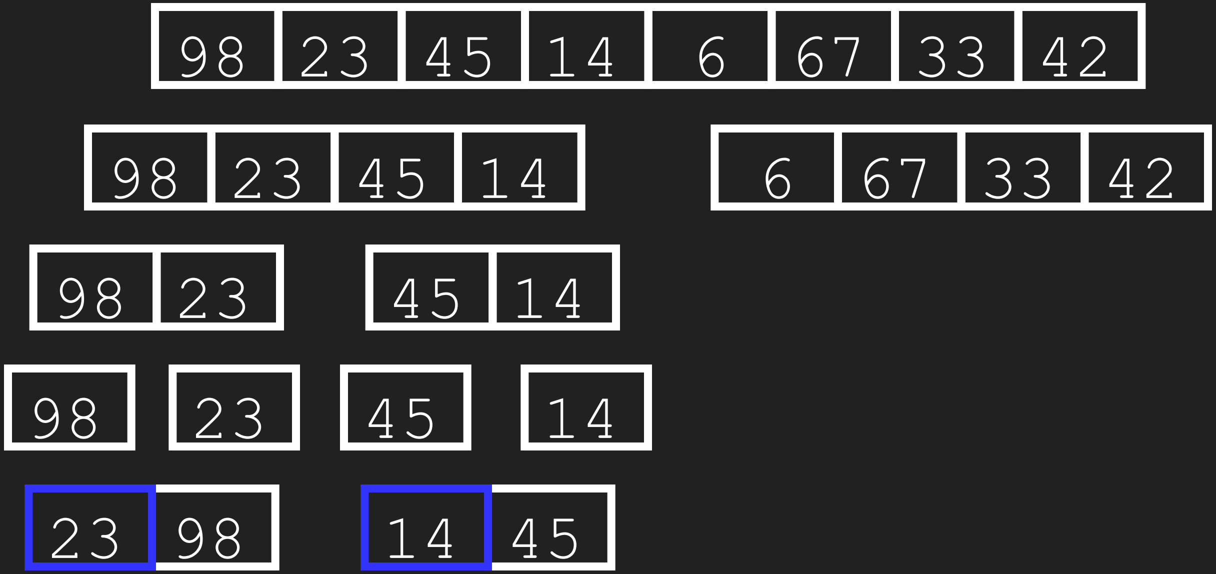
45

14

23 98

14 45

Merge



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

98

23

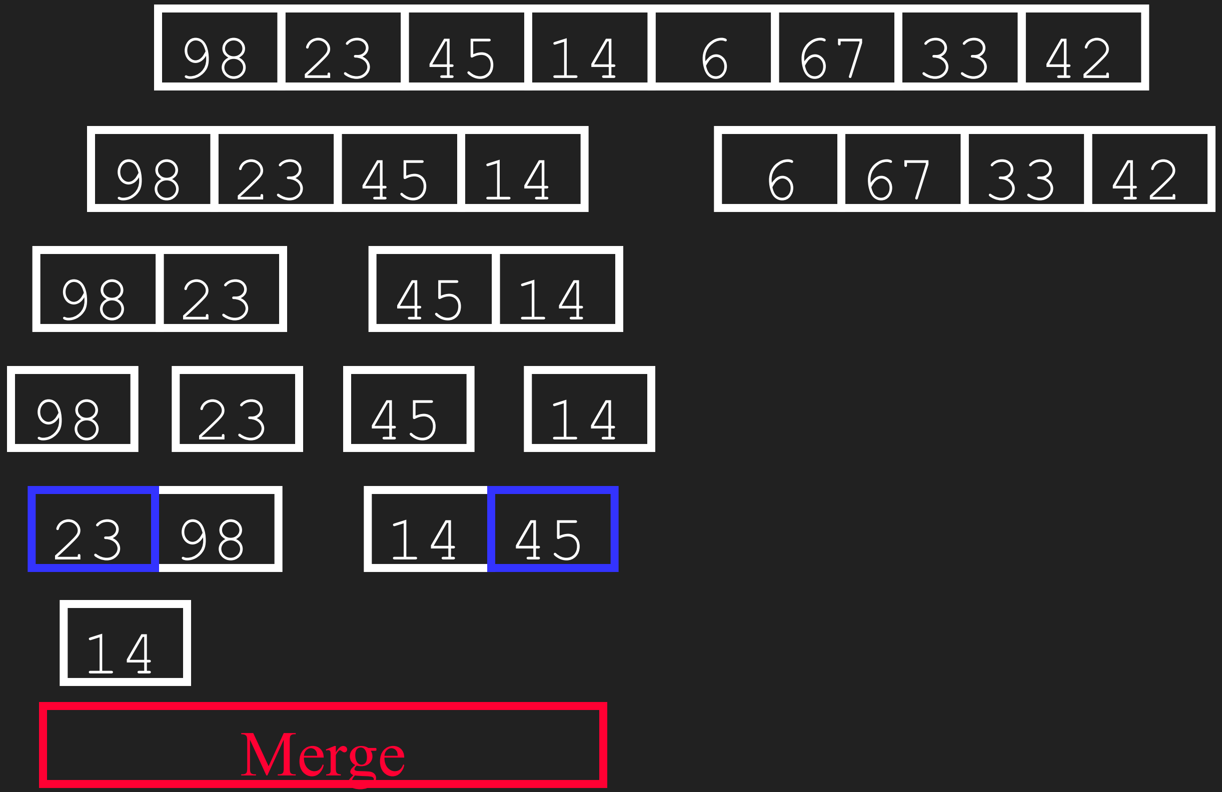
45

14

23 98

14 45

Merge



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

98

23

45

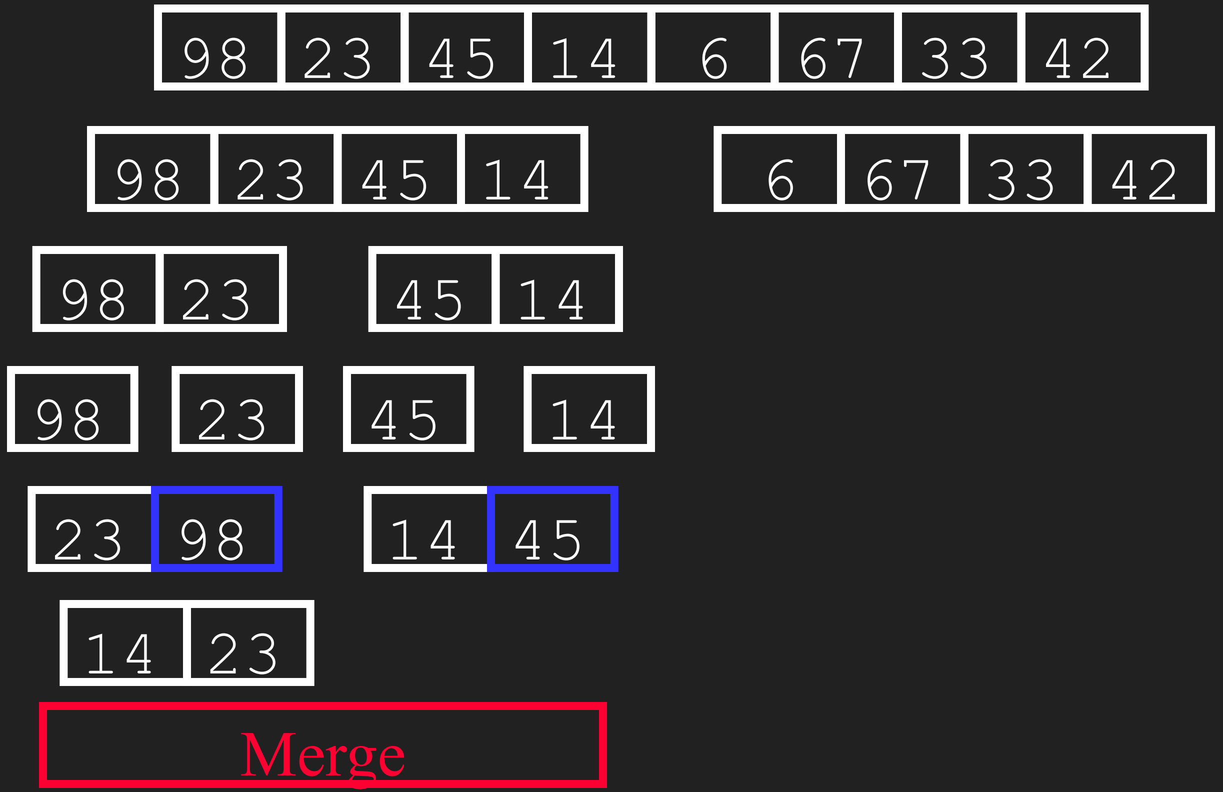
14

23 98

14 45

14

Merge



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

98

23

45

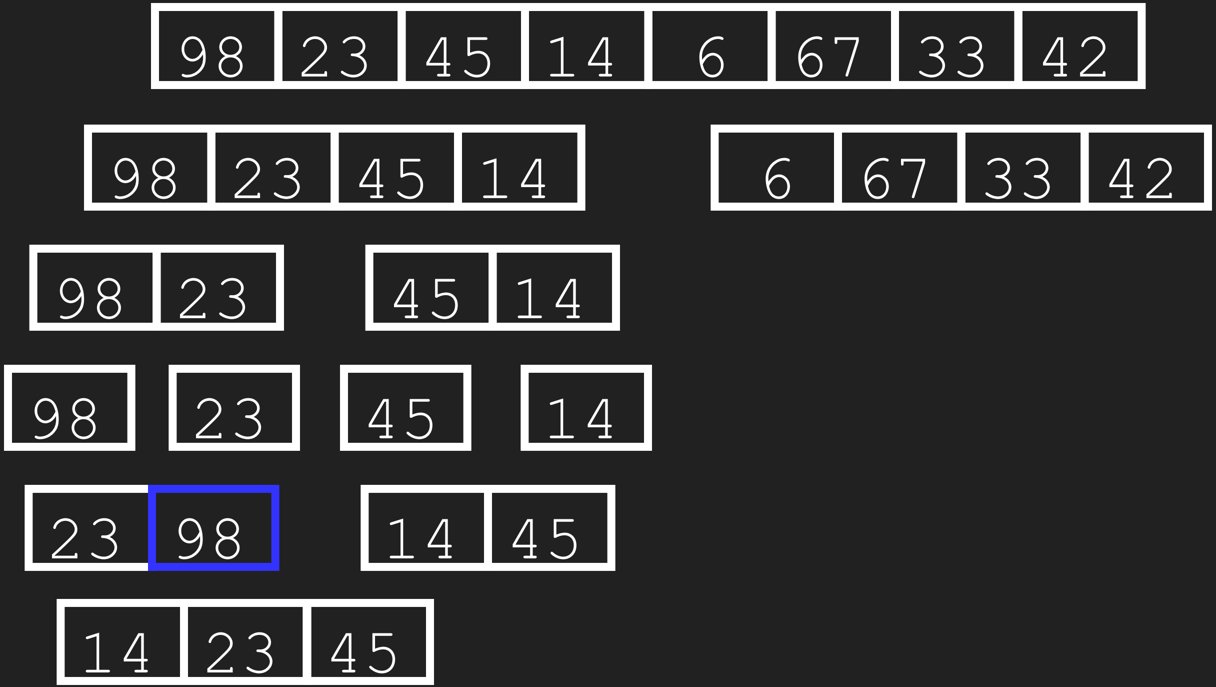
14

23 98

14 45

14 23

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

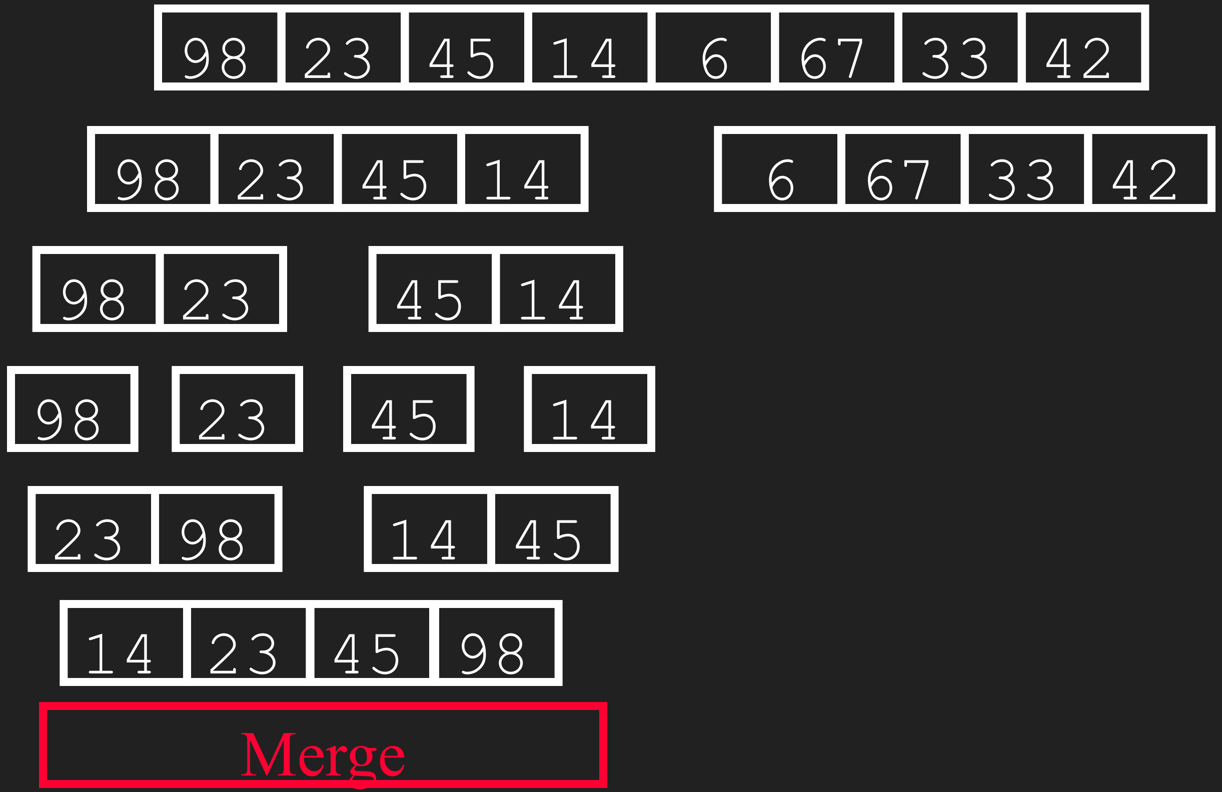
98 23 | 45 14

98 | 23 | 45 | 14

23 98 | 14 45

14 23 45

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

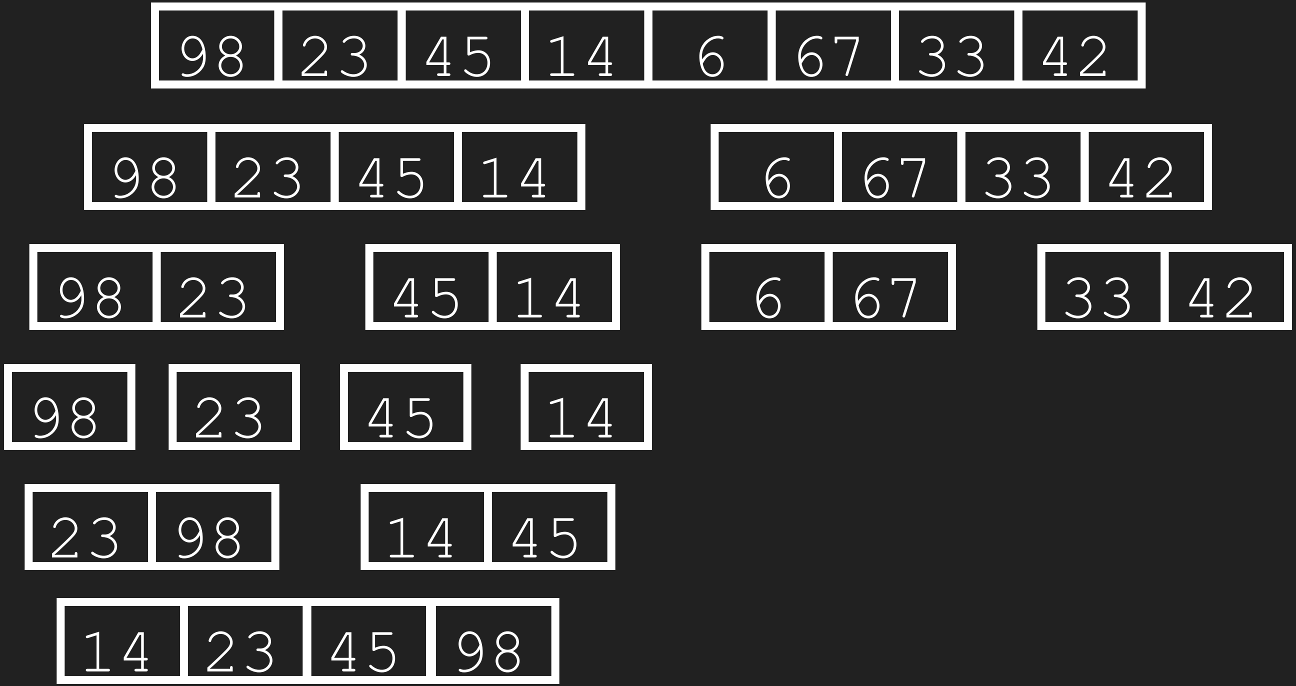
98 23   45 14

98   23   45   14

23 98   14 45

14 23 45 98

Merge



98 23 45 14 6 67 33 42

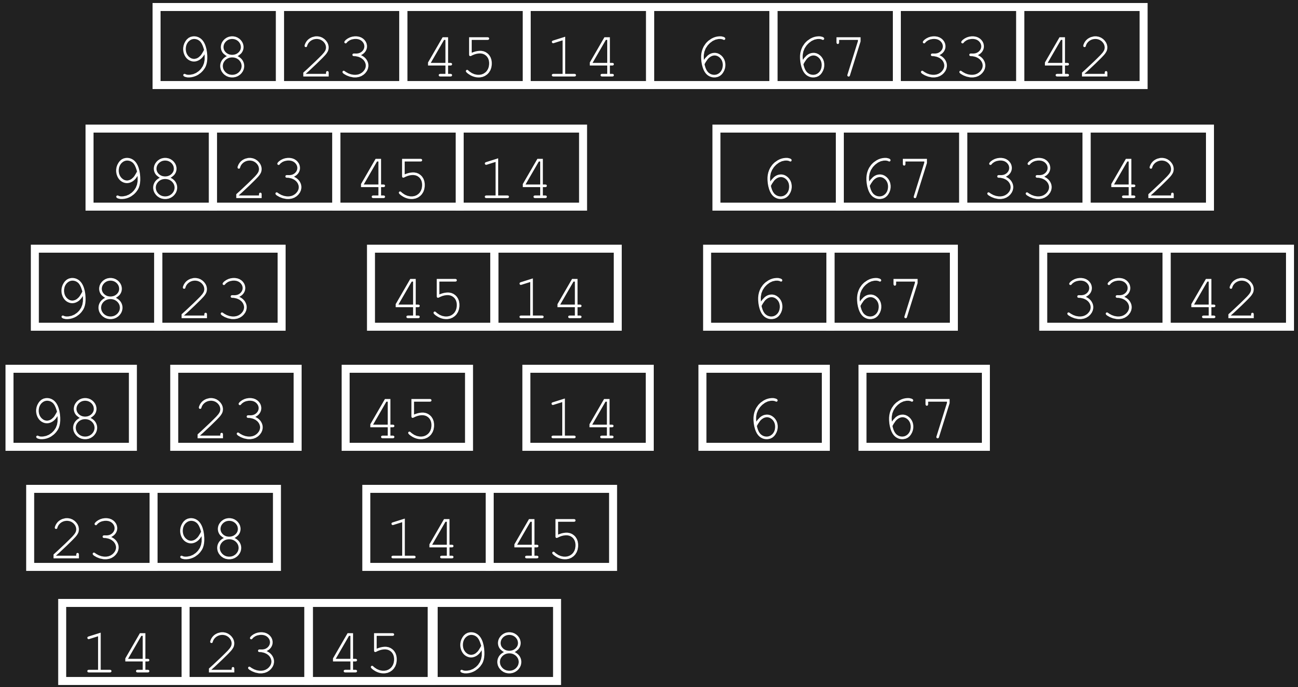
98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14

23 98 | 14 45

14 23 45 98



98 23 45 14 6 67 33 42

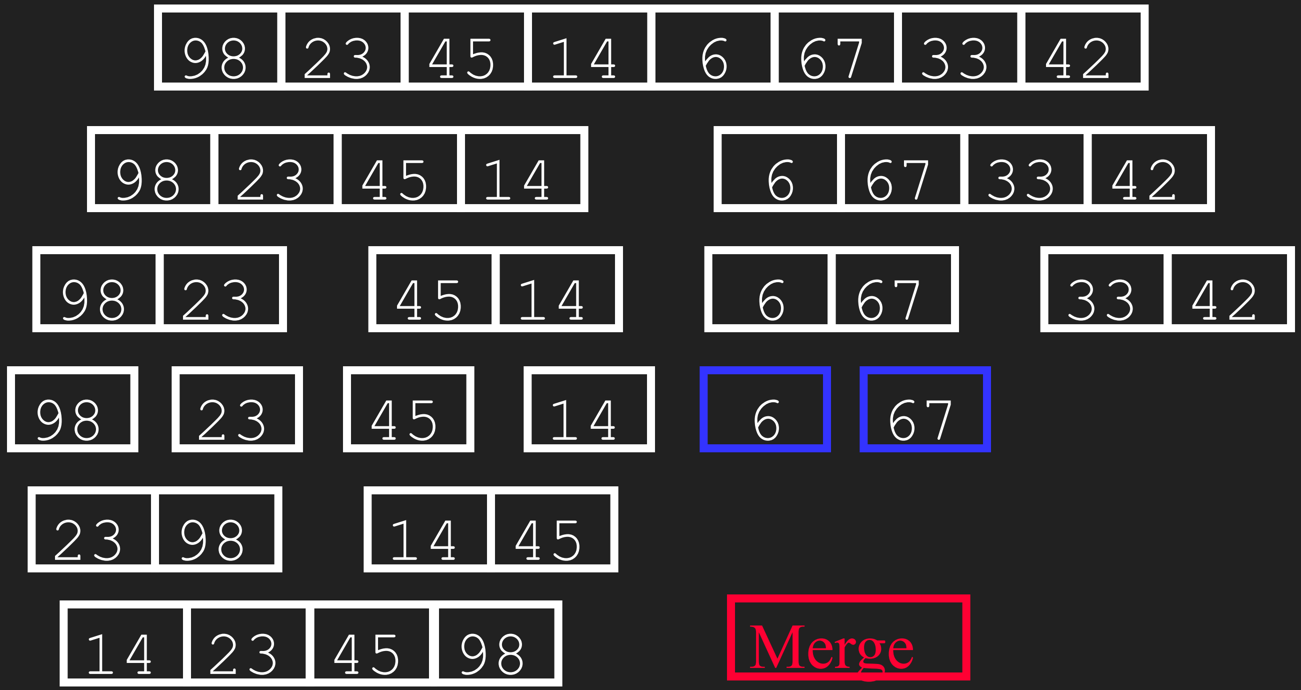
98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67

23 98 | 14 45

14 23 45 98



98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

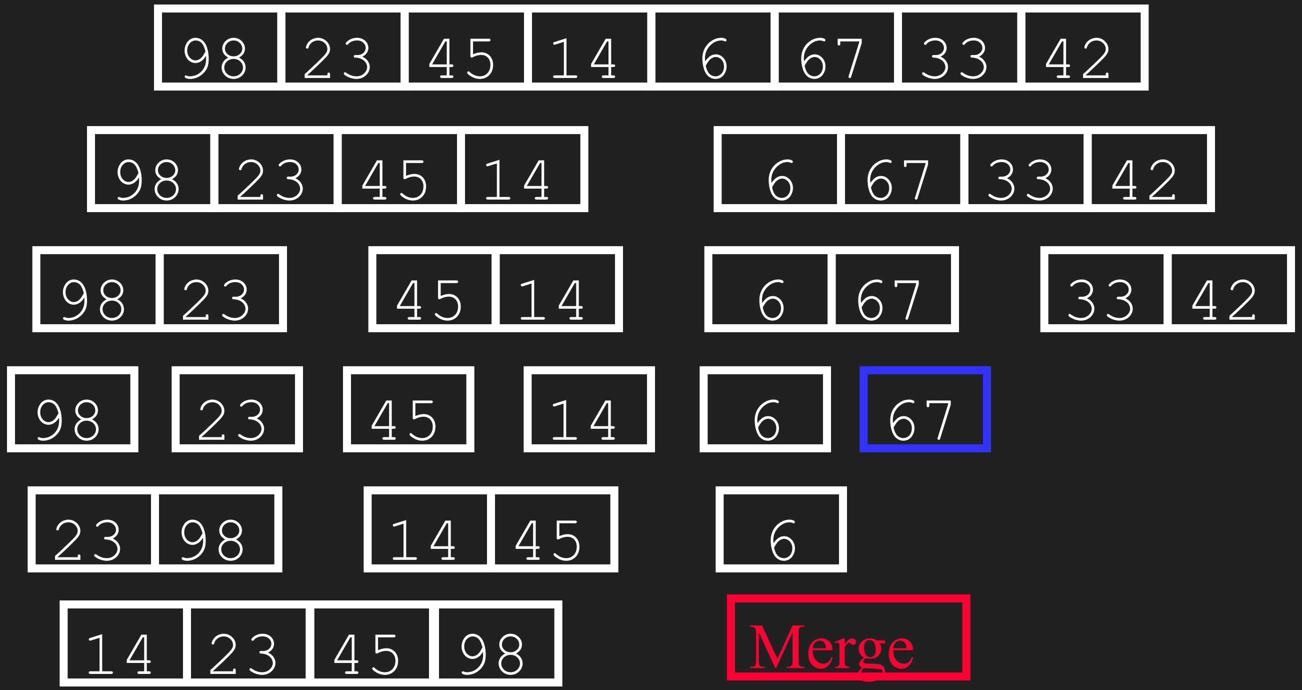
98 23 45 14 6 67 33 42

98 23 45 14 6 67

23 98 14 45

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

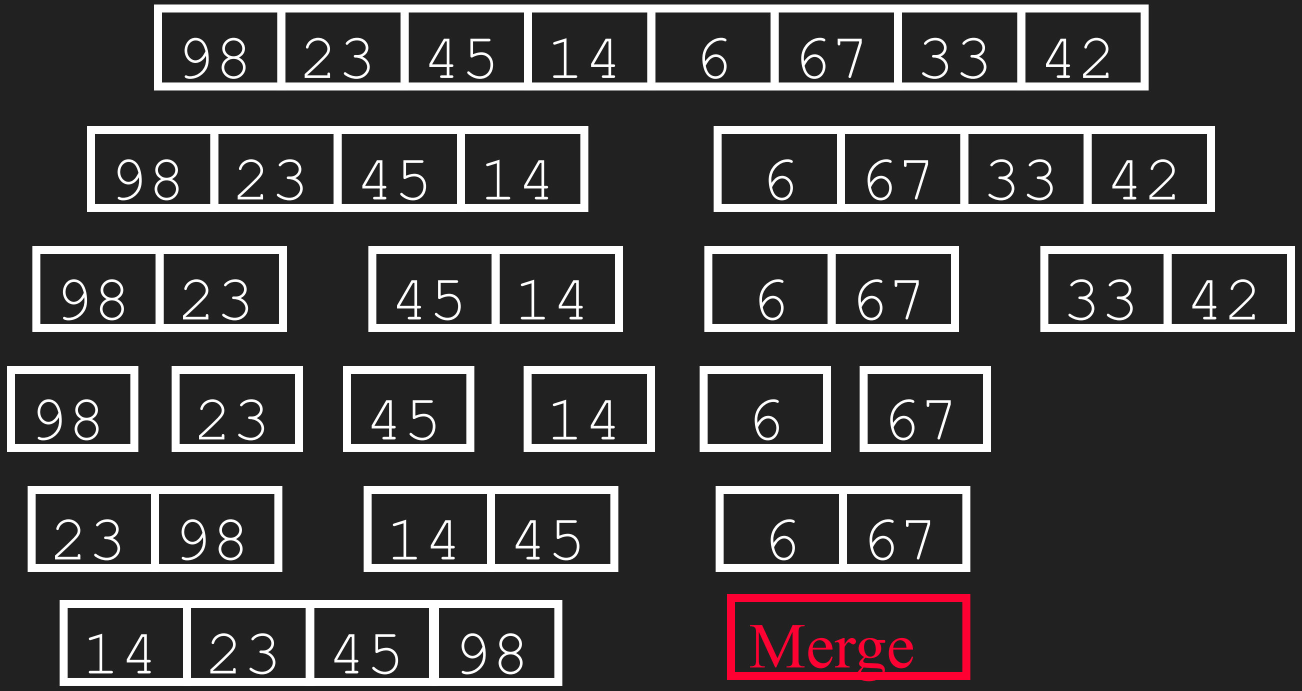
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67

23 98 | 14 45 | 6

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

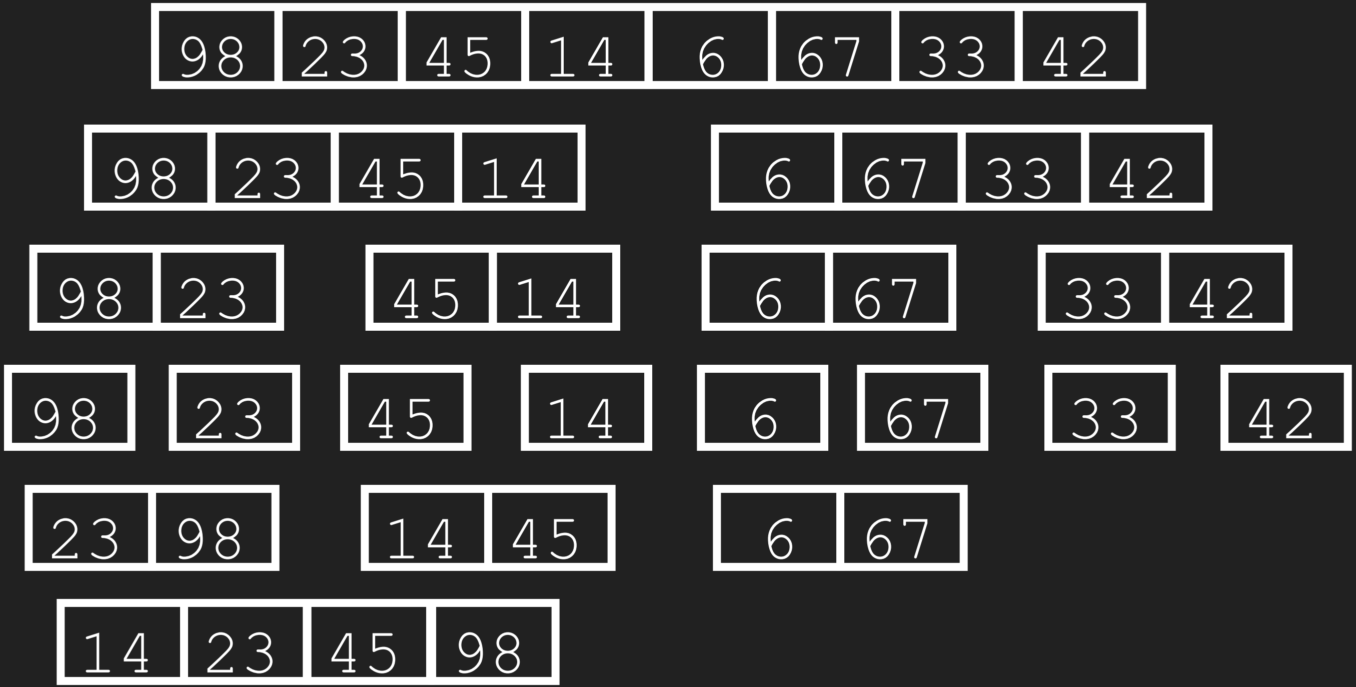
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67

23 98 | 14 45 | 6 67

14 23 45 98

Merge



98 23 45 14 6 67 33 42

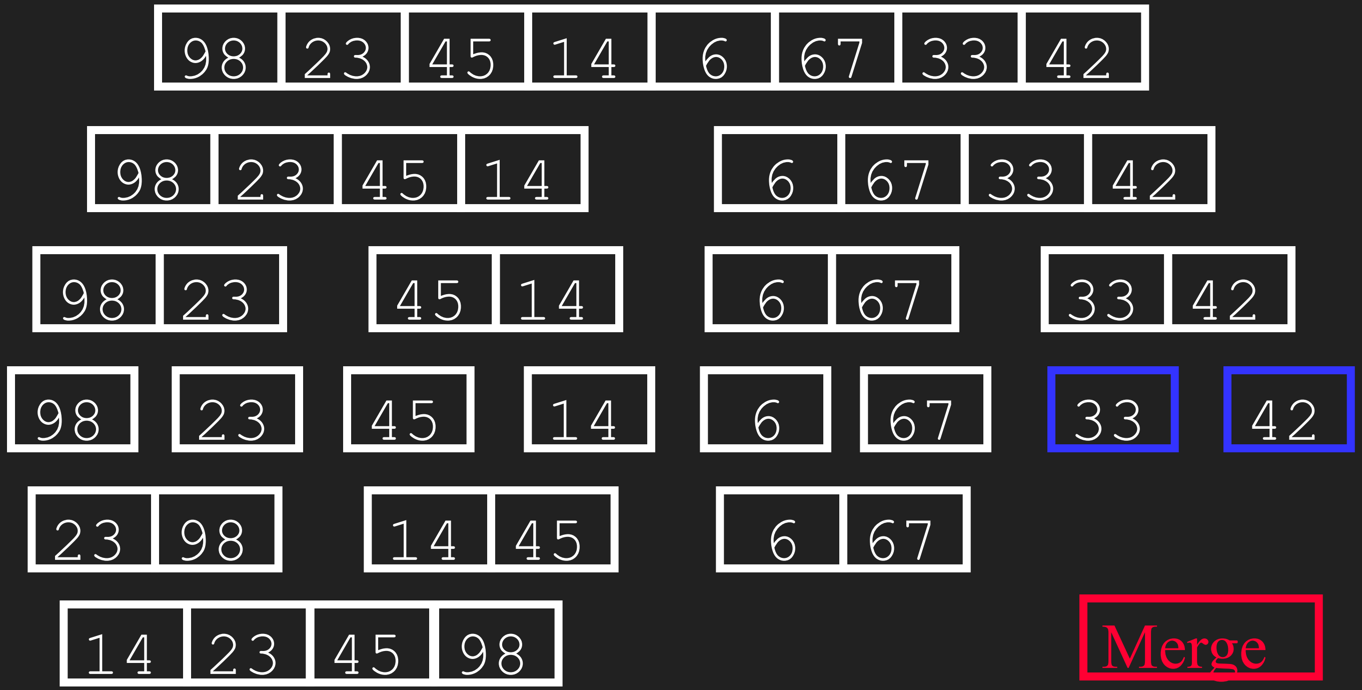
98 23 45 14   6 67 33 42

98 23   45 14   6 67   33 42

98   23   45   14   6   67   33   42

23 98   14 45   6 67

14 23 45 98



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

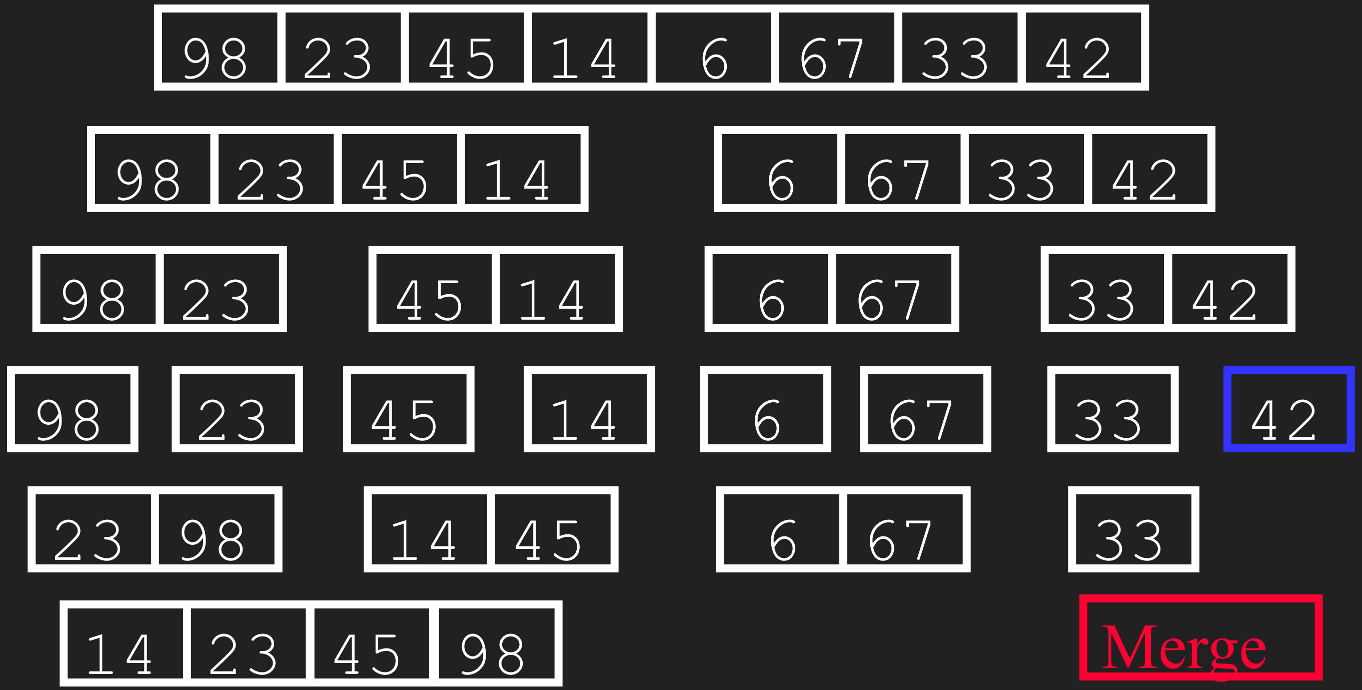
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

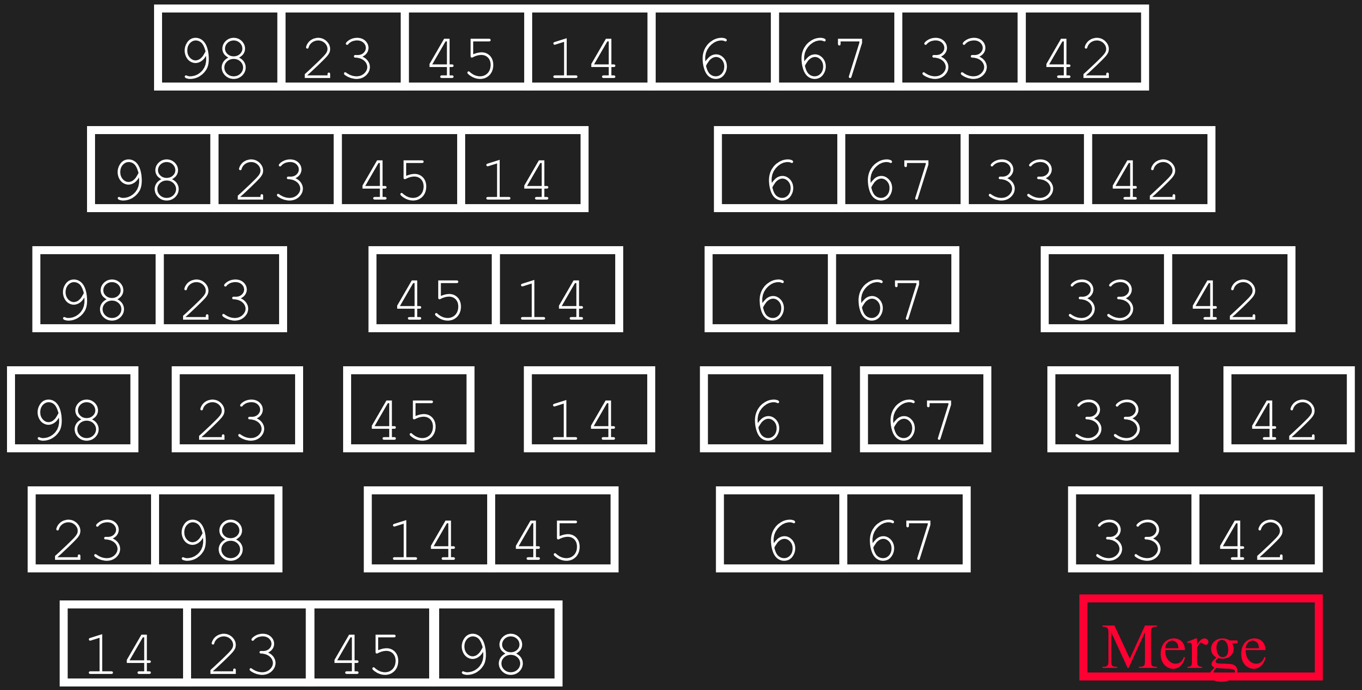
98 23   45 14   6 67   33 42

98   23   45   14   6   67   33   42

23 98   14 45   6 67   33

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

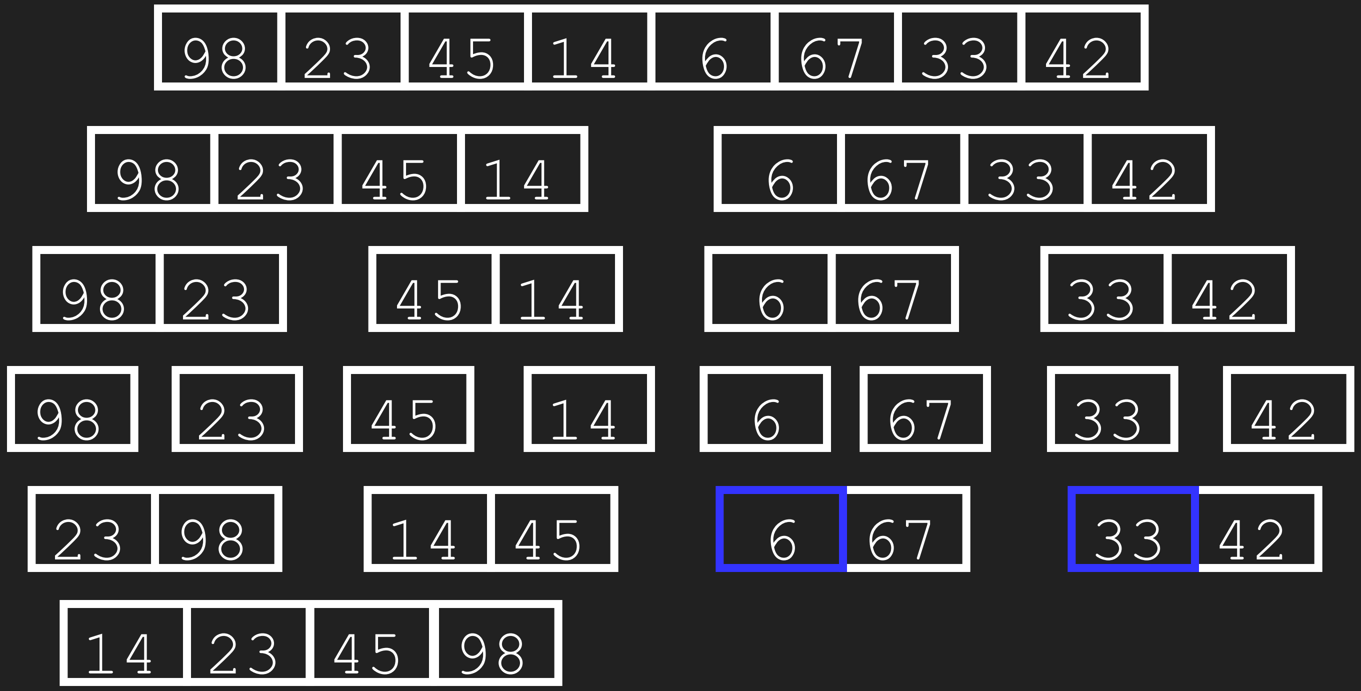
98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

23 98 14 45 6 67 33 42

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

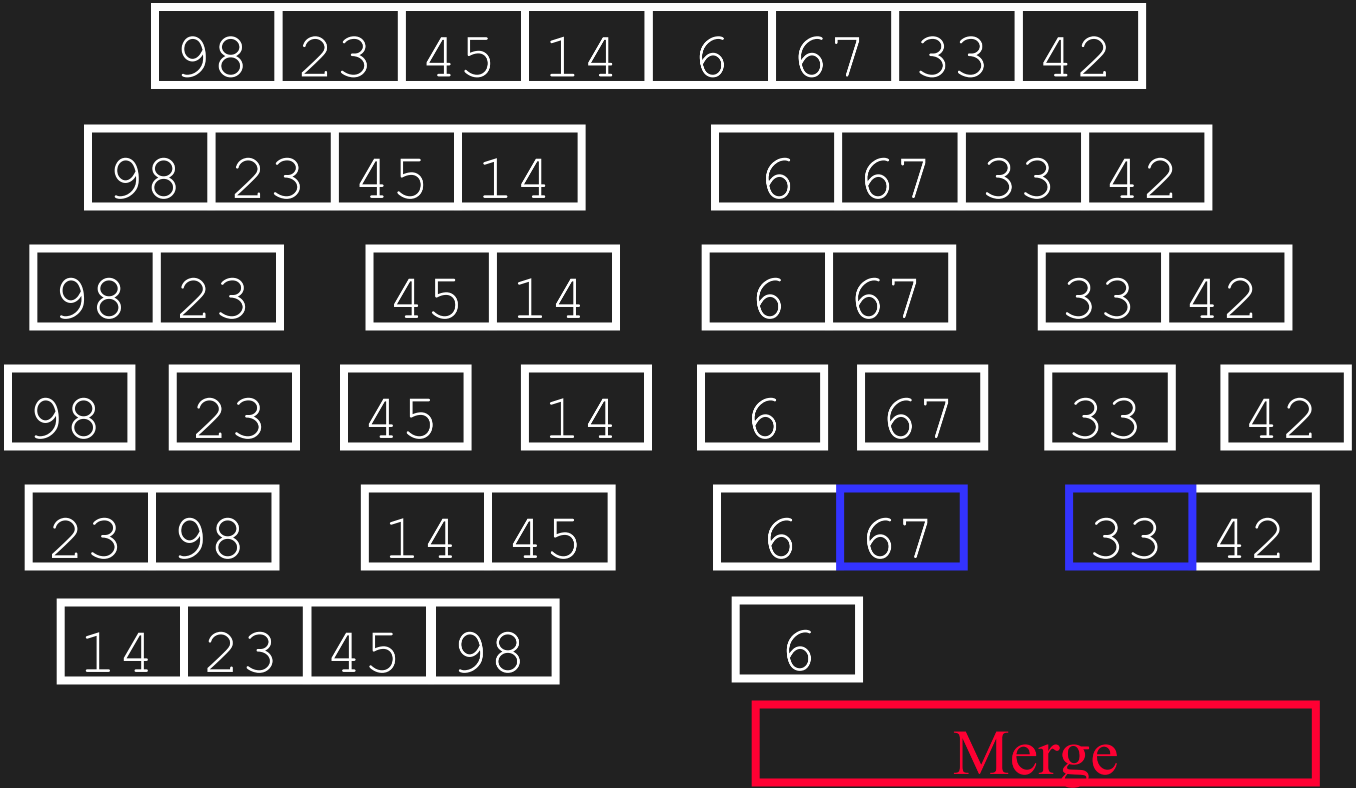
98 23   45 14   6 67   33 42

98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14

6 67 33 42

98 23

45 14

6 67

33 42

98

23

45

14

6

67

33

42

23 98

14 45

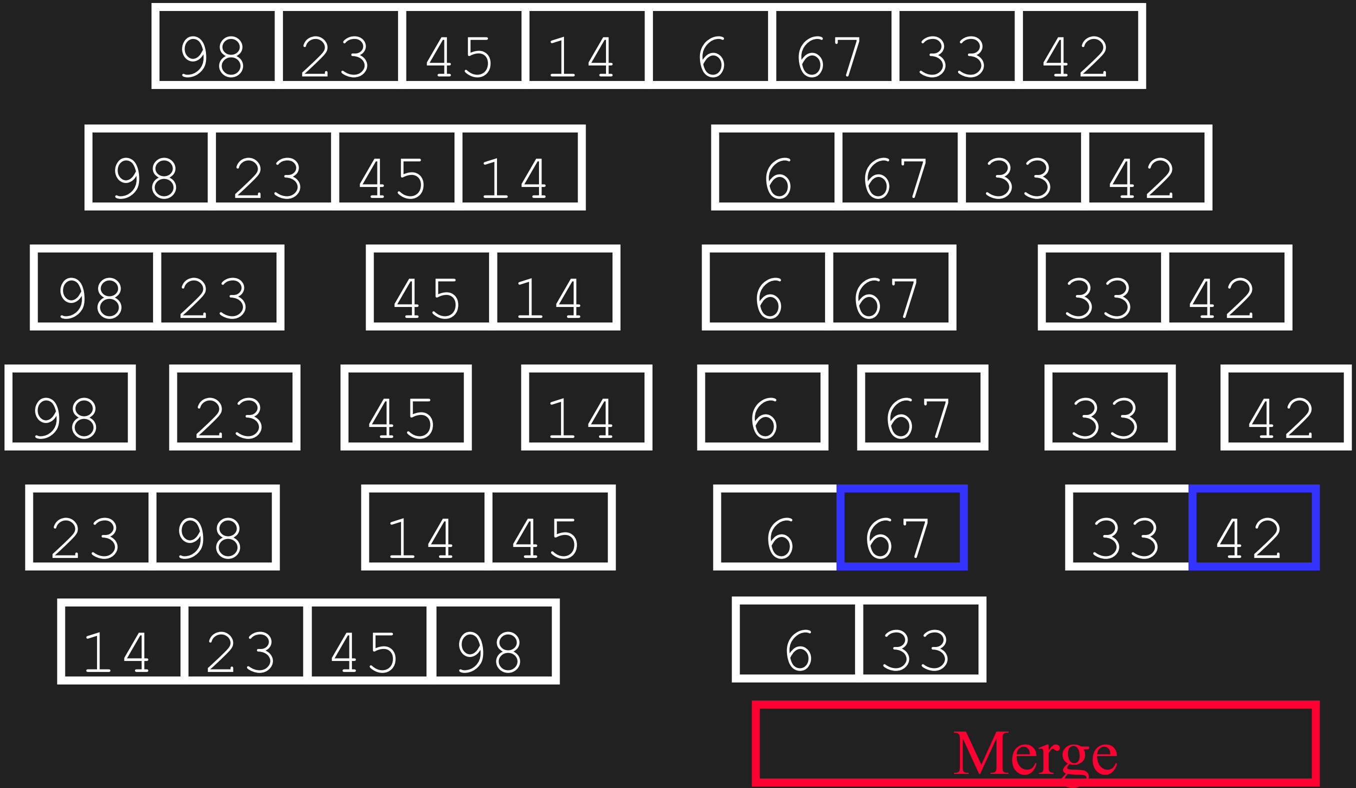
6 67

33 42

14 23 45 98

6

Merge



98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

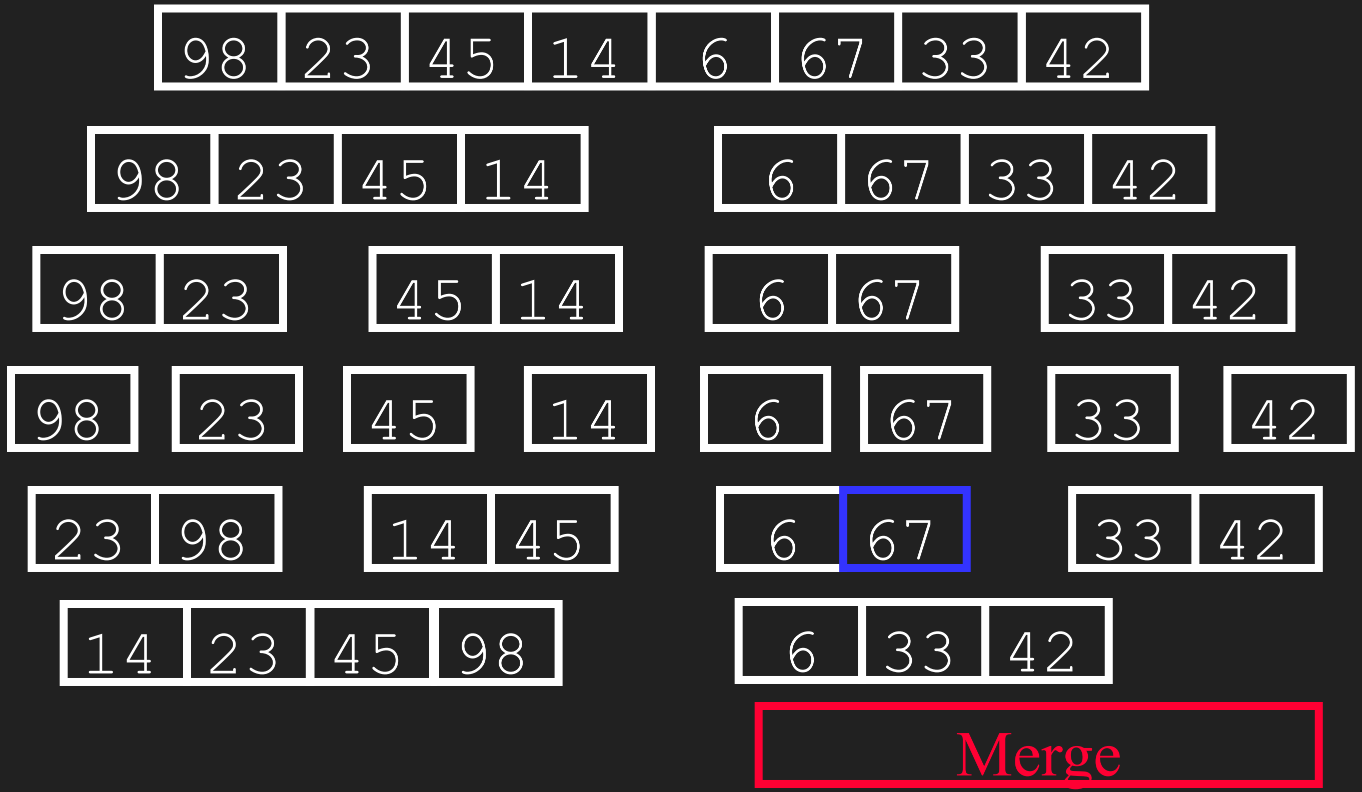
98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

23 98 14 45 6 67 33 42

14 23 45 98 6 33

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

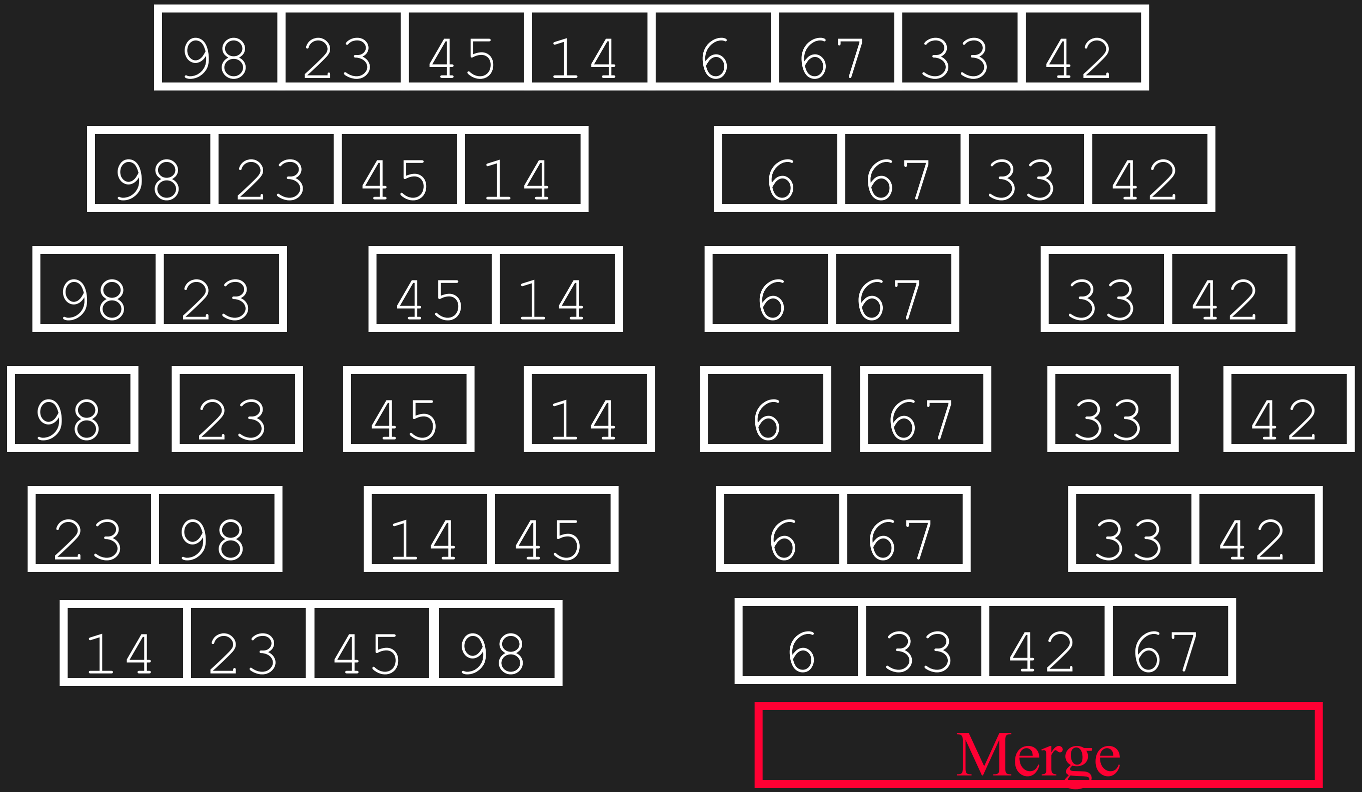
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

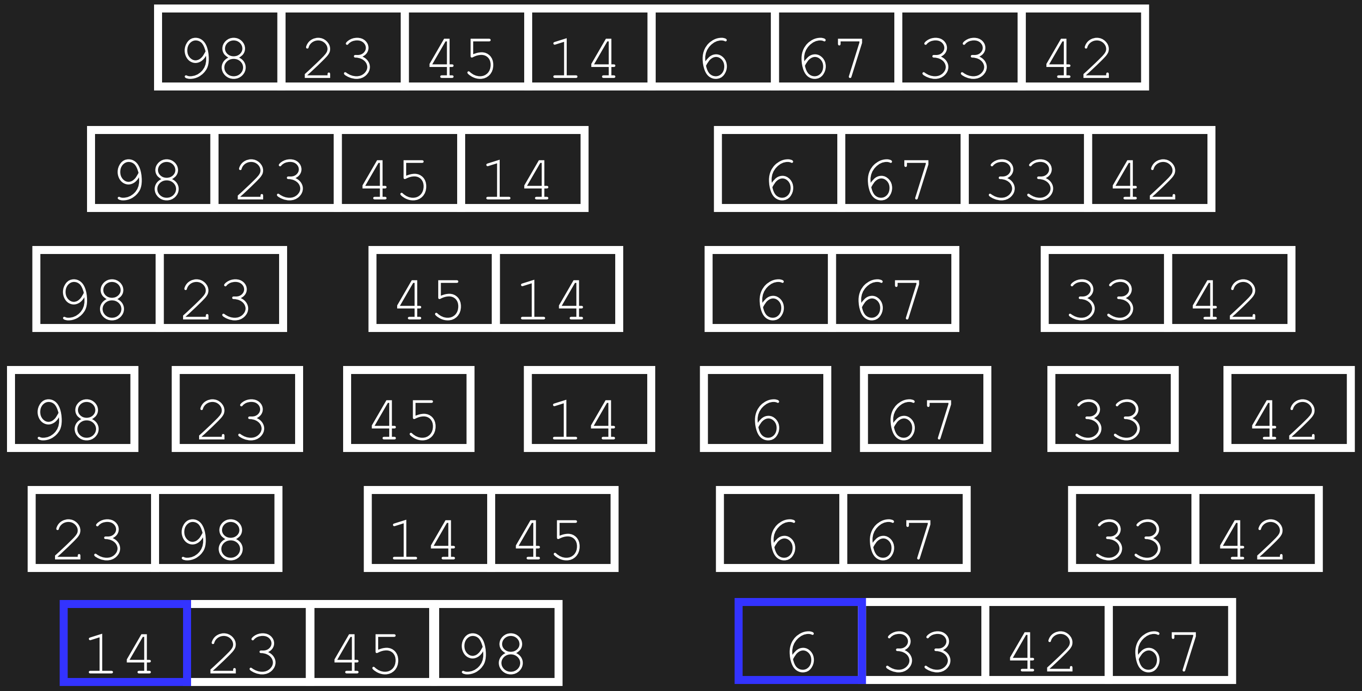
98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

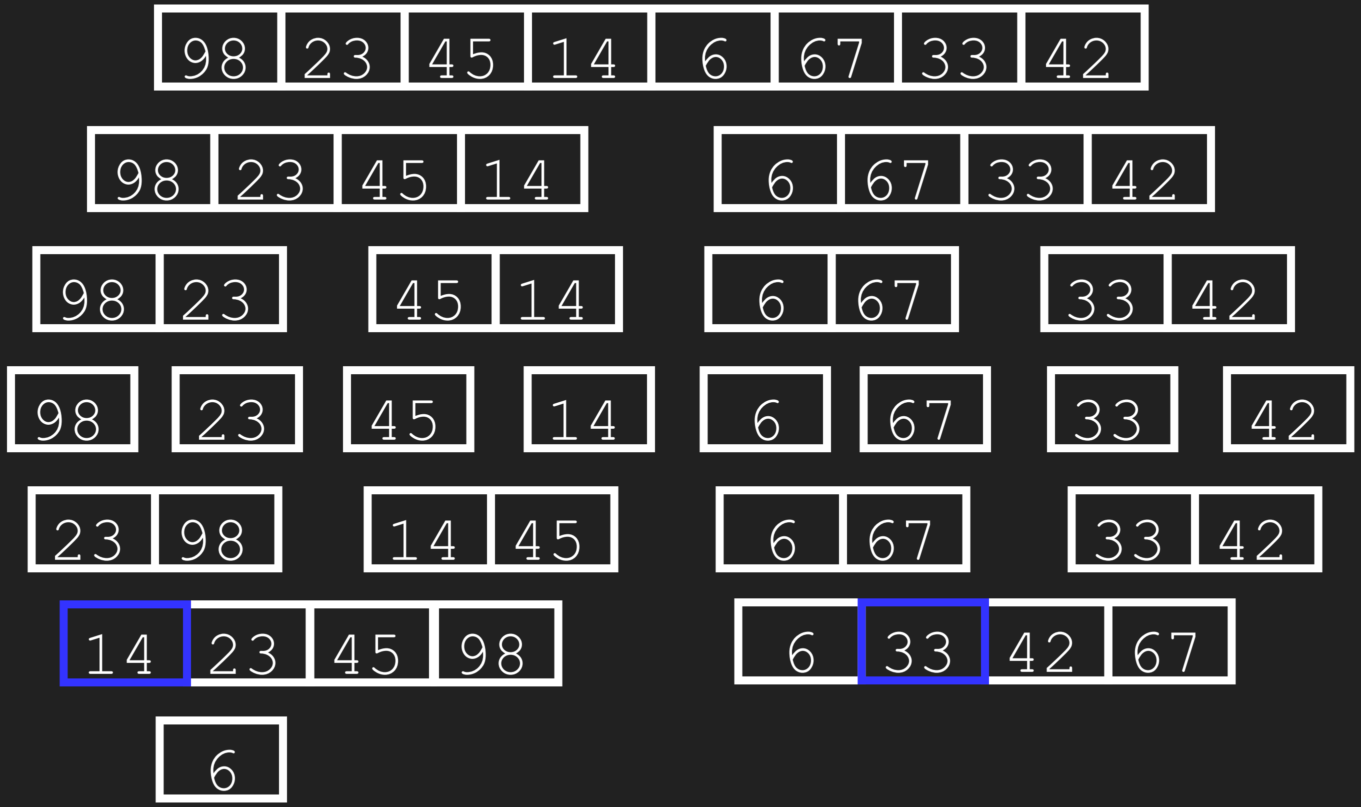
98 23   45 14   6 67   33 42

98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6 33 42 67

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

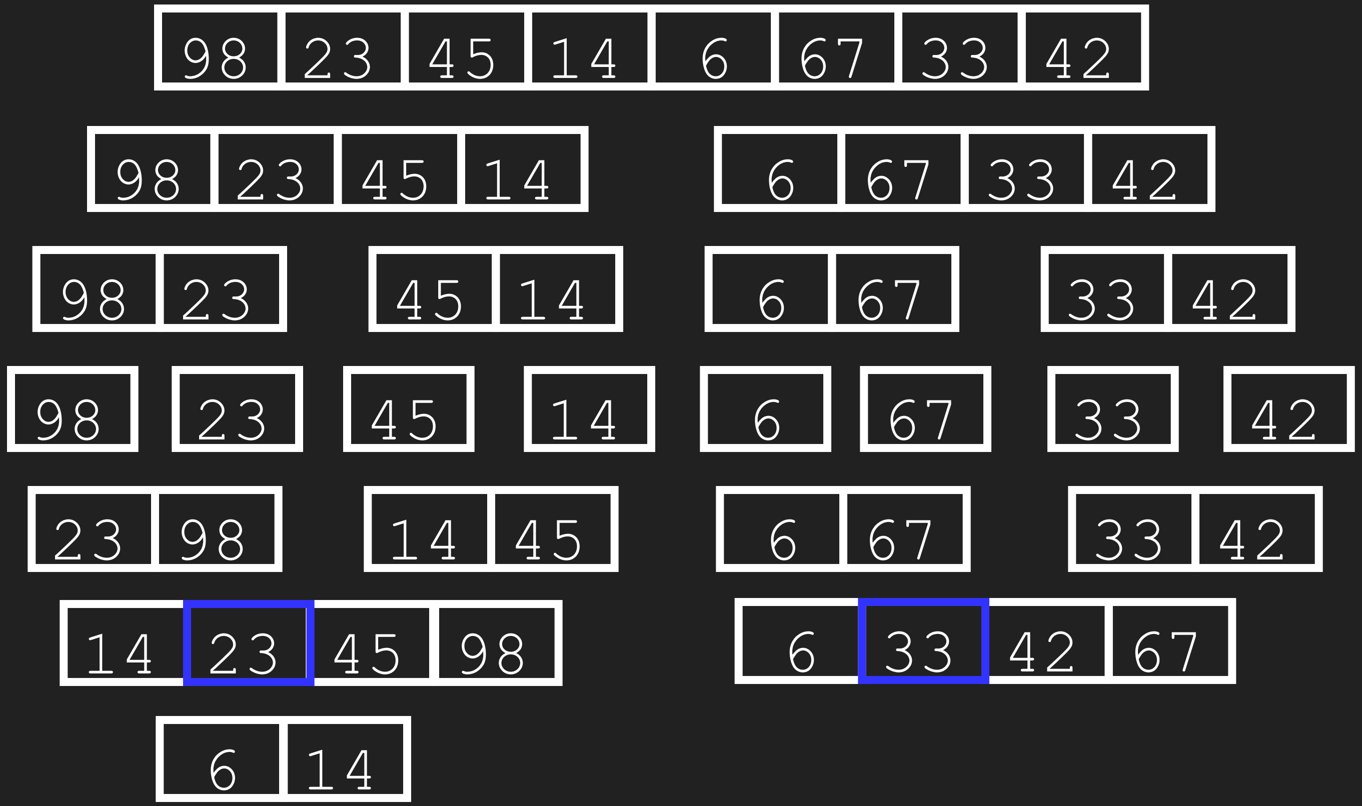
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

98 23   45 14   6 67   33 42

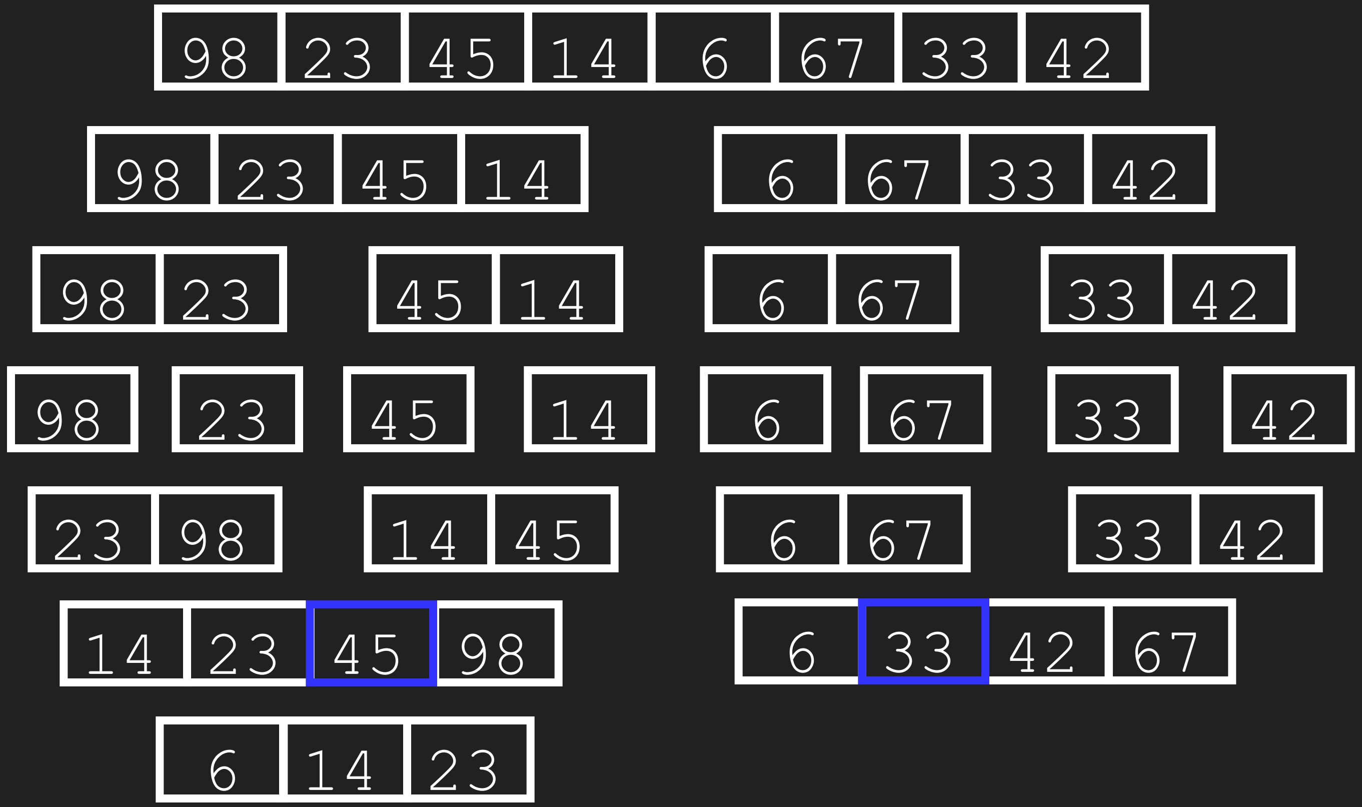
98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6 33 42 67

6 14

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

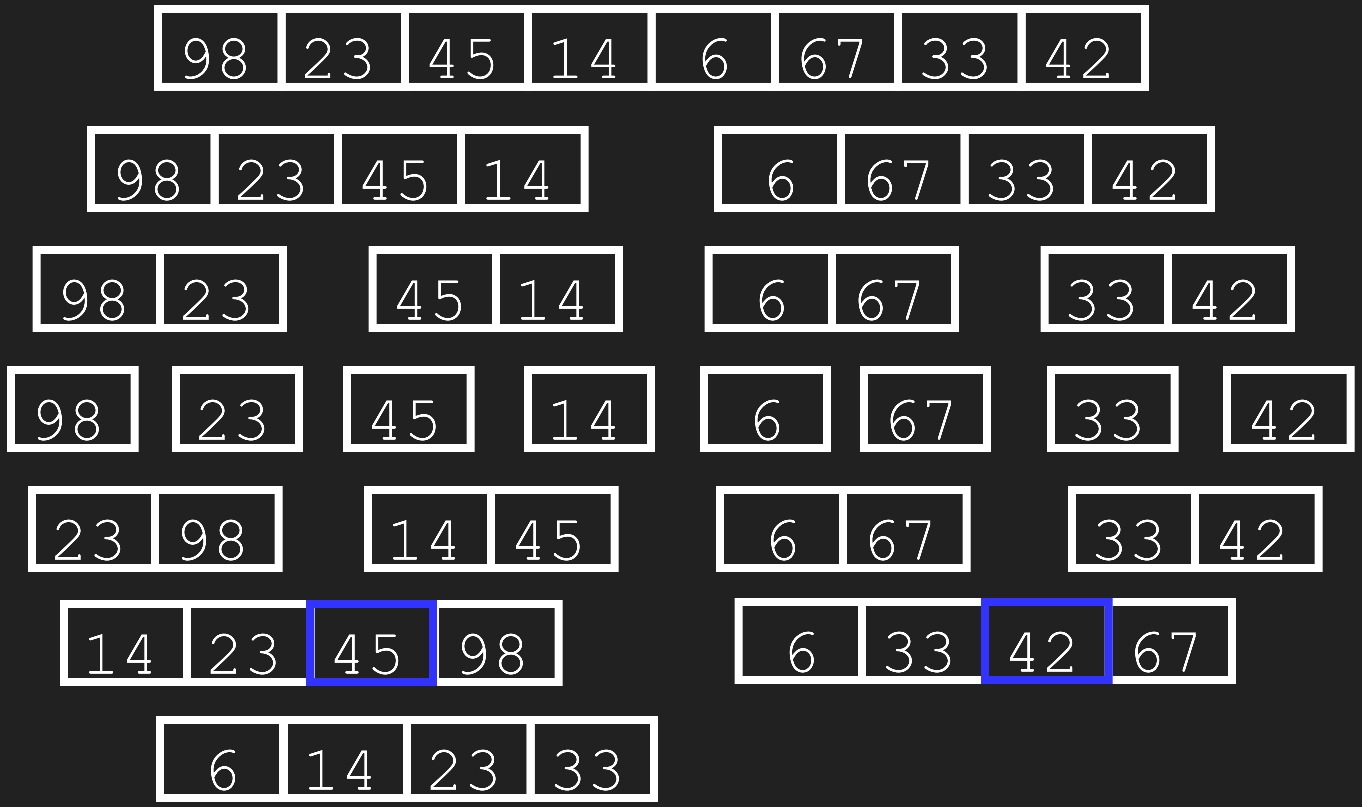
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23

Merge



98 23 45 14 6 67 33 42

98 23 45 14   6 67 33 42

98 23   45 14   6 67   33 42

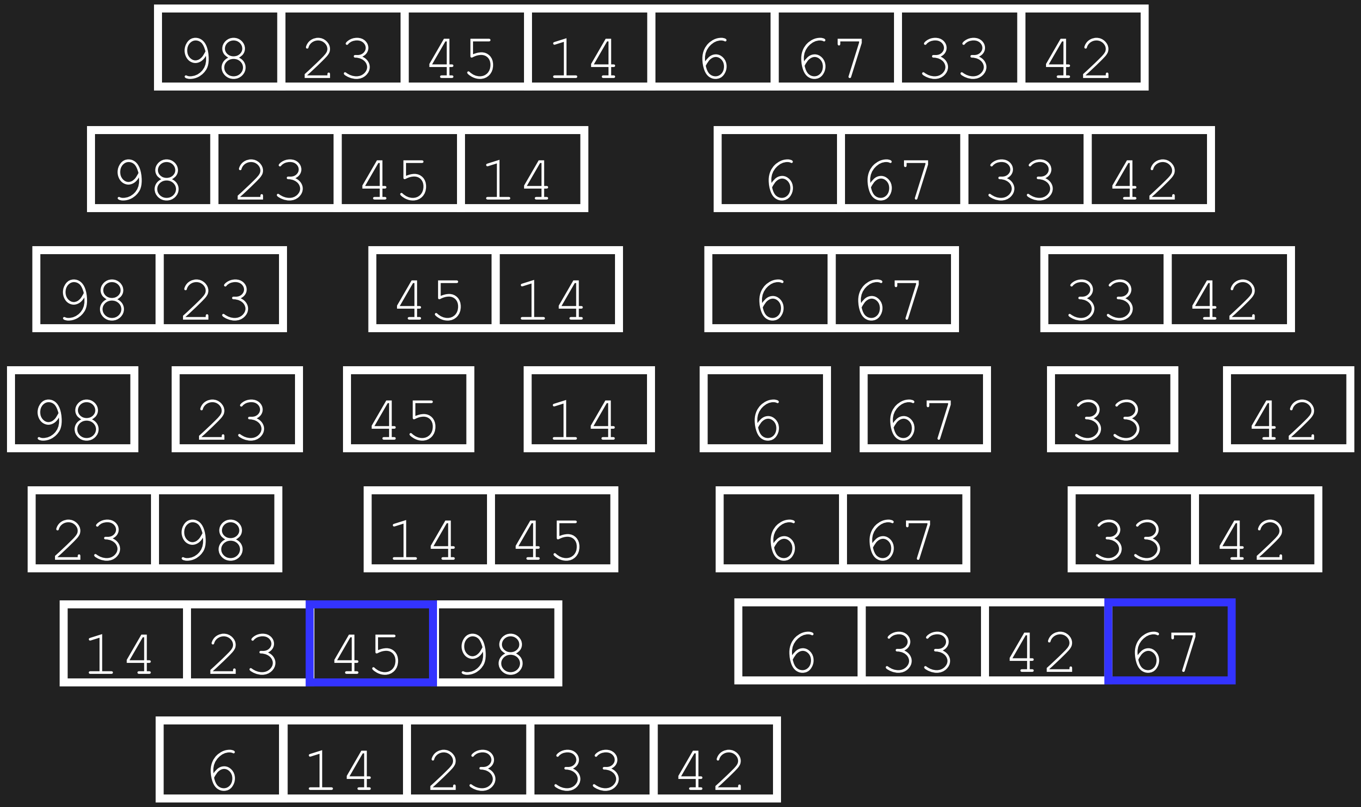
98   23   45   14   6   67   33   42

23 98   14 45   6 67   33 42

14 23 45 98   6 33 42 67

6 14 23 33

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

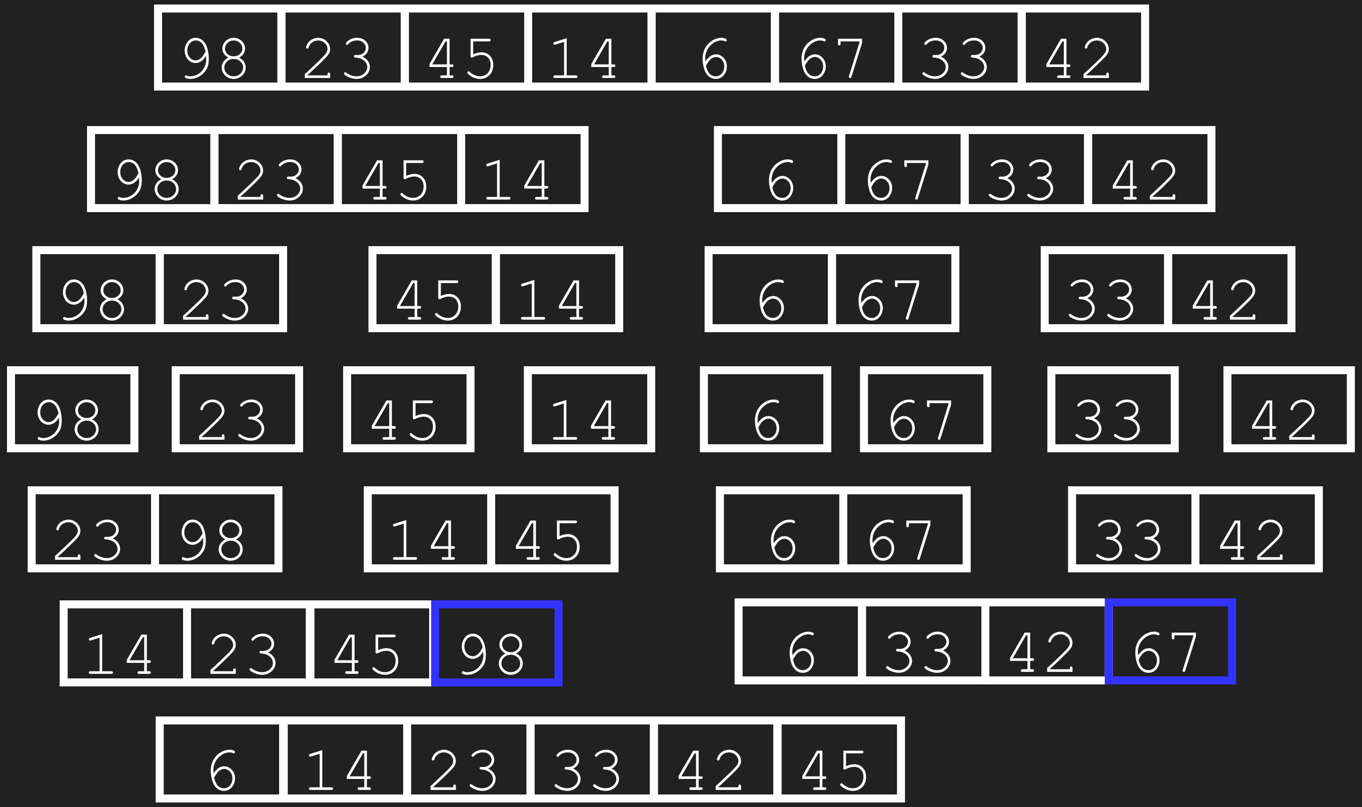
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

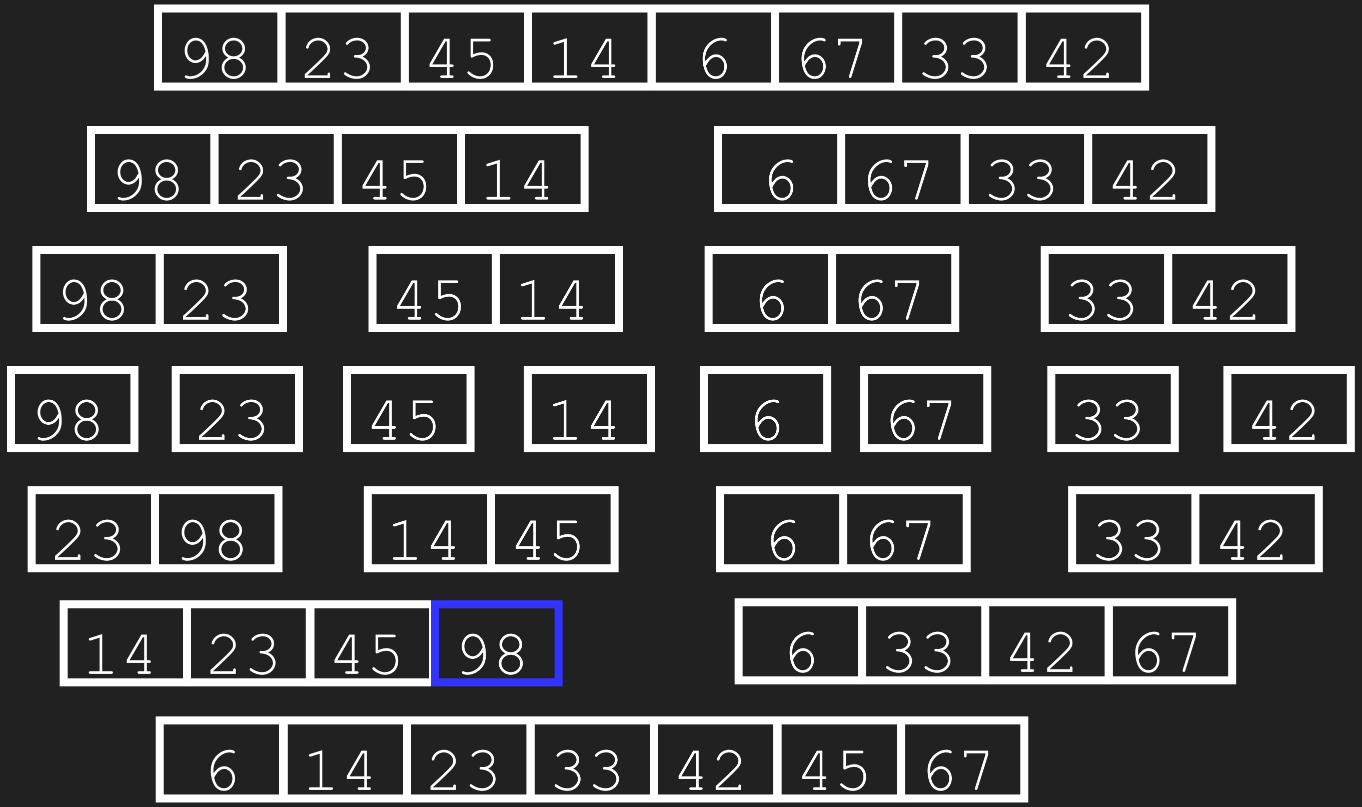
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42 45

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

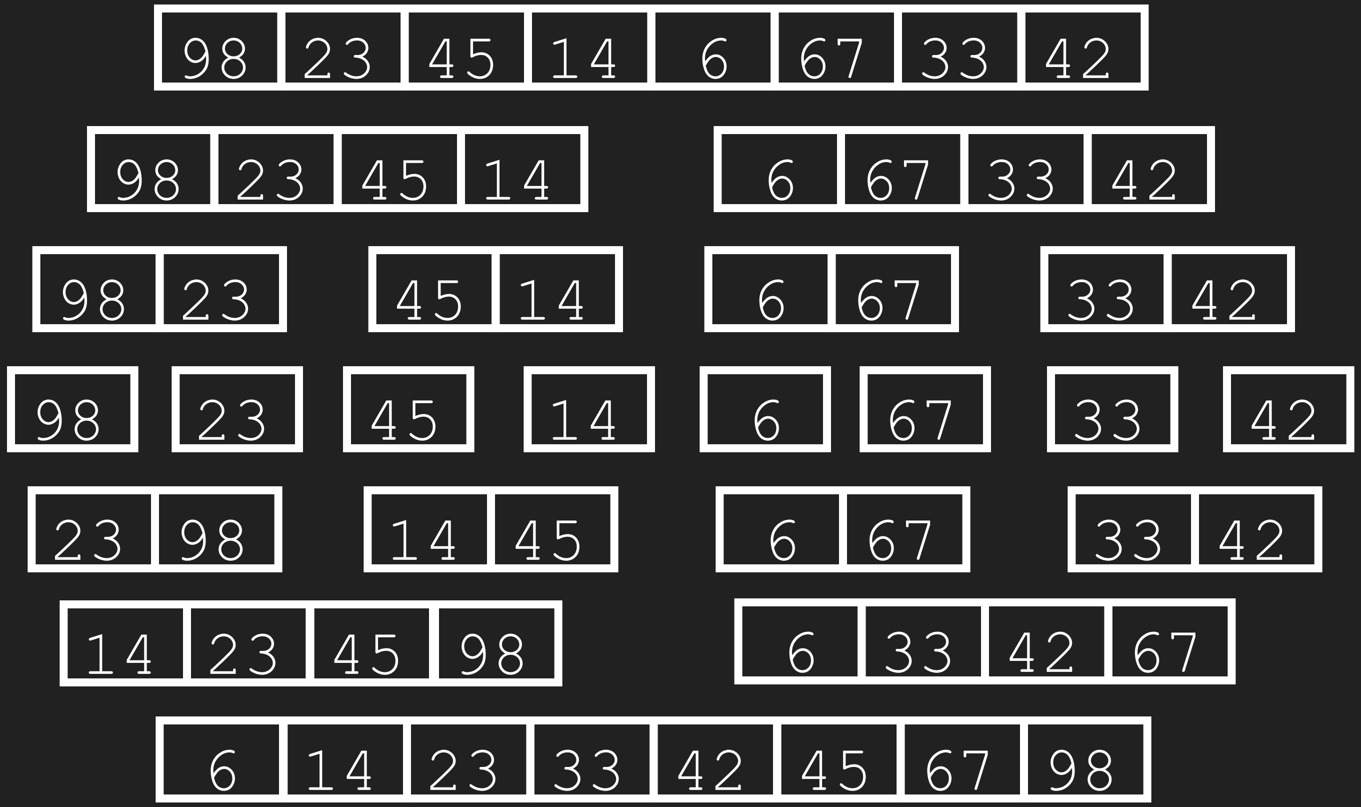
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42 45 67

Merge



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

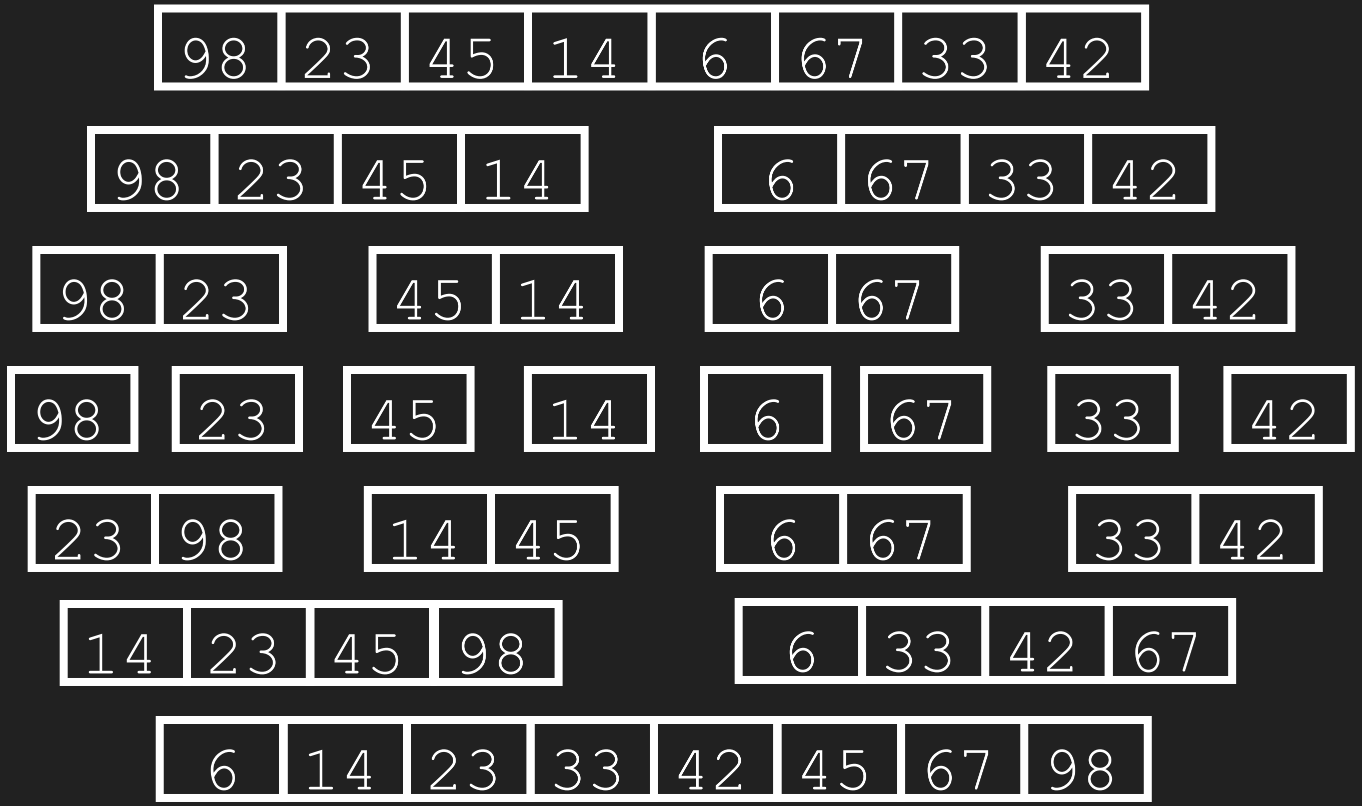
98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42 45 67 98

Merge



98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

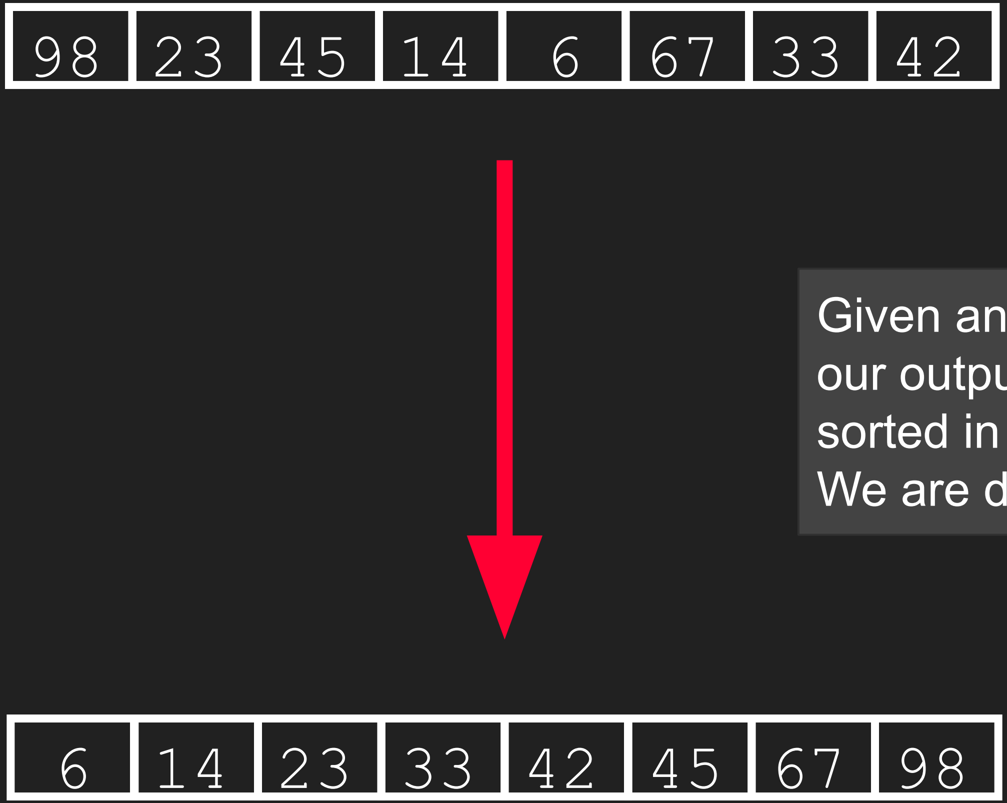
98 23 45 14 6 67 33 42

98 23 45 14 6 67 33 42

23 98 14 45 6 67 33 42

14 23 45 98 6 33 42 67

6 14 23 33 42 45 67 98



98	23	45	14	6	67	33	42

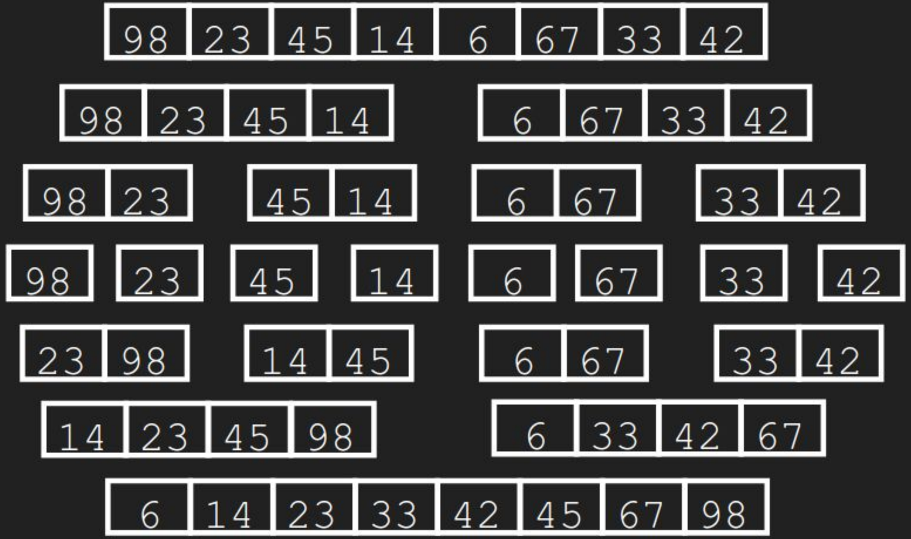
6	14	23	33	42	45	67	98

Given an input of ‘n’ items, our output of ‘n’ items is sorted in ascending order. We are done!

Merge Sort Analysis

Merge Sort Analysis (1/5)

- Do we believe mergesort is really $n*\log_2(n)$ complexity?
- From the pseudo-code, it’s a bit hard to tell
  - Let’s take a look at the complexity class merge sort falls in.



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

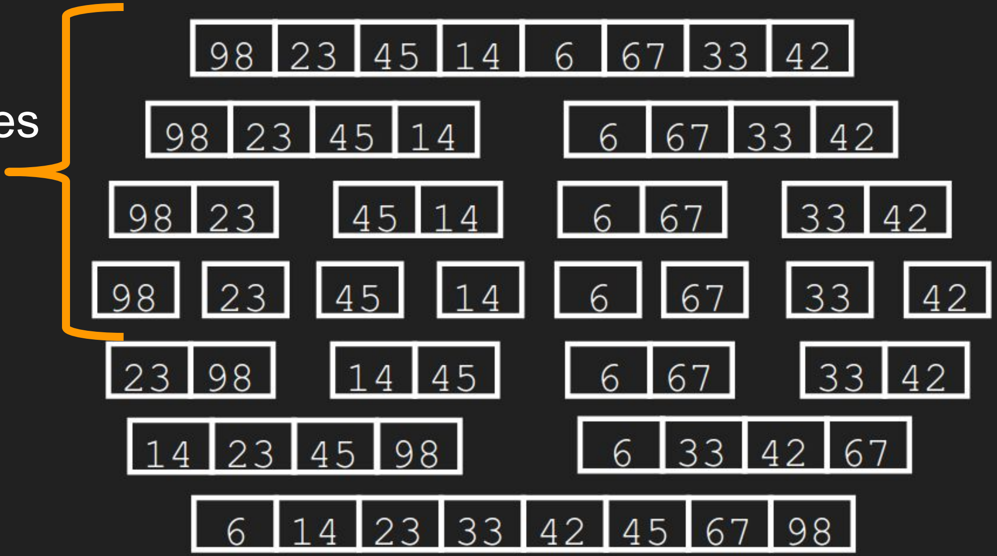
14 23 45 98 | 6 33 42 67

6 14 23 33 42 45 67 98

```
MergeSort(Array, left, right)
  if Array's size > 1
      Divide array Array in halves
      Call MergeSort on first half.
      Call MergeSort on second half.
      Merge two results (combine).
```

Merge Sort Analysis (2/5)

- $\log_2(n)$ levels where we break down our array into very tiny pieces



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

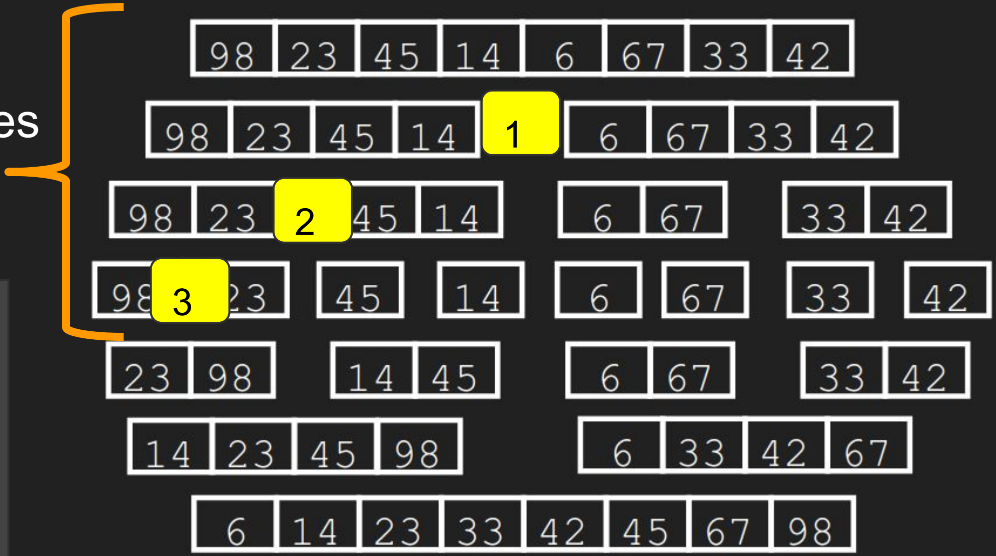
14 23 45 98 | 6 33 42 67

6 14 23 33 42 45 67 98

```
MergeSort(Array, left, right)
  if Array's size > 1
        Divide array Array in halves
        Call MergeSort on first half.
        Call MergeSort on second half.
        Merge two results (combine).
```

Merge Sort Analysis (3/5)

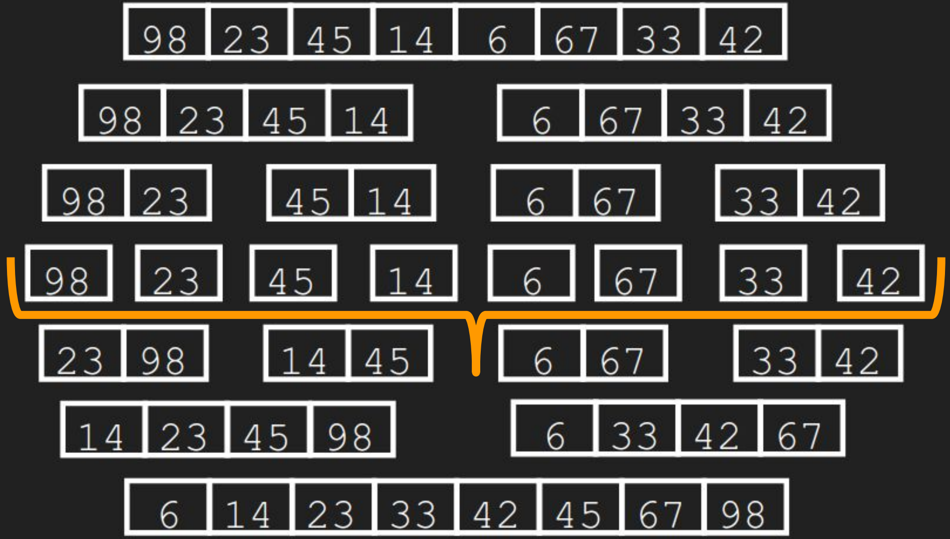
- $\log_2(n)$ levels where we break down our array into very tiny pieces



- $\log_2(8) = 3$
- We divided our array in ‘3’ stages
  - (See the divides in yellow)

Merge Sort Analysis (4/5)

- $\log_2(n)$ levels where we break down our array into very tiny pieces
- Then we make ‘n’ selections combining our array



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

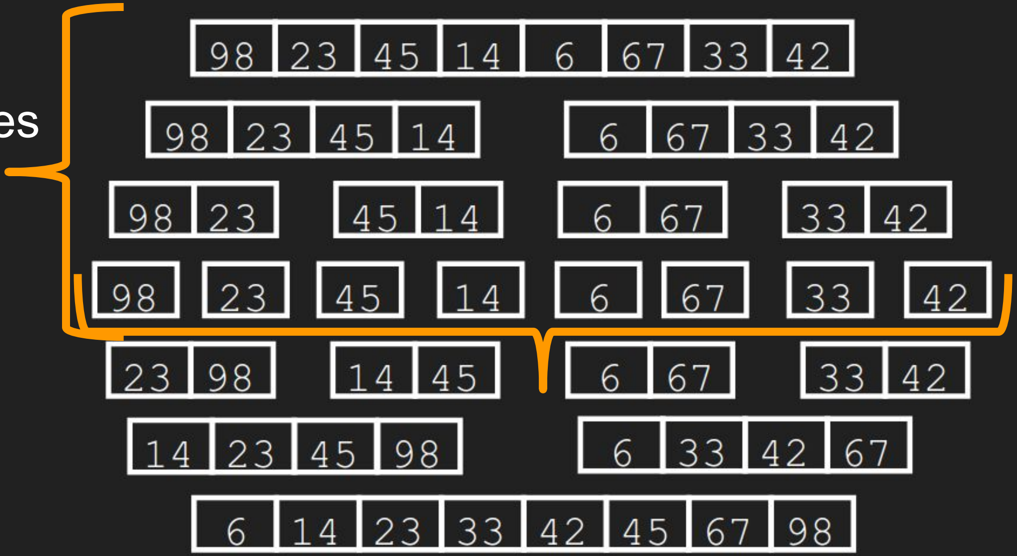
14 23 45 98 | 6 33 42 67

6 14 23 33 42 45 67 98

```
MergeSort(Array, left, right)
  if Array’s size > 1
       Divide array Array in halves
       Call MergeSort on first half.
       Call MergeSort on second half.
       Merge two results (combine).
```

Merge Sort Analysis (5/5)

- $\log_2(n)$ levels where we break down our array into very tiny pieces
- Then we make ‘n’ selections combining our array
- Thus $n*\log_2(n)$ complexity



98 23 45 14 6 67 33 42

98 23 45 14 | 6 67 33 42

98 23 | 45 14 | 6 67 | 33 42

98 | 23 | 45 | 14 | 6 | 67 | 33 | 42

23 98 | 14 45 | 6 67 | 33 42

14 23 45 98 | 6 33 42 67

6 14 23 33 42 45 67 98

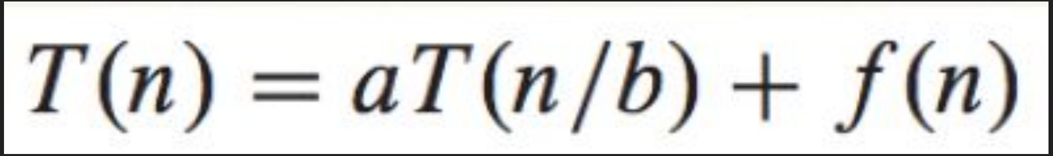
```
MergeSort(Array, left, right)
  if Array's size > 1
      Divide array Array in halves
      Call MergeSort on first half.
      Call MergeSort on second half.
      Merge two results (combine).
```

Merge Sort Recurrence Relation (1/15)

- Mergesort is a ‘recursive’ algorithm
  - (or at least lends itself very nicely to it when implementing)
- If we want to more formally analyze merge sort, we need to look at the recurrence
  - i.e. how much work are we doing at each recursive step
- Understanding the recurrence ‘proves the complexity’ of the algorithm.

Merge Sort Recurrence Relation (2/15)

- The form of the recurrence we are working with is as follows:



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

Merge Sort Recurrence Relation (3/15)

- The form of the recurrence we are working with is as follows:

$$T(n) = \boxed{a}T(n/b) + f(n)$$

‘a’ is the number of ‘subproblems’

a=2 for merge sort, we split into 2 problems (i.e. two arrays)

Merge Sort Recurrence Relation (4/15)

- The form of the recurrence we are working with is as follows:

$$T(n) = aT\boxed{(n/b)} + f(n)$$

‘a’ is the number of ‘subproblems’
a
a=2 for merge sort, we split into 2 problems (i.e. two arrays)

‘n/b’ is the size of each ‘subproblems’

e.g. We end up with an array of n/2 when we divide our array in the first mergesort

Merge Sort Recurrence Relation (4/15)

- The form of the recurrence we are working with is as follows:

$$T(n) = aT(n/b) + \boxed{f(n)}$$

‘a’ is the number of ‘subproblems’
a
a=2 for merge sort, we split into 2 problems (i.e. two arrays)

‘n/b’ is the size of each ‘subproblems’

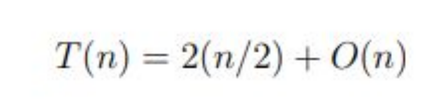
e.g. We end up with an array of n/2 when we divide our array in the first mergesort

f(n) is any work done to ‘divide’ our problem up.

e.g. creating new arrays where we are copying data into

Merge Sort Recurrence Relation (6/15)

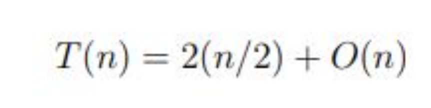
- For merge sort, our recurrence looks like the following:



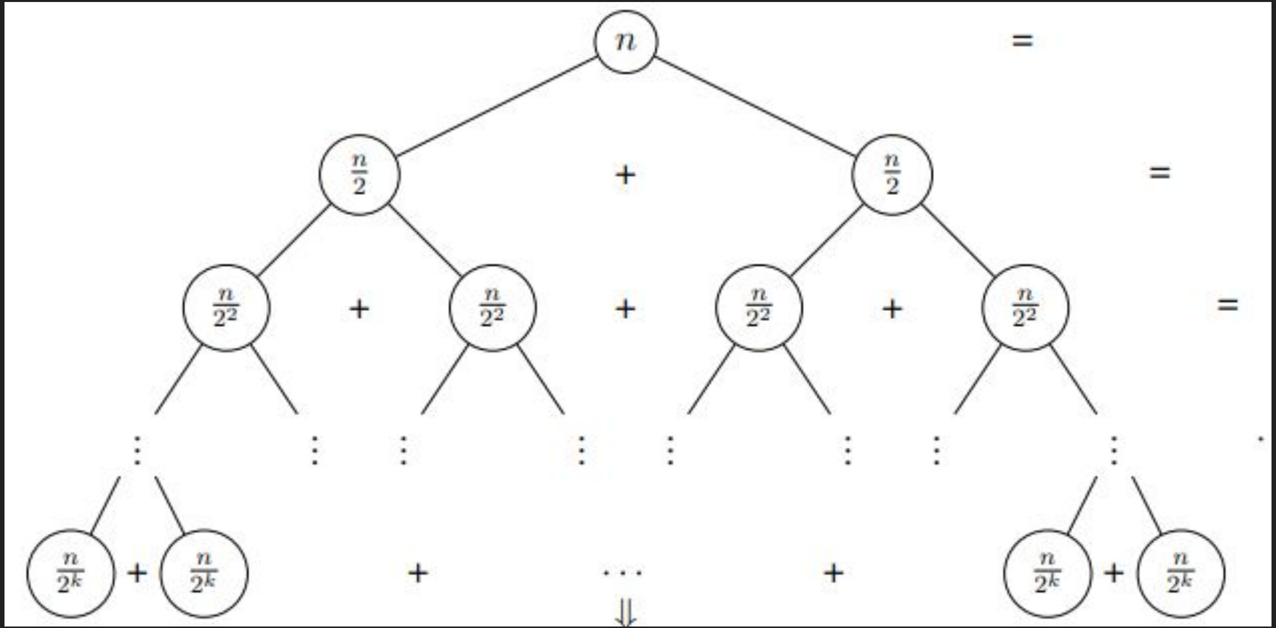
$$T(n) = 2(n/2) + O(n)$$

Merge Sort Recurrence Relation (7/15)

- Each sub-problem can get divided into a ‘recursion tree’ (i.e. tree method)



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

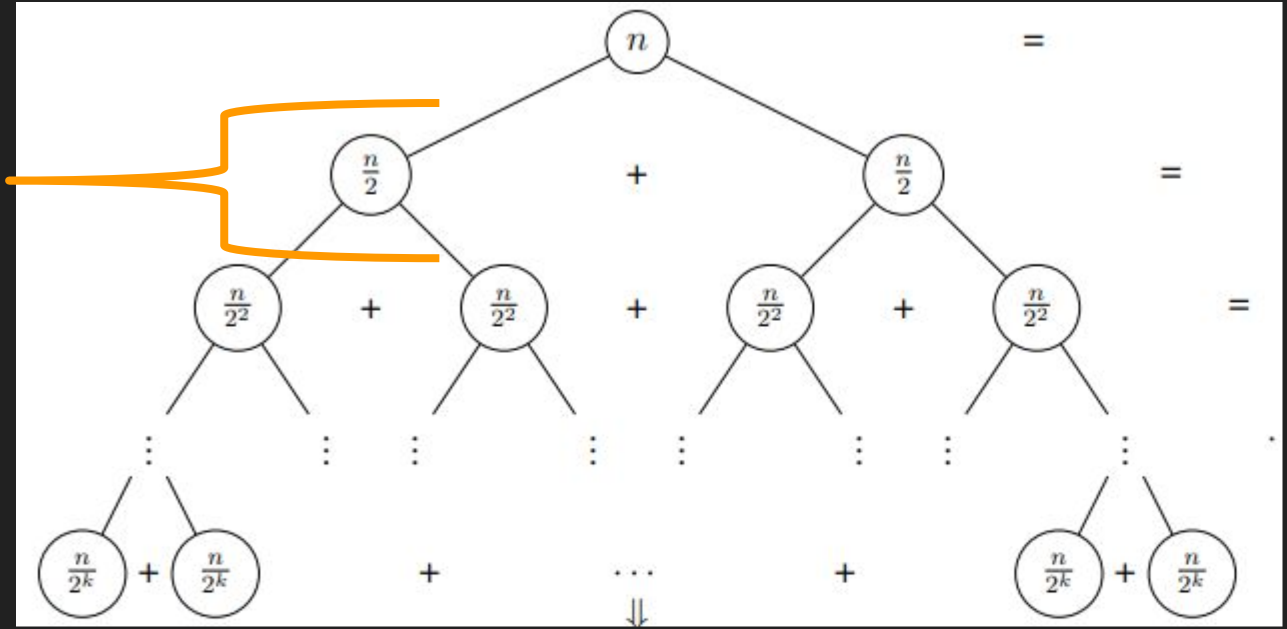

n
=
$\frac{n}{2}$ + $\frac{n}{2}$
=
$\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$
=
$\frac{n}{2^k}$ + $\frac{n}{2^k}$ + $\cdots$ + $\frac{n}{2^k}$ + $\frac{n}{2^k}$
⇓

Merge Sort Recurrence Relation (8/15)

- Each sub-problem can get divided into a ‘recursion tree’

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

Again, observe each level we are dividing our initial array.


n
$=$
$\frac{n}{2}$ + $\frac{n}{2}$
$=$
$\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$
$=$
$\frac{n}{2^k}$ + $\frac{n}{2^k}$ + + $\cdots$ + $\frac{n}{2^k}$ + $\frac{n}{2^k}$
$\Downarrow$

Merge Sort Recurrence Relation (9/15)

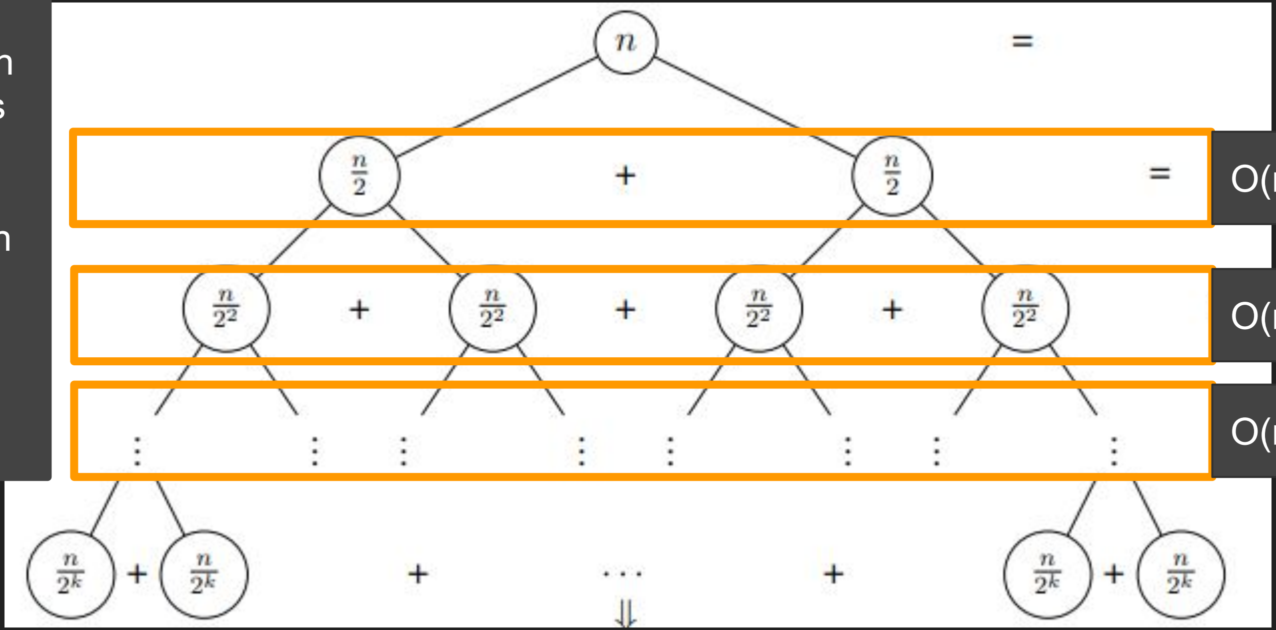
- Each sub-problem can get divided into a ‘recursion tree’

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

Note that at each level, the work is O(n)

e.g. n/2 + n/2 = n

Thus ‘n’ work $\log_2(n)$ times



O(n) Work

O(n) Work

O(n) etc.

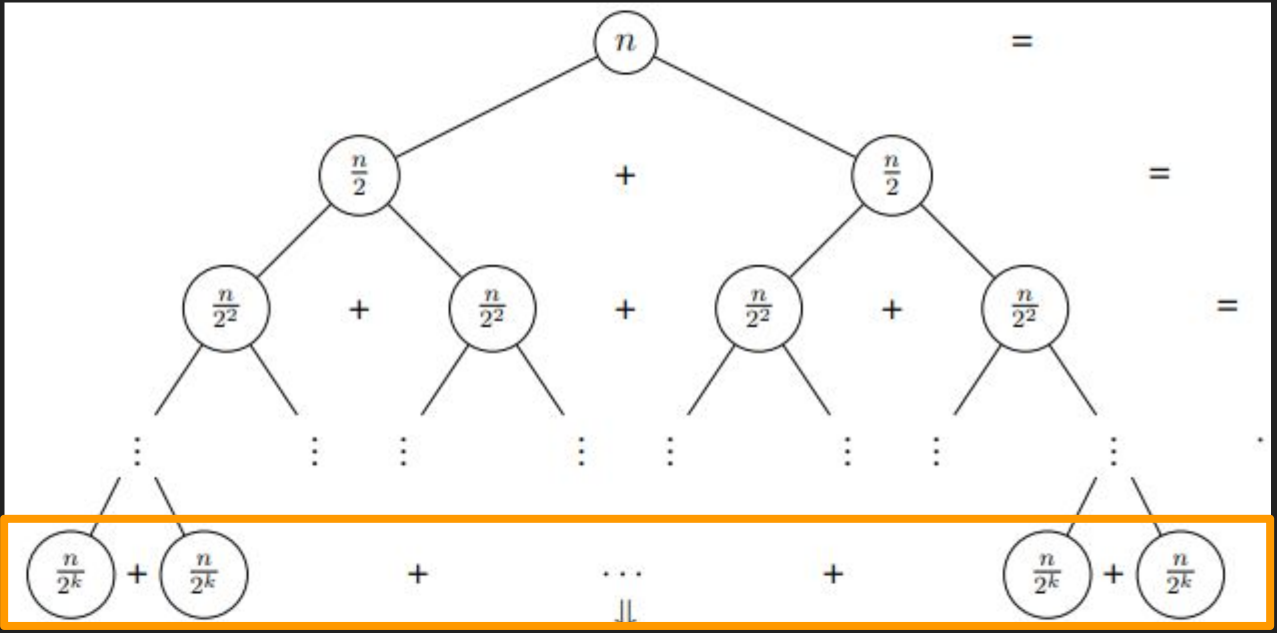
Merge Sort Recurrence Relation (10/15)

- Each sub-problem can get divided into a ‘recursion tree’

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

At the bottom level we have a summation of problems each n/2^k the original size (i.e. ‘1’ in mergesort)

$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right)$$


n
$=$
$\frac{n}{2}$ + $\frac{n}{2}$
$=$
$\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$
$=$
$\frac{n}{2^k}$ + $\frac{n}{2^k}$ + … + $\frac{n}{2^k}$ + $\frac{n}{2^k}$
$\Downarrow$

Merge Sort Recurrence Relation (11/15)

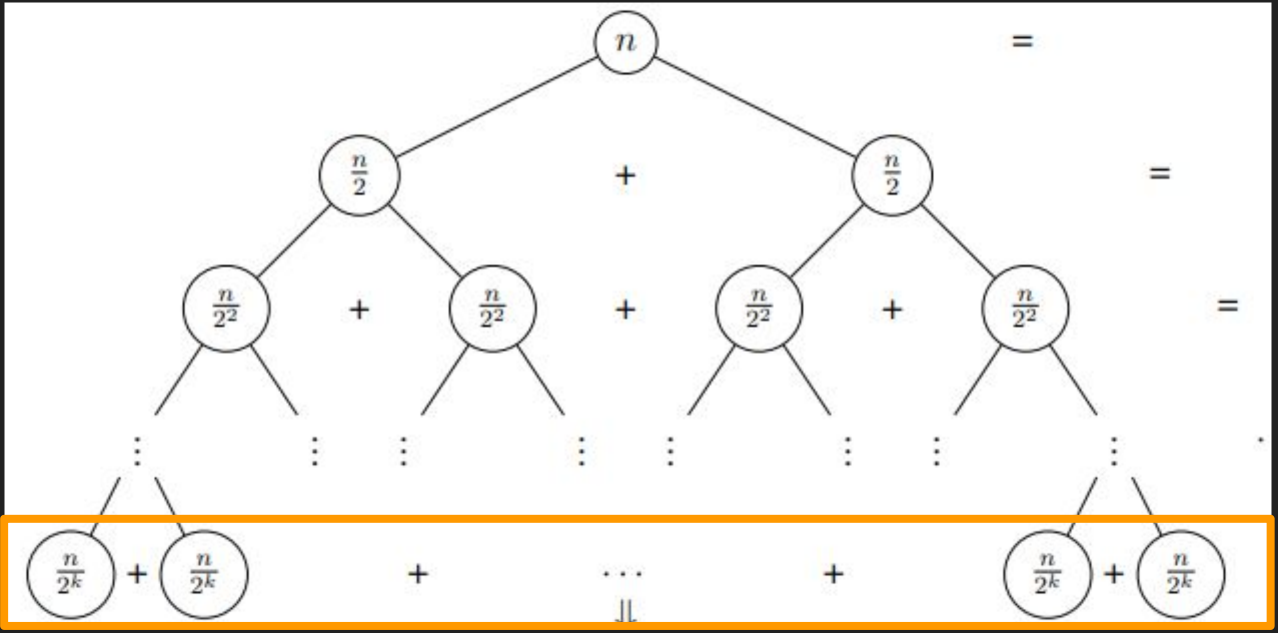
- Each sub-problem can get divided into a ‘recursion tree’

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

And in total, we have n/2^i problems, that are each n/2^k in size

k=number of levels in our recursion tree

$$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right)$$



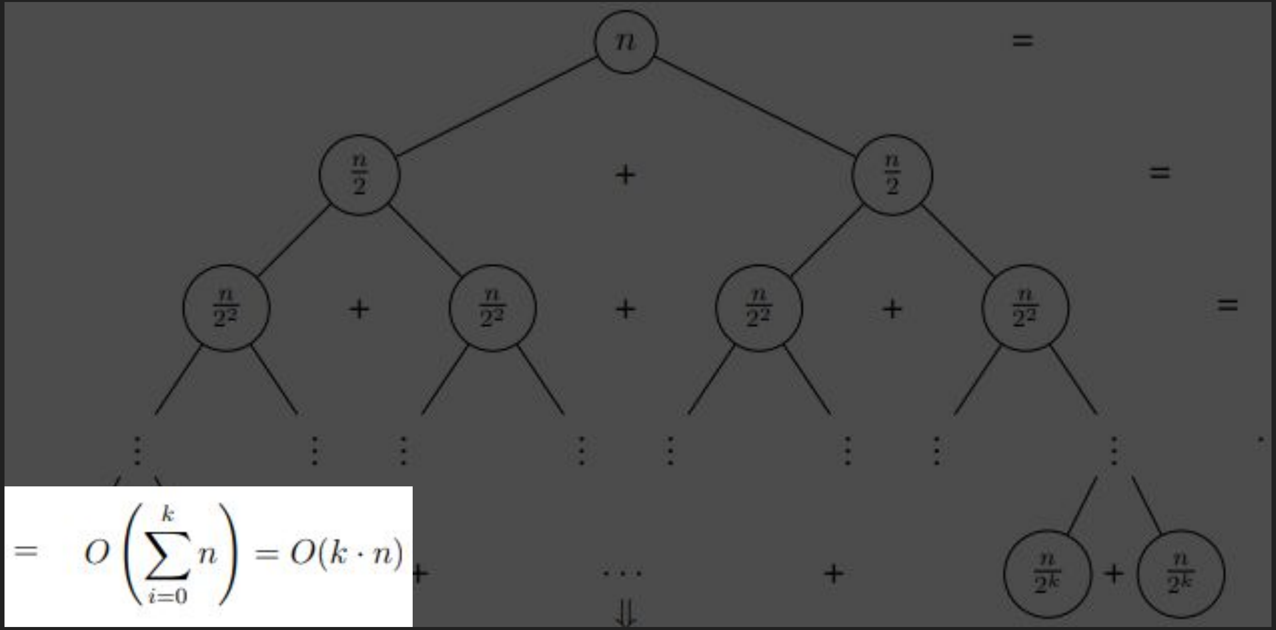
n
=
$\frac{n}{2}$ + $\frac{n}{2}$
=
$\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$
=
$\frac{n}{2^k}$ + $\frac{n}{2^k}$ + … + $\frac{n}{2^k}$ + $\frac{n}{2^k}$
⇓

Merge Sort Recurrence Relation (12/15)

- Each sub-problem can get divided into a ‘recursion tree’

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

If I cross out the common factors, I get ‘n’

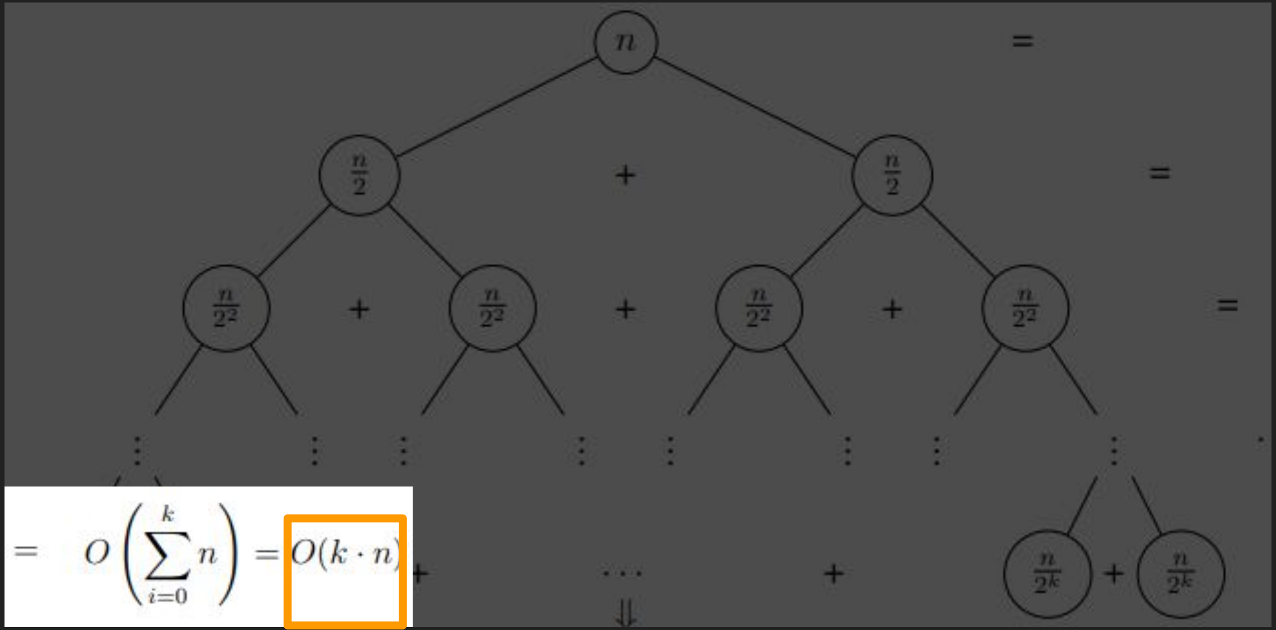


$$O\left(\sum_{i=0}^{k} \cancel{2^i} \cdot \frac{n}{\cancel{2^i}}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n)$$

Merge Sort Recurrence Relation (13/15)

- Each sub-problem can get divided into a ‘recursion tree’

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



n
$=$
$\frac{n}{2}$
$+$
$\frac{n}{2}$
$=$
$\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$
$=$
$+$ $\cdots$ $+$ $\frac{n}{2^k}$ + $\frac{n}{2^k}$
$\Downarrow$

$$= O\left(\sum_{i=0}^{k} n\right) = \boxed{O(k \cdot n)}$$

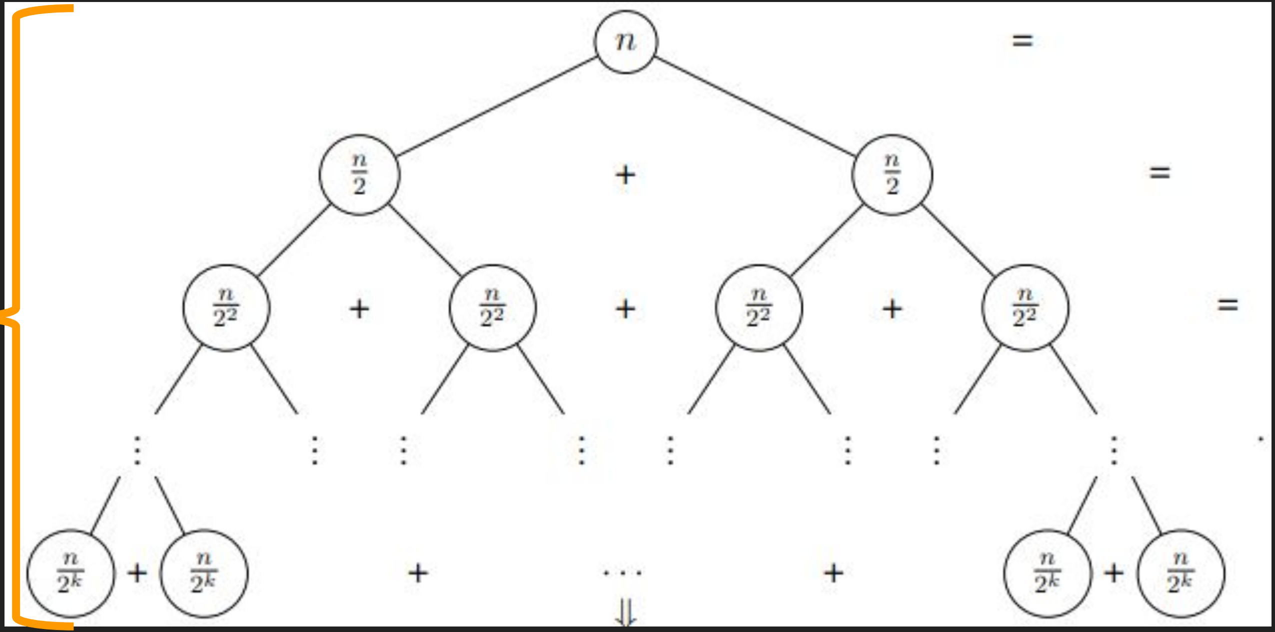
So now I am left with ‘n’ at the bottom like we previously saw, times some ‘constant factor’

Merge Sort Recurrence Relation (14/15)

- Each sub-problem can get divided into a ‘recursion tree’

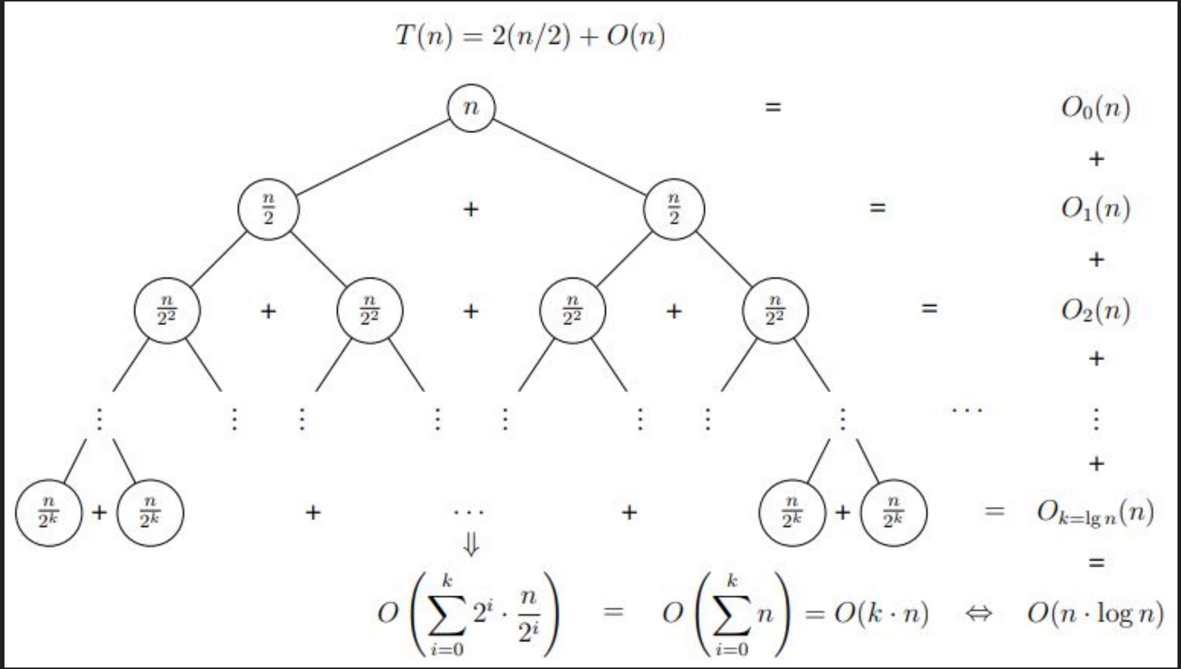
$$T(n) = 2(n/2) + O(n)$$

And again, ‘$\log_2(n)$’ levels

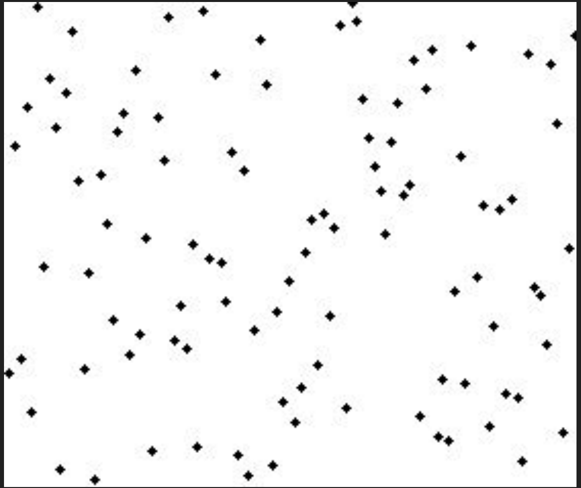


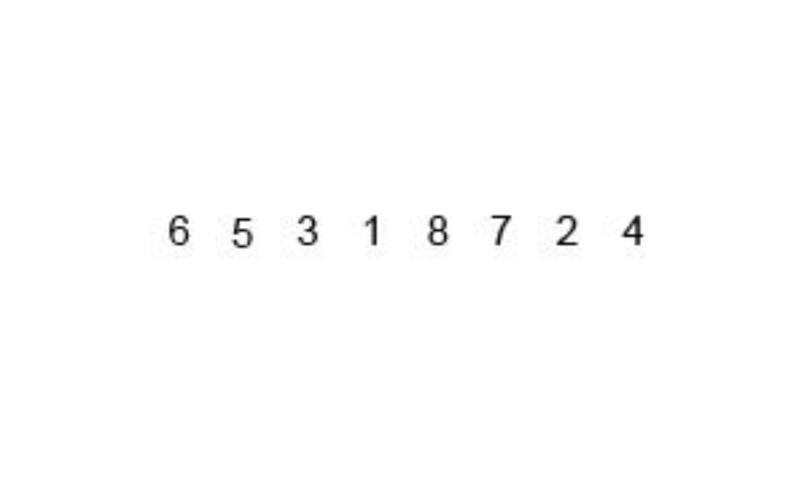
Merge Sort Recurrence Relation (15/15)

- The full picture


$T(n) = 2(n/2) + O(n)$
n = $O_0(n)$
+
$\frac{n}{2}$ + $\frac{n}{2}$ = $O_1(n)$
+
$\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$ + $\frac{n}{2^2}$ = $O_2(n)$
+
$\vdots$ $\cdots$ $\vdots$
+
$\frac{n}{2^k}$ + $\frac{n}{2^k}$ + $\cdots$ + $\frac{n}{2^k}$ + $\frac{n}{2^k}$ = $O_{k=\lg n}(n)$
$\Downarrow$ =
$O\left(\sum_{i=0}^{k} 2^i \cdot \frac{n}{2^i}\right) = O\left(\sum_{i=0}^{k} n\right) = O(k \cdot n) \Leftrightarrow O(n \cdot \log n)$

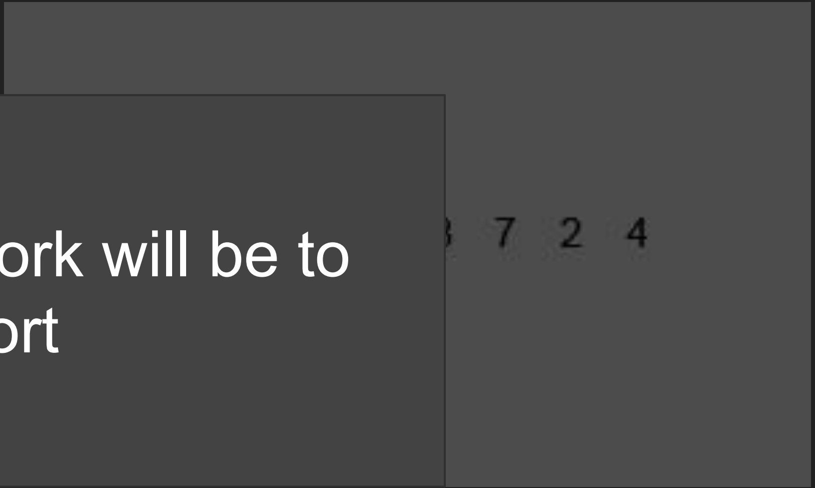
Merge Sort Visual - Second Look (1/2)





- (left) Each ‘dot’ is an element in a list that you can see being slowly divided and combined
- (right) A more concrete example

Merge Sort Visual - Second Look (2/2)

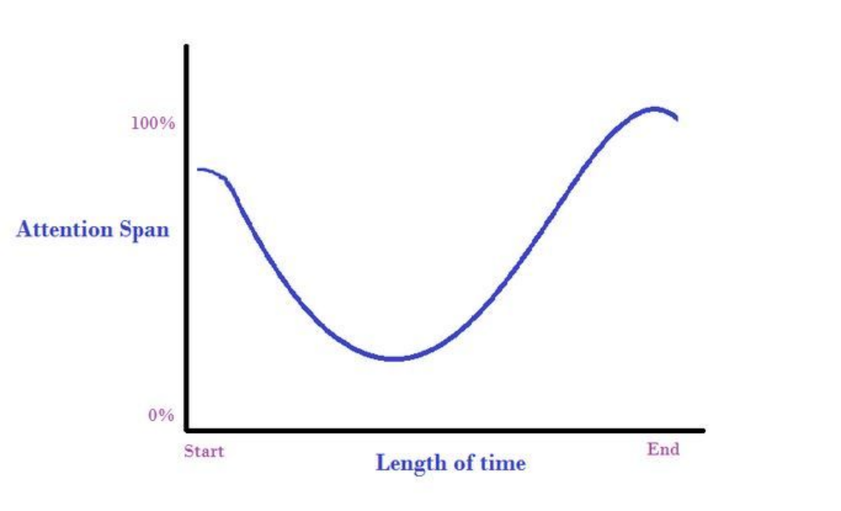


Part of your homework will be to implement merge sort

- (left) Each ‘dot’ is an element in a list that you can see being slowly divided and combined
- (right) A more concrete example

Short 5 minute break

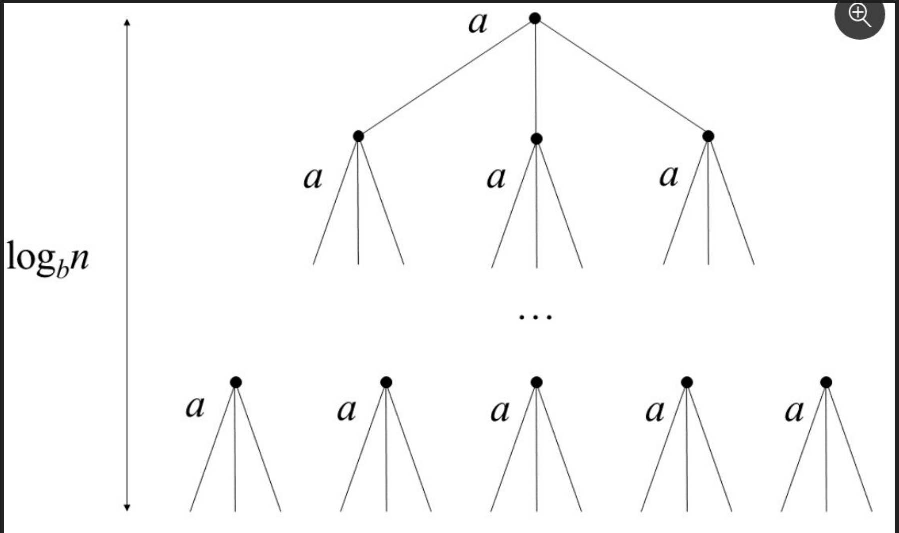
- 3 hours is a long time.
- I will try to never lecture for more than half of that time without some sort of ‘break’ or transition to an in-class activity/lab.
- Use this time to stretch, check your phones, eat/drink something, etc.



Master Theorem

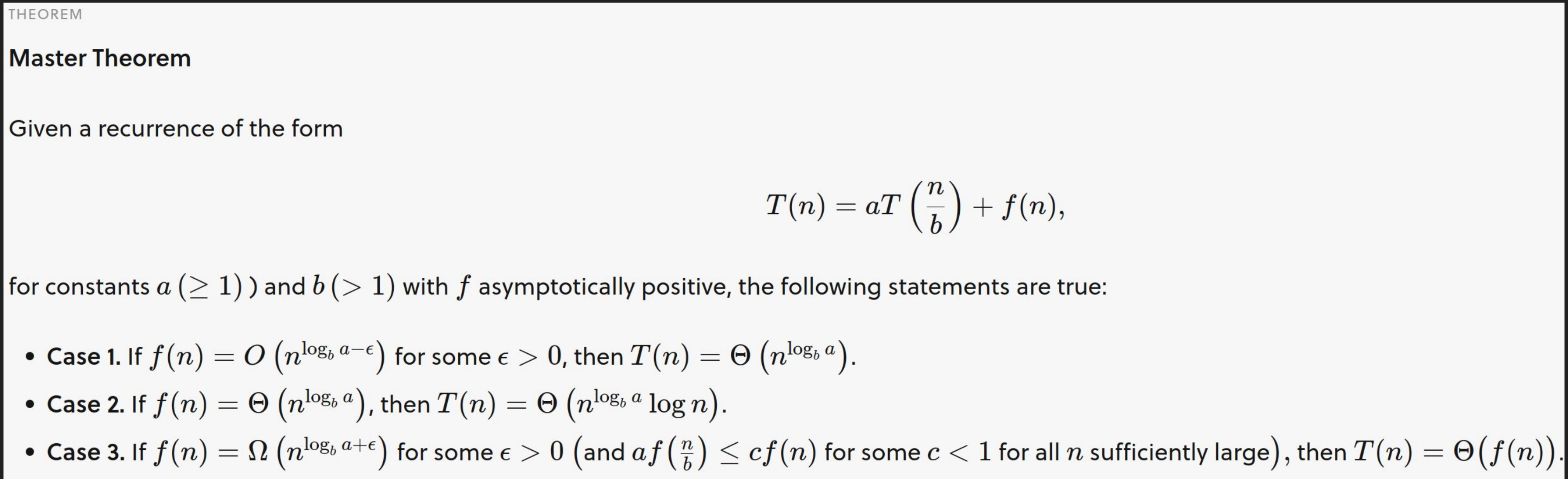
Master Theorem Explained

- Around 1980 several computer scientists discovered that divide and conquer algorithms with particular recurrences followed a pattern.
- Based off of the recurrence, we could figure out the complexity of the algorithm.
- https://en.wikipedia.org/wiki/Master_theorem_(analysis_of_algorithms)



https://brilliant.org/wiki/master-theorem/

Master Theorem Common Cases



THEOREM

Master Theorem

Given a recurrence of the form

$$T(n) = aT\left(\frac{n}{b}\right) + f(n),$$

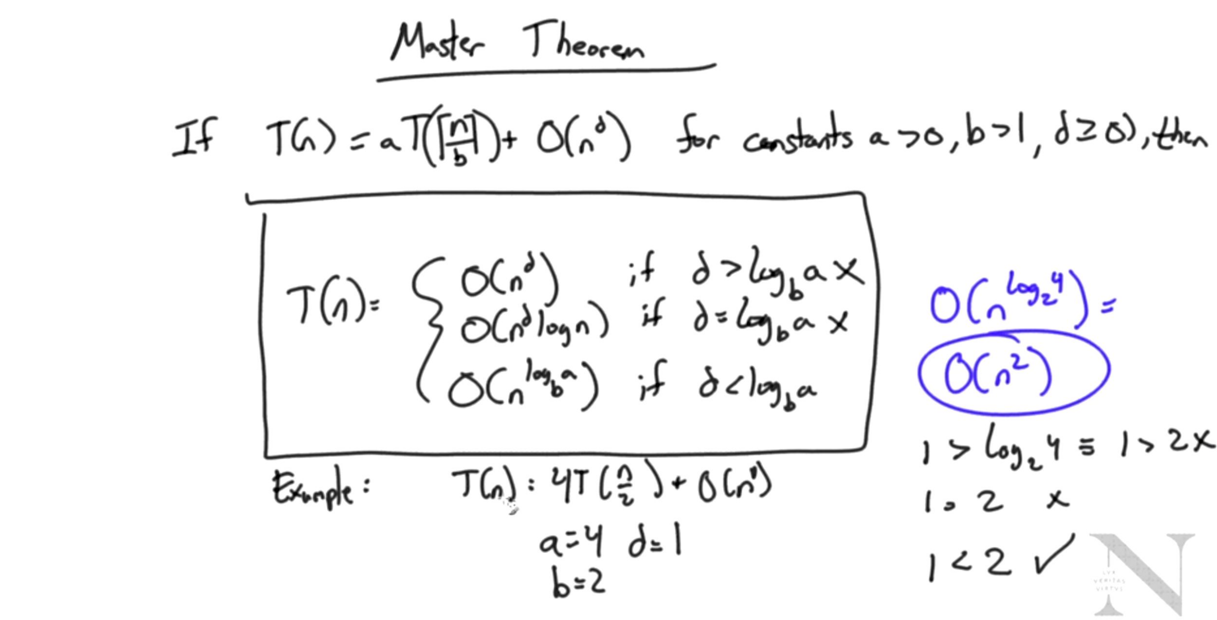
for constants $a\,(\geq 1)$) and $b\,(> 1)$ with f asymptotically positive, the following statements are true:

- **Case 1.** If $f(n) = O\left(n^{\log_b a - \epsilon}\right)$ for some $\epsilon > 0$, then $T(n) = \Theta\left(n^{\log_b a}\right)$.
- **Case 2.** If $f(n) = \Theta\left(n^{\log_b a}\right)$, then $T(n) = \Theta\left(n^{\log_b a} \log n\right)$.
- **Case 3.** If $f(n) = \Omega\left(n^{\log_b a + \epsilon}\right)$ for some $\epsilon > 0$ $\big($and $af\left(\frac{n}{b}\right) \leq cf(n)$ for some $c < 1$ for all n sufficiently large$\big)$, then $T(n) = \Theta\big(f(n)\big)$.

https://brilliant.org/wiki/master-theorem/

Master Theorem

- Live example in class



Master Theorem

If $T(n) = aT(\lceil \frac{n}{b} \rceil) + O(n^d)$ for constants $a > 0, b > 1, d \geq 0$), then

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

Example: $T(n) : 4T(\frac{n}{2}) + O(n^1)$

$a = 4 \quad d = 1$

$b = 2$

$O(n^{\log_2 4}) = \boxed{O(n^2)}$

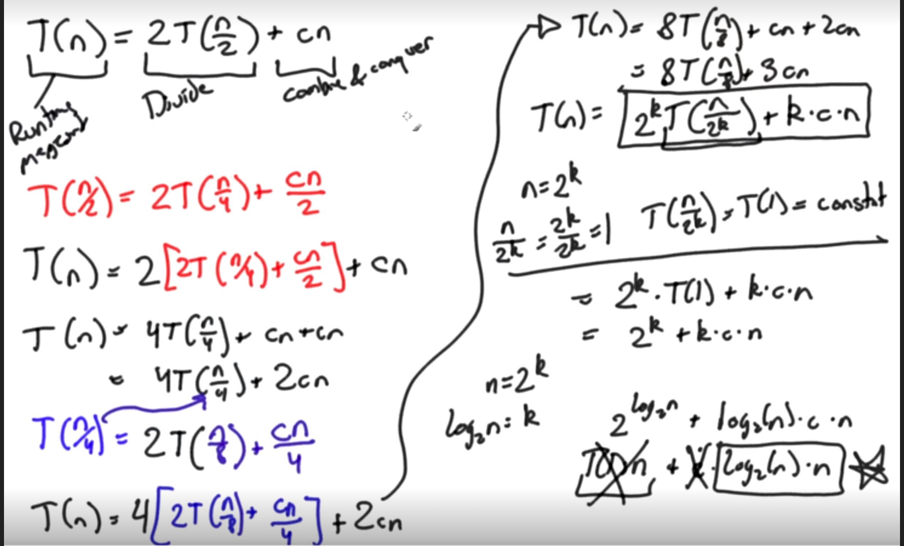
$1 > \log_2 4 = 1 > 2 \times$

$1 = 2 \quad \times$

$1 < 2$ ✓

Substitution Method

- When the master theorem does not work for a particular recurrence (i.e. it does not match one of the cases), then we resort to the substitution method.
- Idea:
  - Keep solving recurrence by hand until you find a pattern.
  - See Module 10 Lesson 6 for full example



$$\underbrace{T(n)}_{\text{Running mergsort}} = \underbrace{2T\left(\frac{n}{2}\right)}_{\text{Divide}} + \underbrace{cn}_{\text{combine \& conquer}}$$

$$T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \frac{cn}{2}$$

$$T(n) = 2\left[2T\left(\frac{n}{4}\right) + \frac{cn}{2}\right] + cn$$

$$T(n) = 4T\left(\frac{n}{4}\right) + cn + cn$$

$$= 4T\left(\frac{n}{4}\right) + 2cn$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \frac{cn}{4}$$

$$T(n) = 4\left[2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right] + 2cn$$

$$T(n) = 8T\left(\frac{n}{8}\right) + cn + 2cn$$

$$= 8T\left(\frac{n}{8}\right) + 3cn$$

$$T(n) = \boxed{2^k \underbrace{T\left(\frac{n}{2^k}\right)} + k \cdot c \cdot n}$$

$n = 2^k$

$\frac{n}{2^k} = \frac{2^k}{2^k} = 1 \quad T\left(\frac{n}{2^k}\right) = T(1) = \text{constnt}$

$$= 2^k \cdot T(1) + k \cdot c \cdot n$$

$$= 2^k + k \cdot c \cdot n$$

$n = 2^k$

$\log_2 n = k$

$$2^{\log_2 n} + \log_2(n) \cdot c \cdot n$$

$$\underbrace{\cancel{T(1)n}} + \cancel{c} \cdot \boxed{\log_2(n) \cdot n} ☆$$

Randomized Algorithms

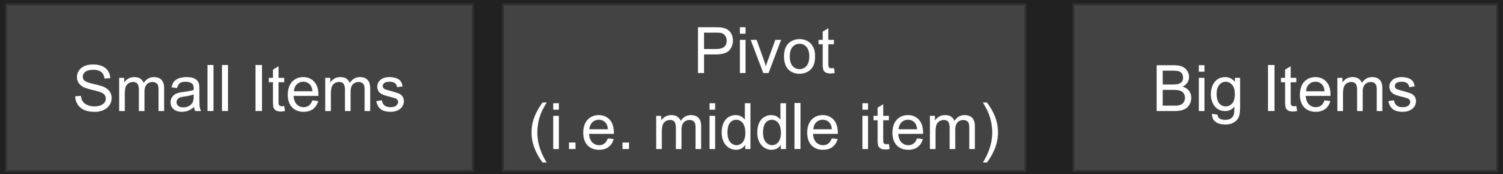
quicksort

Google is impressed

- The engineers at Google are impressed with your understanding different sorting algorithms
- And now they are going to recruit you to help solve yet another problem
  - They need to sort their data even more *quickly*

Quick Sort Overview (1/2) [reference]

- Quicksort is a divide and conquer algorithm.
- What is different is it uses a ‘pivot’ to half elements.
  - Smaller elements go on one side of the pivot, and larger elements go on the remaining side.



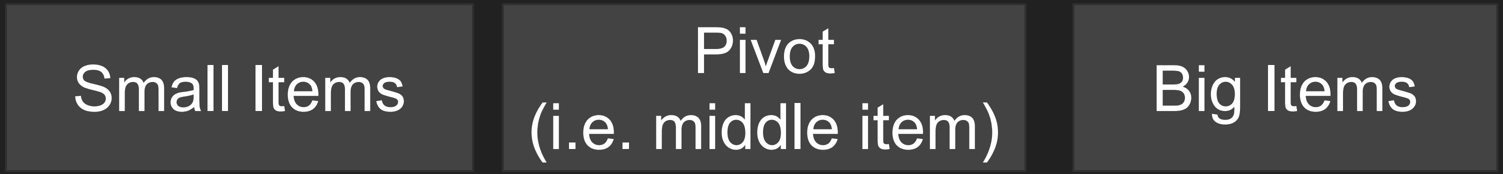
Small Items

Pivot
(i.e. middle item)

Big Items

Quick Sort Overview (2/2) [reference]

- Overall quicksort is another divide-and-conquer algorithm
  - again, similar to merge sort
- The divide step is done by ‘partitioning’ our data into two arrays.
  - One array with items smaller than a ‘pivot’ point
  - One array with items greater than a ‘pivot’ point
- The conquer step is sorting recursively smaller arrays
- The combine step is-...actually there is no combine step, the sort happens in place.



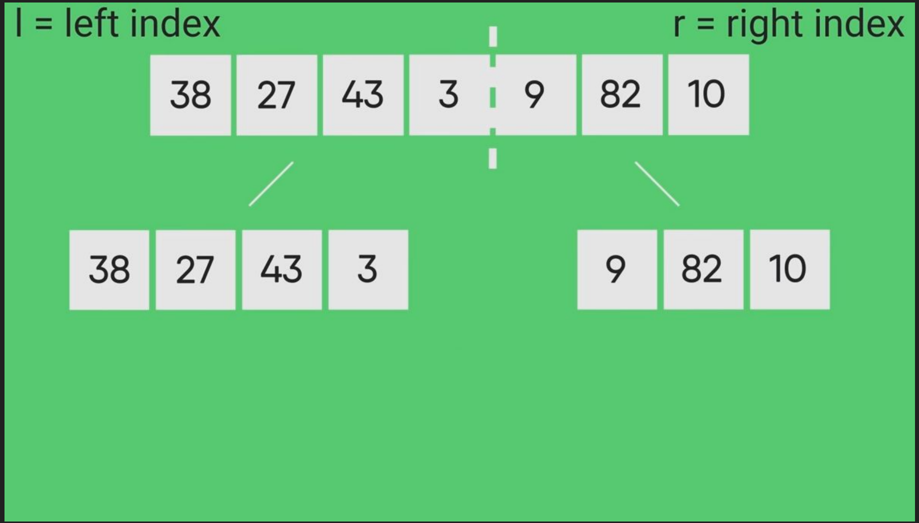
Small Items

Pivot
(i.e. middle item)

Big Items

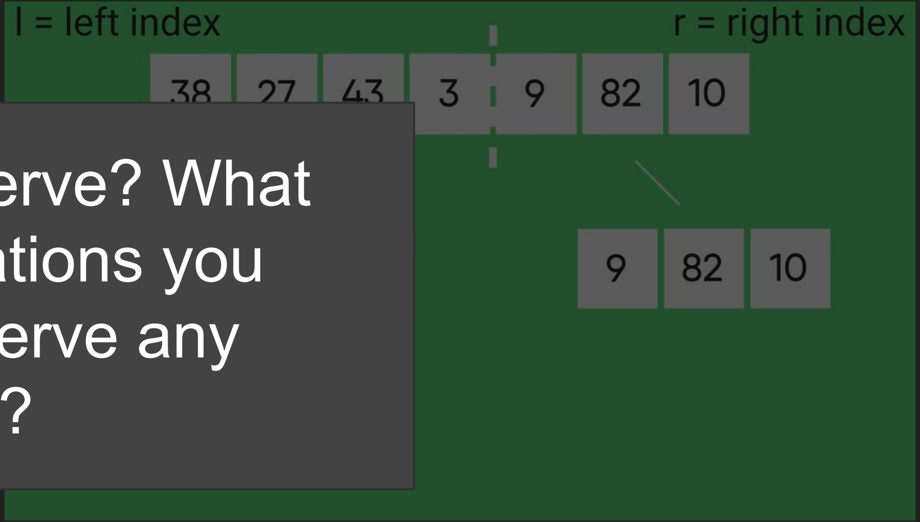
Quick Sort Visual (1/2)

- https://www.youtube.com/watch?v=PgBzjlCcFvc (3:04 min)



Quick Sort Visual (2/2)

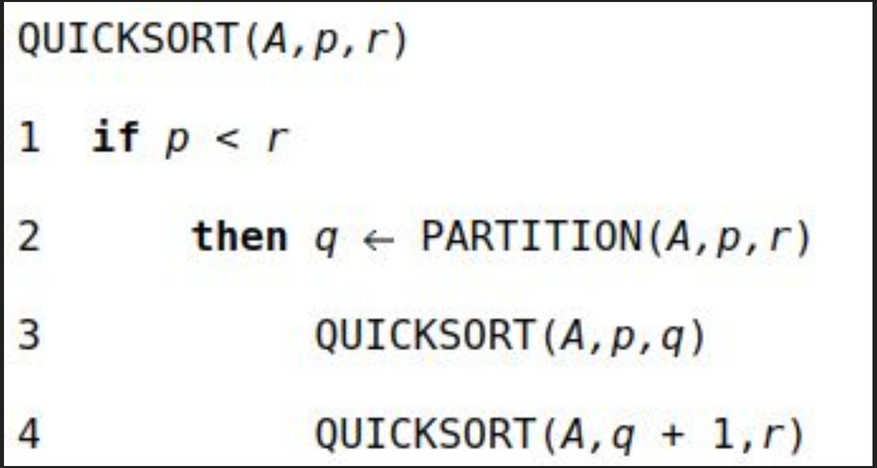
- https://www.youtube.com/watch?v=PgBzjlCcFv (2:04 min)



What did you observe? What was the key operations you saw? Did you observe any loops or recursion?

quick sort - Pseudocode

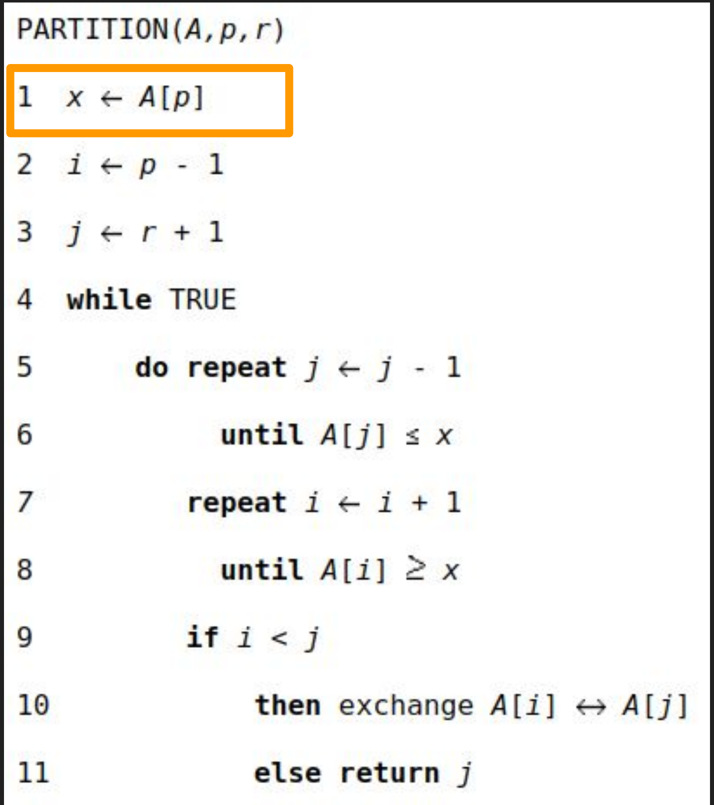
- Quick sort can be done with three function calls
- The partition returns the index in our array for which we ‘pivot’ around.
- Then we recursively call quicksort on the left and right sides



```
QUICKSORT($A,p,r$)
if $p < r$
    then $q \leftarrow$ PARTITION($A,p,r$)
        QUICKSORT($A,p,q$)
        QUICKSORT($A,q + 1,r$)
```

Partition Pseudocode (1/2)

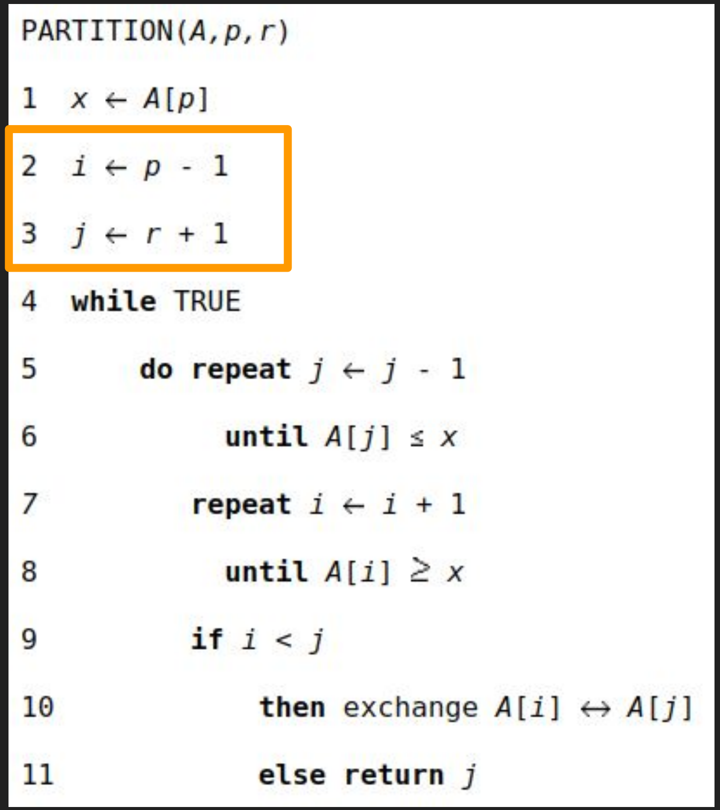
- The partition portion is probably the more interesting part
- We initially have our pivot point at line 1



```
PARTITION($A,p,r$)
1  $x \leftarrow A[p]$
2  $i \leftarrow p - 1$
3  $j \leftarrow r + 1$
4  while TRUE
5      do repeat $j \leftarrow j - 1$
6              until $A[j] \leq x$
7          repeat $i \leftarrow i + 1$
8              until $A[i] \geq x$
9          if $i < j$
10             then exchange $A[i] \leftrightarrow A[j]$
11             else return $j$
```

Partition Pseudocode (2/2)

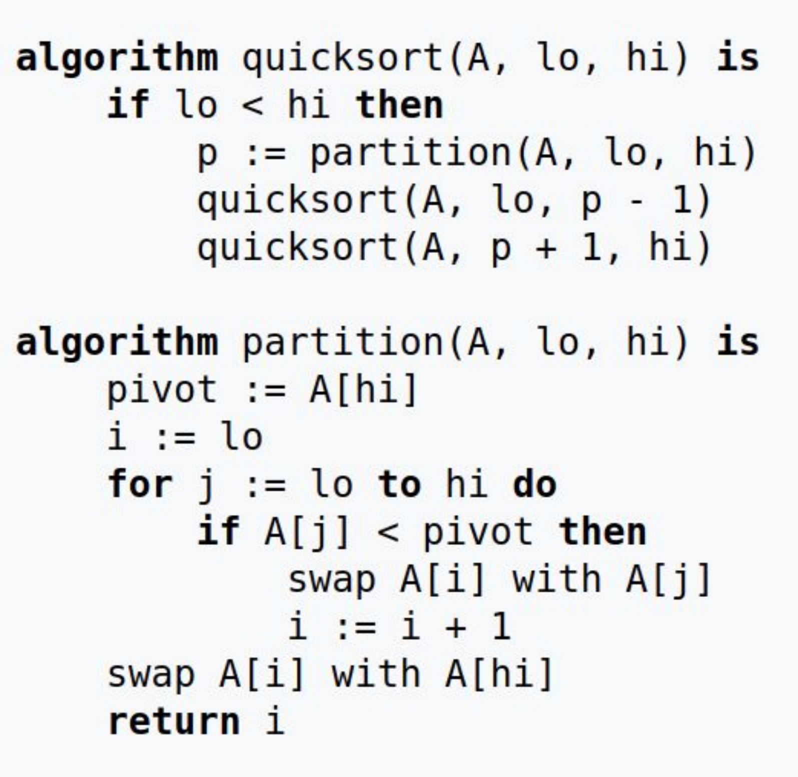
- The partition portion is probably the more interesting part
- We initially have our pivot point at line 1
- Then we have two ‘counters’ (i and j) to walk through the array
- We use these to swap items to the left or right of our pivot



```
PARTITION($A$,$p$,$r$)
1  $x \leftarrow A[p]$
2  $i \leftarrow p - 1$
3  $j \leftarrow r + 1$
4  while TRUE
5        do repeat $j \leftarrow j - 1$
6              until $A[j] \leq x$
7           repeat $i \leftarrow i + 1$
8              until $A[i] \geq x$
9           if $i < j$
10             then exchange $A[i] \leftrightarrow A[j]$
11             else return $j$
```

quick sort - Pseudocode

- Another example
- https://en.wikipedia.org/wiki/Quicksort



```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

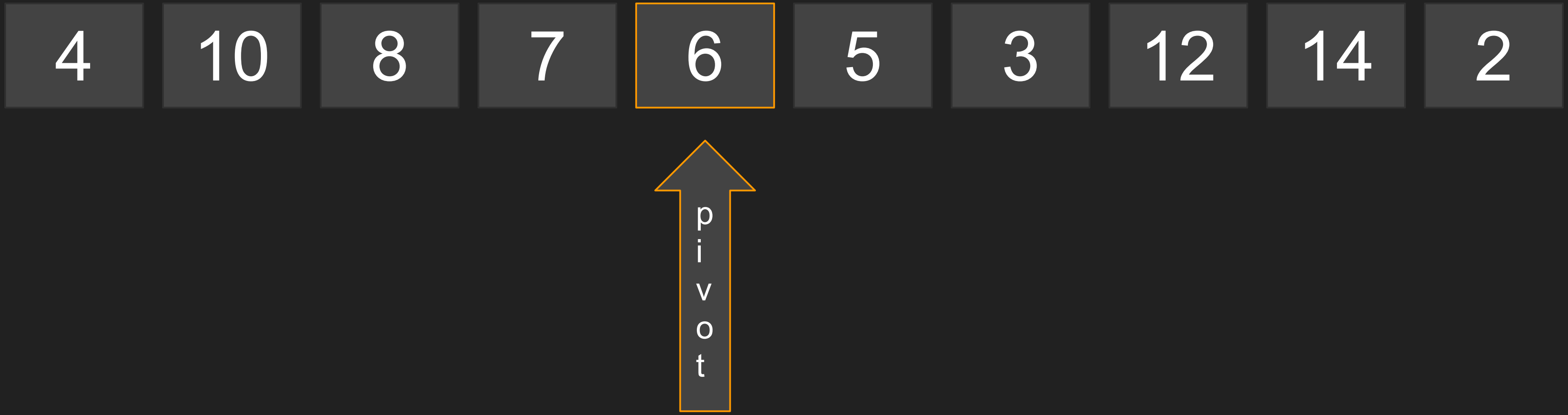
Quicksort example 1

Quicksort example

4	10	8	7	6	5	3	12	14	2

Narration: Here we are given an unsorted list

Quicksort example

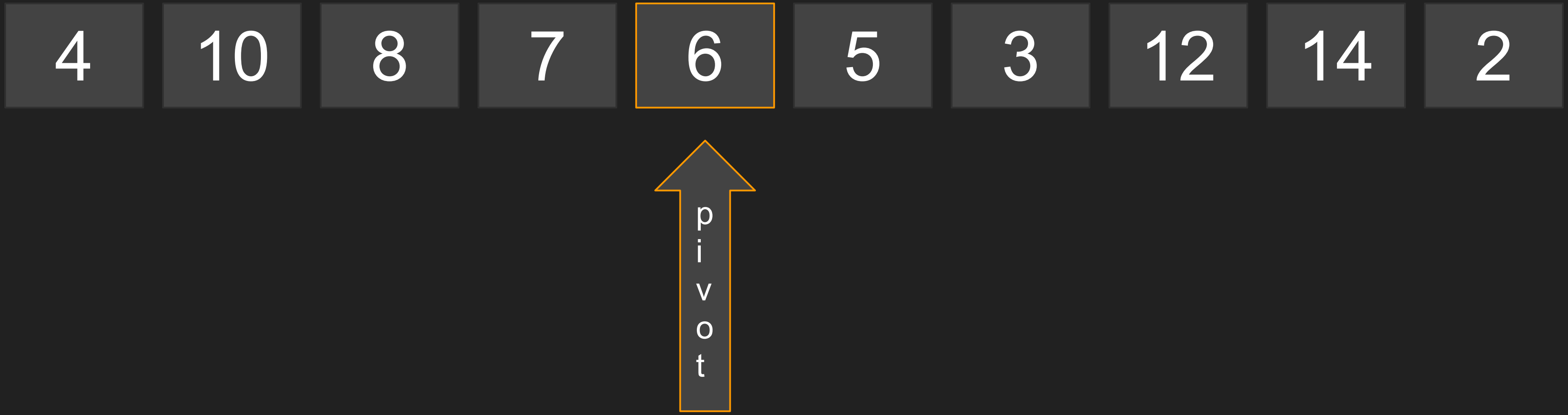


4	10	8	7	6	5	3	12	14	2

pivot

Narration: We need to select some value to ‘pivot’ around. How about our midpoint?

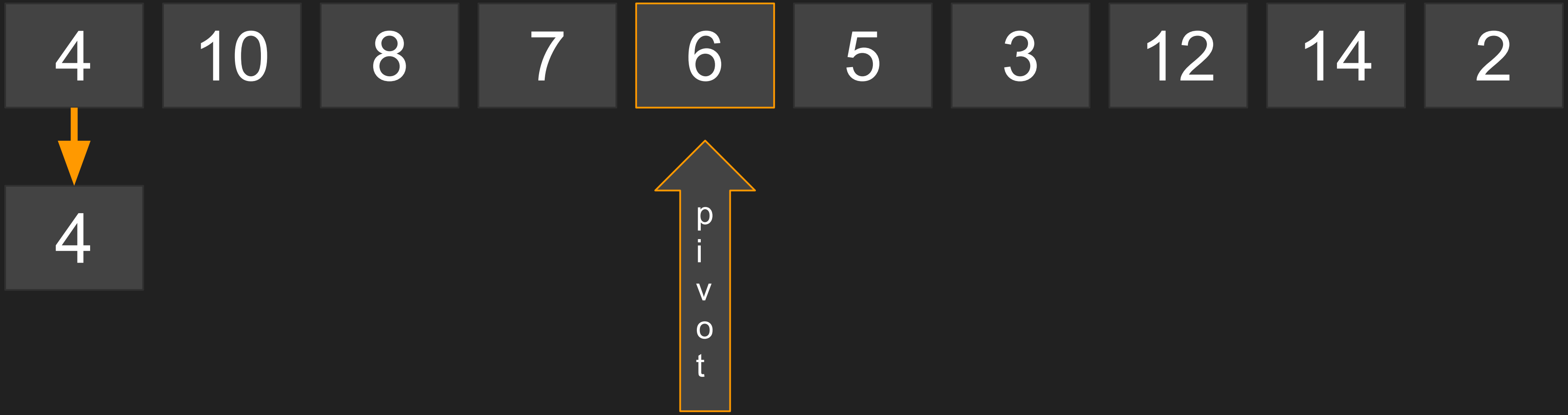
Quicksort example



4	10	8	7	6	5	3	12	14	2

Narration: Now we just move all of the ‘smaller’ elements than 6 to the left, and the bigger elements to the right

Quicksort example



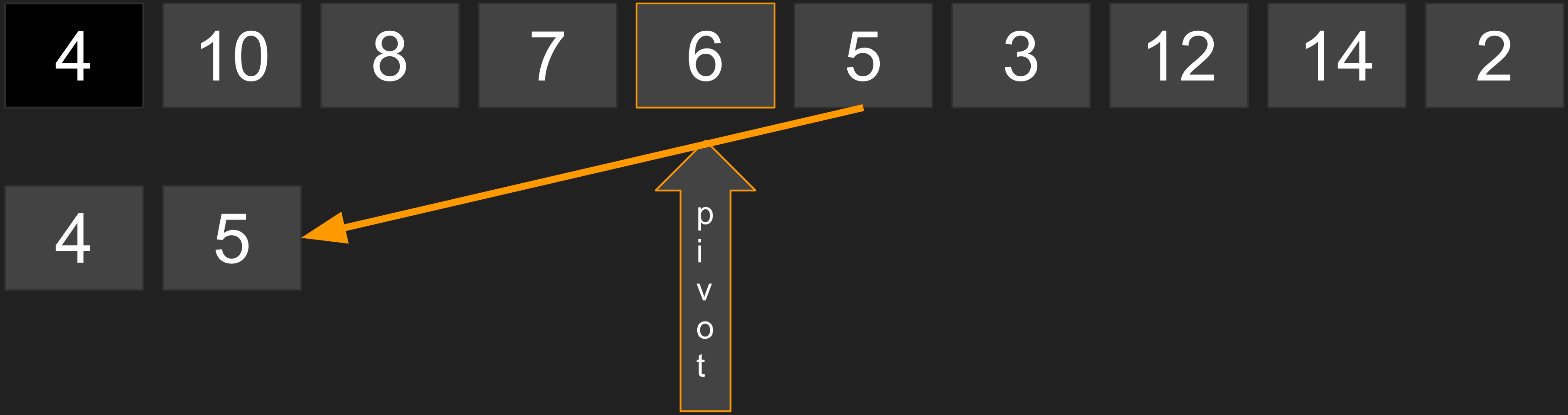
4	10	8	7	6	5	3	12	14	2

4

pivot

Narration: 4 < 6, keep left

Quicksort example

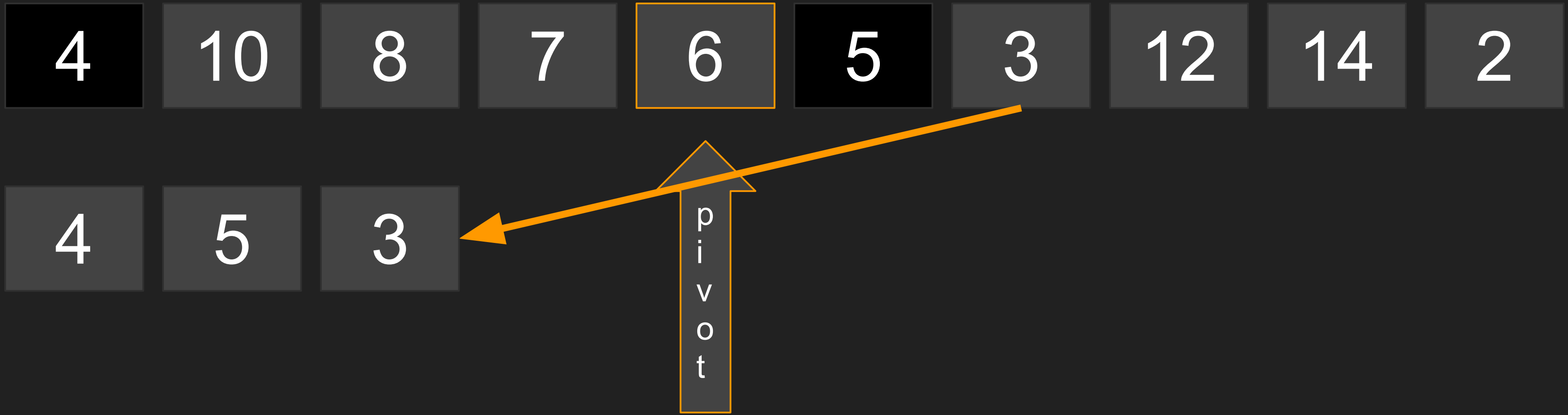


4	10	8	7	6	5	3	12	14	2
4	5								

pivot

Narration: 5 < 6

Quicksort example

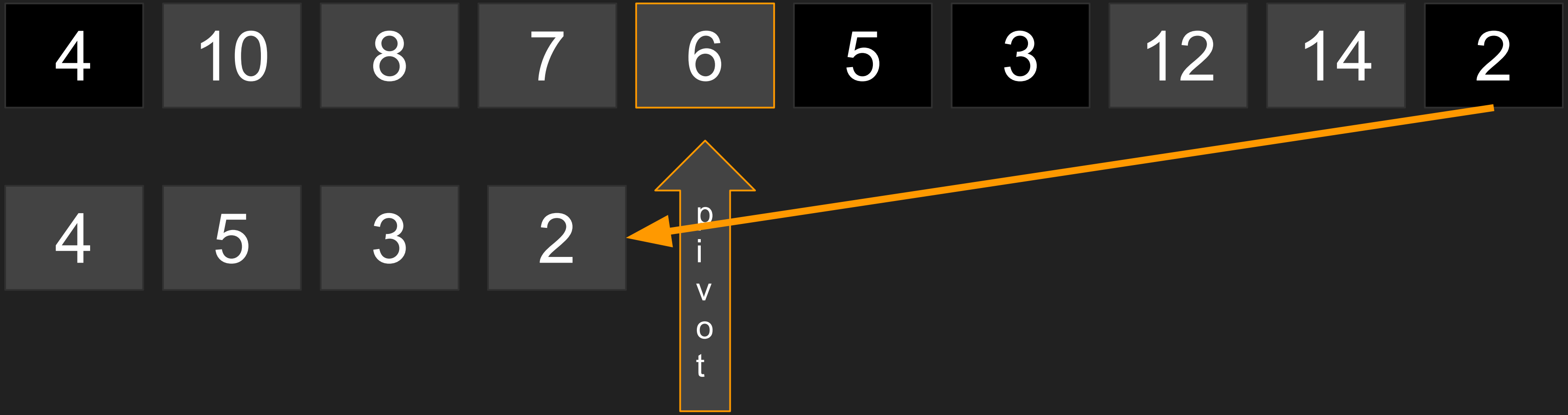


4	10	8	7	6	5	3	12	14	2

4	5	3

Narration: 3 < 6

Quicksort example



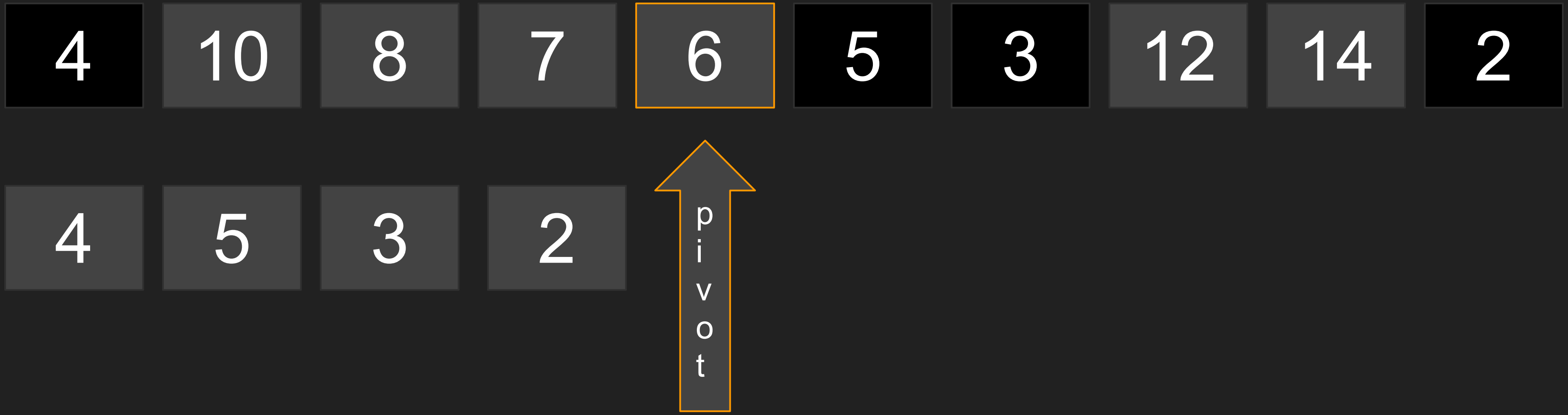
4 10 8 7 6 5 3 12 14 2

4 5 3 2

pivot

Narration: 2 < 6

Quicksort example



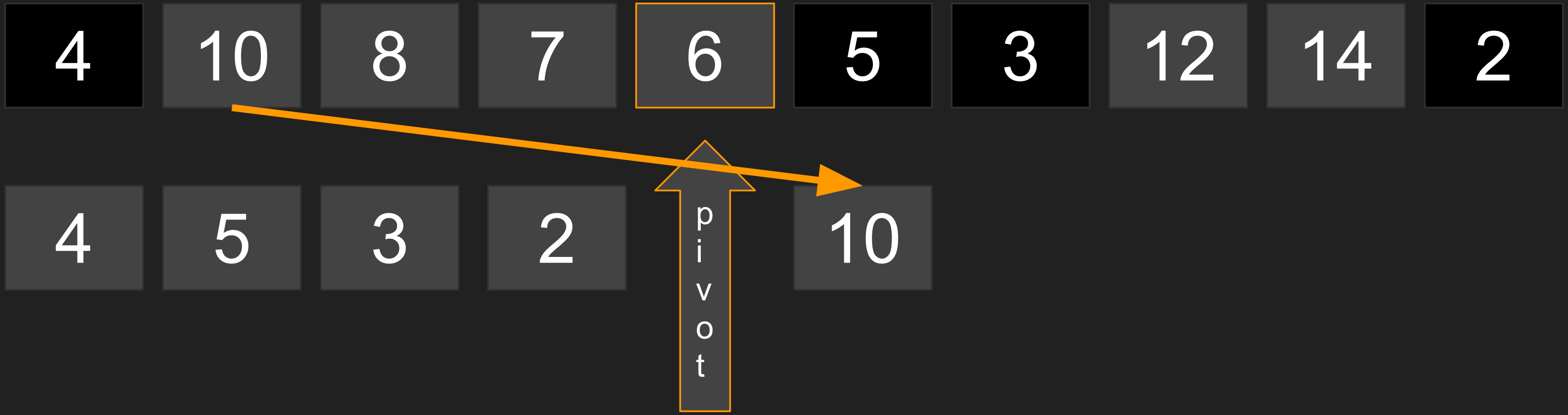
4 10 8 7 6 5 3 12 14 2

4 5 3 2

pivot

Narration: Okay all of the small items are on the left, now let's move everything to the right

Quicksort example

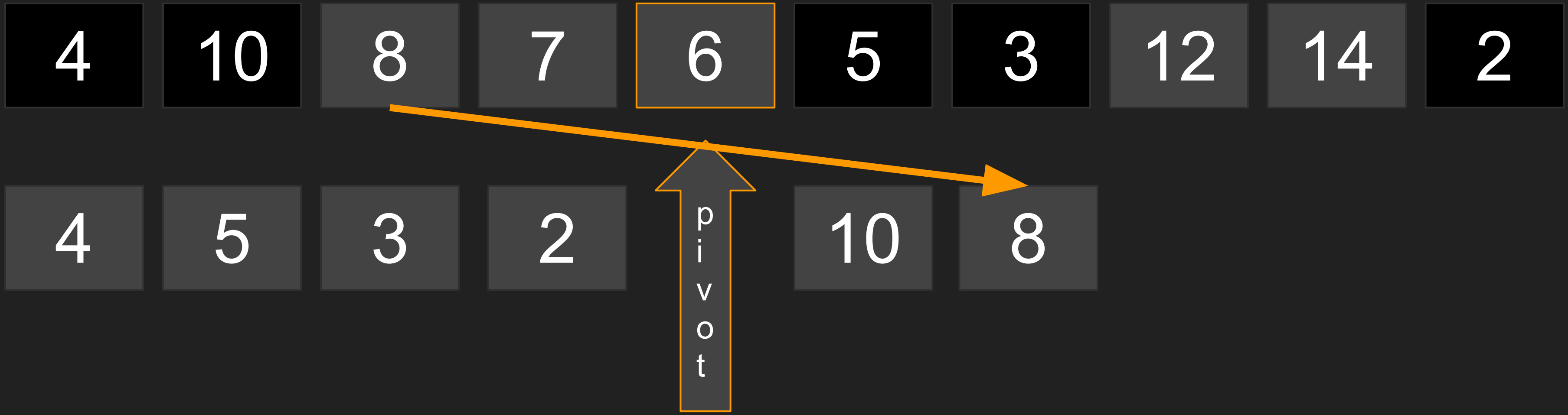


4	10	8	7	6	5	3	12	14	2

4	5	3	2		10

Narration: 10 > 6

Quicksort example

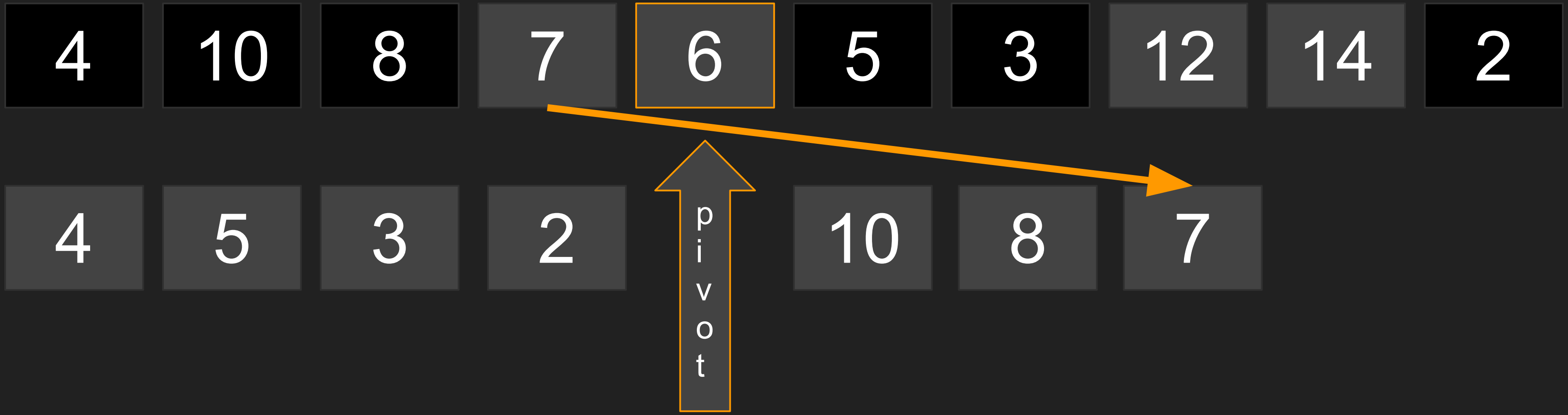


4 10 8 7 6 5 3 12 14 2

4 5 3 2 pivot 10 8

Narration: 8 > 6

Quicksort example

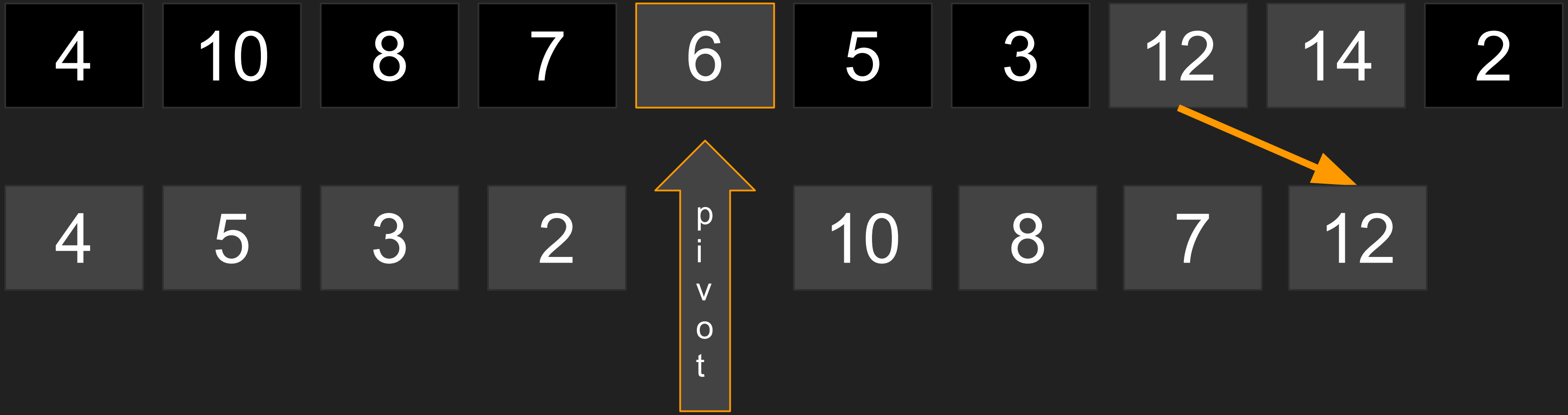


4 10 8 7 6 5 3 12 14 2

4 5 3 2 pivot 10 8 7

Narration: 7 > 6

Quicksort example

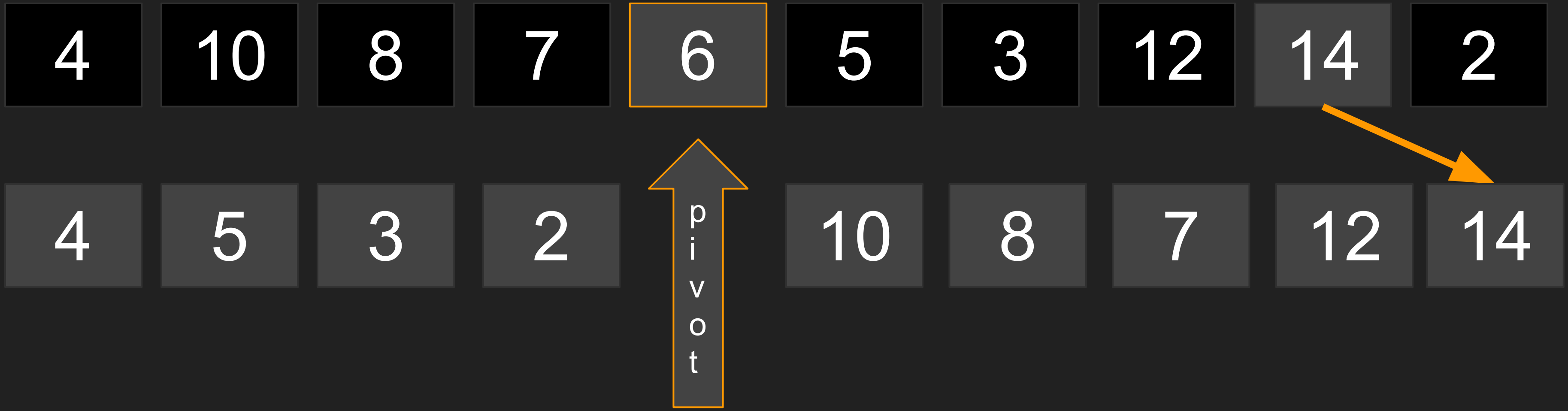


4 10 8 7 6 5 3 12 14 2

4 5 3 2 pivot 10 8 7 12

Narration: 12 > 6

Quicksort example

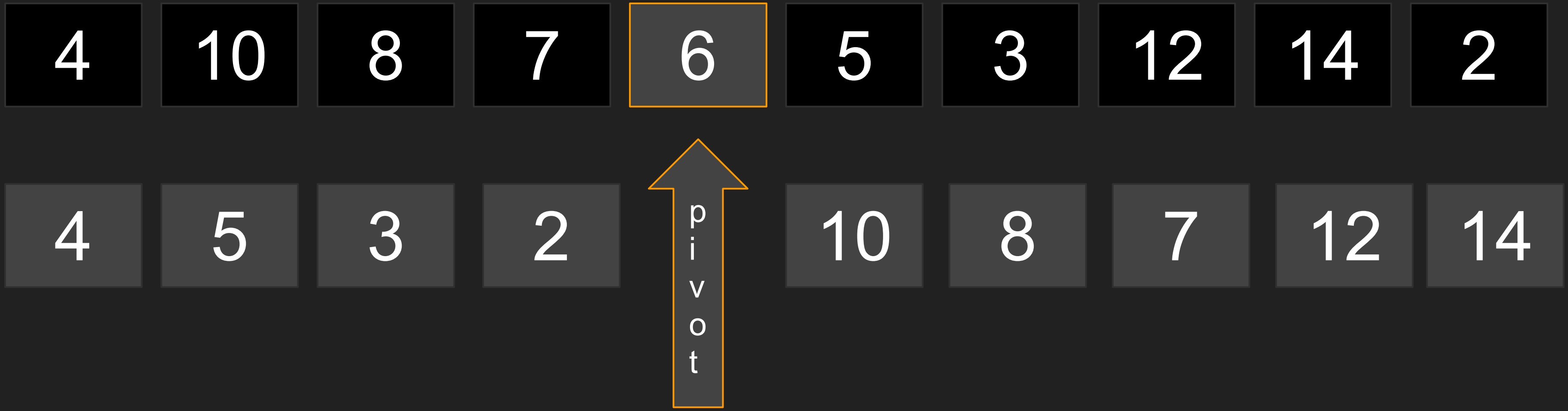


4 10 8 7 6 5 3 12 14 2

4 5 3 2 pivot 10 8 7 12 14

Narration: 14 > 6

Quicksort example



4	10	8	7	6	5	3	12	14	2

4	5	3	2	pivot	10	8	7	12	14

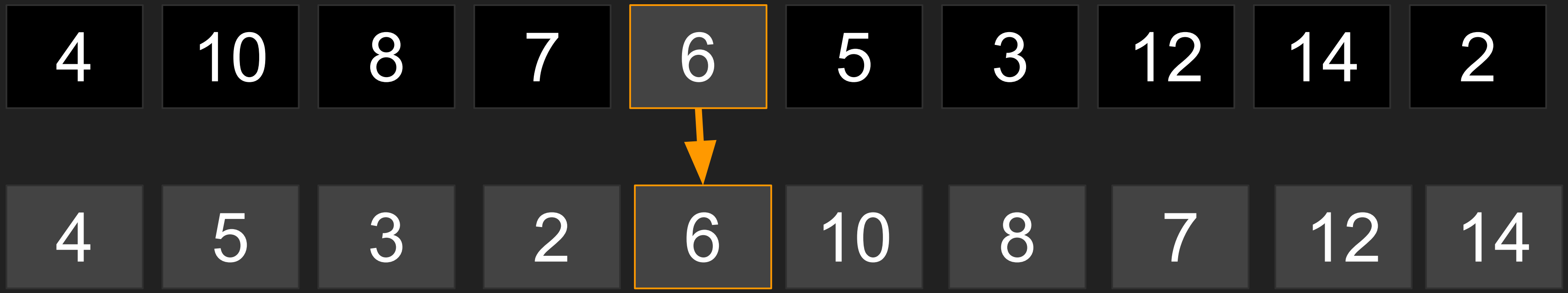
Narration: Okay all of the big items are on the right

Quicksort example

4	10	8	7	6	5	3	12	14	2
4	5	3	2		10	8	7	12	14

Narration: Finally, move our pivot in its final position.

Quicksort example

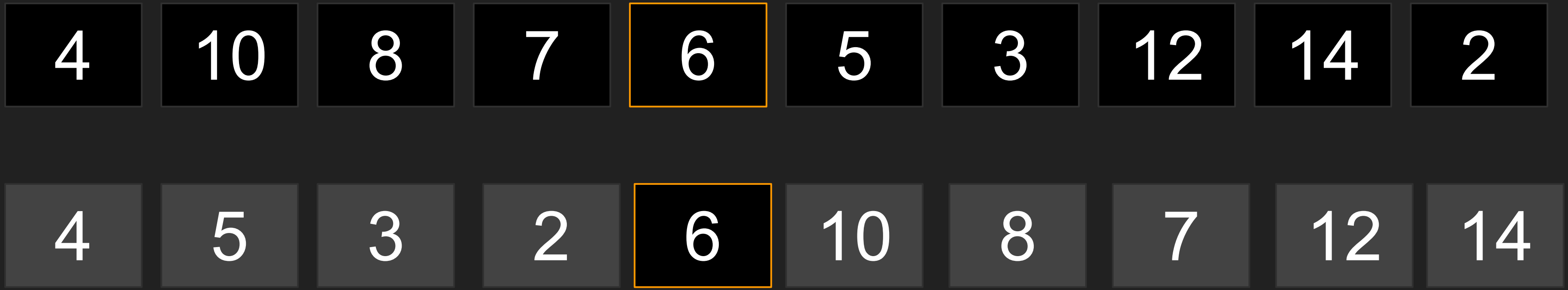


4 10 8 7 6 5 3 12 14 2

4 5 3 2 6 10 8 7 12 14

Narration: Round 1 done!

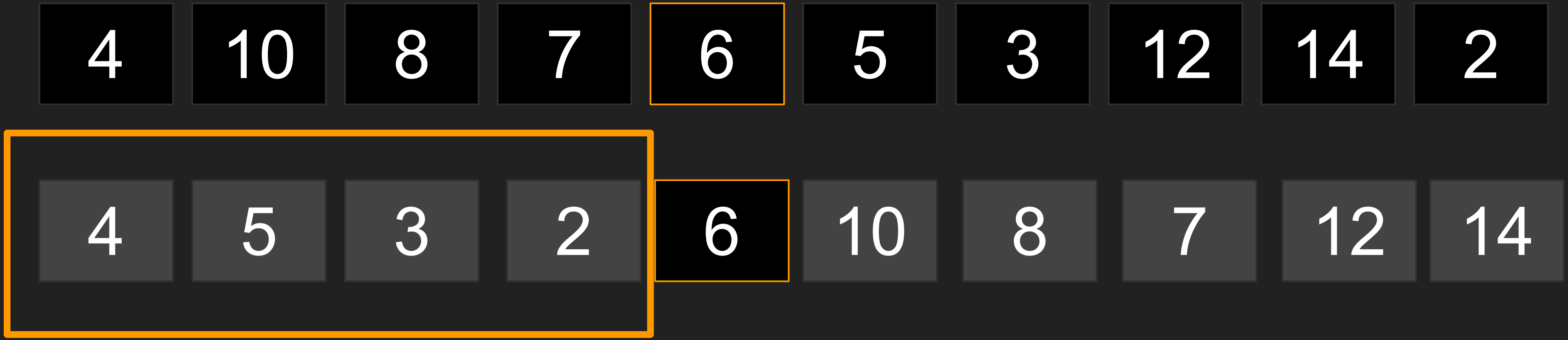
Quicksort example



4	10	8	7	6	5	3	12	14	2
4	5	3	2	6	10	8	7	12	14

Narration: Now we never need to sort around a pivot again. Additionally, we never need to compare the items on the left of the pivot with those of the right

Quicksort example

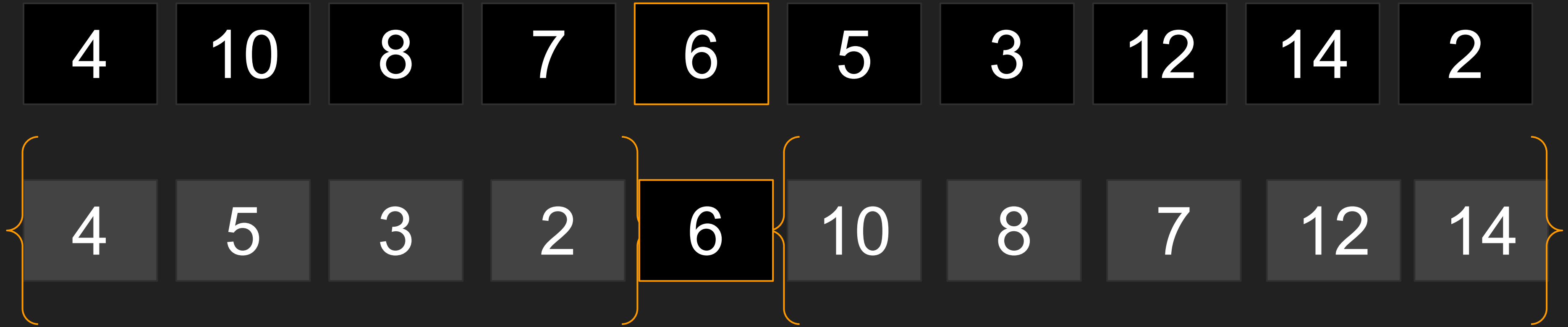


4 10 8 7 6 5 3 12 14 2

4 5 3 2 6 10 8 7 12 14

Narration: We have halved the amount of items we need to look at. This is $\log_2(n)$ behavior!

Quicksort example

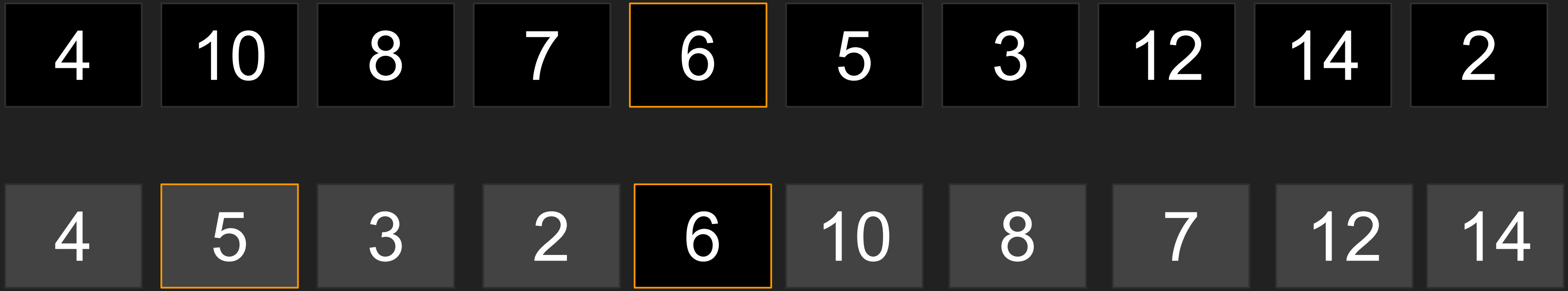


4 10 8 7 6 5 3 12 14 2

4 5 3 2 6 10 8 7 12 14

Narration: We can now repeat this procedure recursing on the left side, and then the right side.

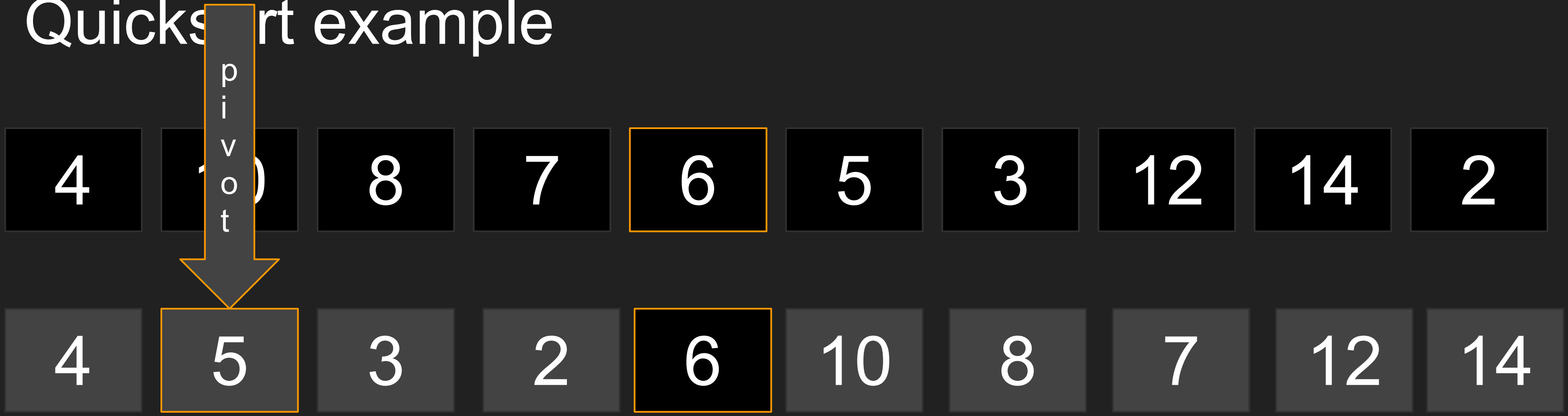
Quicksort example



4	10	8	7	6	5	3	12	14	2
4	5	3	2	6	10	8	7	12	14

Narration: Select a new pivot

Quicksort example

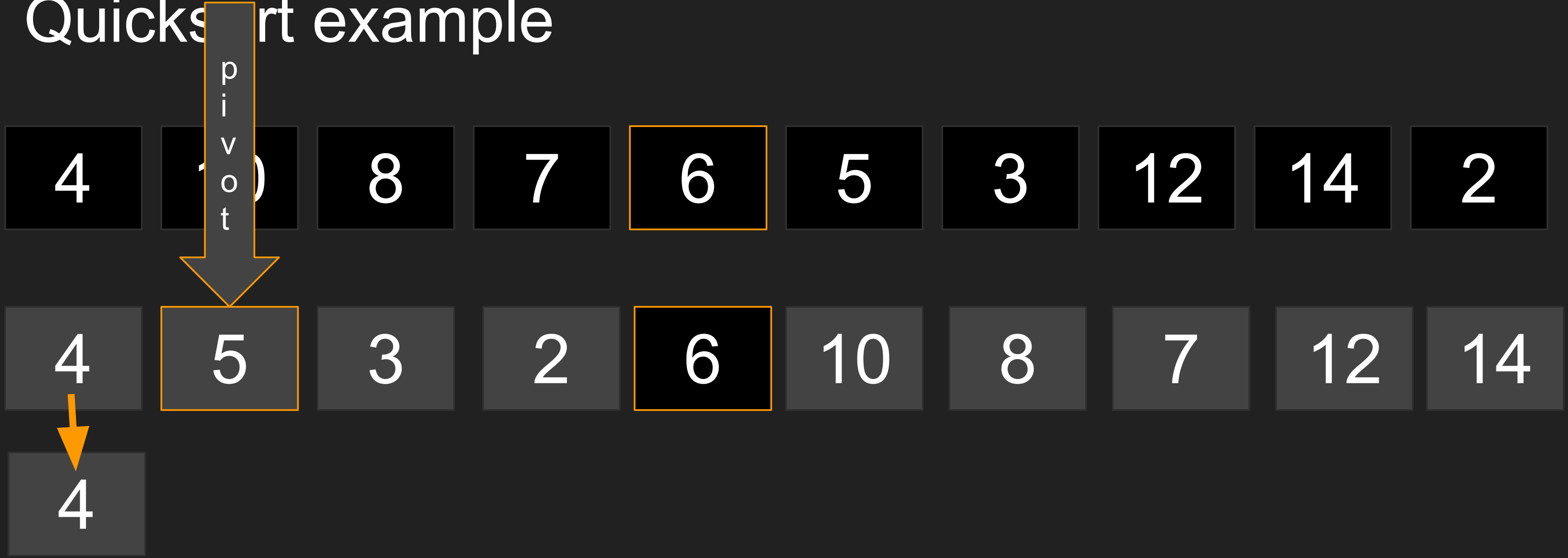


pivot

4	10	8	7	6	5	3	12	14	2
4	5	3	2	6	10	8	7	12	14

Narration:

Quicksort example



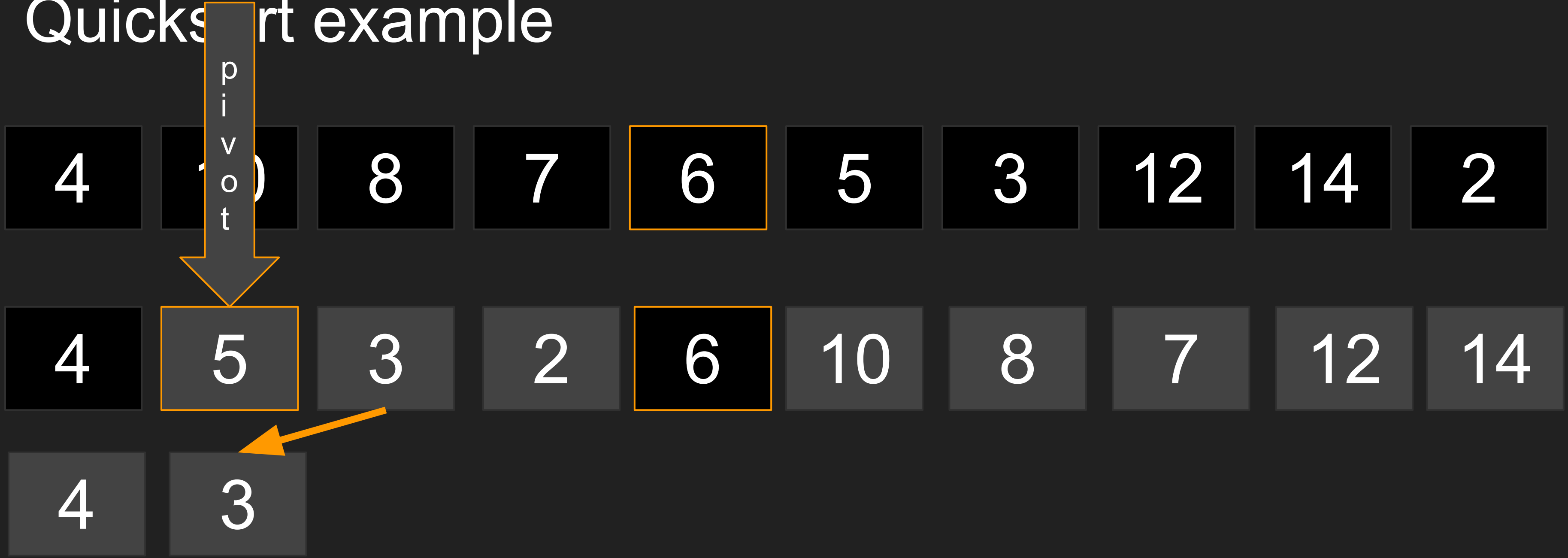
4 10 8 7 6 5 3 12 14 2

4 5 3 2 6 10 8 7 12 14

4

Narration: 4 < 5

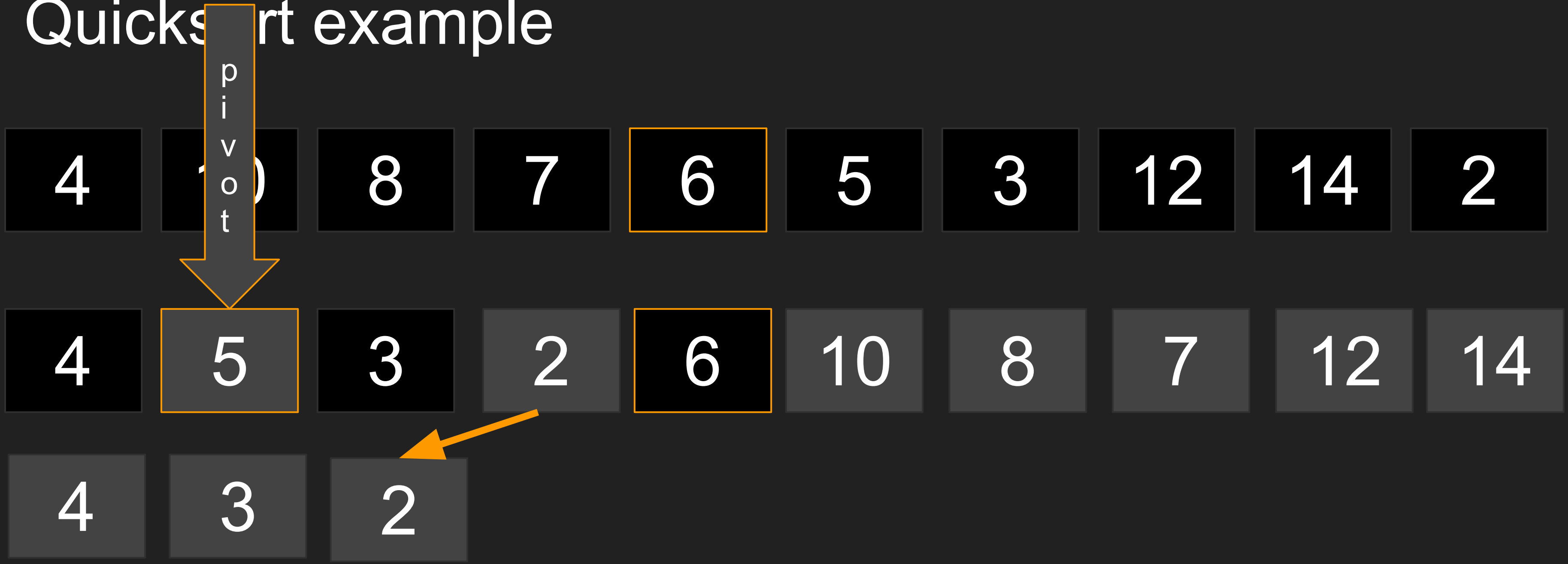
Quicksort example



4	10	8	7	6	5	3	12	14	2
4	5	3	2	6	10	8	7	12	14
4	3								

Narration: 3 < 5

Quicksort example



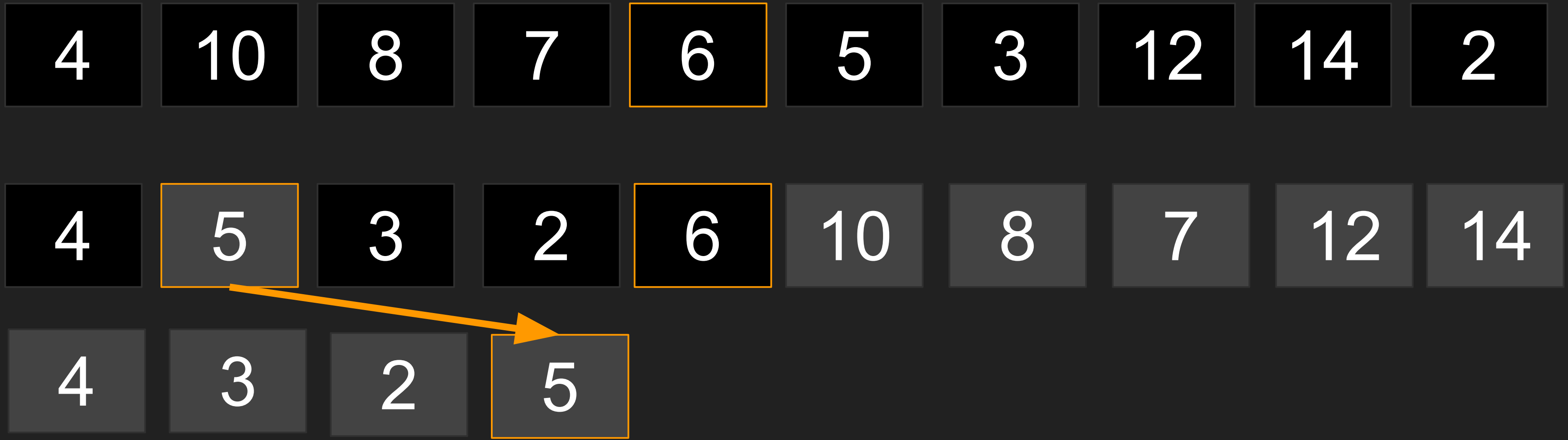
4 10 8 7 6 5 3 12 14 2

4 5 3 2 6 10 8 7 12 14

4 3 2

Narration: 2 < 5

Quicksort example



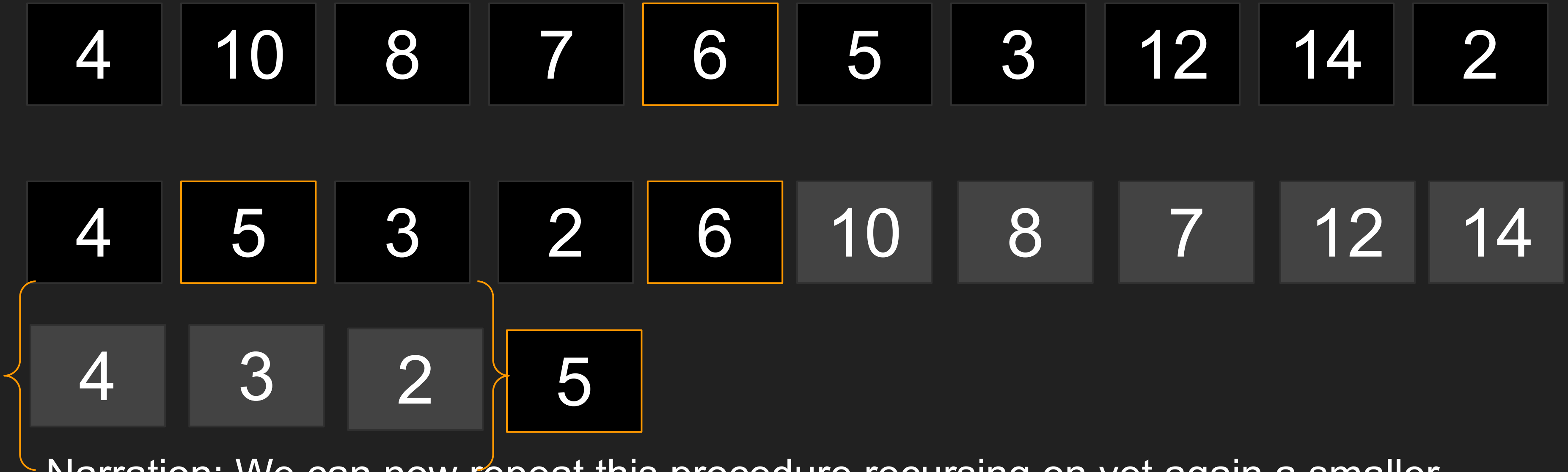
4 10 8 7 6 5 3 12 14 2

4 5 3 2 6 10 8 7 12 14

4 3 2 5

Narration: 5 slides all the way to the right

Quicksort example



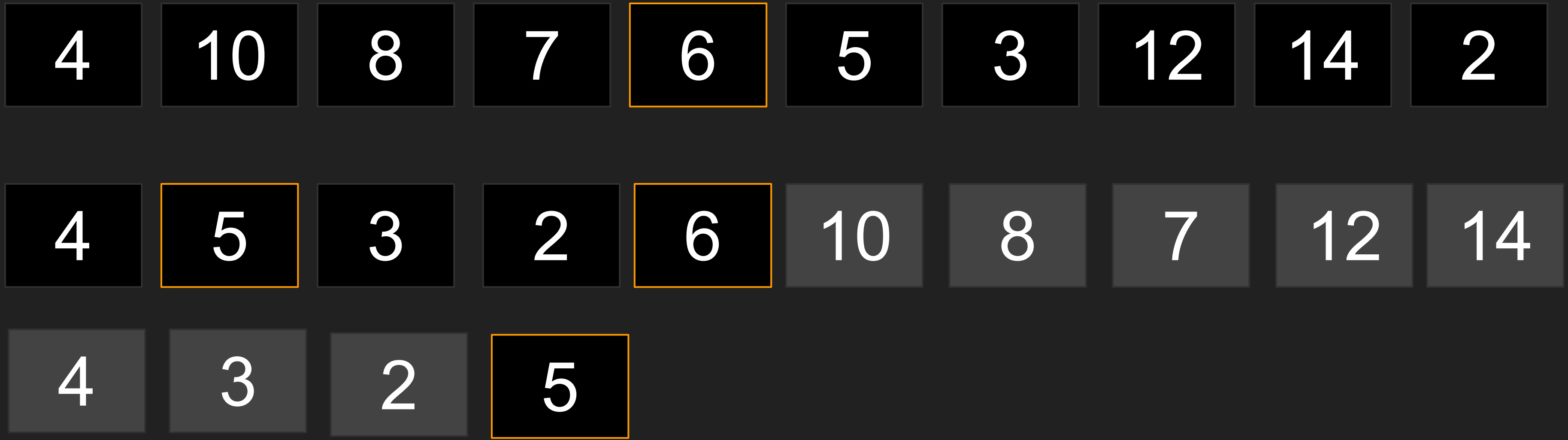
4 10 8 7 6 5 3 12 14 2

4 5 3 2 6 10 8 7 12 14

4 3 2 5

Narration: We can now repeat this procedure recursing on yet again a smaller subset of items

Quicksort example



4 10 8 7 6 5 3 12 14 2

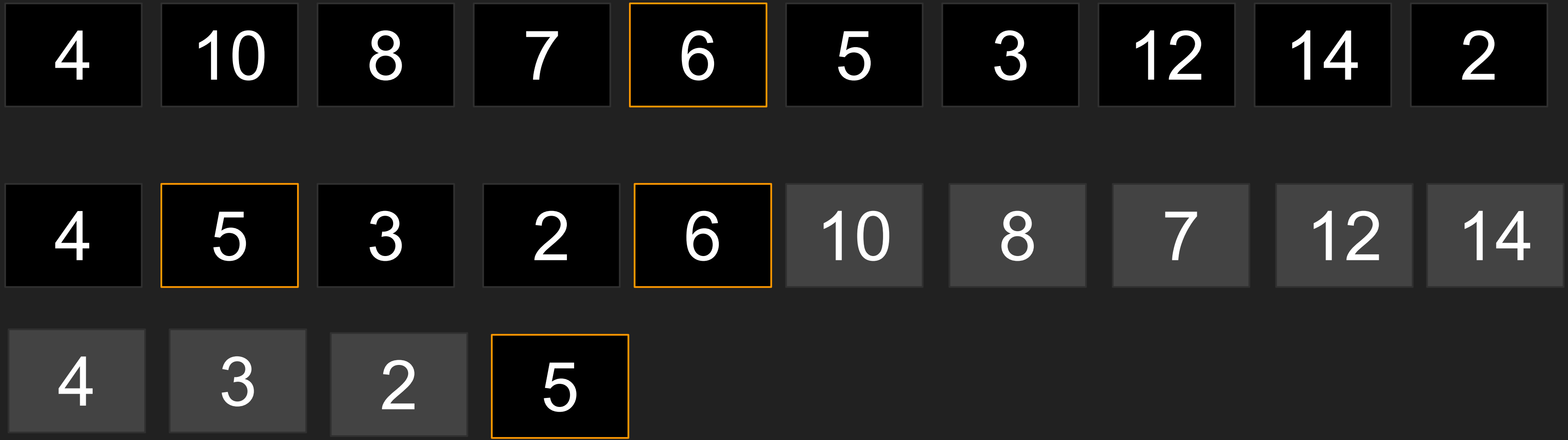
4 5 3 2 6 10 8 7 12 14

4 3 2 5

Narration: Let’s recurse again on the left side.

Narration:

Quicksort example



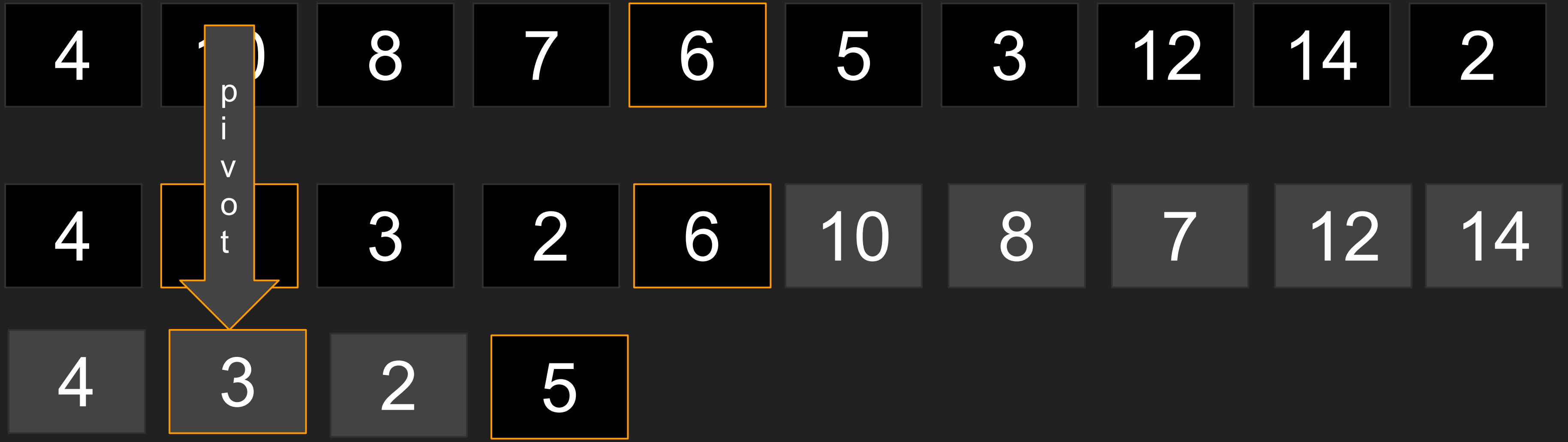
4 10 8 7 6 5 3 12 14 2

4 5 3 2 6 10 8 7 12 14

4 3 2 5

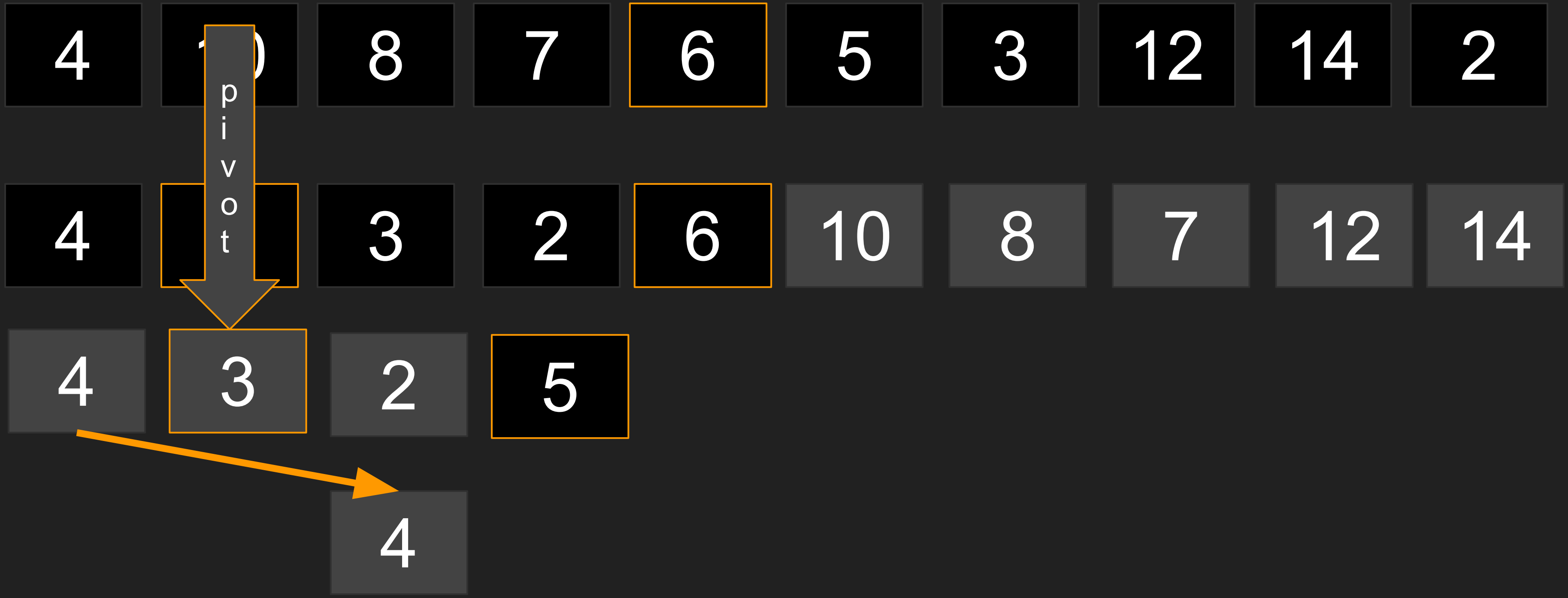
Narration:

Quicksort example



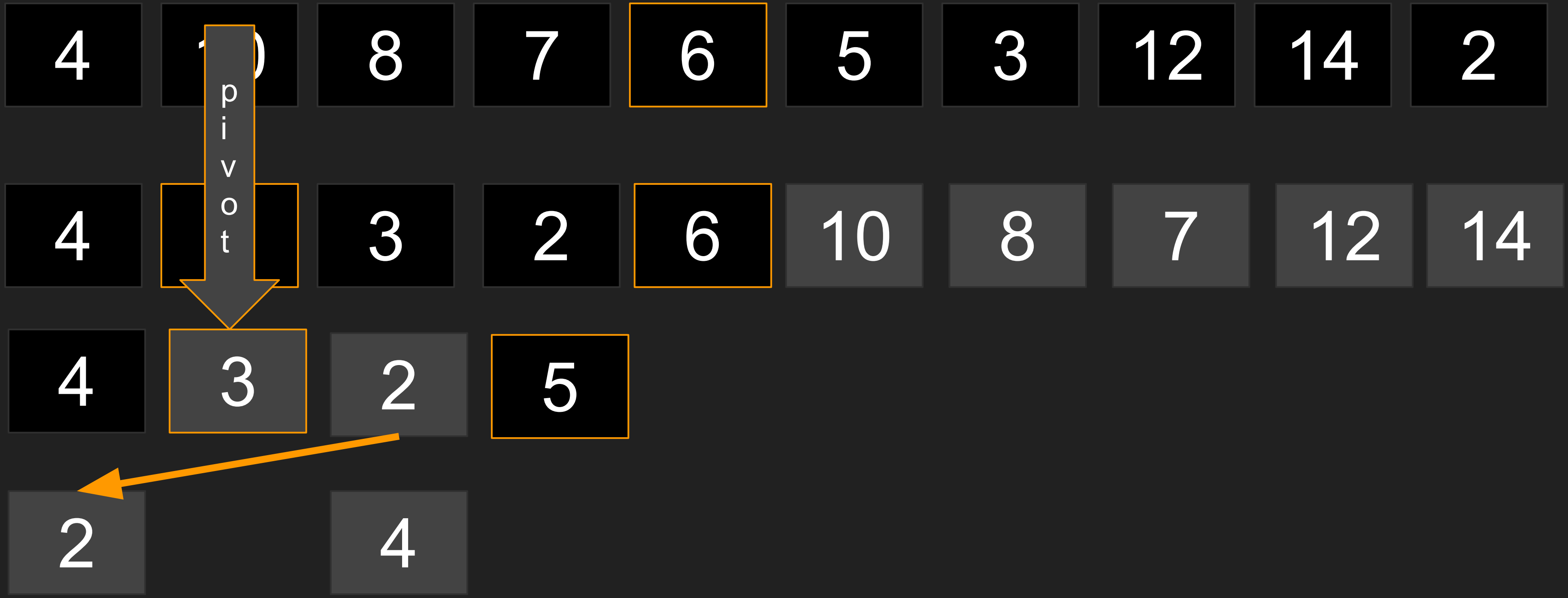
Quicksort example

Narration:



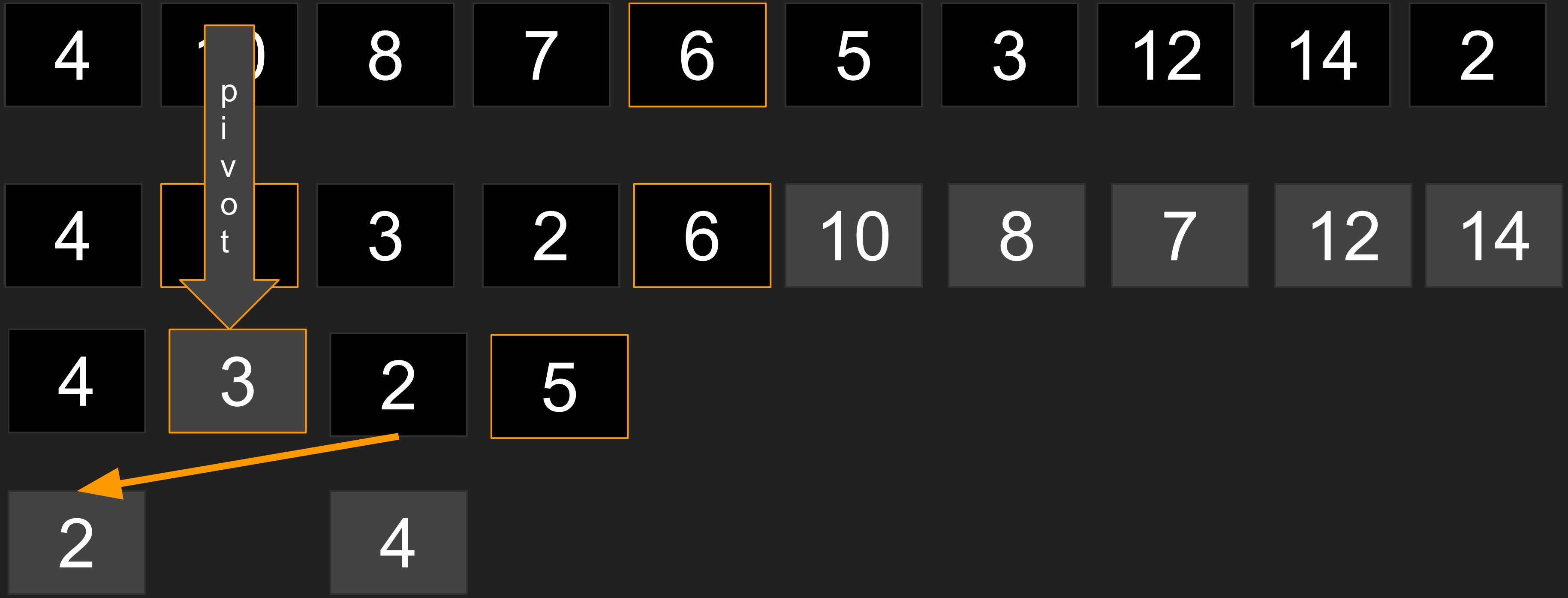
Narration:

Quicksort example



Narration:

Quicksort example



4 10 8 7 6 5 3 12 14 2

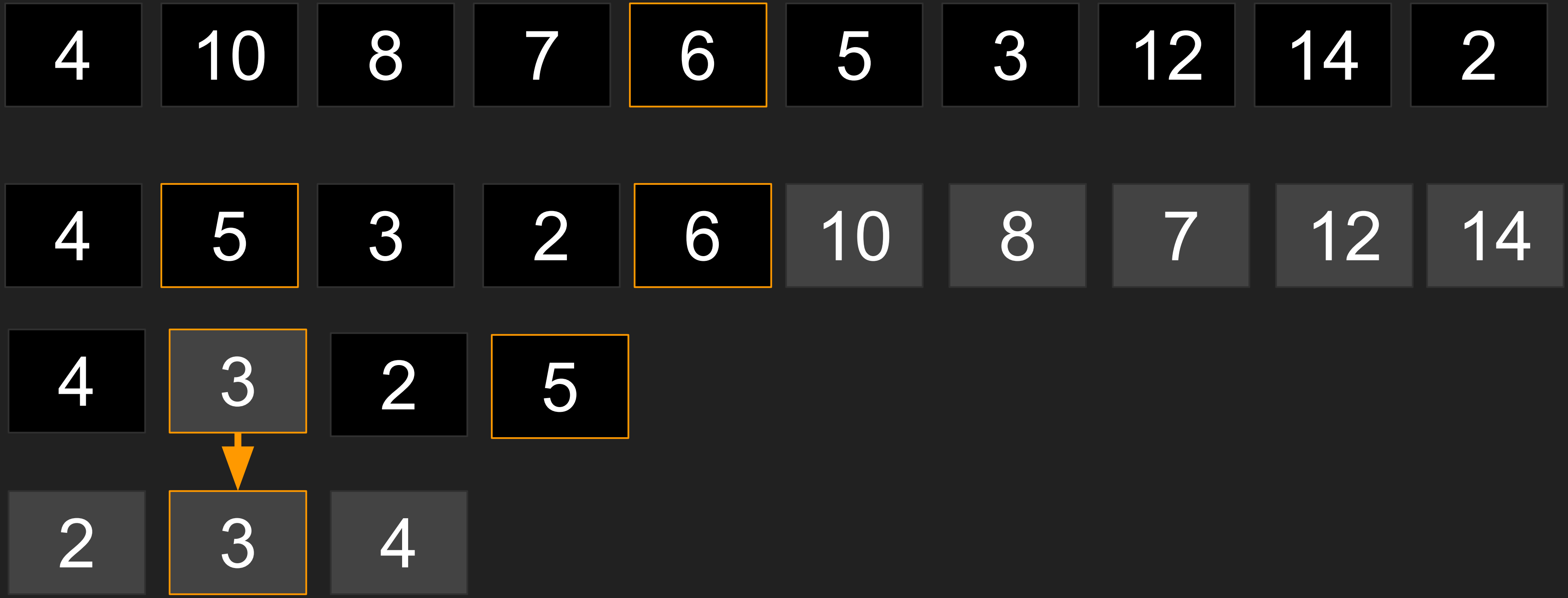
4 3 2 6 10 8 7 12 14

4 3 2 5

2 4

Quicksort example

Narration: We have now sorted the left side!



4 10 8 7 6 5 3 12 14 2

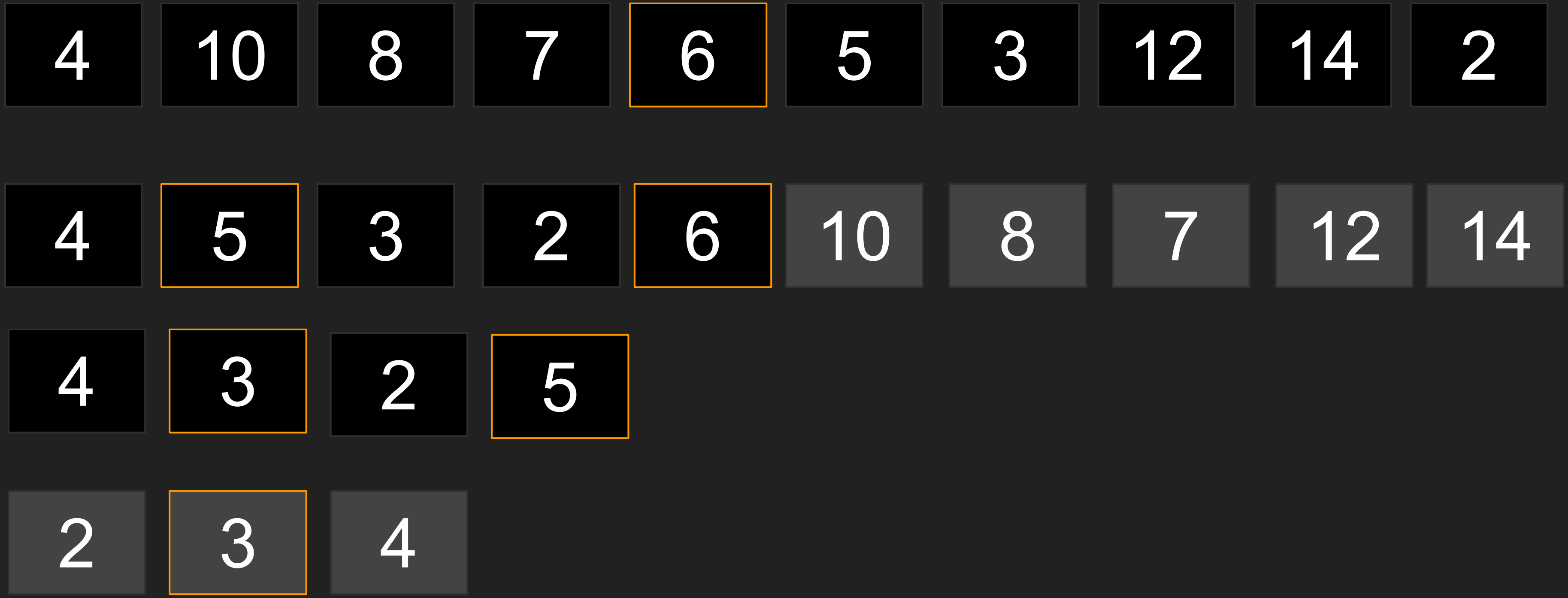
4 5 3 2 6 10 8 7 12 14

4 3 2 5

2 3 4

Quicksort example

Narration: We have now sorted the left side!



4 10 8 7 6 5 3 12 14 2

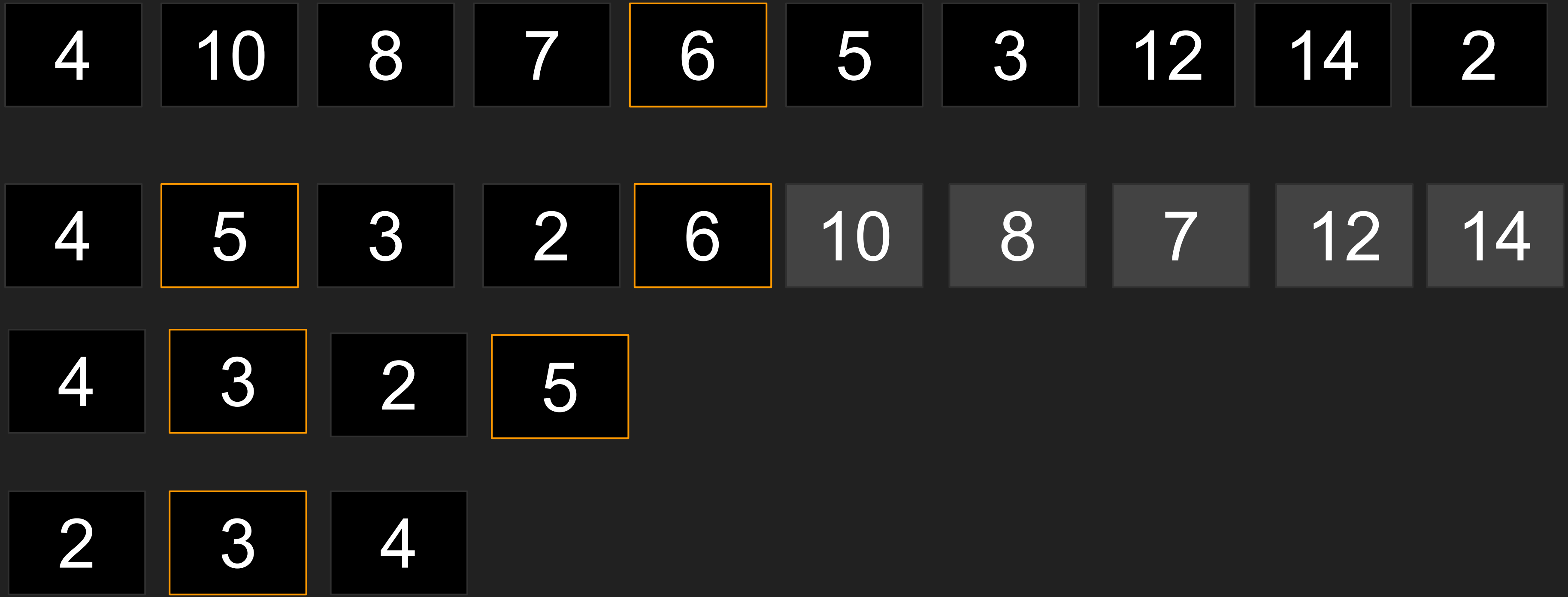
4 5 3 2 6 10 8 7 12 14

4 3 2 5

2 3 4

Narration:

Quicksort example



4 10 8 7 6 5 3 12 14 2

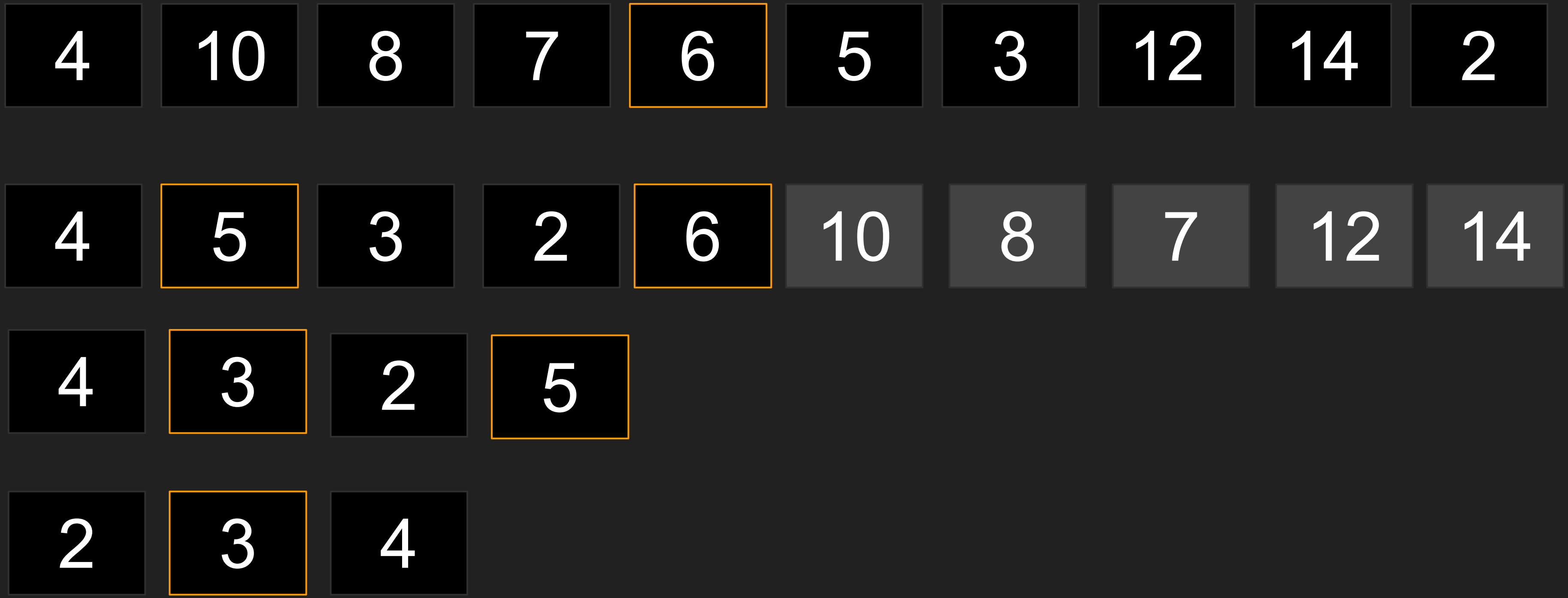
4 5 3 2 6 10 8 7 12 14

4 3 2 5

2 3 4

Quicksort example

Narration: I will now move down the pivots, and then we will recurse on the right side.



4 10 8 7 6 5 3 12 14 2

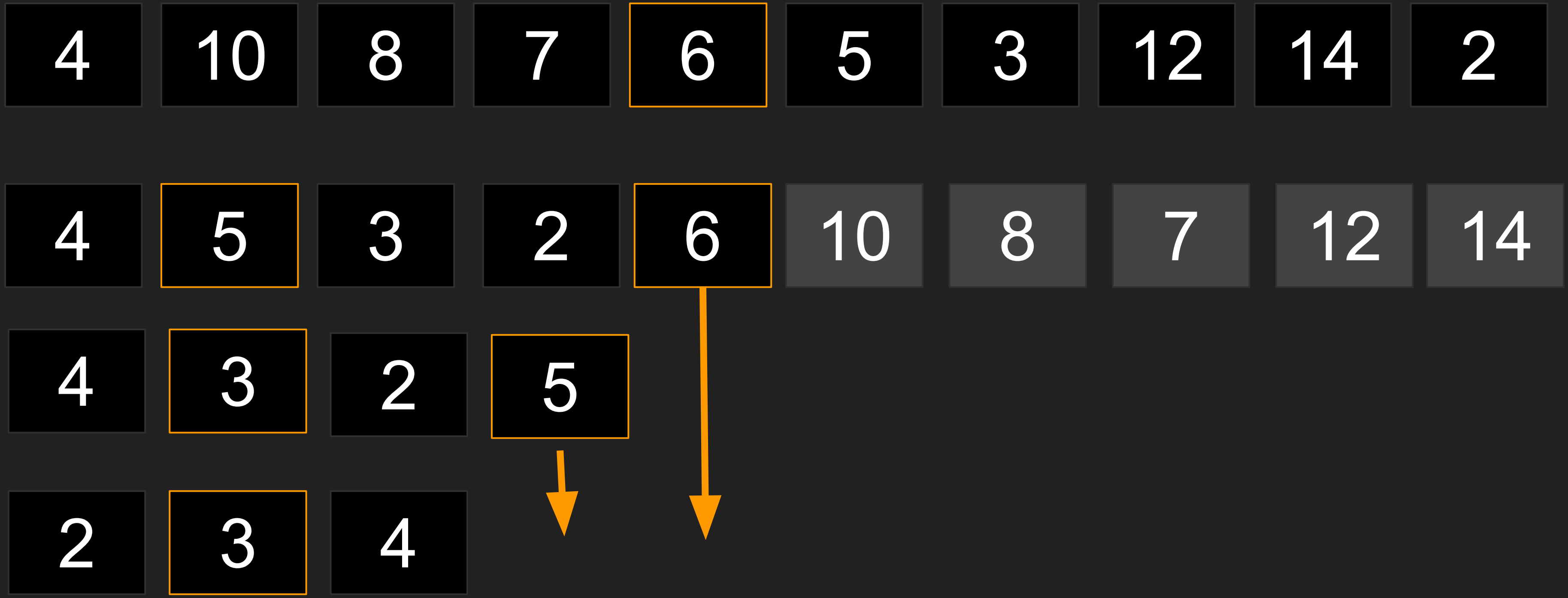
4 5 3 2 6 10 8 7 12 14

4 3 2 5

2 3 4

Quicksort example

Narration: I will now move down the pivots, and then we will recurse on the right side.



4 10 8 7 6 5 3 12 14 2

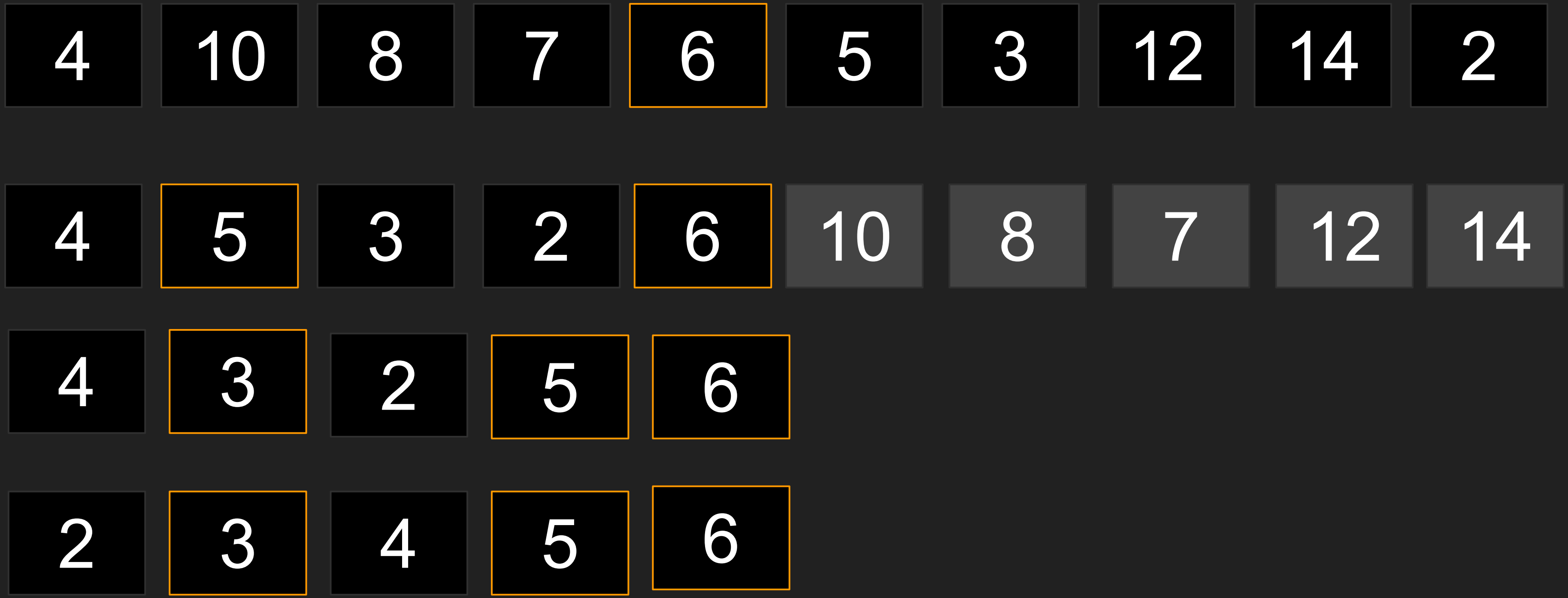
4 5 3 2 6 10 8 7 12 14

4 3 2 5

2 3 4

Quicksort example

Narration: Recurse on the right side!



4 10 8 7 6 5 3 12 14 2

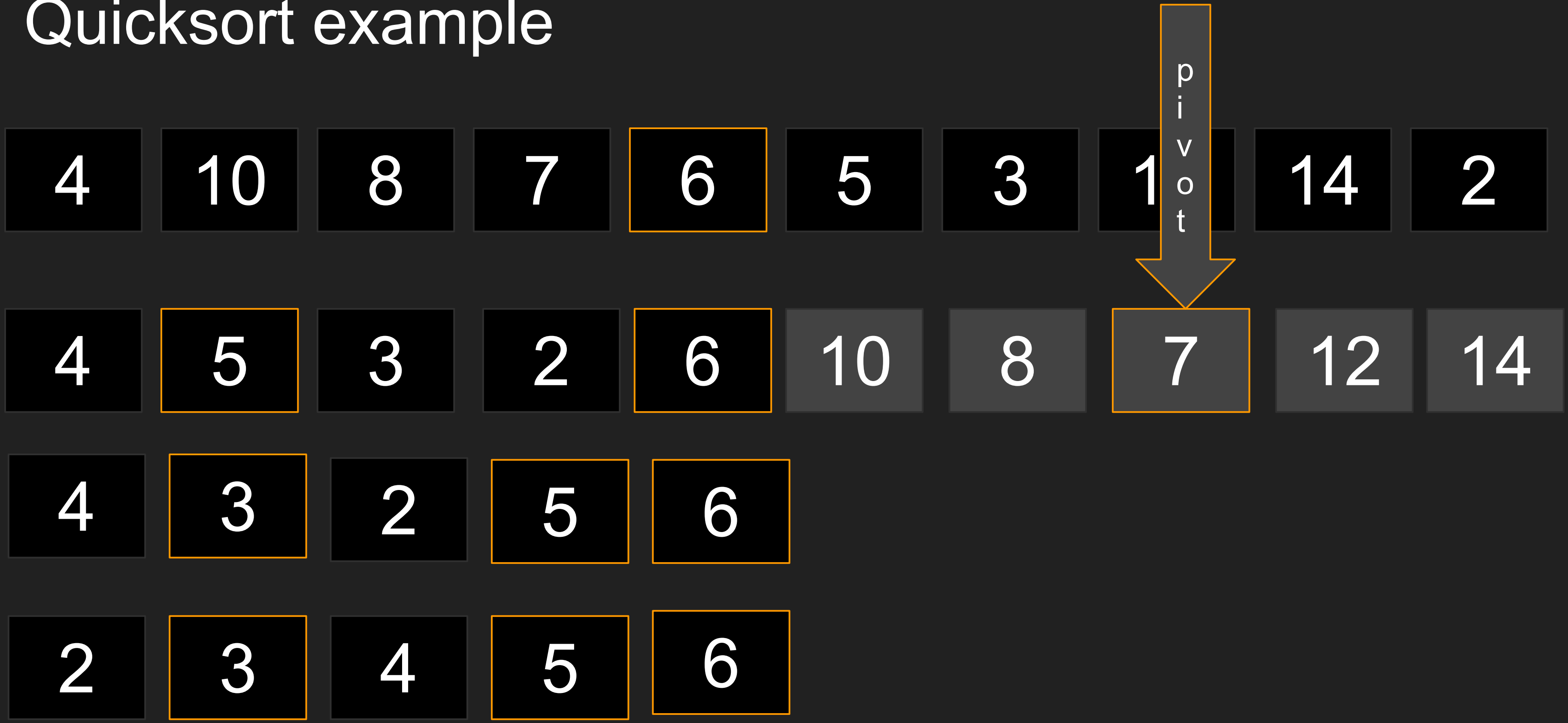
4 5 3 2 6 10 8 7 12 14

4 3 2 5 6

2 3 4 5 6

Narration:

Quicksort example



4 10 8 7 6 5 3 1 14 2

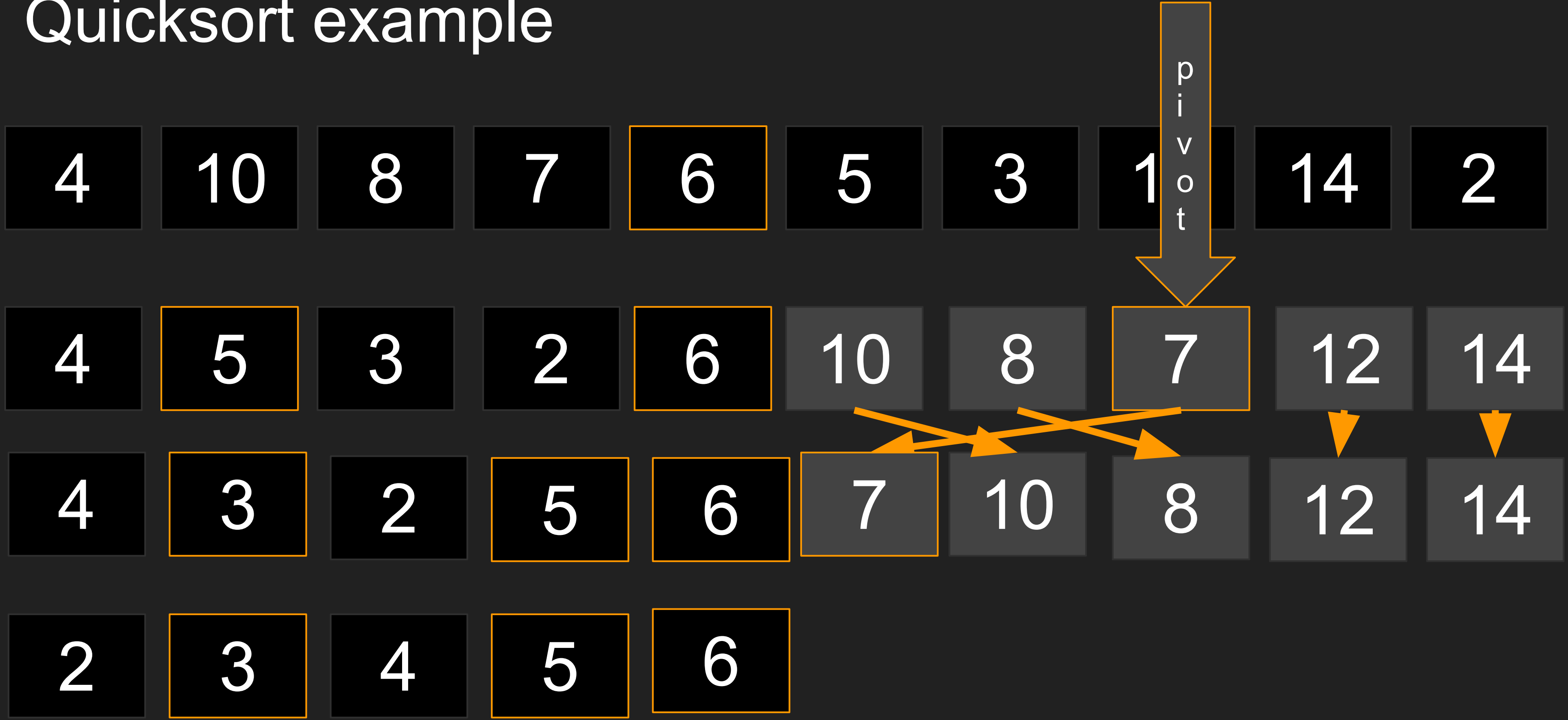
4 5 3 2 6 10 8 7 12 14

4 3 2 5 6

2 3 4 5 6

Narration:

Quicksort example



4 10 8 7 6 5 3 1 14 2

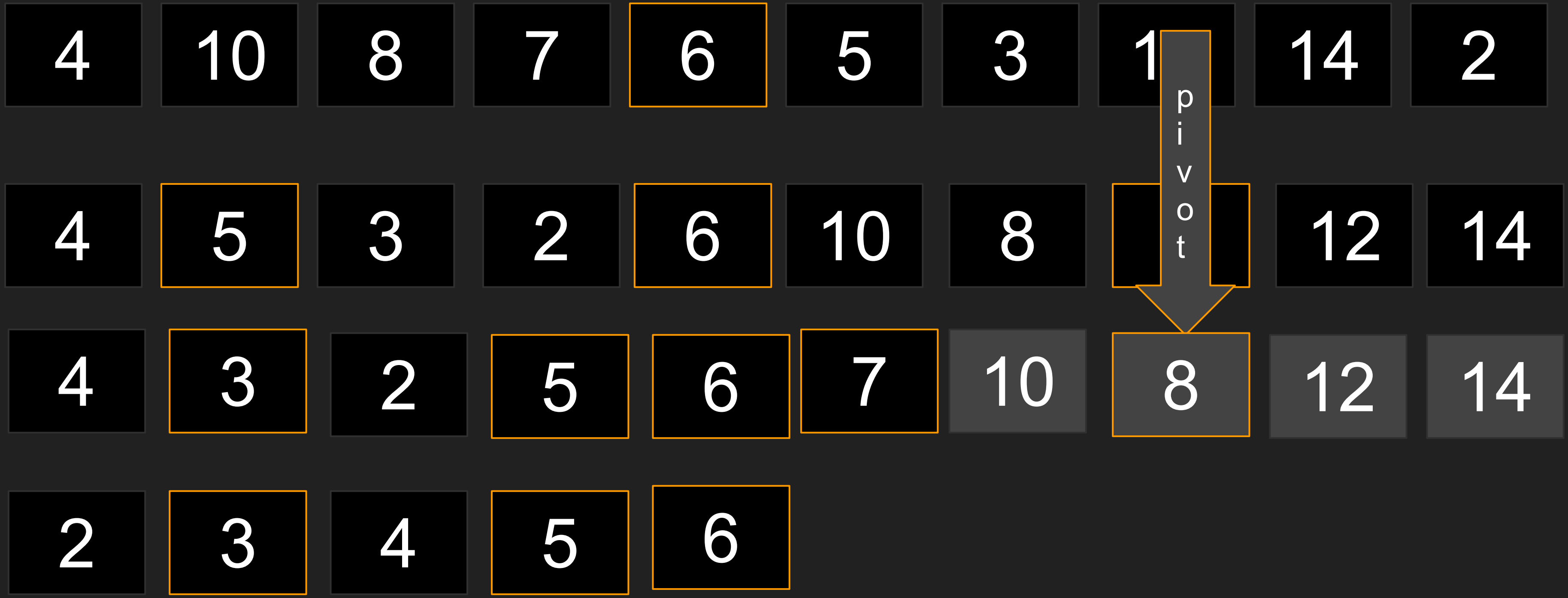
4 5 3 2 6 10 8 7 12 14

4 3 2 5 6 7 10 8 12 14

2 3 4 5 6

Narration: Choose another pivot

Quicksort example



4 10 8 7 6 5 3 1 14 2

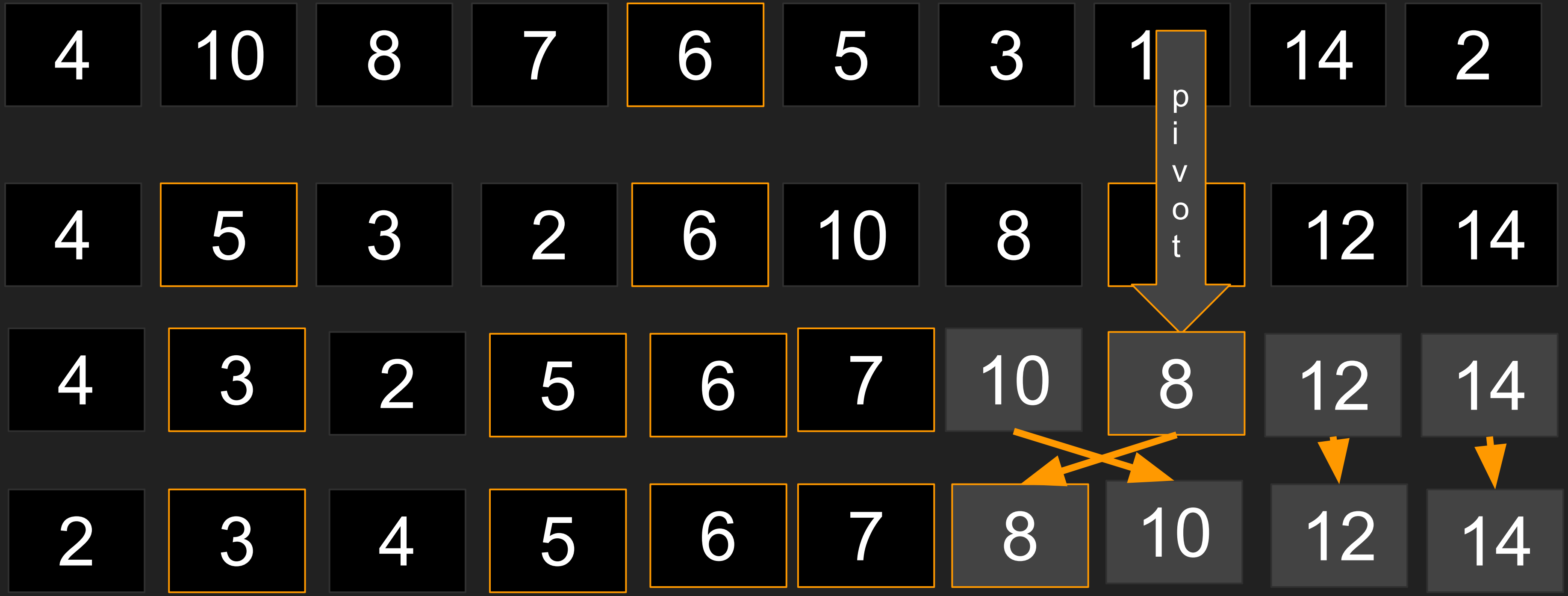
4 5 3 2 6 10 8 12 14

4 3 2 5 6 7 10 8 12 14

2 3 4 5 6

Narration: Choose another pivot

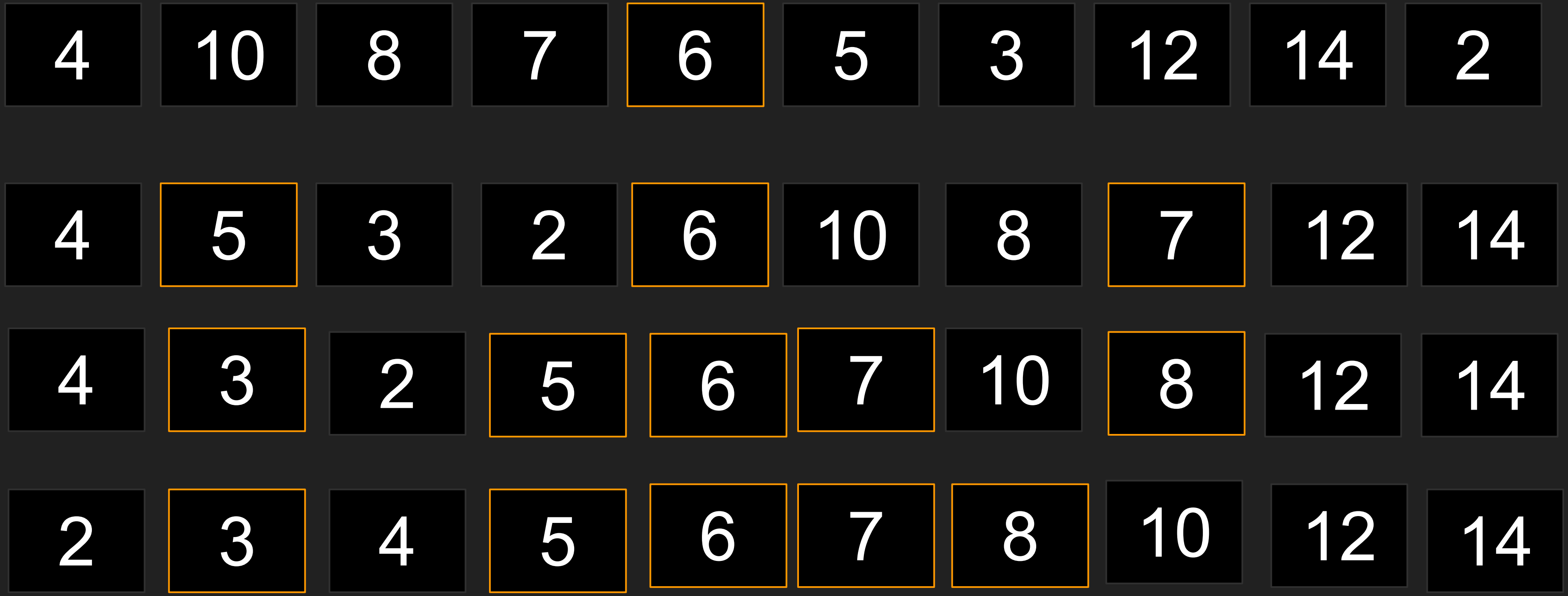
Quicksort example



4	10	8	7	6	5	3	1	14	2
4	5	3	2	6	10	8		12	14
4	3	2	5	6	7	10	8	12	14
2	3	4	5	6	7	8	10	12	14

Narration: Sorted!

Quicksort example



4 10 8 7 6 5 3 12 14 2

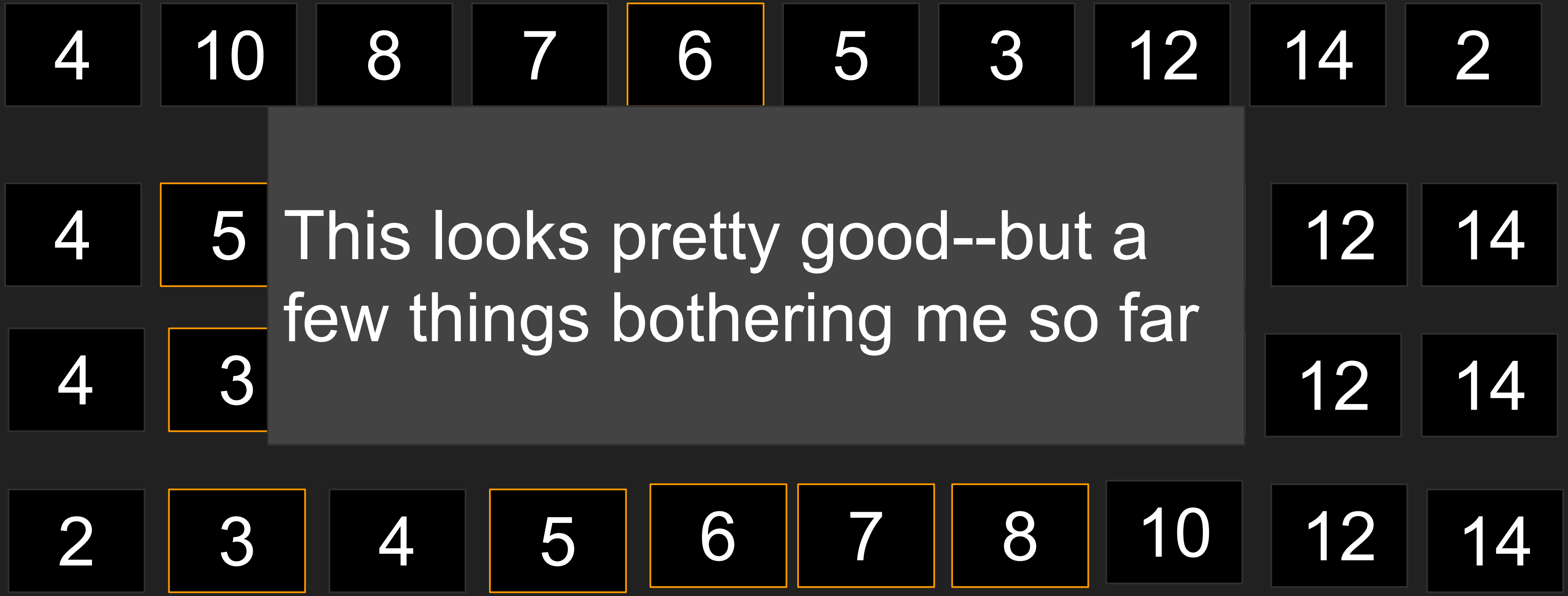
4 5 3 2 6 10 8 7 12 14

4 3 2 5 6 7 10 8 12 14

2 3 4 5 6 7 8 10 12 14

Narration: Sorted!

Quicksort example



4 10 8 7 6 5 3 12 14 2

4 5 12 14

4 3 12 14

2 3 4 5 6 7 8 10 12 14

This looks pretty good--but a few things bothering me so far

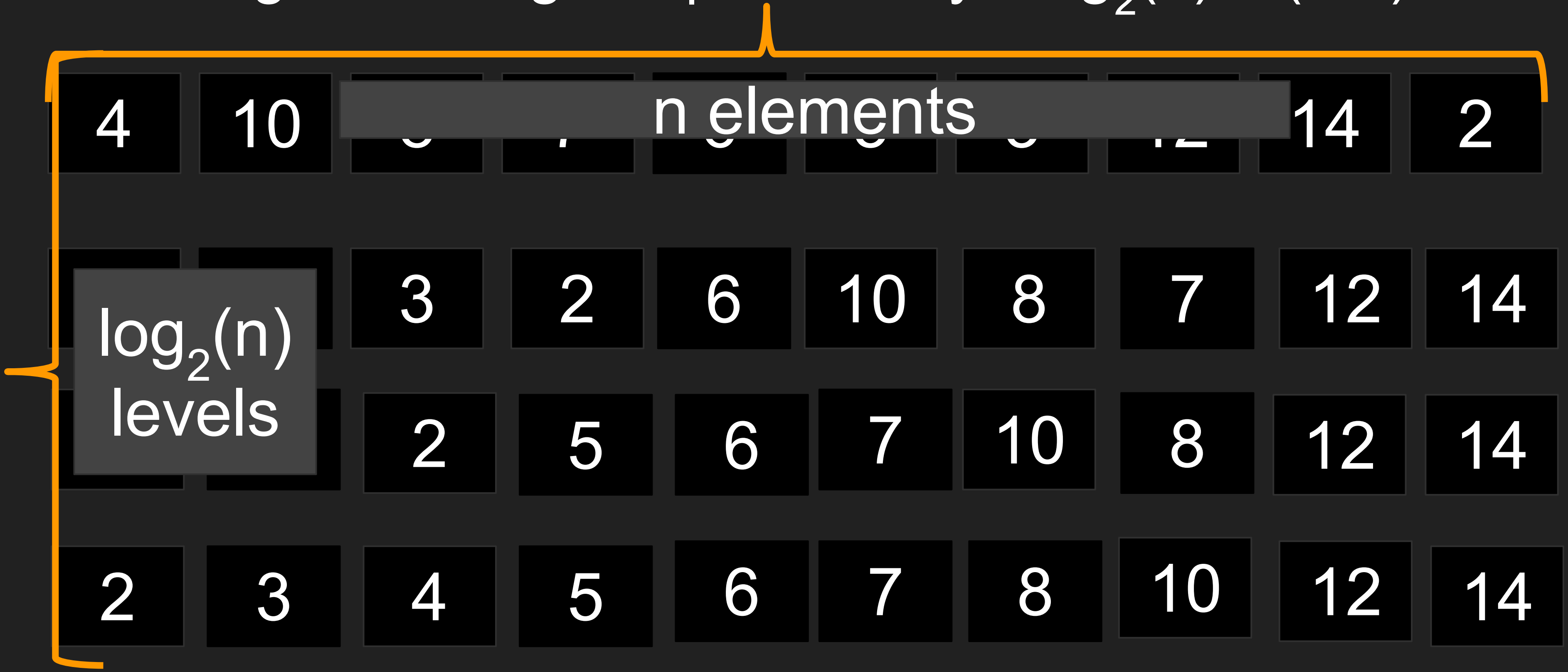
#1 thing bothering me | How to choose the pivot?

- Typically we choose the number in the middle of the list
    - Why? Given data that is unsorted the last element may approximately be in the middle
    - But actually, if the data is randomly sorted, then we could also simply take the last element
    - (Either approach is fine!)
- An additional approach is sometimes taken to try to choose an element that is in the middle
    - Compare three elements (take first, ,middle, and last item, and pick the median)
    - This is the ‘median of 3’ strategy.

#2 thing bothering me | Is it really nlog$_2$(n) ? (1/3)

4	10	8	7	6	5	3	12	14	2
4	5	3	2	6	10	8	7	12	14
4	3	2	5	6	7	10	8	12	14
2	3	4	5	6	7	8	10	12	14

#2 thing bothering me | Is it really $n\log_2(n)$? (2/3)



4 10 14 2

3 2 6 10 8 7 12 14

2 5 6 7 10 8 12 14

2 3 4 5 6 7 8 10 12 14

#2 thing bothering me | Is it really $n\log_2(n)$? (3/3)

- Well, yes as we saw it looked really similar to mergesort
- We’re going to revisit this topic in a moment!
- Let’s look at another example.

#3 thing bothering me | A little too much magic in the previous example

- Okay, let us take a look at a second example that is a little closer to the actual algorithm you would implement.

Quicksort example 2

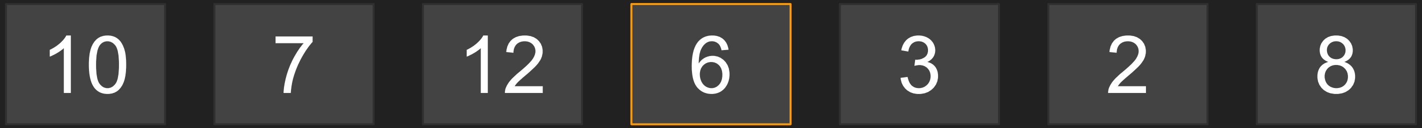
Quicksort round #2

Narration: Here is another set of numbers

10	7	12	6	3	2	8

Quicksort round #2

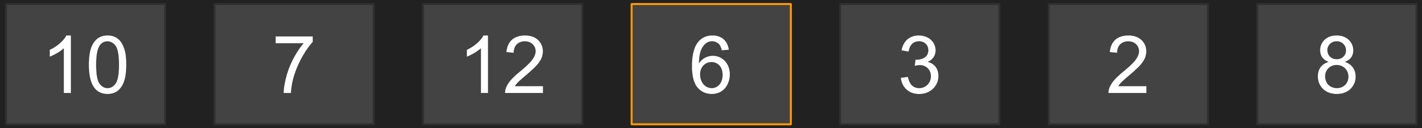
Narration: This time I choose the same middle pivot



10	7	12	6	3	2	8

Quicksort round #2

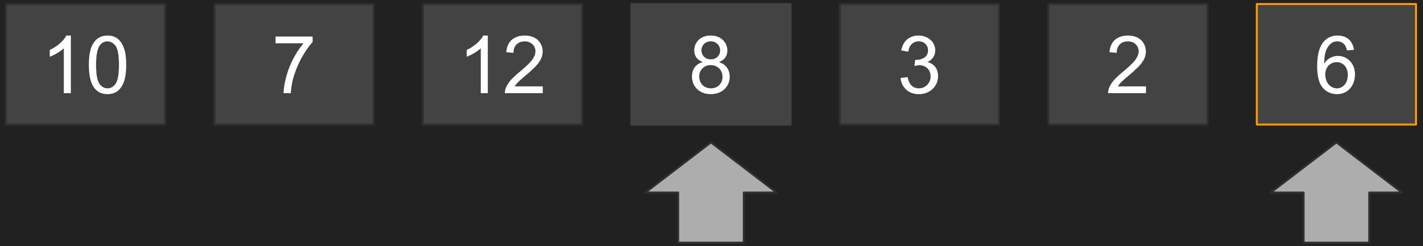
Narration: We move 6 to the end of the list. We could just as easily have chosen 8 however to avoid the move



10	7	12	6	3	2	8

Quicksort round #2

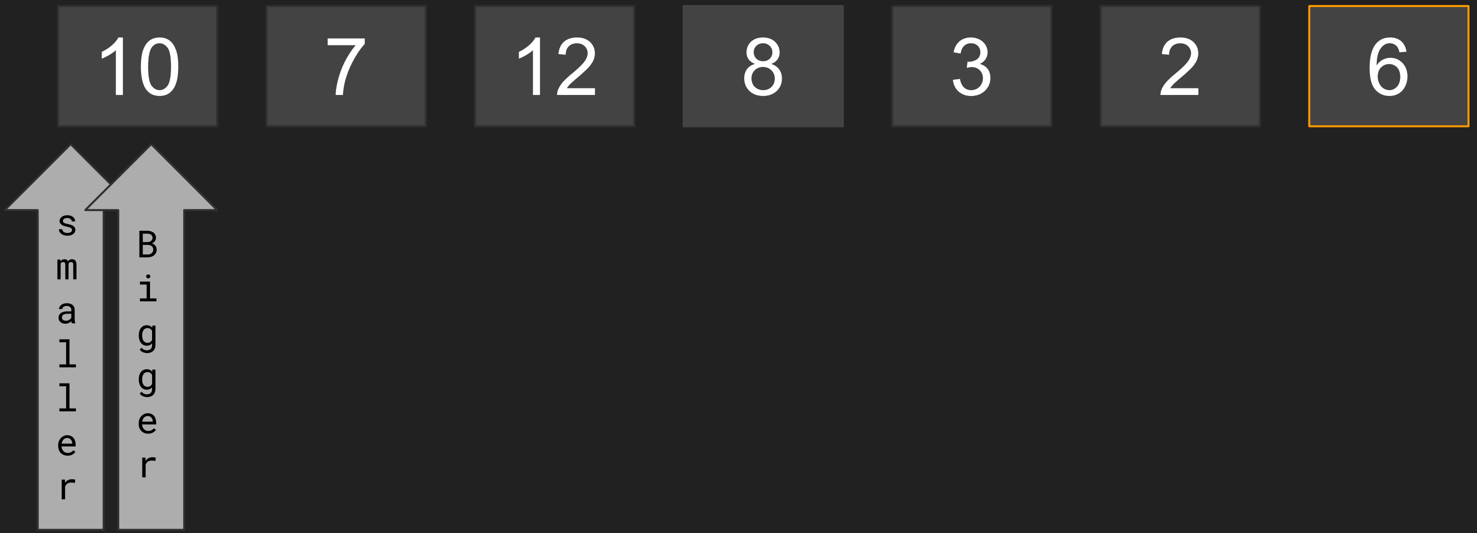
Narration: We move 6 to the end of the list. We could just as easily have chosen 8 however to avoid the move



10	7	12	8	3	2	6

Quicksort round #2

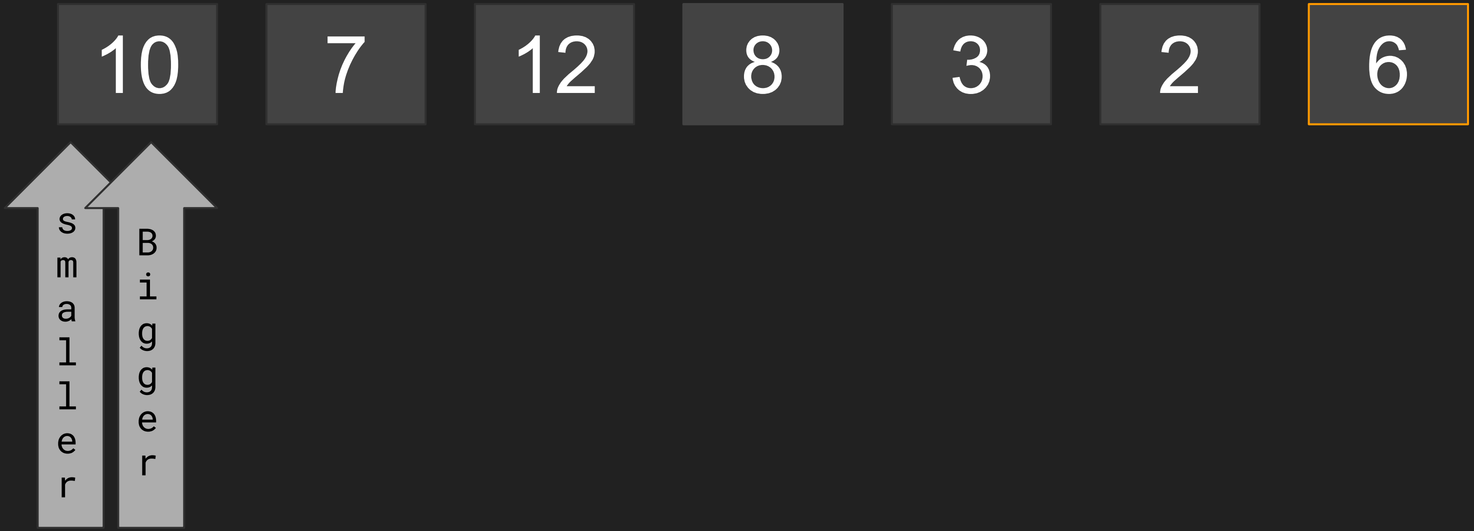
Narration: Now in order to use quicksort ‘in-place’ we will use two counters to iterate through our list. They start at the front of our list.



10	7	12	8	3	2	6

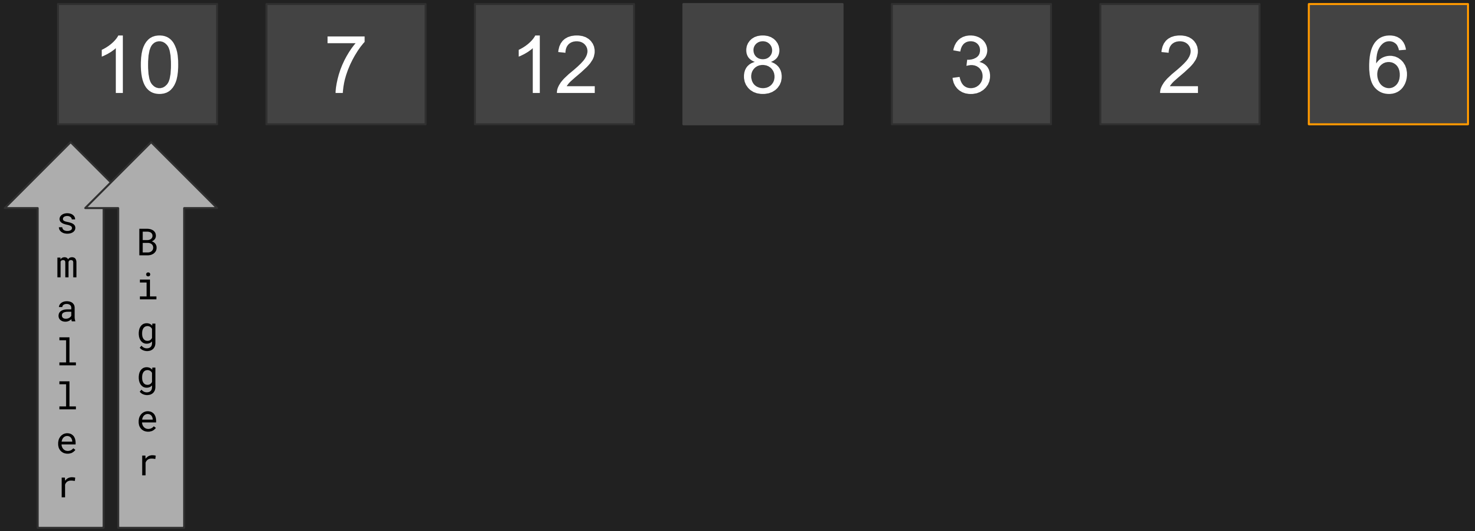
Narration: Compare 10 > 6

Quicksort round #2



Quicksort round #2

Now because 10 is bigger than 6, we will move exactly one of the counters over.



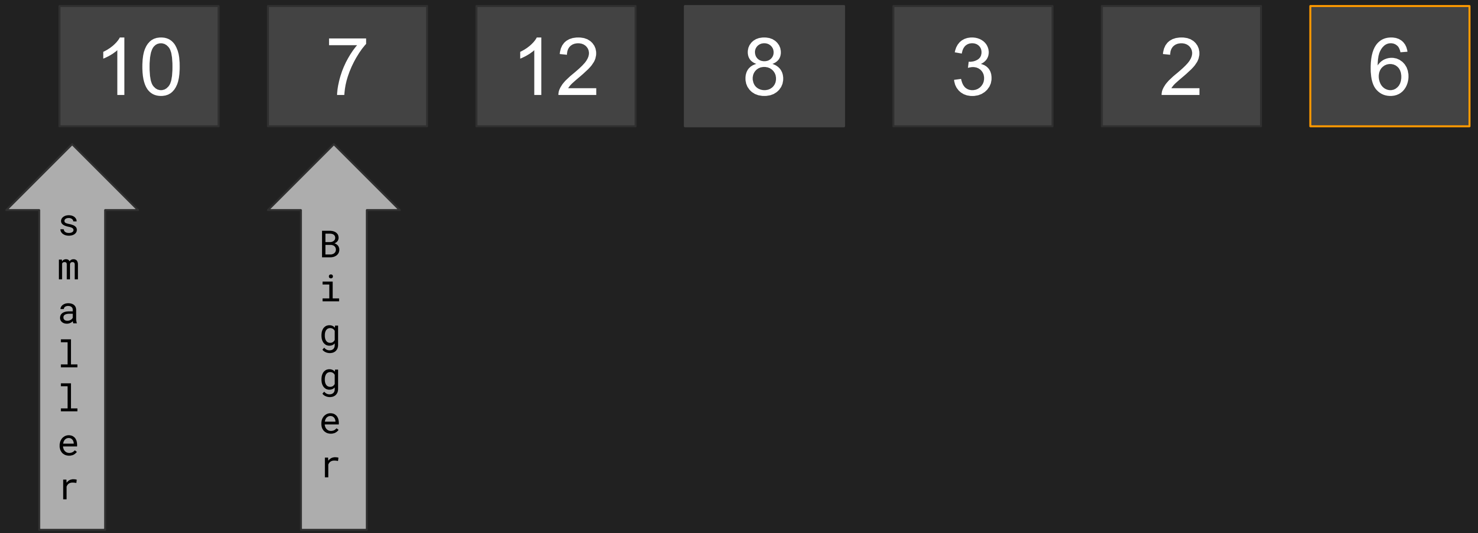
10	7	12	8	3	2	6

Smaller

Bigger

Quicksort round #2

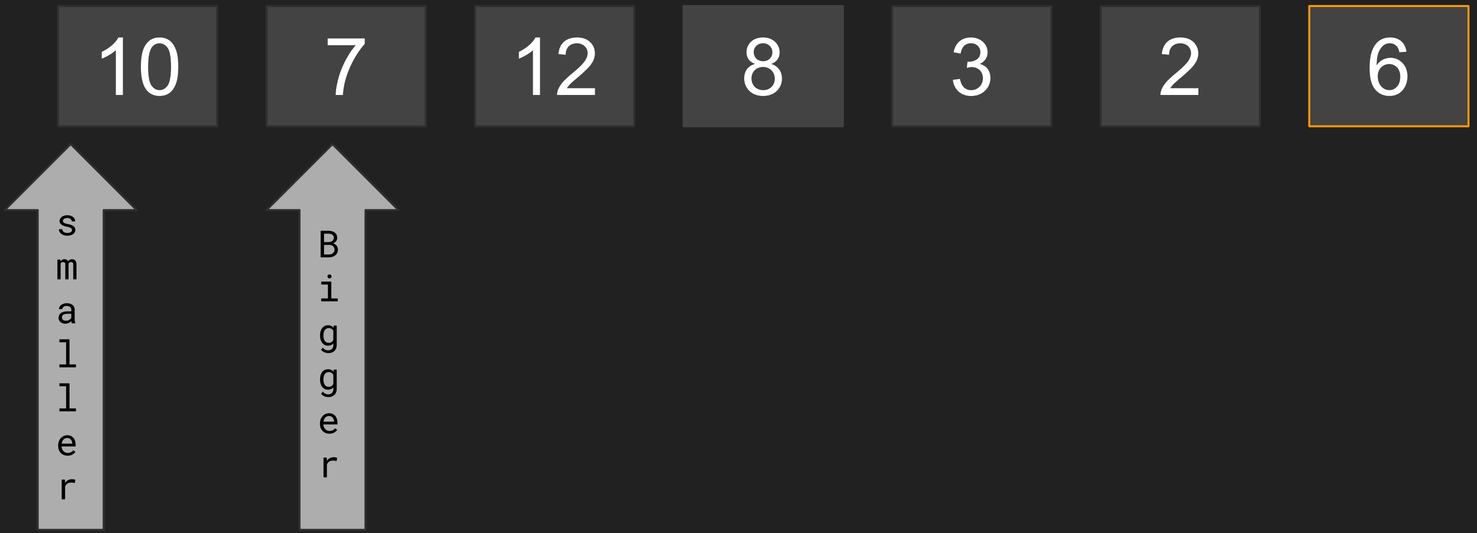
Now because 10 is bigger than 6, we will move exactly one of the counters over.



10	7	12	8	3	2	6

7 > 6, so we move counter again.

Quicksort round #2



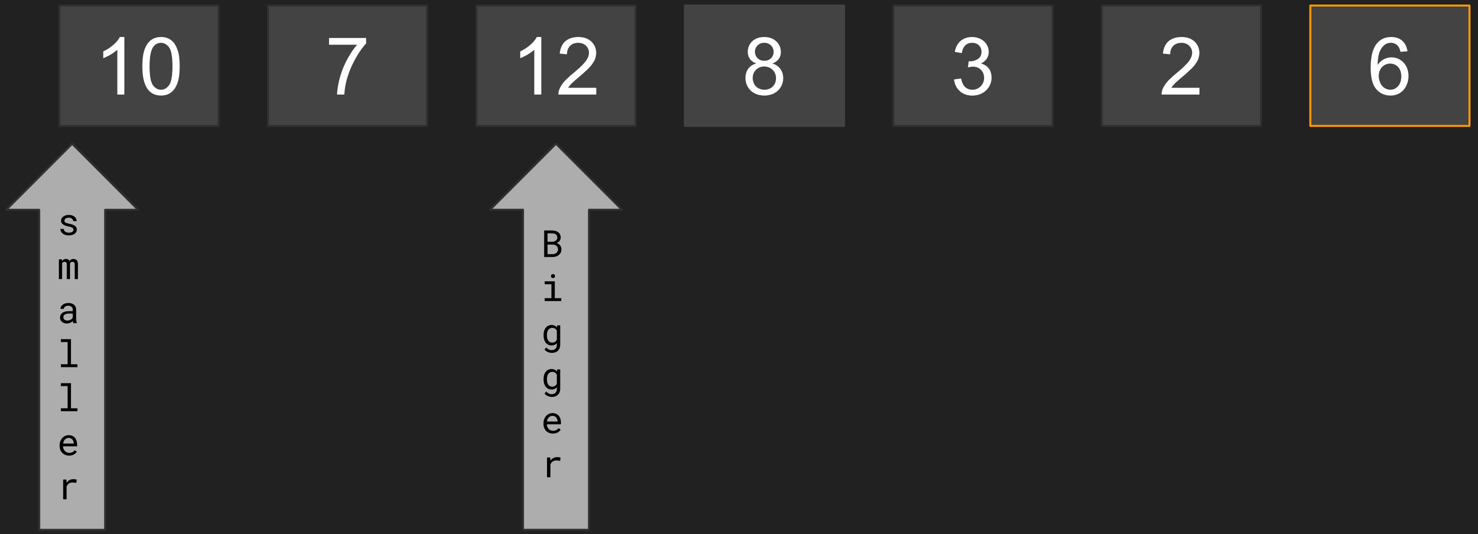
10	7	12	8	3	2	6

smaller

Bigger

7 > 6, so we move counter again.

Quicksort round #2



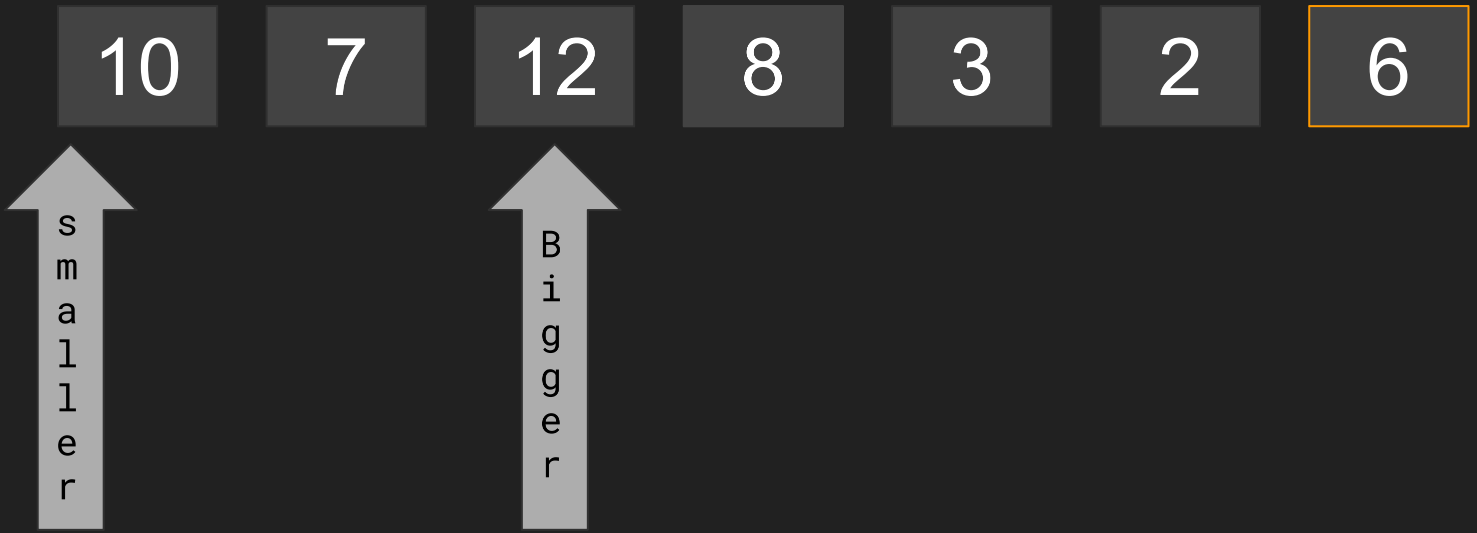
10	7	12	8	3	2	6

Smaller

Bigger

12 > 6, so we move counter again.

Quicksort round #2



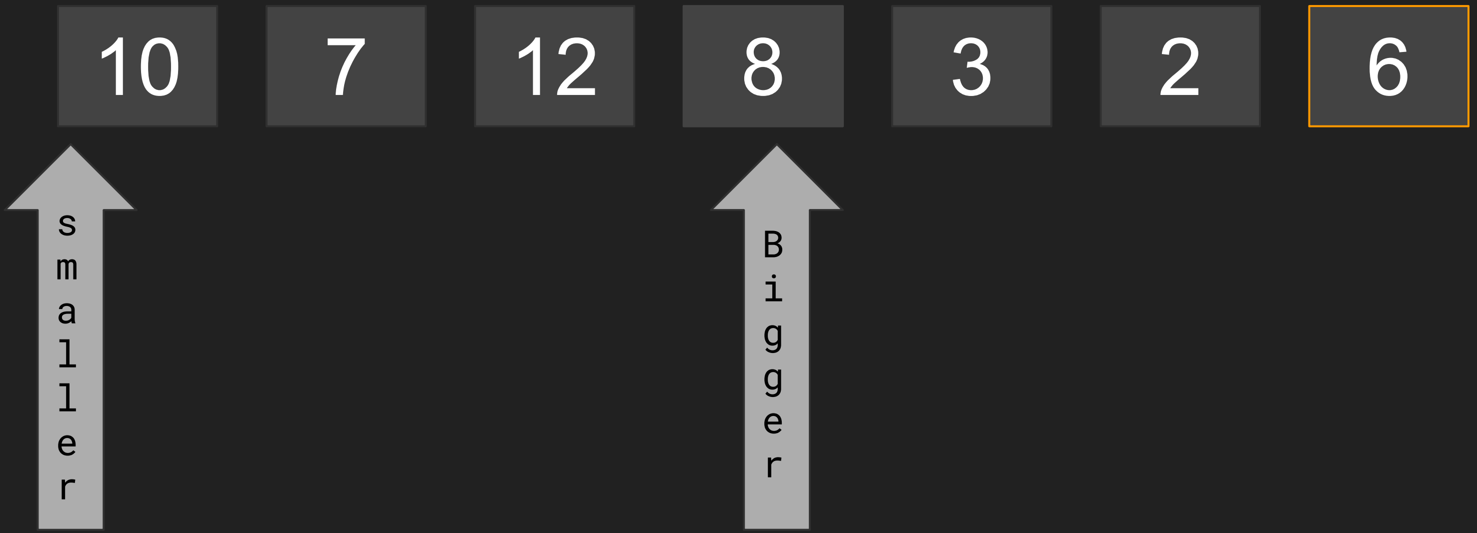
10 7 12 8 3 2 6

smaller

Bigger

12 > 6, so we move counter again.

Quicksort round #2



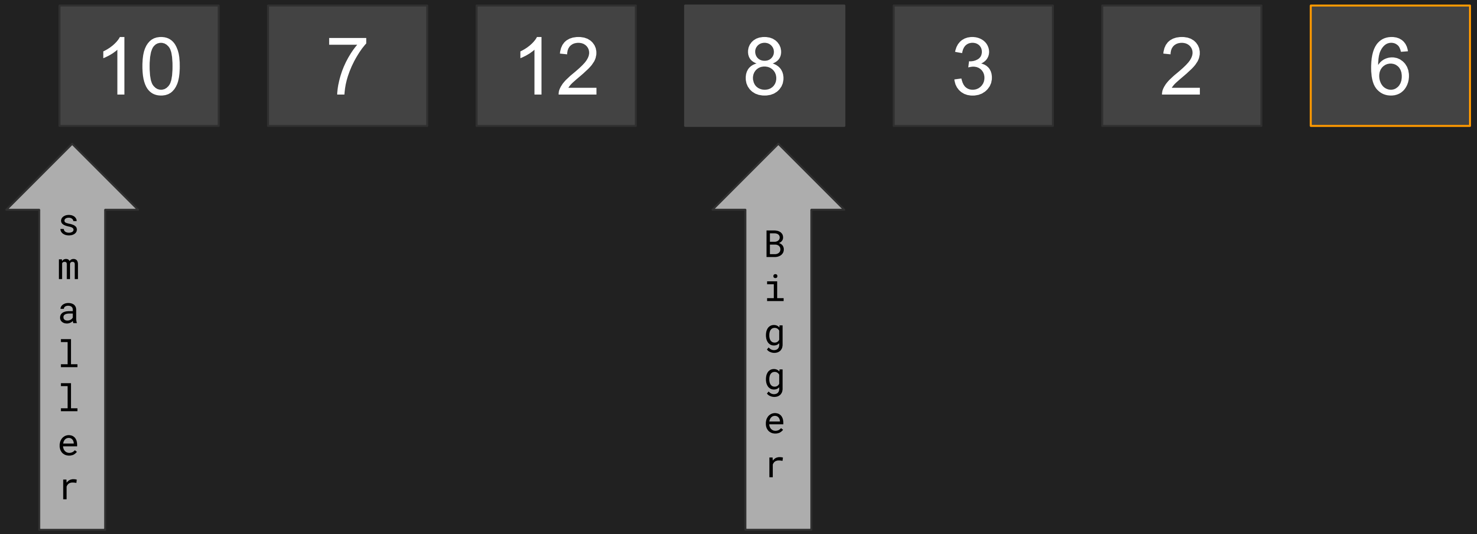
10	7	12	8	3	2	6

smaller

Bigger

8 > 6, so we move counter again.

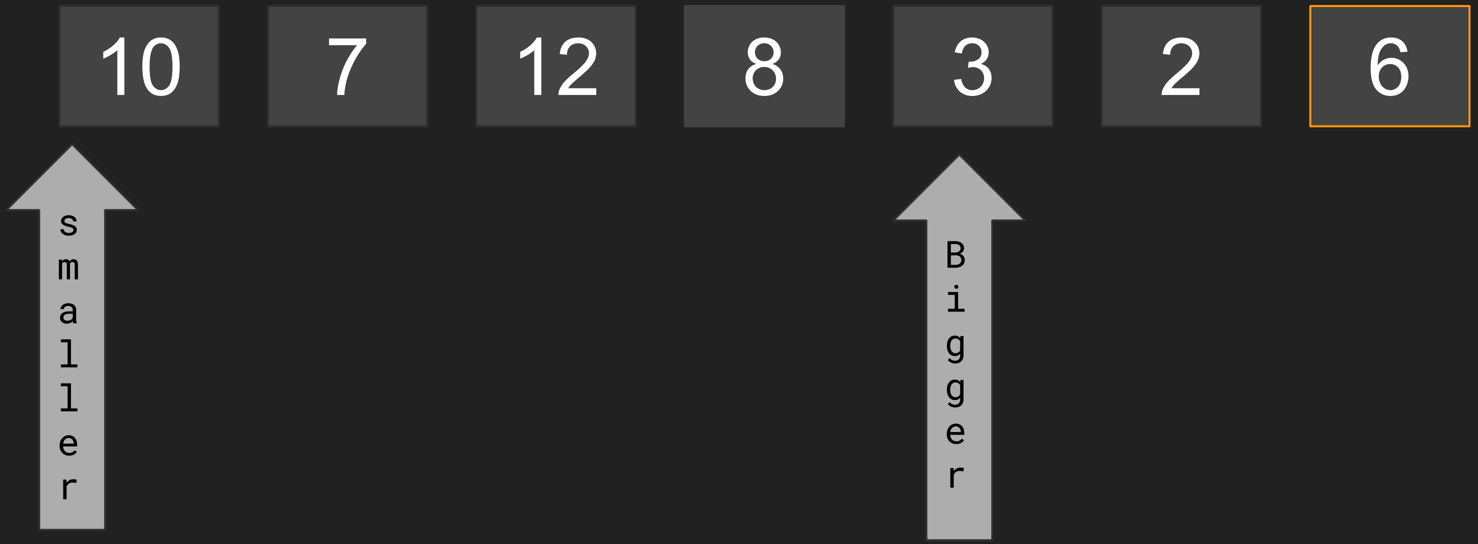
Quicksort round #2



10	7	12	8	3	2	6

8 > 6, so we move counter again.

Quicksort round #2



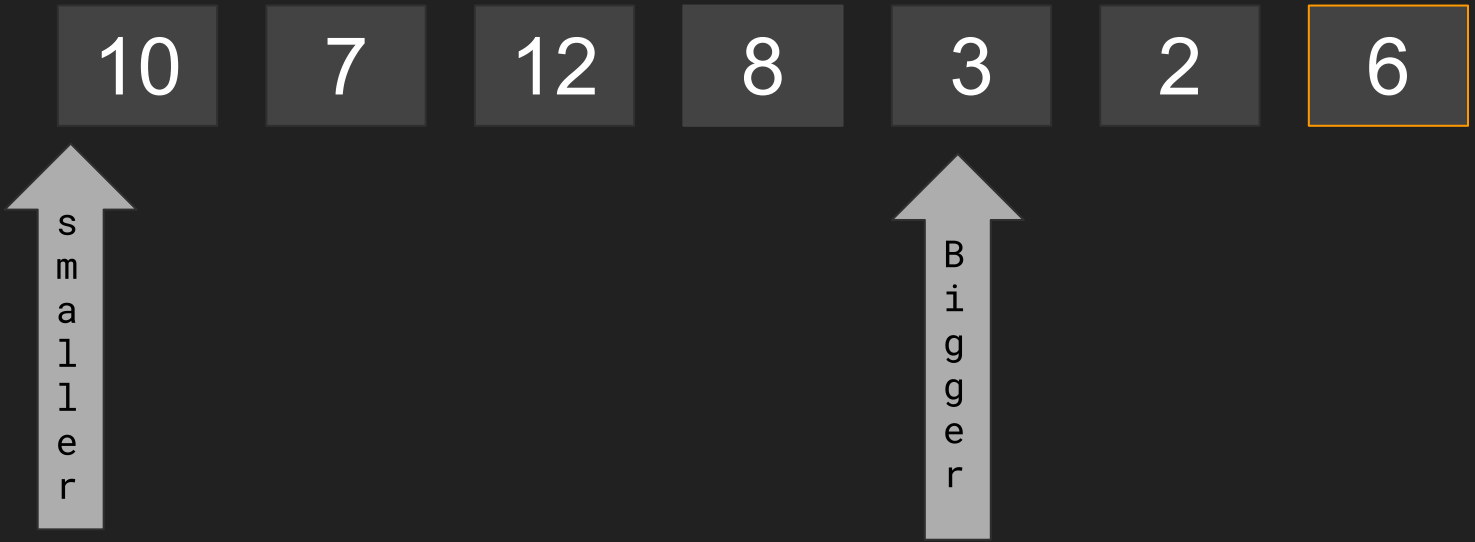
10 7 12 8 3 2 6

smaller

Bigger

Quicksort round #2

6 < 3, so we are going to swap ‘3’ with our ‘smaller counter



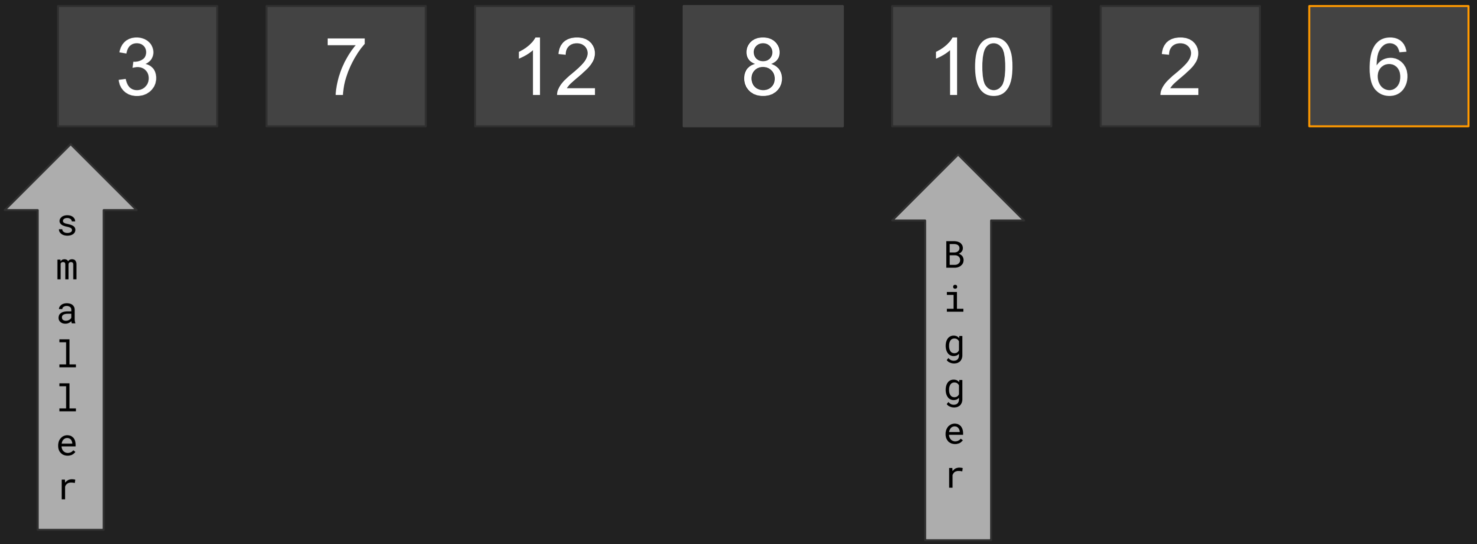
10 7 12 8 3 2 6

smaller

Bigger

Quicksort round #2

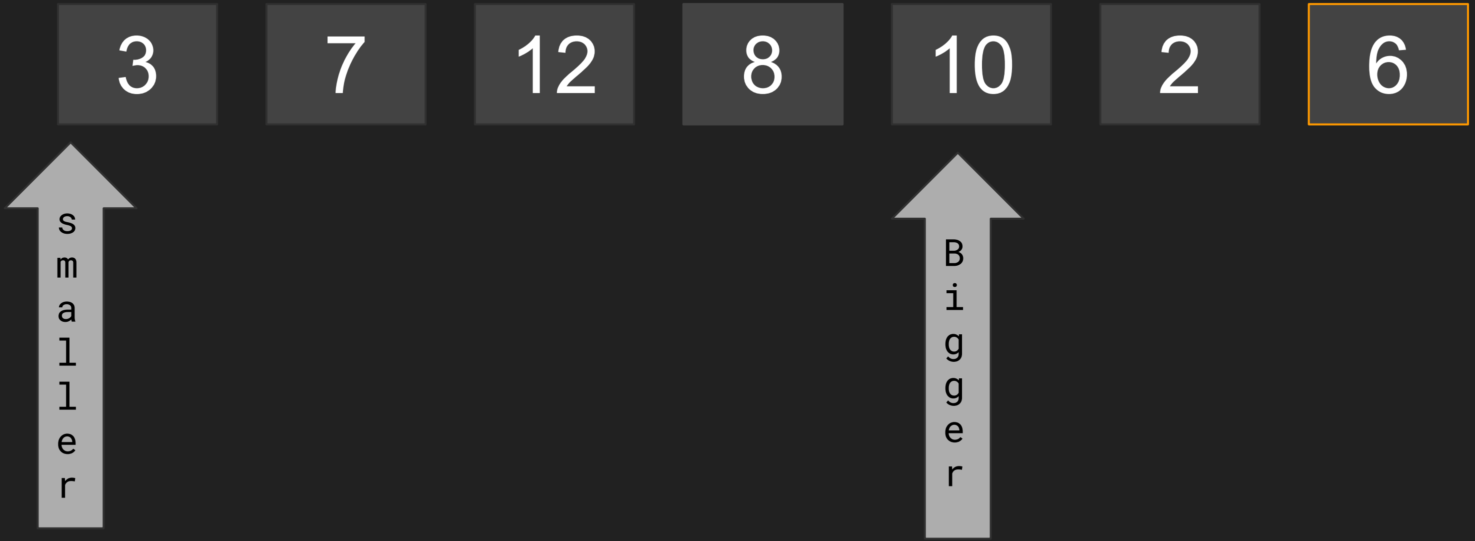
6 < 3, so we are going to swap ‘3’ with our ‘smaller counter



3	7	12	8	10	2	6

Now increment the ‘smaller counter’ once more

Quicksort round #2



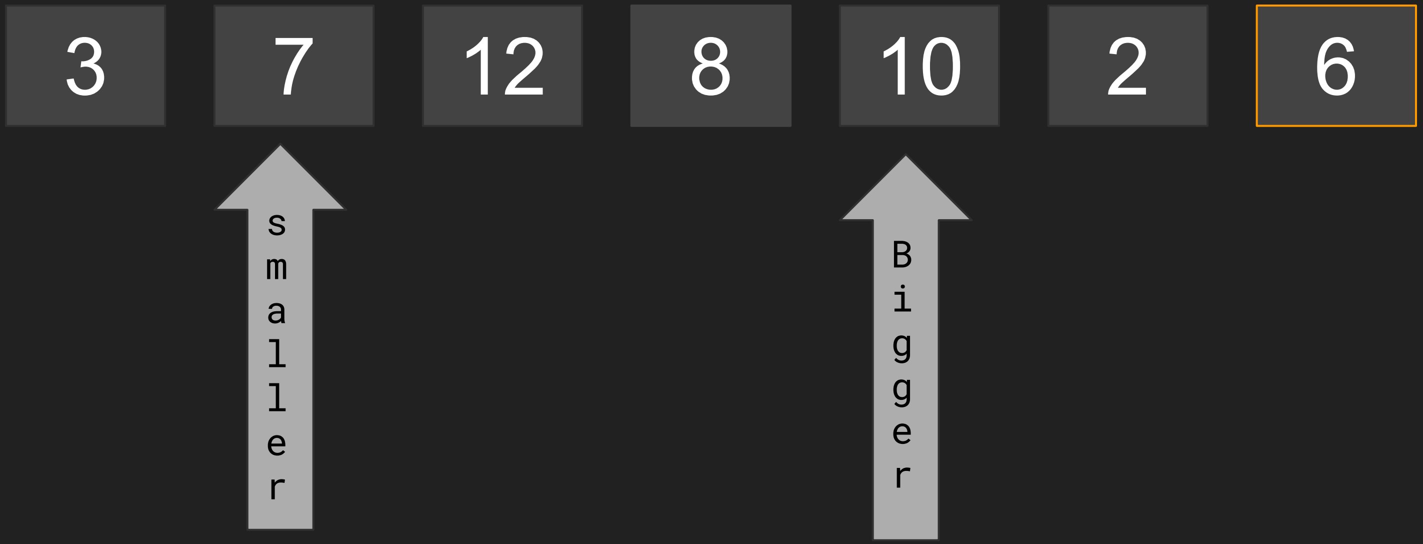
3	7	12	8	10	2	6

Smaller

Bigger

Now increment the ‘smaller counter’ once more

Quicksort round #2



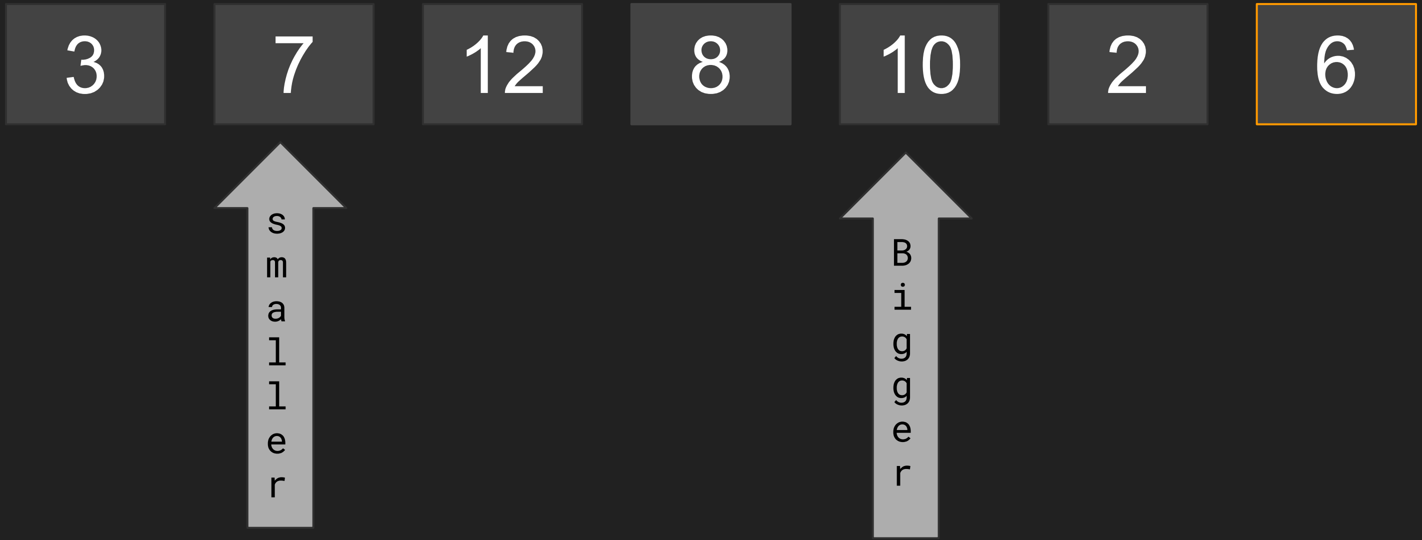
3	7	12	8	10	2	6

smaller

Bigger

Continue our algorithm, moving our ‘bigger’ counter to compare the next item

Quicksort round #2



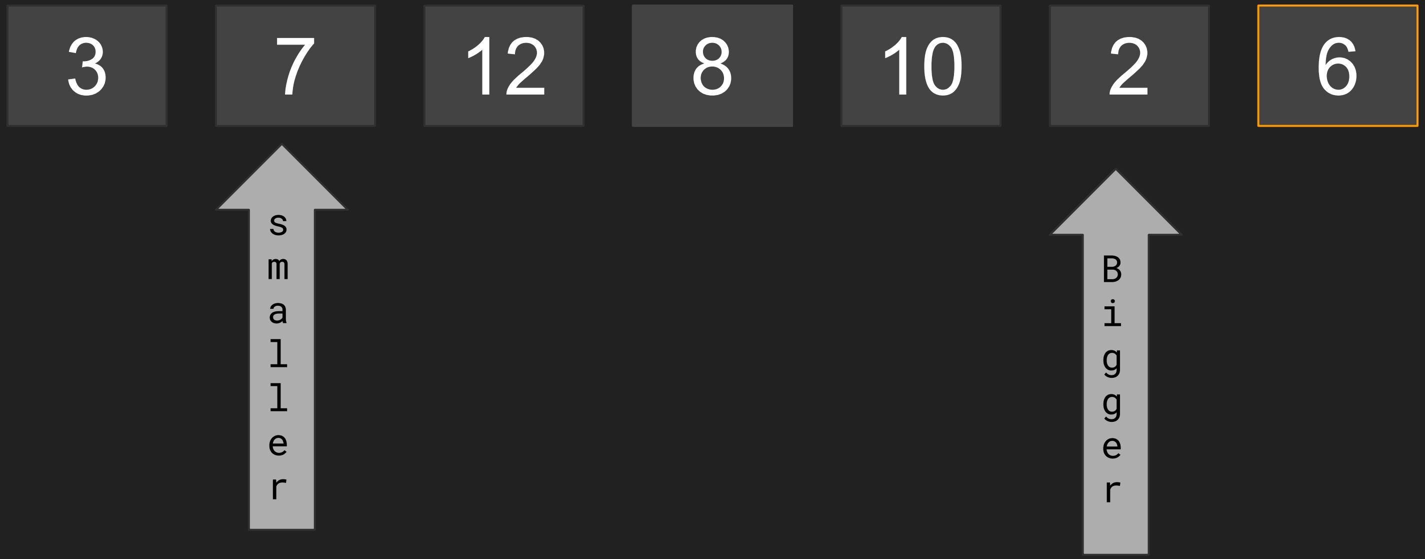
3 7 12 8 10 2 6

smaller

Bigger

2 < 6, so again do the swap

Quicksort round #2



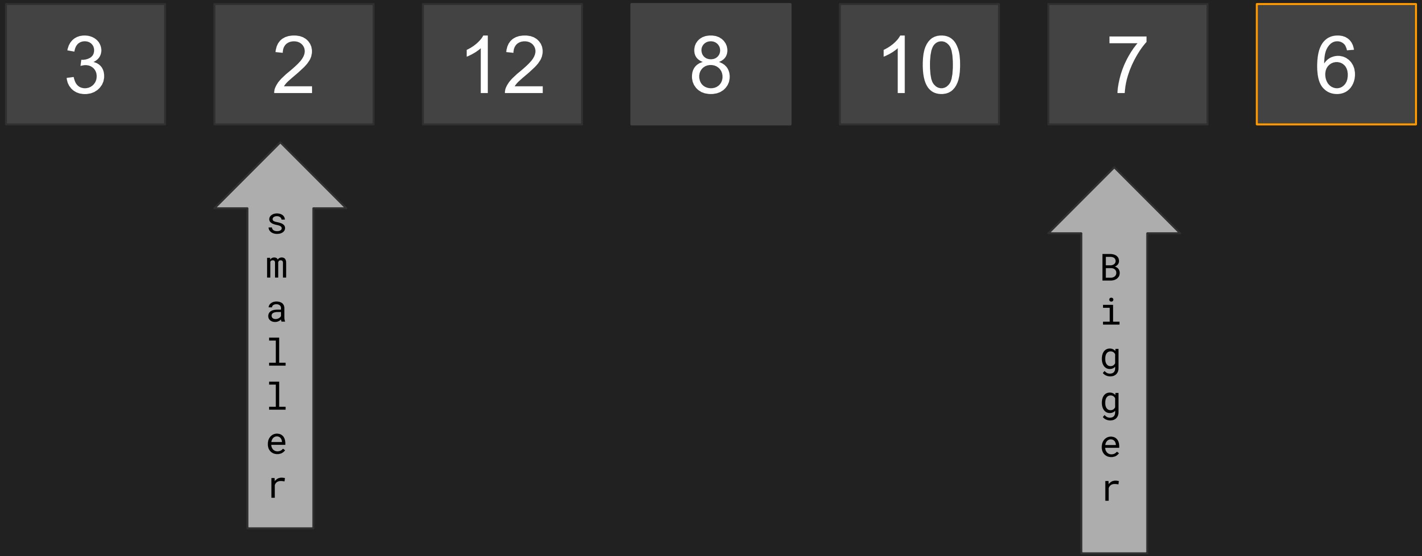
3	7	12	8	10	2	6

smaller

Bigger

2 and 7 swap

Quicksort round #2



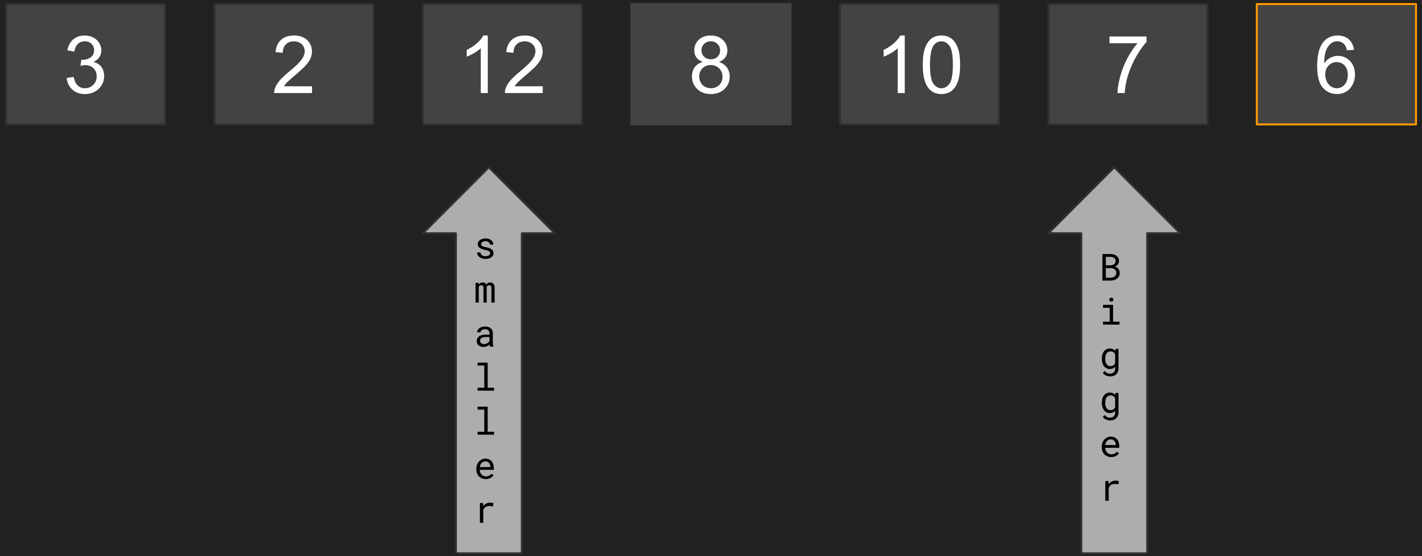
3	2	12	8	10	7	6

Smaller

Bigger

Increment our ‘smaller’ counter

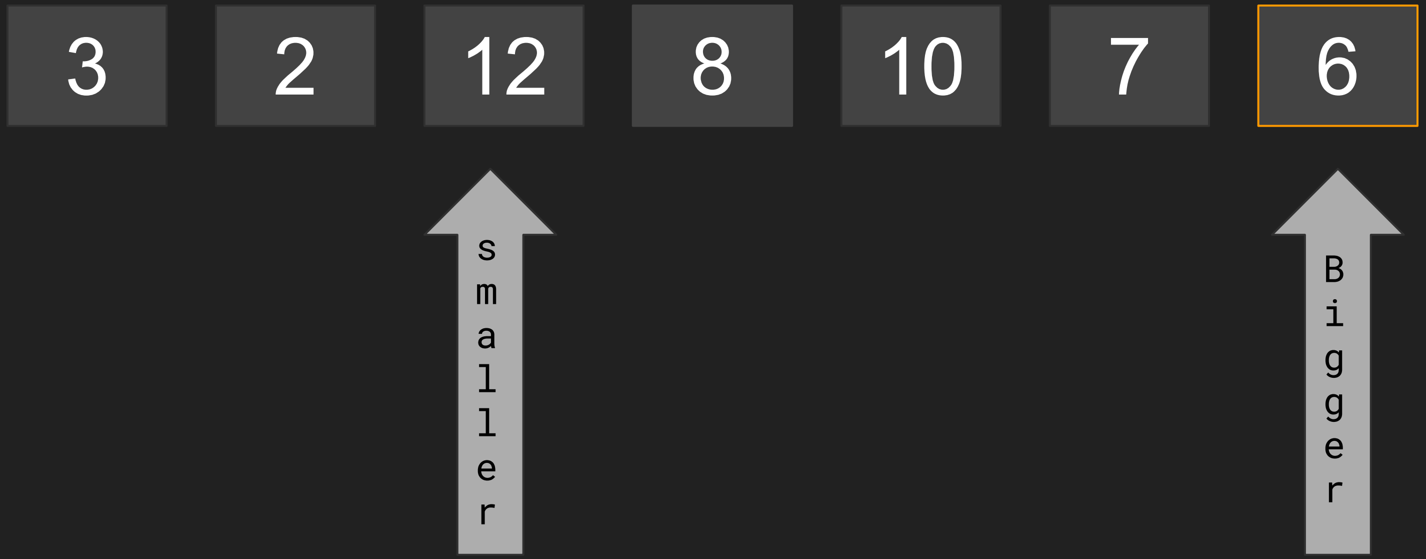
Quicksort round #2



3	2	12	8	10	7	6

Increment our ‘bigger’ counter, and we are at the end

Quicksort round #2



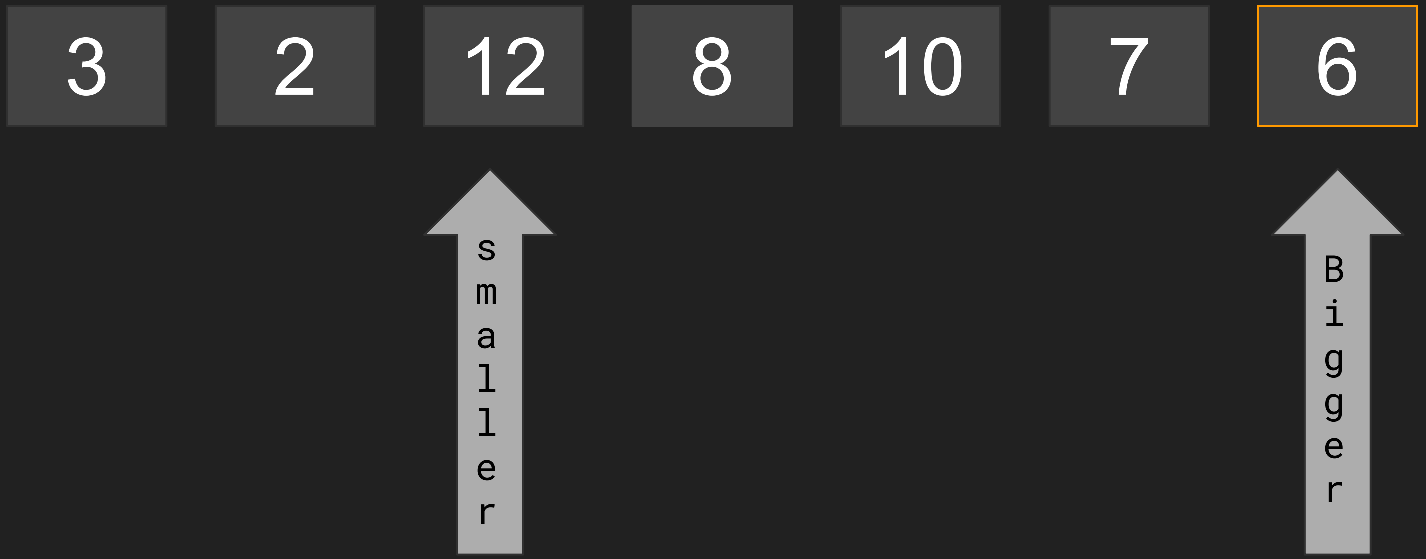
3	2	12	8	10	7	6

smaller

Bigger

Now we swap our pivot point with where our ‘smaller’ counter is

Quicksort round #2



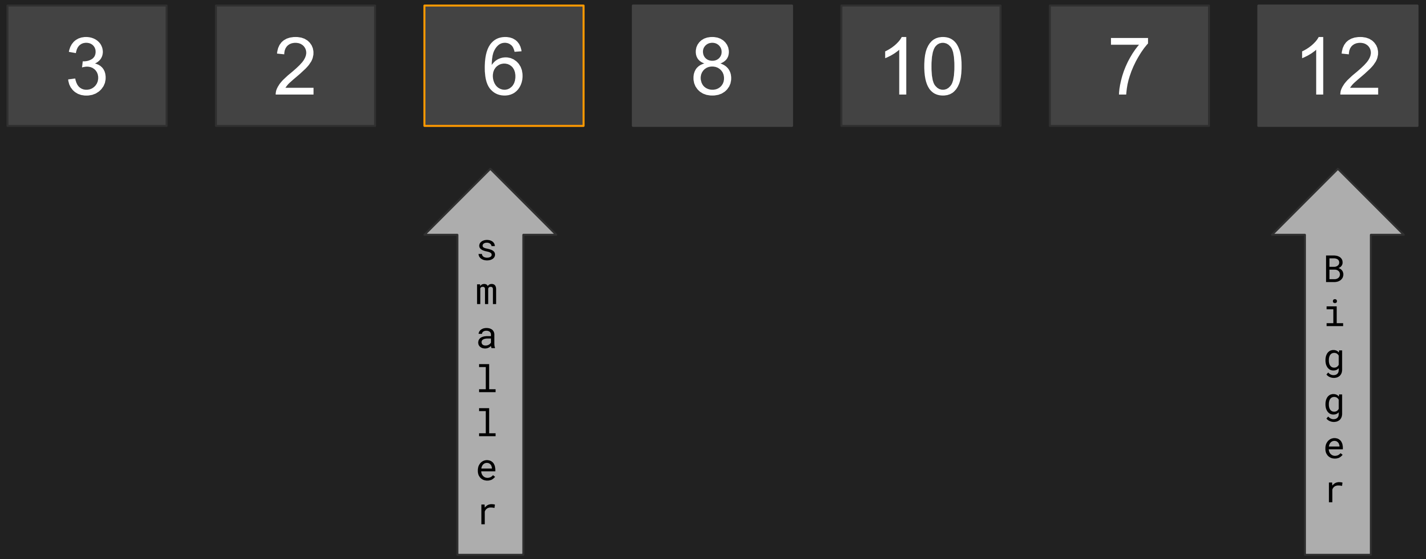
3	2	12	8	10	7	6

smaller

Bigger

6 and 12 swapped

Quicksort round #2



3 2 6 8 10 7 12

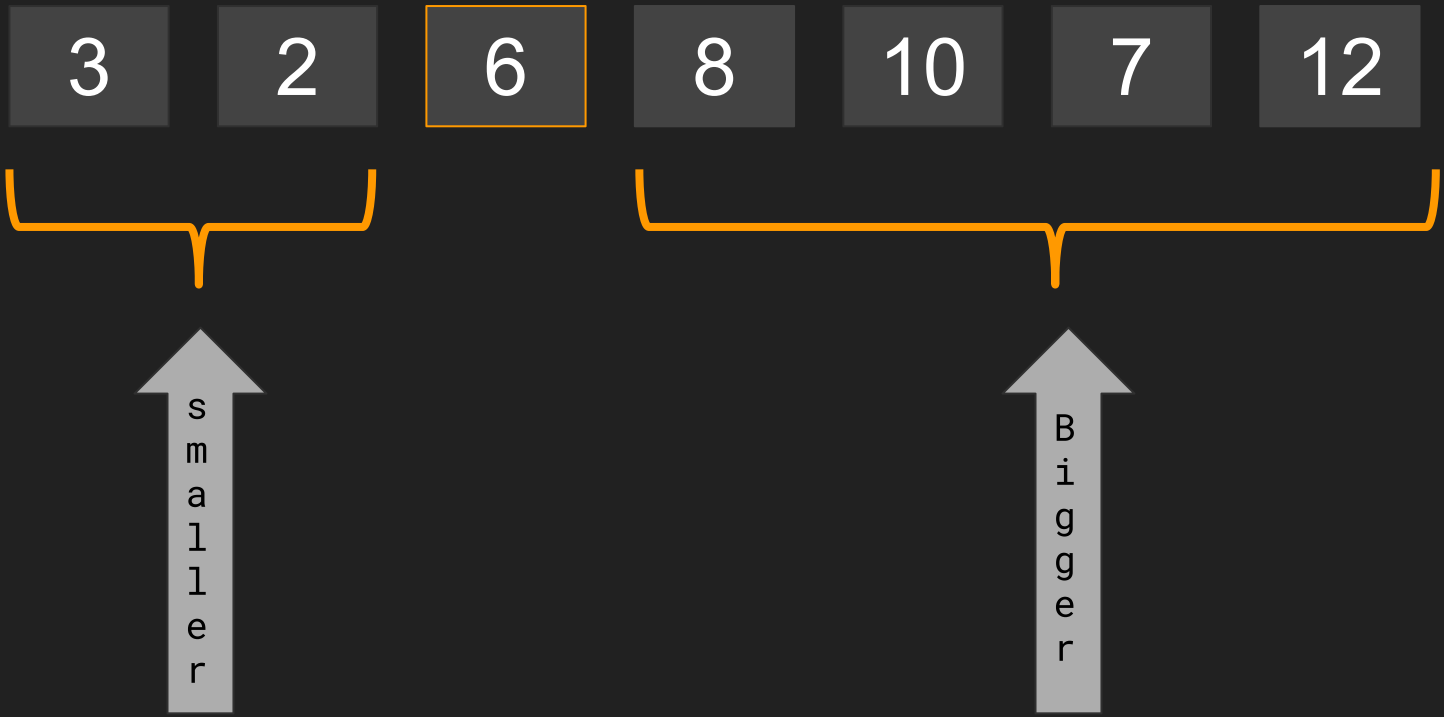
smaller

Bigger

Quicksort round #2

Observe everything to the left of 6 is smaller, and everything to the right is bigger.

Now we can repeat for sublists appropriately.



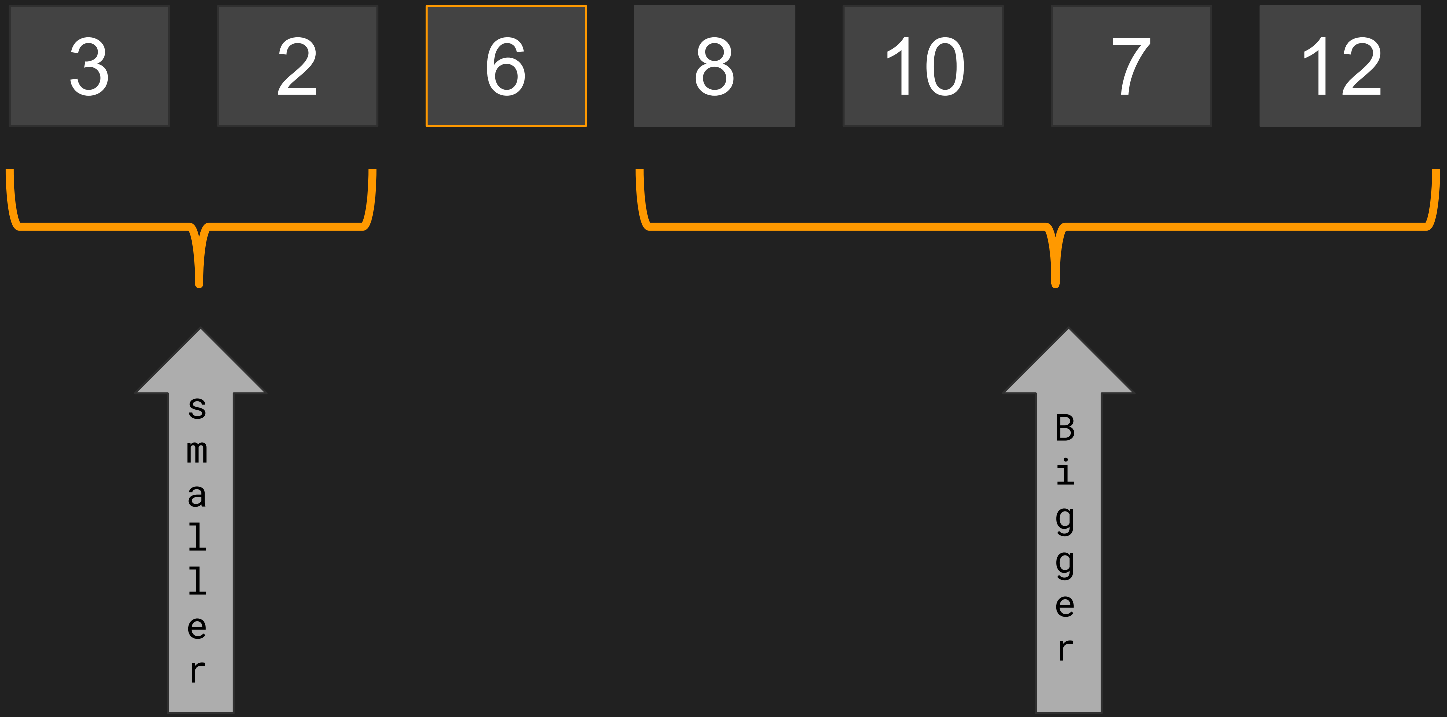
3	2	6	8	10	7	12

smaller

Bigger

Quicksort round #2

One thing that is neat about this implementation, is that it is an ‘in-place’ sort. Meaning we do not need to allocate extra space like with merge sort. We do things in-place!



3	2	6	8	10	7	12

smaller

Bigger

Quicksort in practice

- The expected time (average case, or “*theta*”) is $\Theta(N*Log_2(N))$
  - Note this is our first view of average-case analysis
- Quicksort is in the worst-case actually $O(N^2)$ algorithm
- We’ll need to do a little more formal analysis to confirm with the recurrence.

Worst-case quicksort

(Bad instance of) Quicksort recurrence (1/3)

- The insight is that the partition step could be bad
  - i.e. we get a very unbalanced partition, where we can only sort ‘1’ item at a time.
  - Partitioning itself takes ‘N’ time, that is the latter term
    - We iterate through each object
  - The recurrence could be (n-1) if we do a bad job partitioning
    - (i.e. nothing moves left or right of the pivot)

$$T(n) = T(n - 1) + \Theta(n)$$

(Bad instance of) Quicksort recurrence (2/3)

- The insight is that the partition step could be bad
    - i.e. we get a very unbalanced partition, where we can only sort ‘1’ item at a time.
    - Partitioning itself takes ‘N’ time, that is the latter term
        - We iterate through each object
    - The recurrence could be (n-1) if we do a bad job partitioning
        - (i.e. nothing moves left or right of the pivot)

$$T(n) = T(n - 1) + \Theta(n)$$

The recurrence is what we are recursing on

i.e. how much work we are ‘chopping off’ or dividing up each time through the algorithm.

(Bad instance of) Quicksort recurrence (3/3)

- Thus solving the recurrence is the following form
  - For k = 1 to n
  - We are doing at least (‘k’) work in our partition.
  - If we have to partition, ‘N’ times, because we keep selecting bad partitions, then we get O(N*N) behavior.

$$T(n) = T(n - 1) + \Theta(n)$$

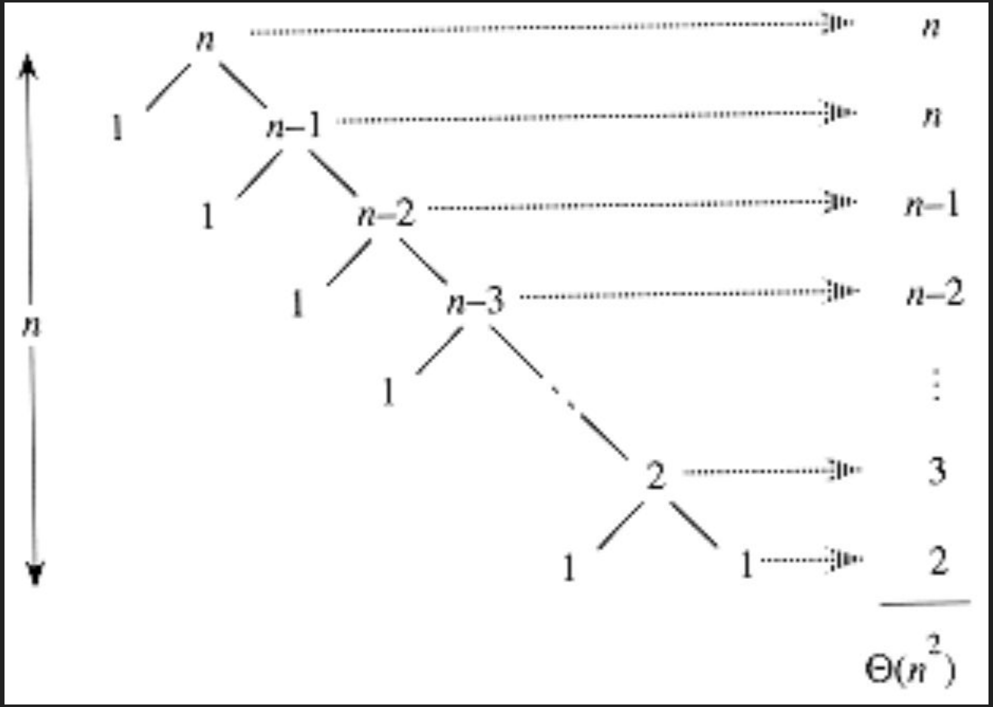
$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) \\ &= \sum_{k=1}^{n} \Theta(k) \\ &= \Theta\left(\sum_{k=1}^{n} k\right) \\ &= \Theta(n^2) \, . \end{aligned}$$

Reminder!

$$\sum_{i=1}^{n} k = \frac{n(n+1)}{2}$$

Recurrence Tree for a (Bad instance of) quick sort

- How many times we have to call ‘partition’ is shown on the left
- You can observer we are only partitioning off ‘1’ thing at a time
  - (i.e. our pivot)
- Thus we are not breaking our work in ‘half’ like in merge sort.

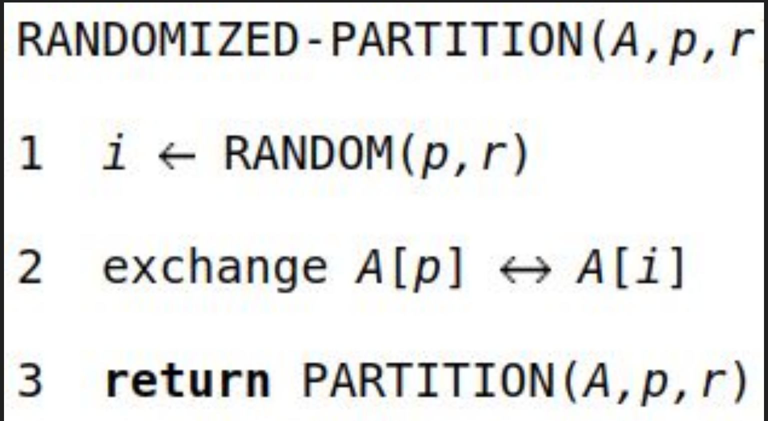


How do we do better?

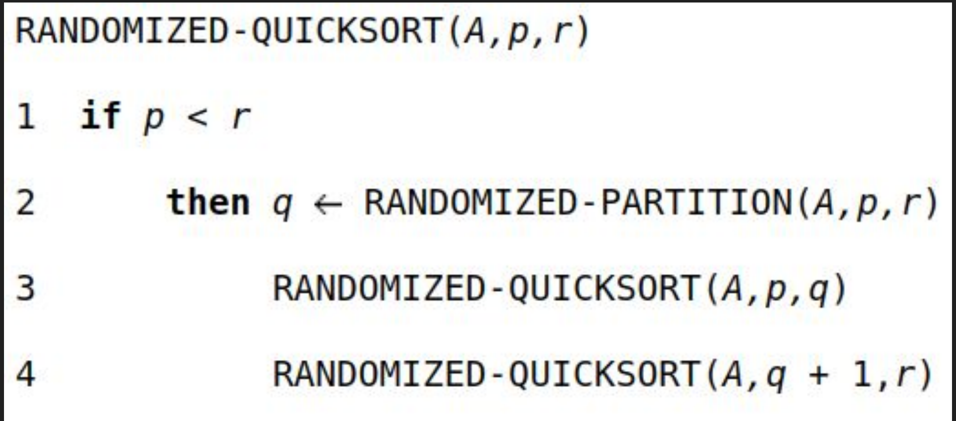
- There is another trick which puts quicksort in the ‘randomized algorithm’ category of algorithms

Randomized quicksort

- Just a small change to select the ‘pivot’ based on a random index.
    - This randomly generated pivot will ensure we are not as likely to keep selecting a bad pivot
        - The best-case, worst-case, and average-case overall stay the same--but this will do slightly better regardless of the ‘sortedness’ of our input.
        - (For either the partitions, or the whole input list)
    - (e.g. we try to sort an already sorted array with say the last index as a ‘fixed pivot’ every iteration)



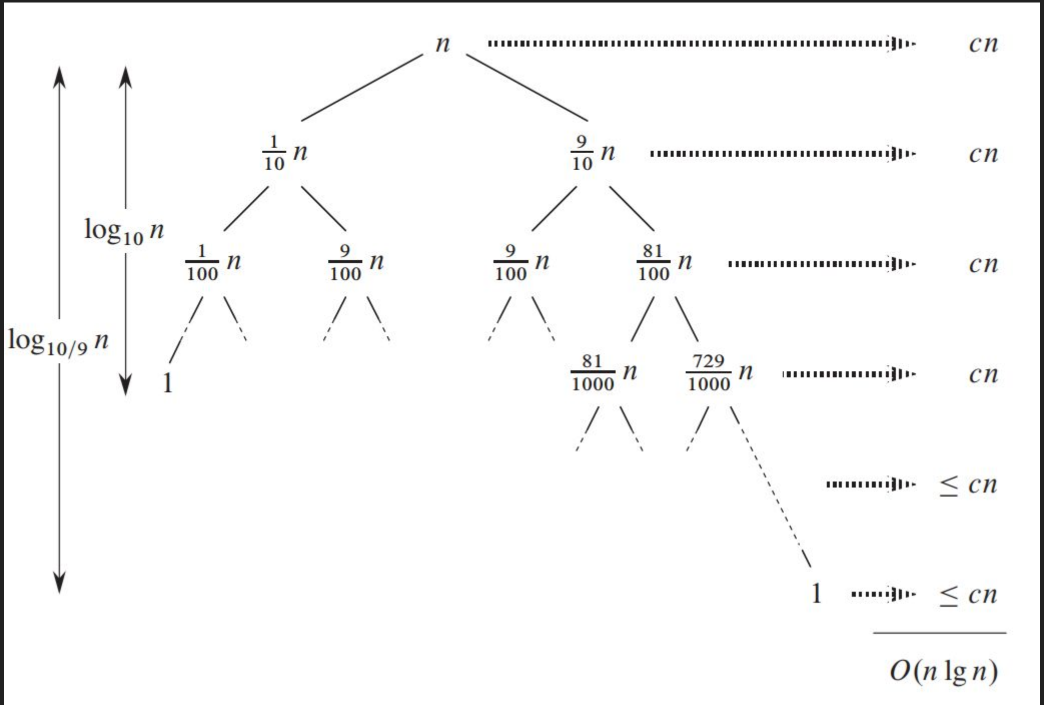
```
RANDOMIZED-PARTITION($A,p,r$
    $i \leftarrow$ RANDOM($p,r$)
    exchange $A[p] \leftrightarrow A[i]$
    return PARTITION($A,p,r$)
```



```
RANDOMIZED-QUICKSORT($A,p,r$)
    if $p < r$
        then $q \leftarrow$ RANDOMIZED-PARTITION($A,p,r$)
            RANDOMIZED-QUICKSORT($A,p,q$)
            RANDOMIZED-QUICKSORT($A,q+1,r$)
```

Even if we get a slightly bad pivot with our randomized strategy

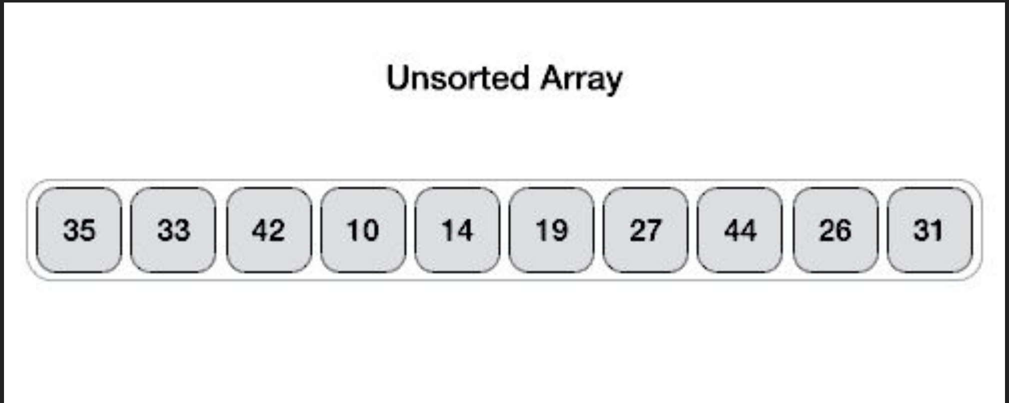
- However, even if we only partition off say 1/10th of our items
  - quicksort is still log behavior
- quicksort in the ‘randomized algorithm’ category of algorithms and thus runs on the average case N*Log(N)



In Summary

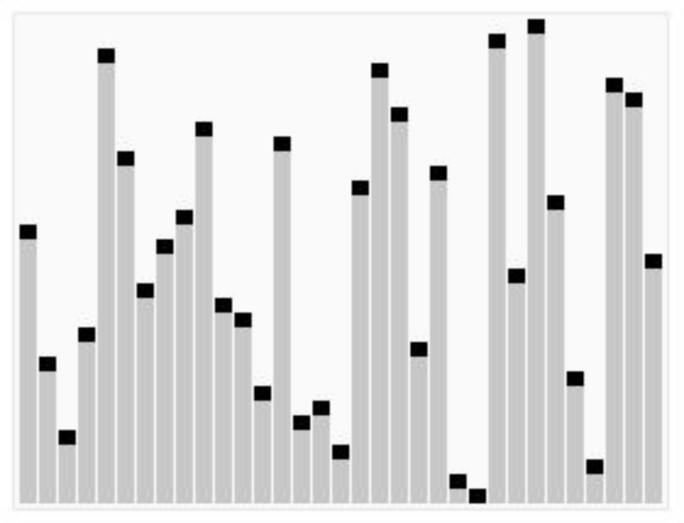
- When to use randomized strategies?
  - In the case of quicksort, when it is hard to make any guarantees, randomized algorithms may be useful.
- Overall, a company like Google working with unsorted data likely uses some variant of quicksort
  - (perhaps highly parallelized)
- If we know the data is mostly sorted--we probably avoid quicksort
  - (And maybe even use insertion sort!)

Quick Sort Visuals



Unsorted Array

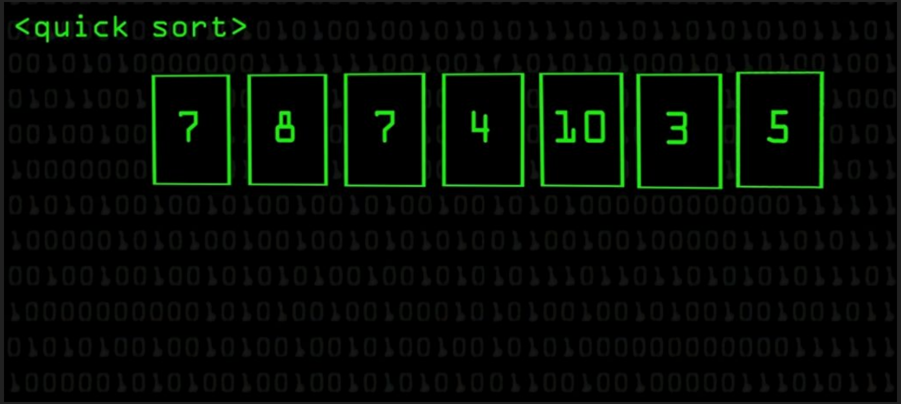
35 33 42 10 14 19 27 44 26 31



Computer Systems Feed

YOUR DAILY FEED

- (An article/image/video/thought injected in each class!)
- https://www.youtube.com/watch?v=XE4VP_8Y0BU



Algorithm, Data Structure, and Proof Toolboxes

For this course, I want you to be able to see how each data structure and algorithm is different.

- For data structures learn how each restriction on how we organize our data causes tradeoffs
- For algorithms, think about the higher level technique

Algorithm Toolbox: Bubble, Selection, Insertion Sort, Linear Search, Binary Search

Comparison Sorts

Bubble Sort - $O(n^2)$

Swap adjacent elements and 'bubble' up element

Selection Sort - $O(n^2)$

Search for minimum element and place in ordered position amongst unordered elements

Insertion Sort - $O(n^2)$

Select each element and place in its sorted position amongst all elements that have been previously placed

Divide and Conquer Sorts

Merge Sort- $O(n*\log_2(n))$

Randomized Algorithms

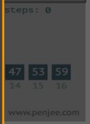
QuickSort- $\Theta(n*\log_2(n))$ ("theta")

$O(n^2)$ - in the worst case

Searches

Linear Search - $O(n)$

ment



$_2(n)$

Binary search
steps: 0
37
1 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
low
mid
high

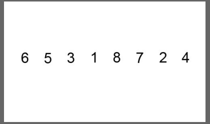
Search sorted data from midpoint, eliminate values less than or greater than 'element' you are search for each step until we match the mid

Two new sorting algorithms to add!

Algorithm Toolbox: Bubble, Selection, Insertion Sort, Linear Search, Binary Search

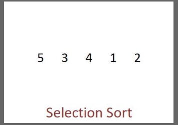
Comparison Sorts

Bubble Sort - $O(n^2)$



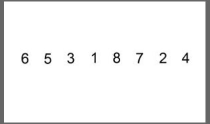
Swap adjacent elements and ‘bubble’ up element

Selection Sort - $O(n^2)$



Search for minimum element and place in ordered position amongst unordered elements

Insertion Sort - $O(n^2)$



Select each element and place in its sorted position amongst all elements that have been previously placed

Divide and Conquer Sorts

Merge Sort- $O(n*\log_2(n))$

Randomized Algorithms

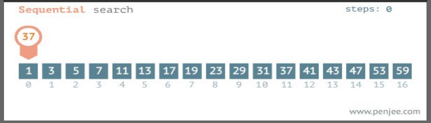
QuickSort- $\Theta(n*\log_2(n))$ (“theta”)

$O(n^2)$ - in the worst case

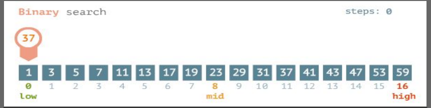
Searches

Linear Search - $O(n)$

Search and compare each element one at a time



Binary Search - $\log_2(n)$



Search sorted data from midpoint, eliminate values less than or greater than ‘element’ you are search for each step until we match the mid

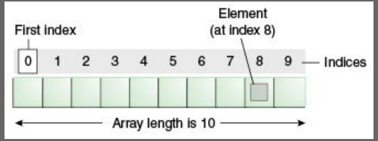
And many more! https://en.wikipedia.org/wiki/List_of_data_structures

Data Structure Toolbox: arrays, linked lists, doubly linked list, queues, stacks, maps

Associative Containers

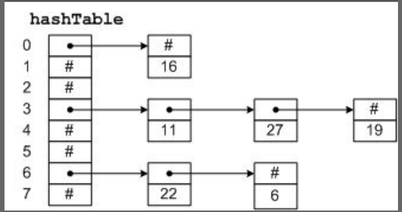
Arrays -

A contiguous block of memory, random access O(1)



Hashmap (chained implementation) -

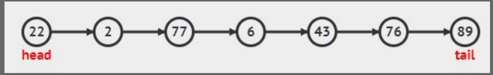
Associative Data Structure with key/value pairs and a ‘hash function’



Sequence Containers

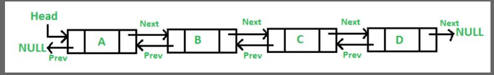
Linked Lists -

A ‘chain’ of nodes, can traverse in one direction



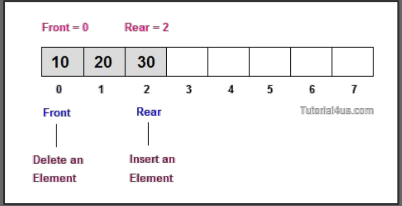
Doubly Linked Lists -

A ‘chain’ of nodes, can traverse in both directions



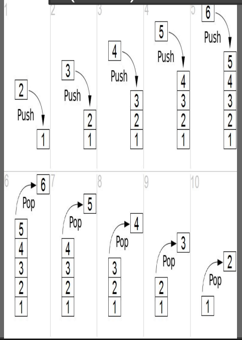
Queues -

A First in, First out data structure (FIFO)



Stacks -

A Last in, last out data structure (LIFO)



Proof Toolbox

- Our tools so far!

Notation

```
∀n - “for all”
| - “such that”
n ∈ ℤ - “n is an element of the integers”
```

Proof Techniques

- Proof by Case
  - Enumerate or test all possible inputs
- Proof by Induction
  - Show that two cases hold
- Proof by Invariant
  - Step through 4 steps of algorithm
- Big-O Analysis
  - Prove run-time complexity

- Recurrence
  - Can be solved with Subsitution Method
- Recurrence Tree
  - “A Visual Proof” (Somewhat informal)
- Master Thereom
  - Proven by definition
- Subsitution Method
  - (Works for any recurrence)

Building Blocks

Definition - Something given, we can assume is true

e.g. let x = 7

Proposition - A true or false statement

e.g.
1+7 = 7 FALSE
2+7 = 9 TRUE

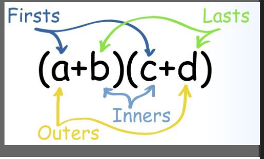
Predicate - A proposition whose truth depends on its input. It is a function that returns true or false.

“P(n) ::= “n is a perfect square”
P(4) thus is true, because 4 is a perfect square
P(3) is false, because it is not a perfect square.

Math Toolbox

Mathematics

Multiplication



$$(a+b)(c+d) = ac + ad + bc + bd$$

Notation

Pi Production Notation

$$n! = \prod_{i=1}^{n} i.$$

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-2) \cdot (n-1) \cdot n$$

Big-O: O(n) - “Worse Case Analysis or upper bound”
Big-Theta: Θ - “Average Case Analysis”
Big-Omega Ω - “Best Case or lower bound”

Factorial (!) $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$

Logs

Logs -

$$\log_a n = \frac{\log_b n}{\log_b a}$$

Usually we work in log base 2, i.e. $\log_2(n)$. The change of base formula is given below

This lecture in summary

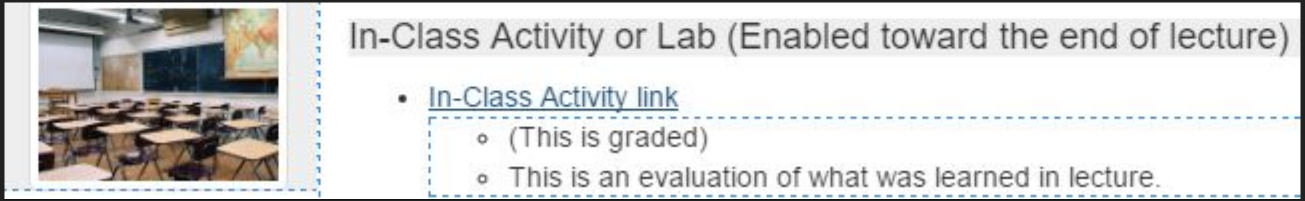
- We have explored
  - Recursion in C
  - Recursion in algorithms pseudo-code
  - Mergesort
  - Quicksort

A Couple of Half-Truths (The Systems side of me must tell you...)

- What is missing and you will have to discover in either CS 5800 or CS 5600
  - How concurrency effects Big-O
    - (Remember, our RAM model of computation is simple!)
  - In practice implementations of algorithms like qsort or Java’s sorting algorithm may actually leverage multiple sorting algorithms for different input sizes or data sizes.

In-Class Activity

1. Complete the in-class activity from the schedule
   a. (Do this during class, not before :))
2. Please take 2-5 minutes to do so
3. These make up a total of 5% of your grade
   a. We will review the answers shortly



In-Class Activity or Lab (Enabled toward the end of lecture)

- In-Class Activity link
  - (This is graded)
  - This is an evaluation of what was learned in lecture.

Lab Time!