COMP 211-212

Fall 2014 and Spring 2015

Write code

Write beautiful code

Two notions of beauty

Structure: code communicates an idea

Efficiency: code instructs a computer

Two notions of beauty

Structure: code communicates an idea

logic : COMP 321, COMP 360-1

Efficiency: code instructs a computer

Two notions of beauty

Structure: code communicates an idea

logic : COMP 321, COMP 360-1

Efficiency: code instructs a computer

combinatorics: MATH 228, COMP 312, COMP 301

write code

write code

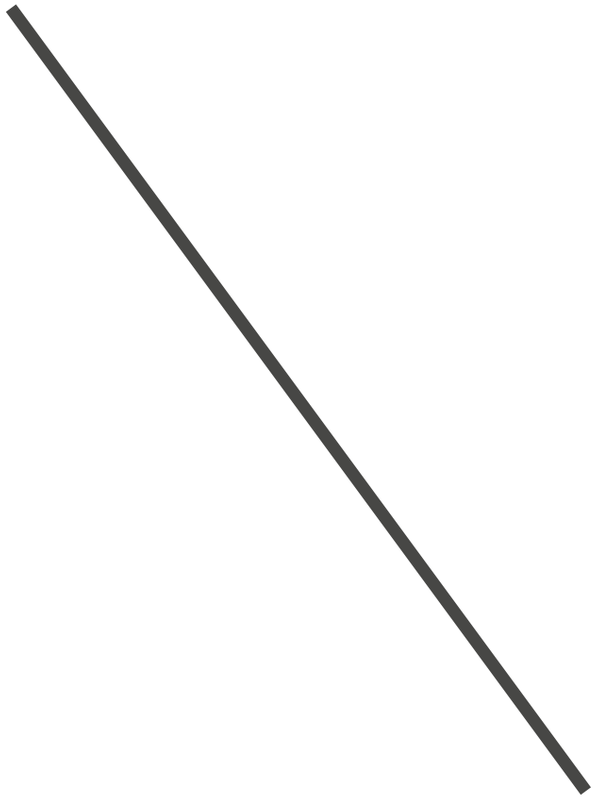
reason about
efficiency

write code

reason about behavior

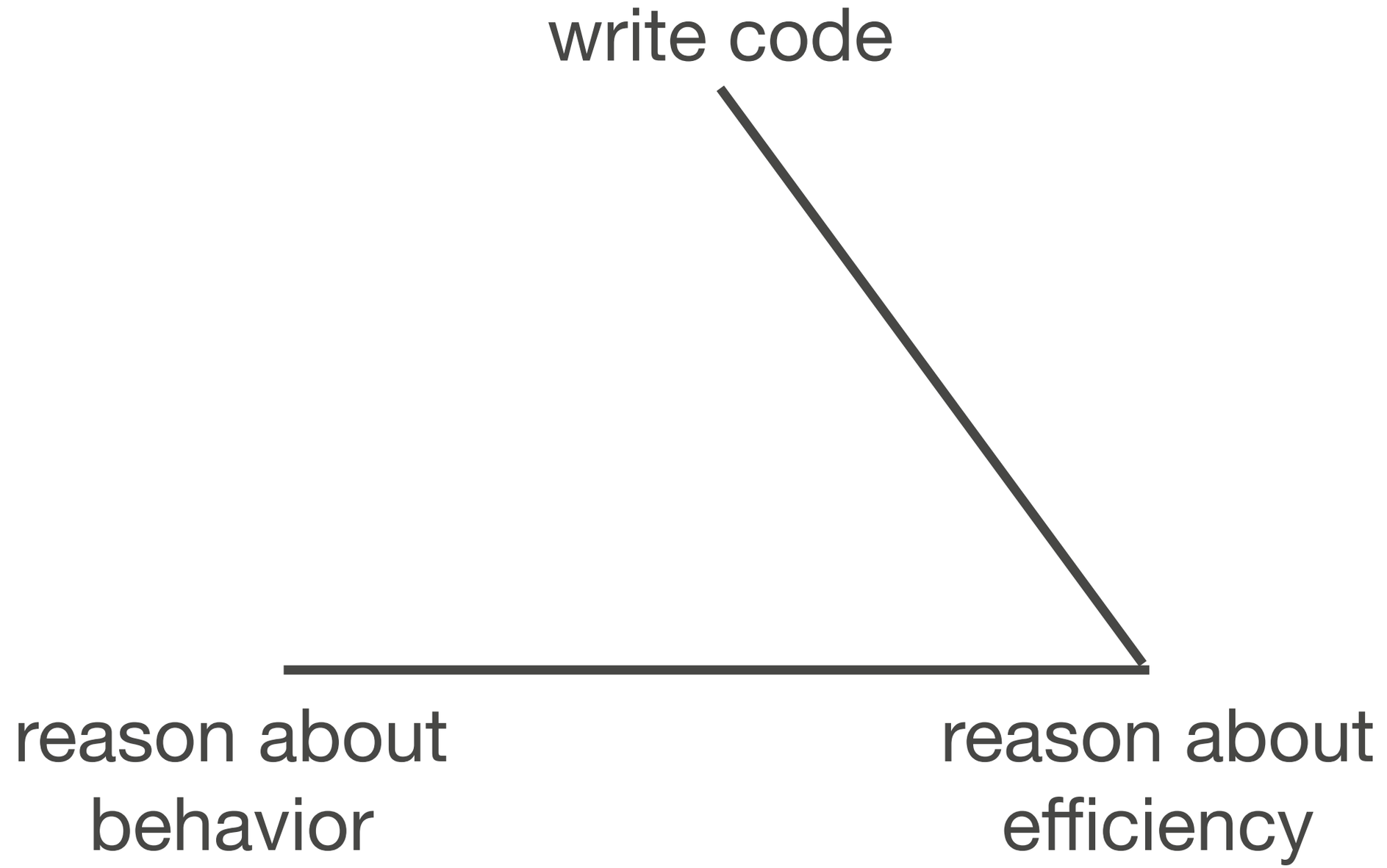
reason about efficiency

write code

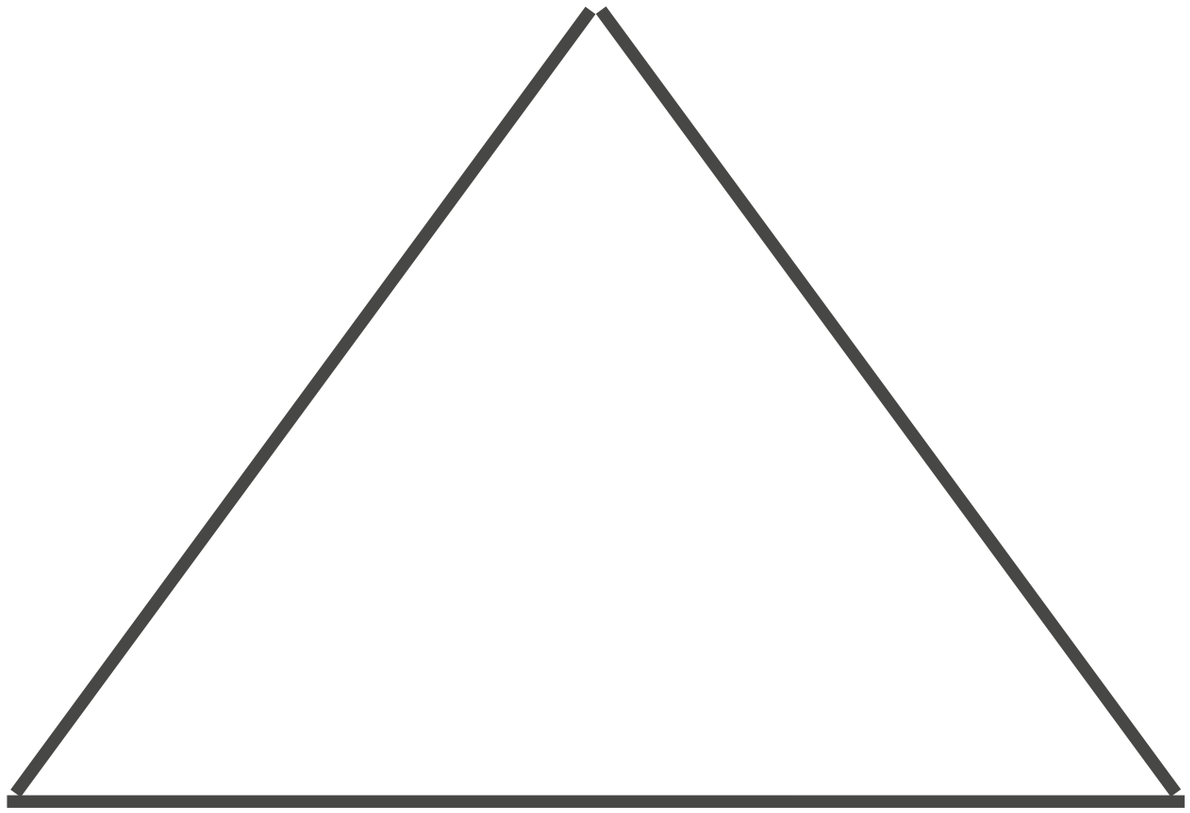


reason about behavior

reason about efficiency


write code
reason about behavior
reason about efficiency

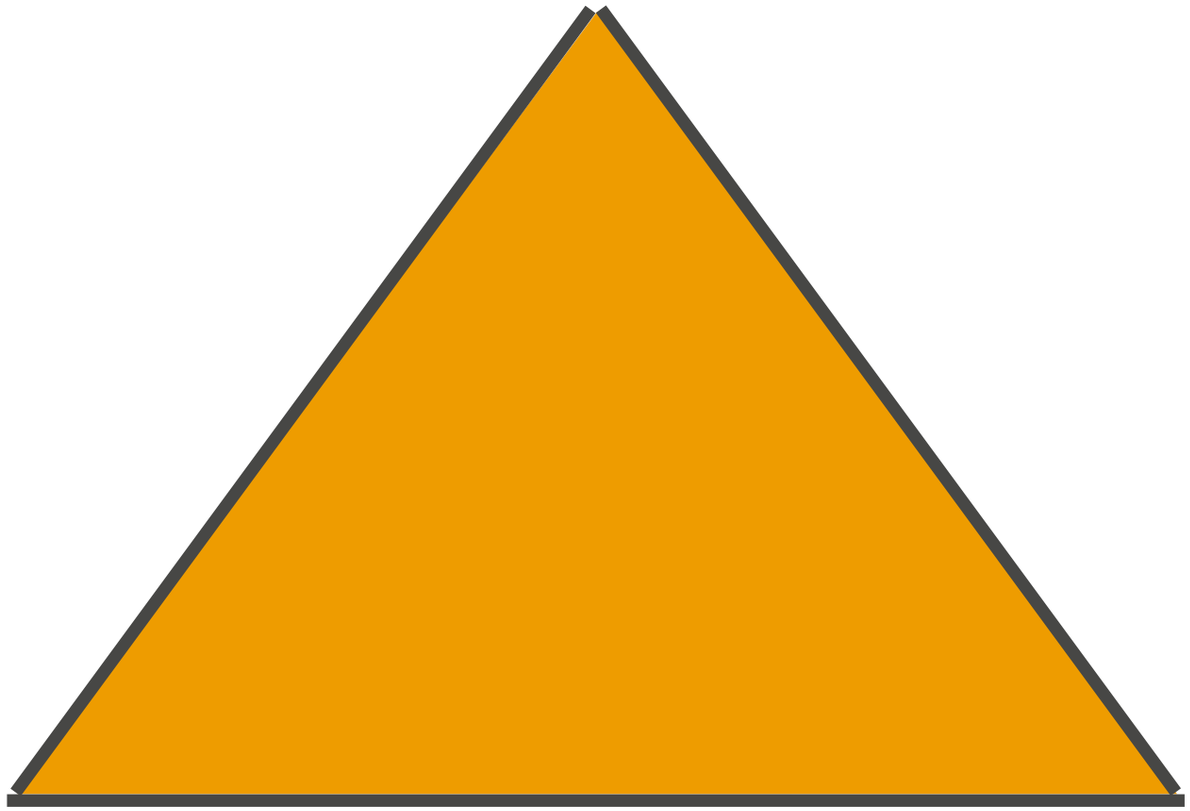
write code



reason about behavior

reason about efficiency

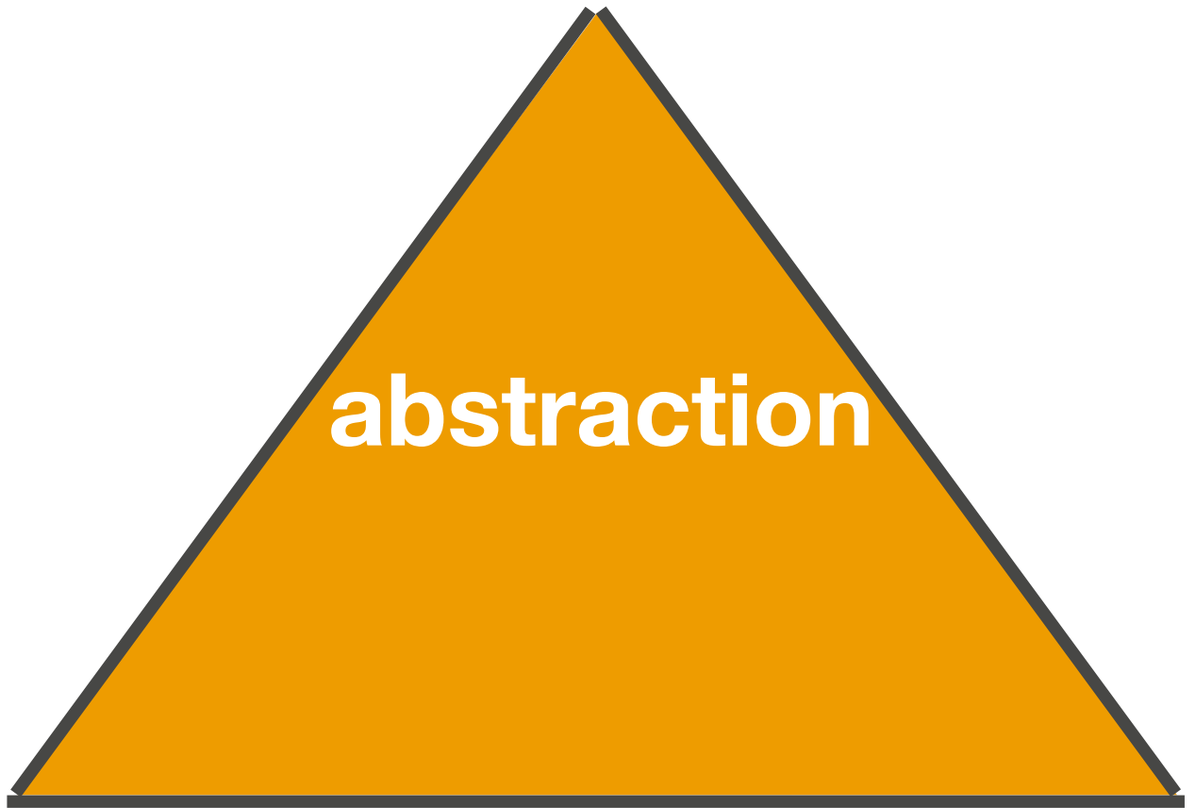
write code



reason about behavior

reason about efficiency

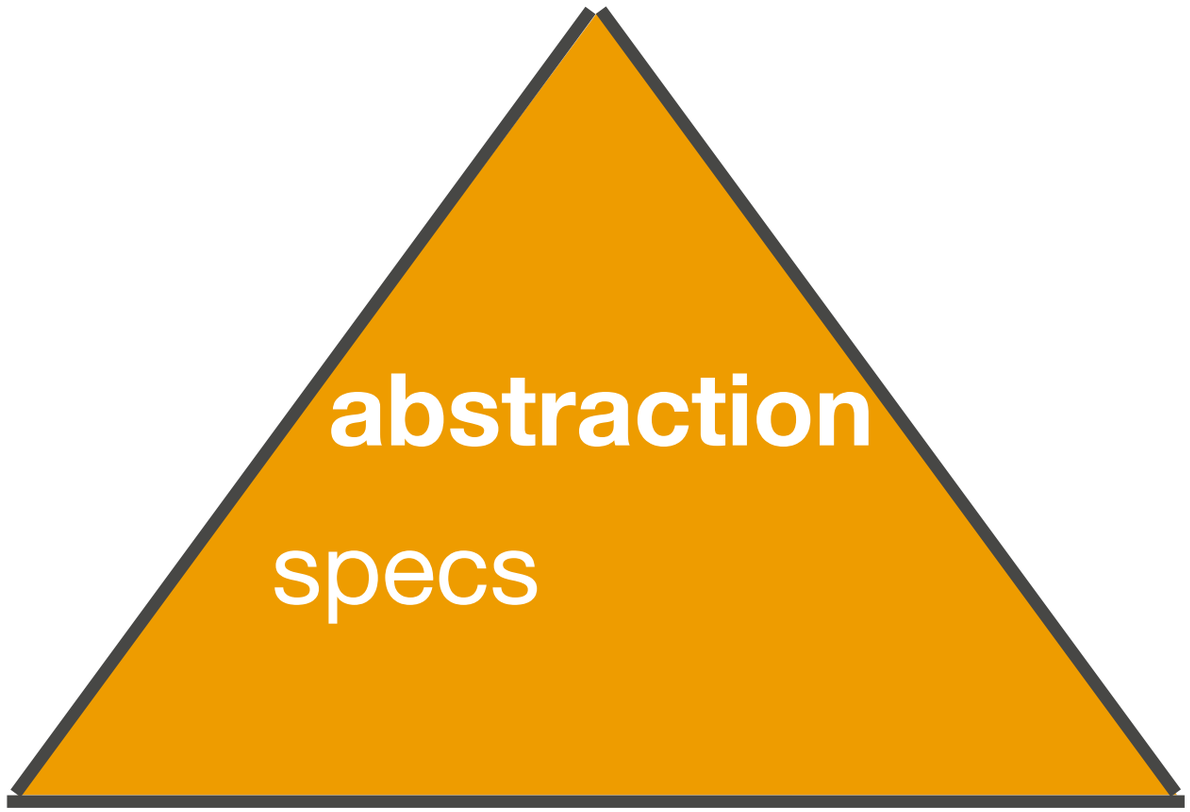
write code



reason about behavior

reason about efficiency

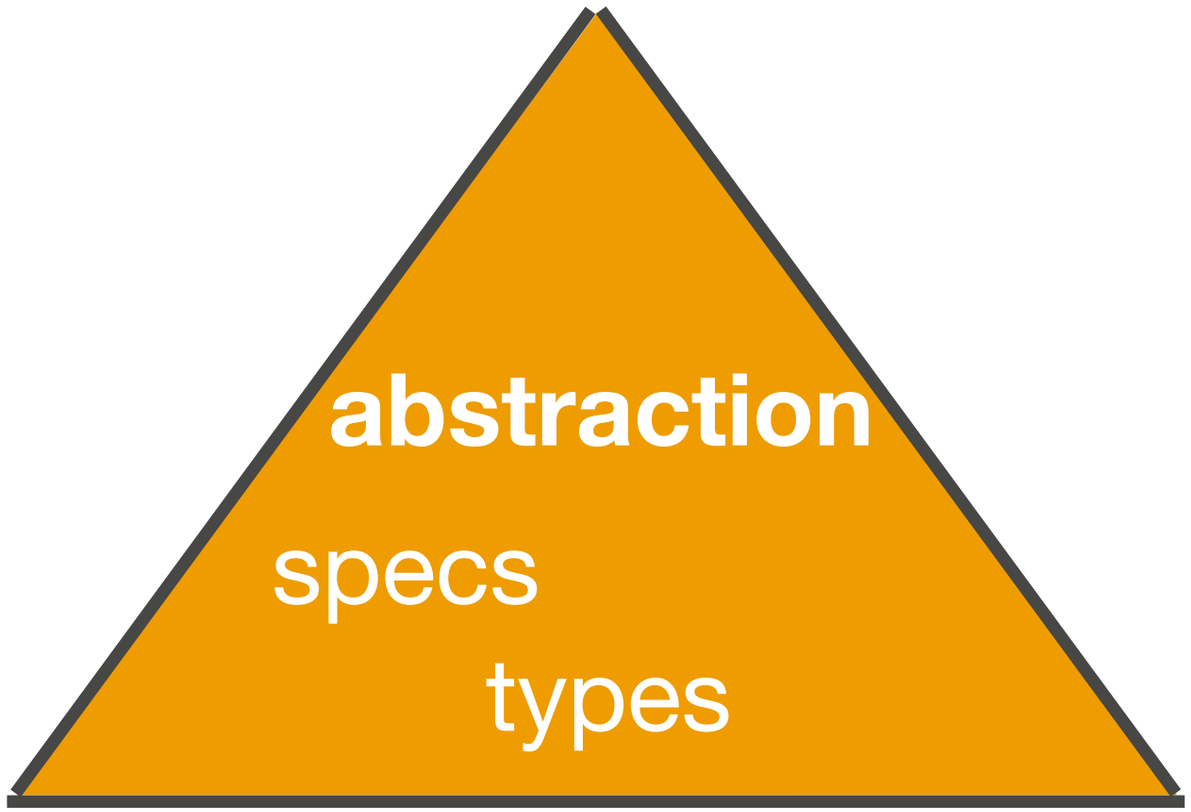
write code



reason about behavior

reason about efficiency

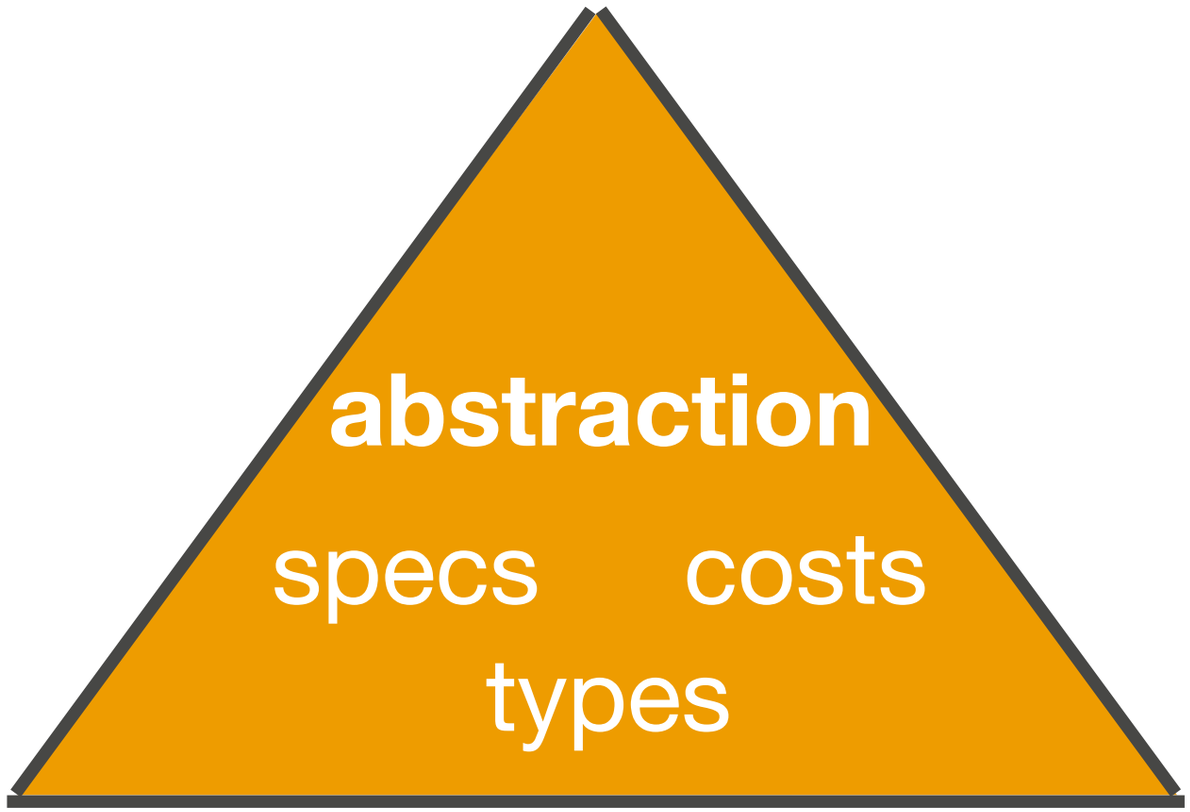
write code



reason about behavior

reason about efficiency

write code



reason about behavior

reason about efficiency

Specifications

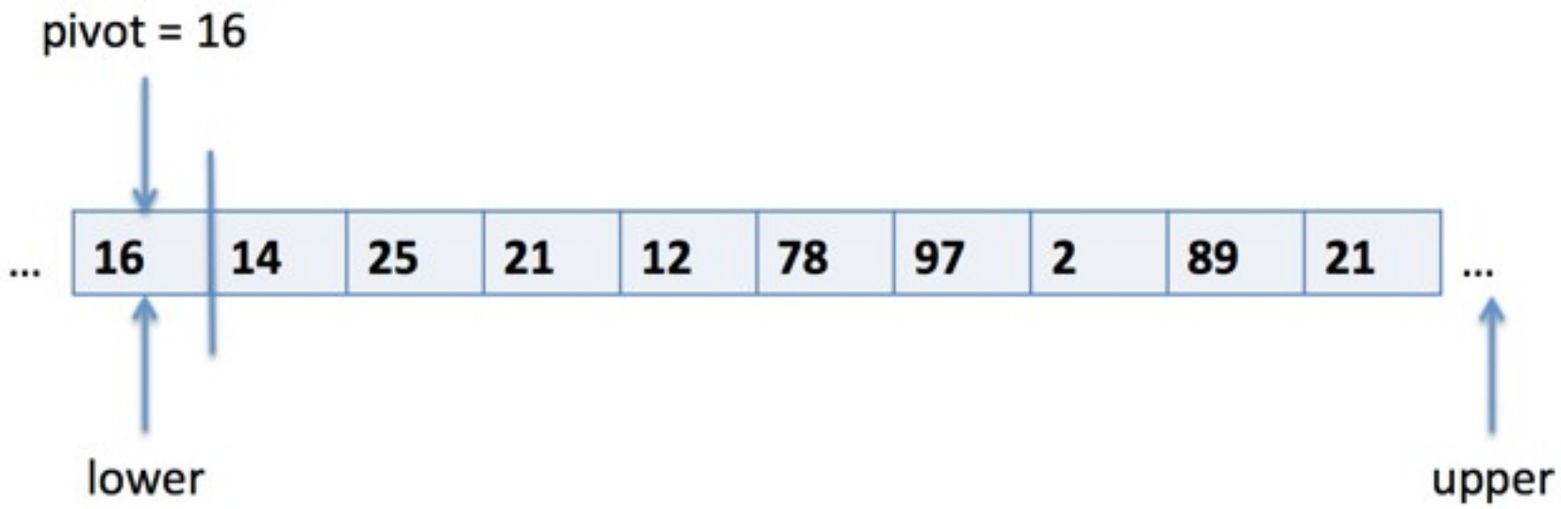
contracts structure code

- pre-conditions, post-conditions
- loop invariants

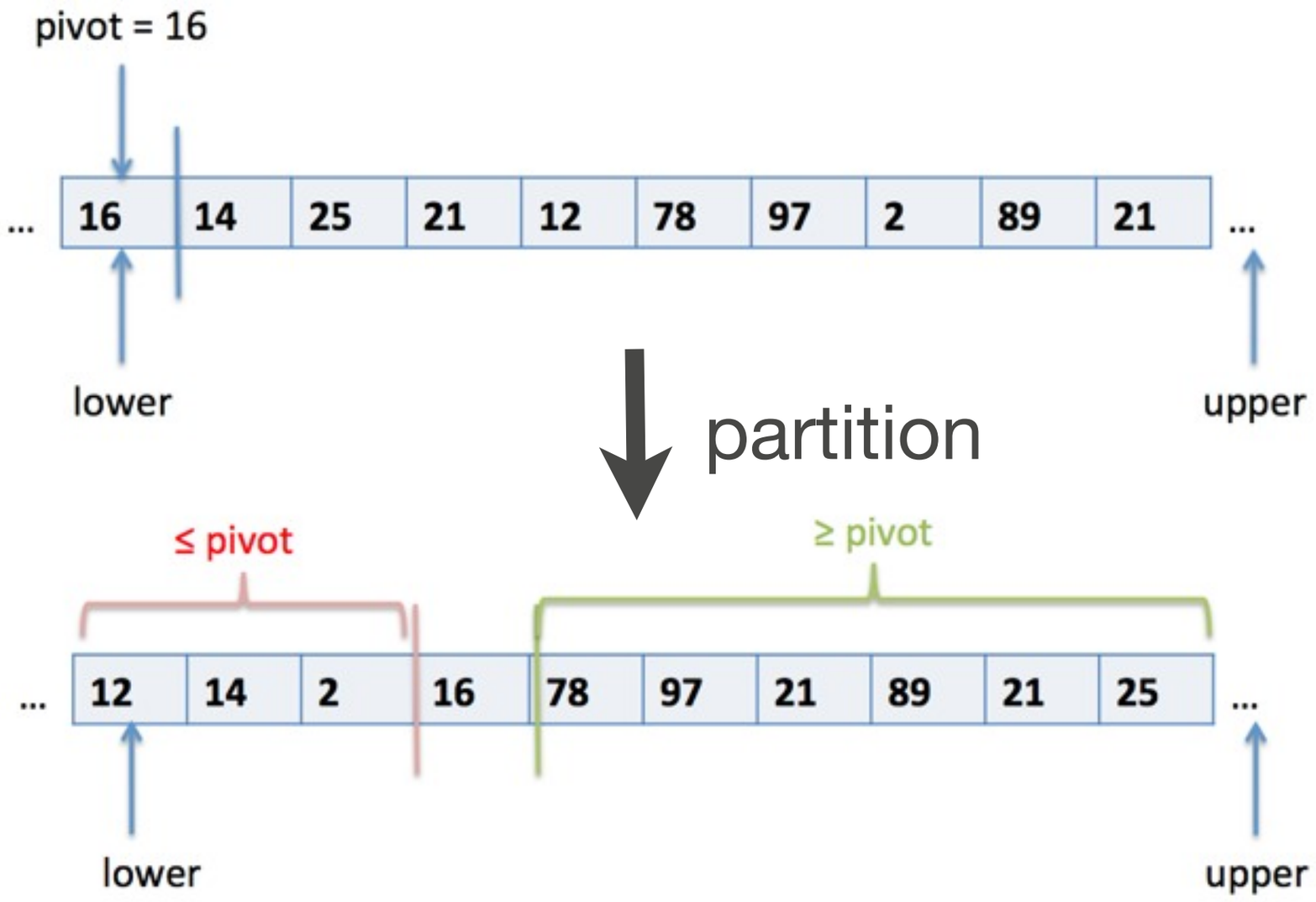
In-place quicksort

- Motivated by space and time efficiency
- Requires specification and verification to communicate an idea

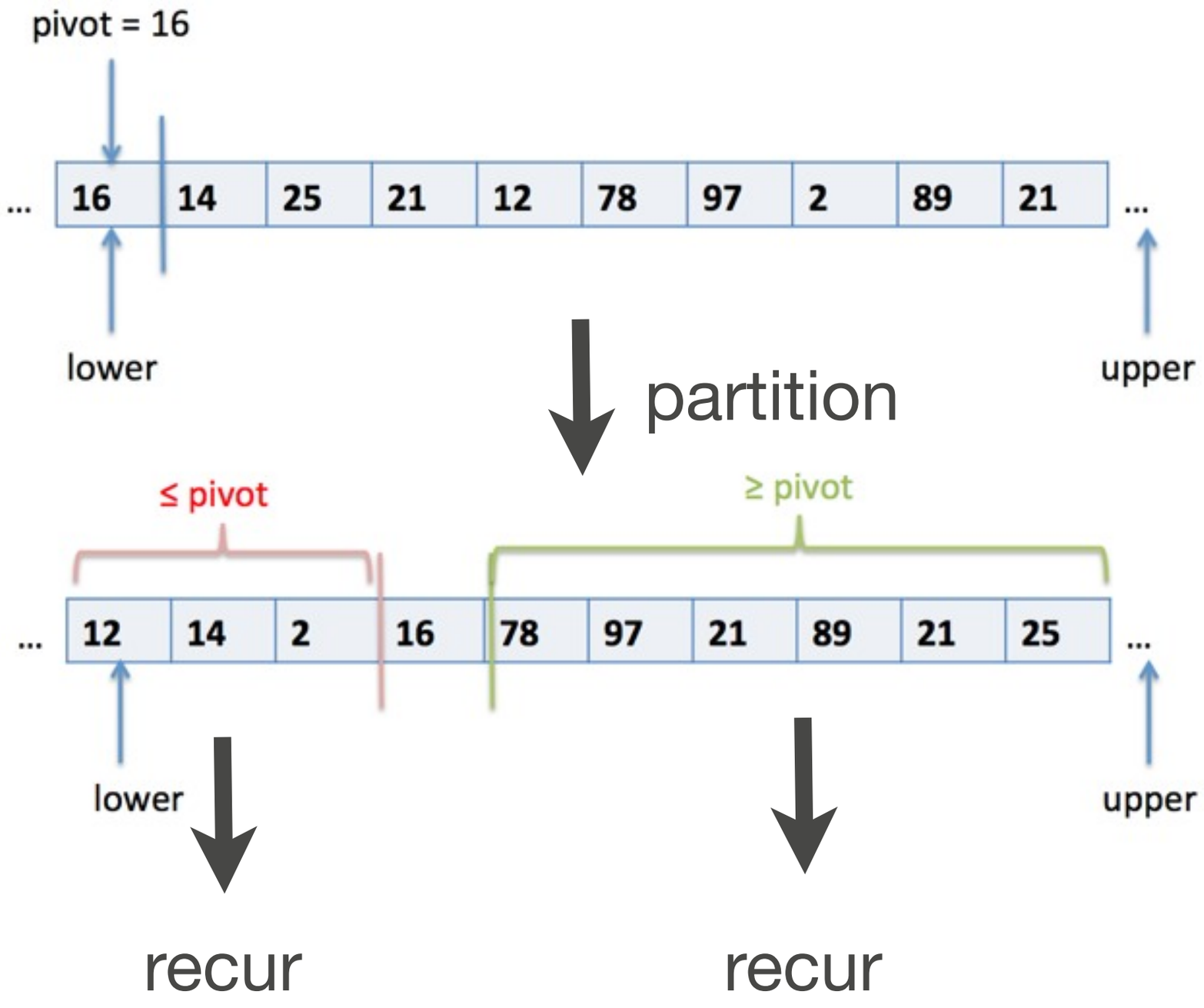
In-place quicksort



In-place quicksort



In-place quicksort



```
int partition(int[] A, int lower, int upper)
{
  int left = lower+1;
  int right = upper;

  while (left < right)
    {
      if (A[left] <= A[lower]) {
        left++;
      } else {
        swap(A,left,right-1);
        right--;
      }
    }

  swap(A,lower,left-1);
  return left-1;
}
```

```
int partition(int[] A, int lower, int upper)
{
  int left = lower+1;
  int right = upper;

  while (left < right)
    {
      if (A[left] <= A[lower]) {
        left++;
      } else {
        swap(A,left,right-1);
        right--;
      }
    }

  swap(A,lower,left-1);
  return left-1;
}
```

O(upper-lower) time

```
int partition(int[] A, int lower, int upper)
{
  int left = lower+1;
  int right = upper;

  while (left < right)
    {
      if (A[left] <= A[lower]) {
        left++;
      } else {
        swap(A,left,right-1);
        right--;
      }
    }

  swap(A,lower,left-1);
  return left-1;
}
```

O(upper-lower) time

O(1) space

```
int partition(int[] A, int lower, int upper)
{
  int left = lower+1;
  int right = upper;

  while (left < right)
    {
      if (A[left] <= A[lower]) {
        left++;
      } else {
        swap(A,left,right-1);
        right--;
      }
    }

  swap(A,lower,left-1);
  return left-1;
}
```

O(upper-lower) time

O(1) space

what is it supposed to do?

why does it do that?

```
int partition(int[] A, int lower, int upper)
{
  int left = lower+1;
  int right = upper;

  while (left < right)
    {
      if (A[left] <= A[lower]) {
        left++;
      } else {
        swap(A,left,right-1);
        right--;
      }
    }

  swap(A,lower,left-1);
  return left-1;
}
```

O(upper-lower) time

O(1) space

what is it supposed to do?

why does it do that?

```
int partition(int[] A, int lower, int upper)
{
  int left = lower+1;
  int right = upper;

  while (left < right)
    {
      if (A[left] <= A[lower]) {
        left++;
      } else {
        swap(A,left,right-1);
        right--;
      }
    }

  swap(A,lower,left-1);
  return left-1;
}
```

O(upper-lower) time

O(1) space

what is it supposed to do?

why does it do that?

```
int partition(int[] A, int lower, int upper)
{
  int left = lower+1;
  int right = upper;

  while (left < right)
    {
      if (A[left] <= A[lower]) {
        left++;
      } else {
        swap(A,left,right-1);
        right--;
      }
    }

  swap(A,lower,left-1);
  return left-1;
}
```

O(upper-lower) time

O(1) space

what is it supposed to do?

why does it do that?

```
int partition(int[] A, int lower, int upper)
{
  int left = lower+1;
  int right = upper;

  while (left < right)
    {
      if (A[left] <= A[lower]) {
        left++;
      } else {
        swap(A,left,right-1);
        right--;
      }
    }

  swap(A,lower,left-1);
  return left-1;
}
```

O(upper-lower) time

O(1) space

what is it supposed to do?

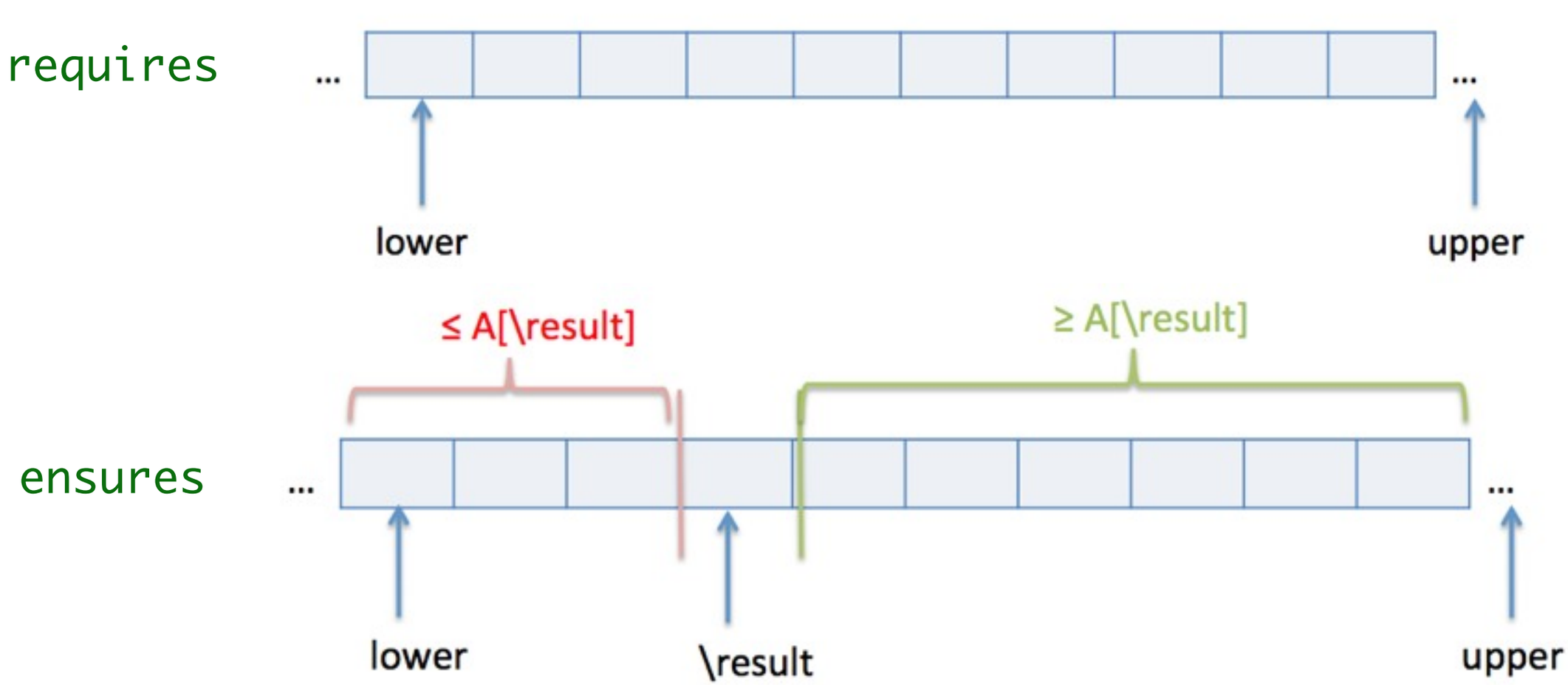
why does it do that?

```
int partition(int[] A, int lower, int upper)
//@requires 0 <= lower < upper <= \length(A);
//@ensures lower <= \result < upper;
//@ensures A[lower, \result)
//          <= A[\result]
//          <= A[\result,upper);
```

```
int partition(int[] A, int lower, int upper)

//@requires 0 <= lower < upper <= \length(A);
//@ensures lower <= \result < upper;
//@ensures A[lower, \result)
//          <= A[\result]
//          <= A[\result,upper);
```

requires

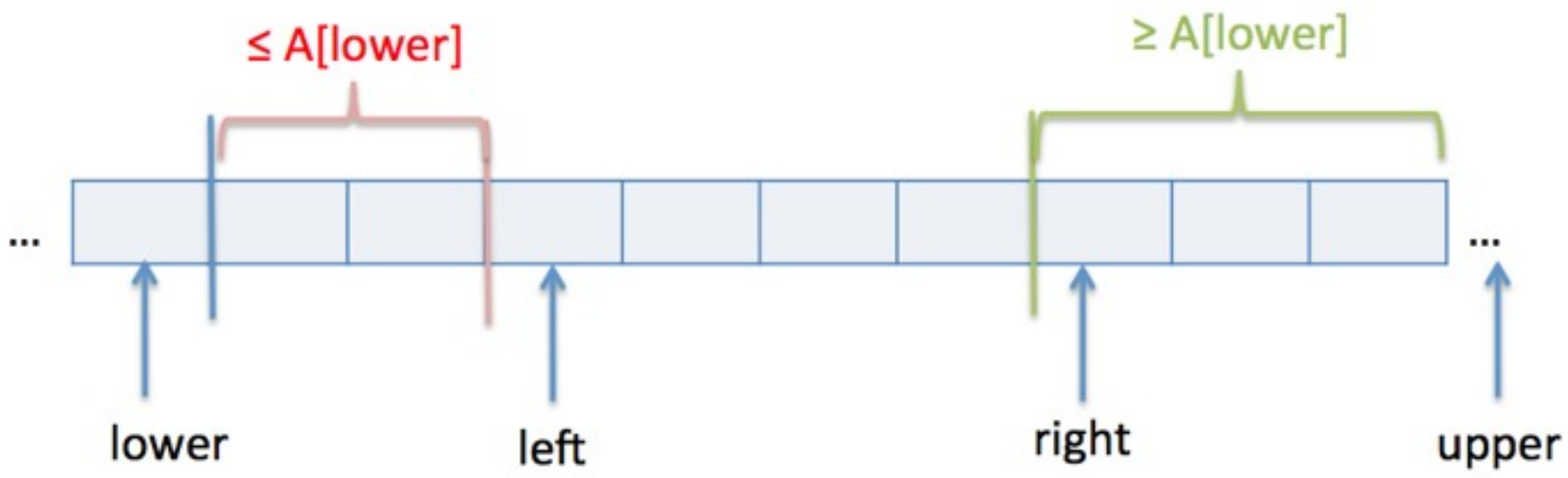
```
int partition(int[] A, int lower, int upper)
//@requires 0 <= lower < upper <= \length(A);
//@ensures lower <= \result < upper;
//@ensures A[lower, \result)
//          <= A[\result]
//          <= A[\result,upper);
```



```
while (left < right)
   {
     if (A[left] <= A[lower]) {
       left++;
     } else {
       swap(A,left,right-1);
       right--;
     }
   }
```

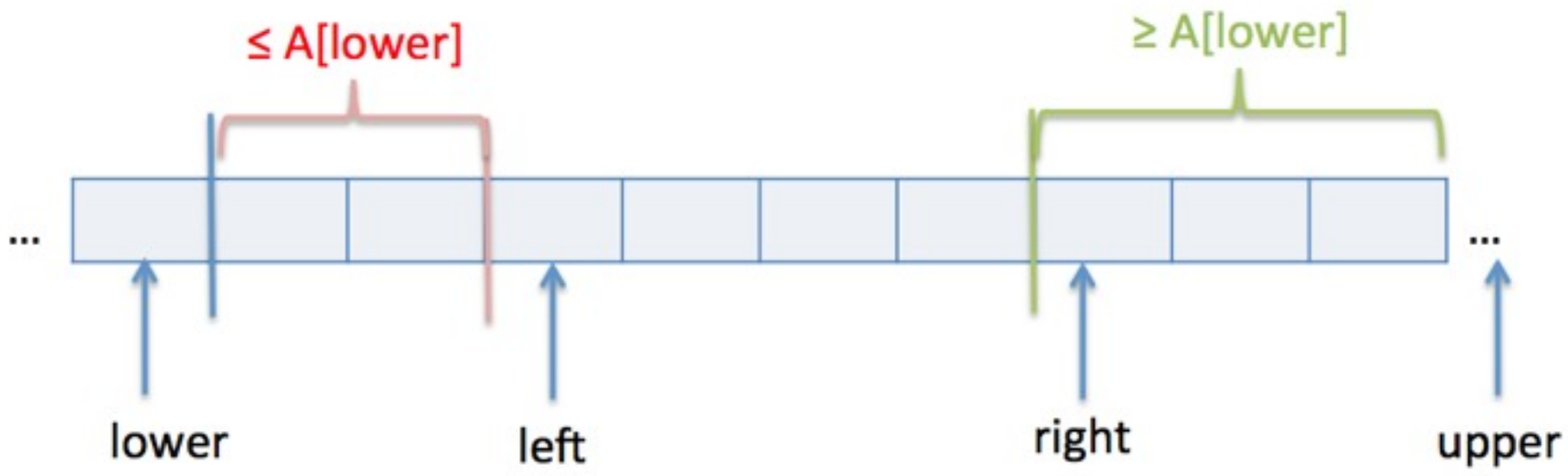
```
//@loop_invariant lower < left <= right <= upper;
//@loop_invariant     A[lower+1,left)
//                 <= A[lower]
//                 <= A[right,upper);
```

```
while (left < right)
   {
     if (A[left] <= A[lower]) {
       left++;
     } else {
       swap(A,left,right-1);
       right--;
     }
   }
```



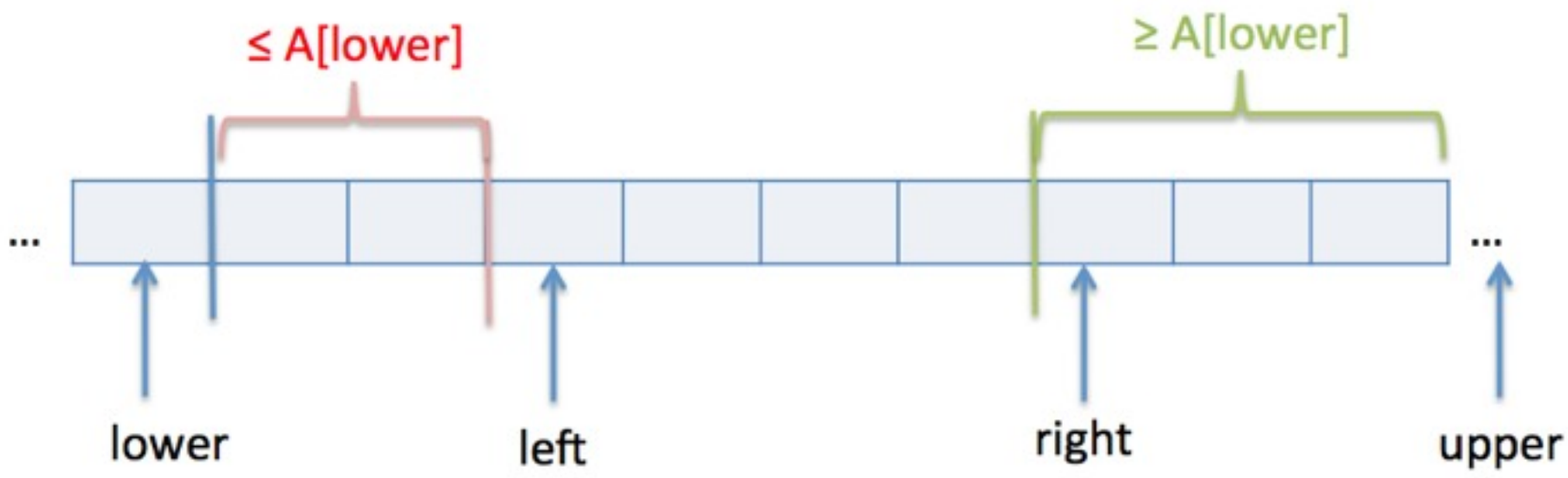
```
while (left < right)
  {
    if (A[left] <= A[lower]) {
      left++;
    } else {
      swap(A,left,right-1);
      right--;
    }
  }
```

purpose of a test is to establish a fact



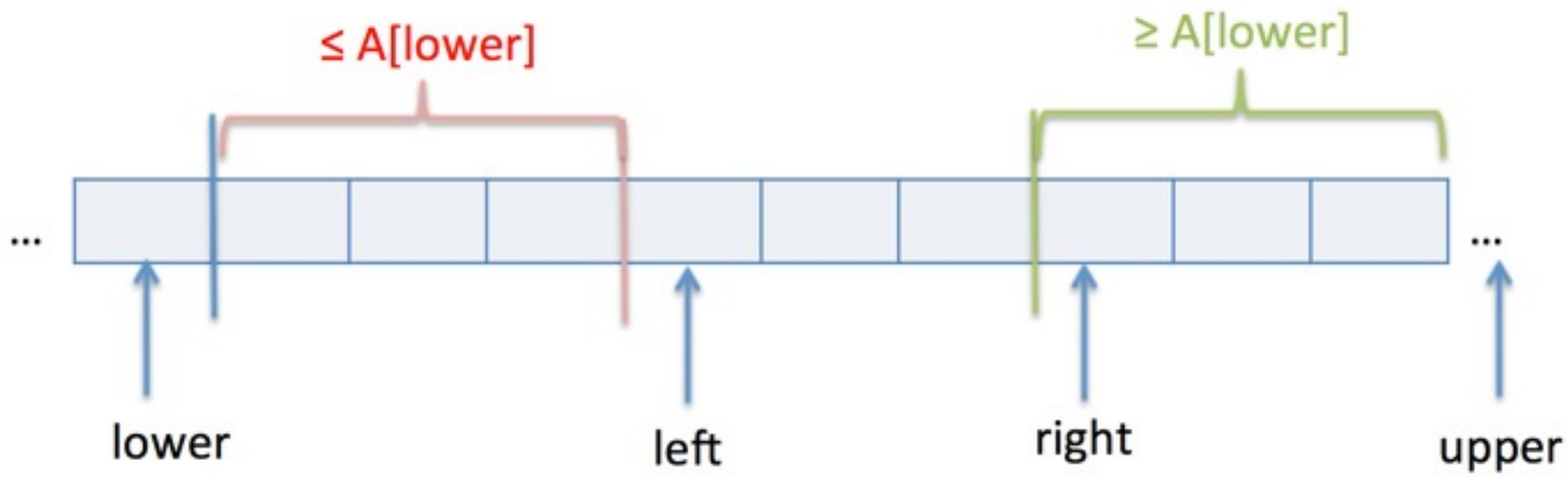
```
while (left < right)
   {
     if (A[left] <= A[lower]) {
        left++;
     } else {
        swap(A,left,right-1);
        right--;
     }
   }
```

justifies expanding left



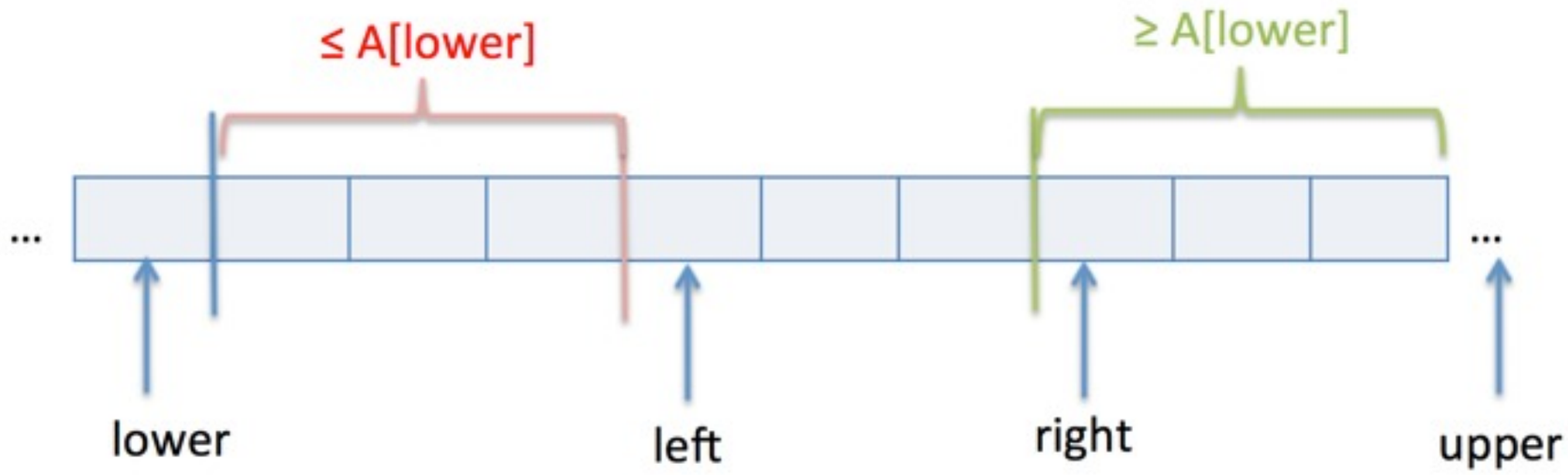
```
while (left < right)
   {
     if (A[left] <= A[lower]) {
        left++;
     } else {
        swap(A,left,right-1);
        right--;
     }
   }
```

justifies expanding left



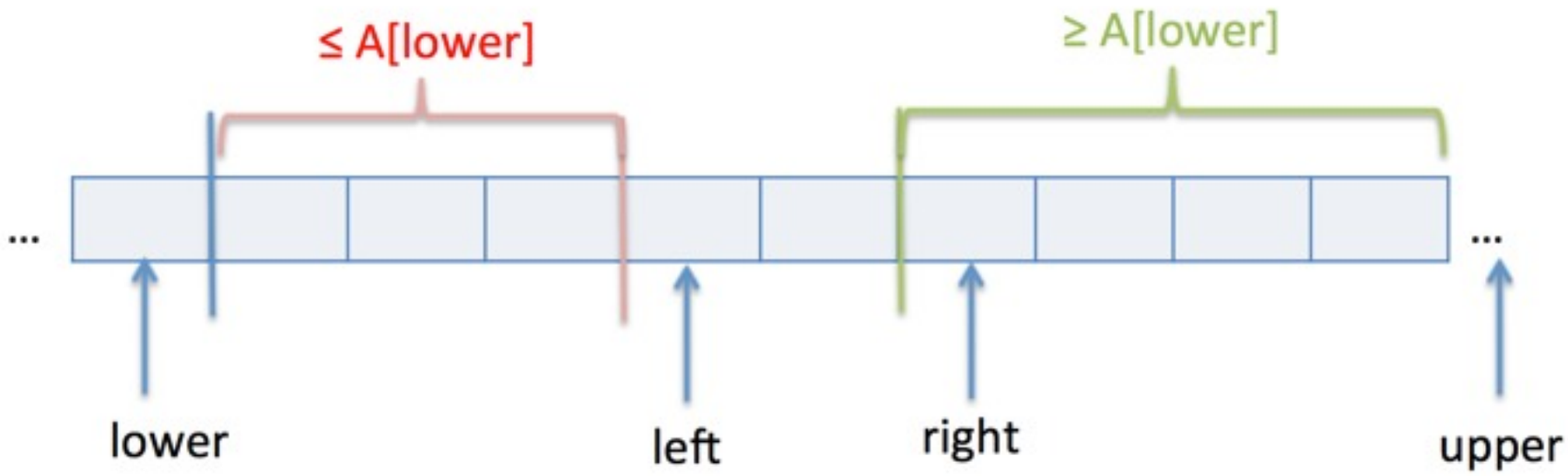
```
while (left < right)
   {
     if (A[left] <= A[lower]) {
        left++;
     } else {
        swap(A,left,right-1);
        right--;
     }
   }
```

test and swap
justify
expanding right



```
while (left < right)
   {
     if (A[left] <= A[lower]) {
        left++;
     } else {
        swap(A,left,right-1);
        right--;
     }
   }
```

test and swap
justify
expanding right

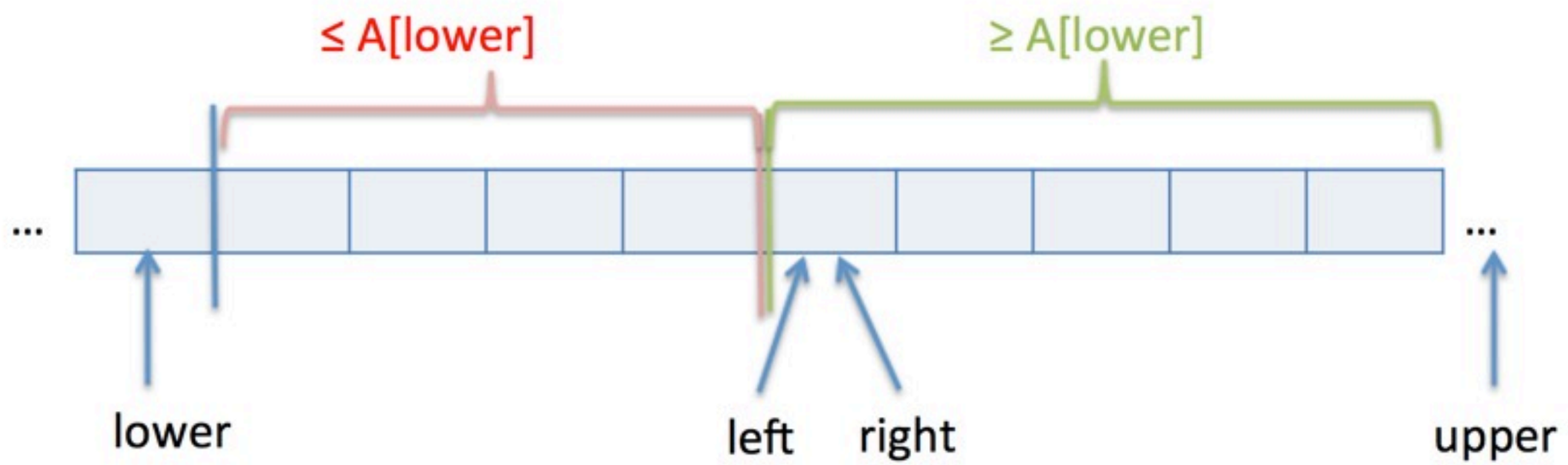


```
swap(A,lower,left-1);
return left-1;
```

when loop exits, invariant is true and loop test is false

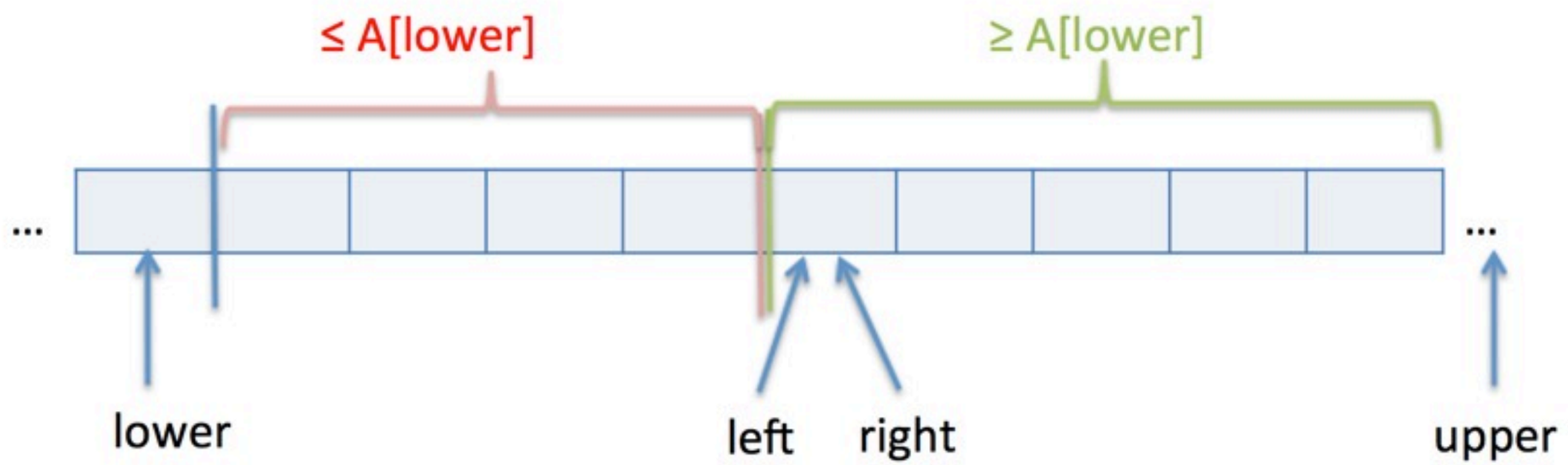
```
swap(A,lower,left-1);
return left-1;
```

when loop exits, invariant is true and loop test is false



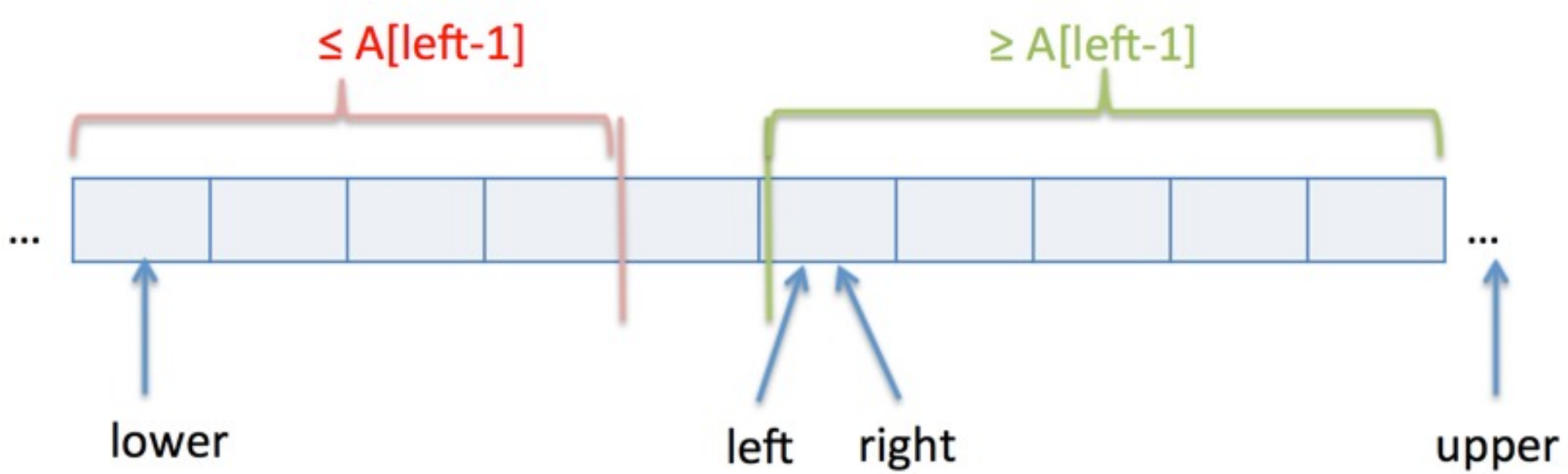
```
swap(A,lower,left-1);
return left-1;
```

swap makes postcondition true

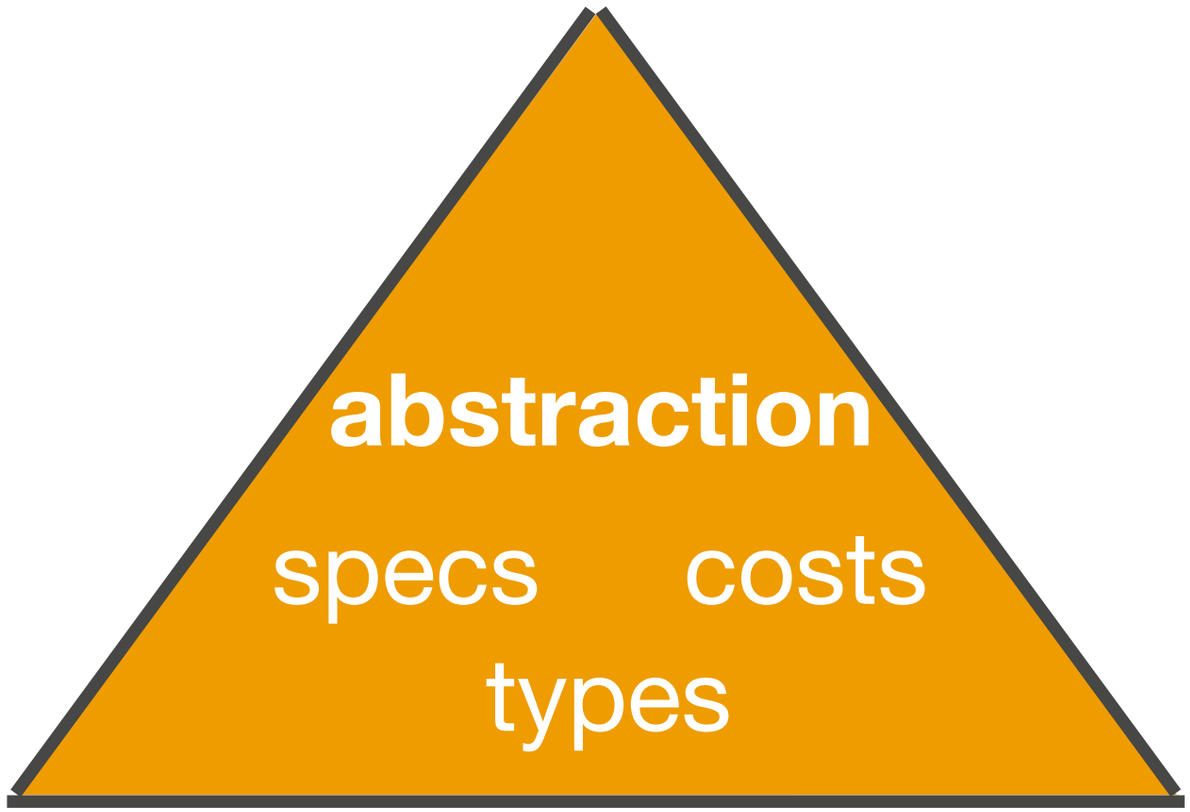


```
swap(A,lower,left-1);
return left-1;
```

swap makes postcondition true



write code



reason about behavior

reason about efficiency

Types

Intro: how do I make something of that type?

Elim: how do I use something of that type?

Types structure code

```
fun f (t : tree) =
  case t of
    Empty => …
  | Leaf x => …
  | Node(l,r) => … f l … f r …
```

Types structure reasoning

Theorem: For all t:tree, …

Proof: structural induction on t.

Case for Empty:

Case for Leaf x:

Case for Node(L,R):
    IH on L:
    IH on R:

Beyond requires/ensures

Theorem: `(map g) o (map f) = map (g o f)`

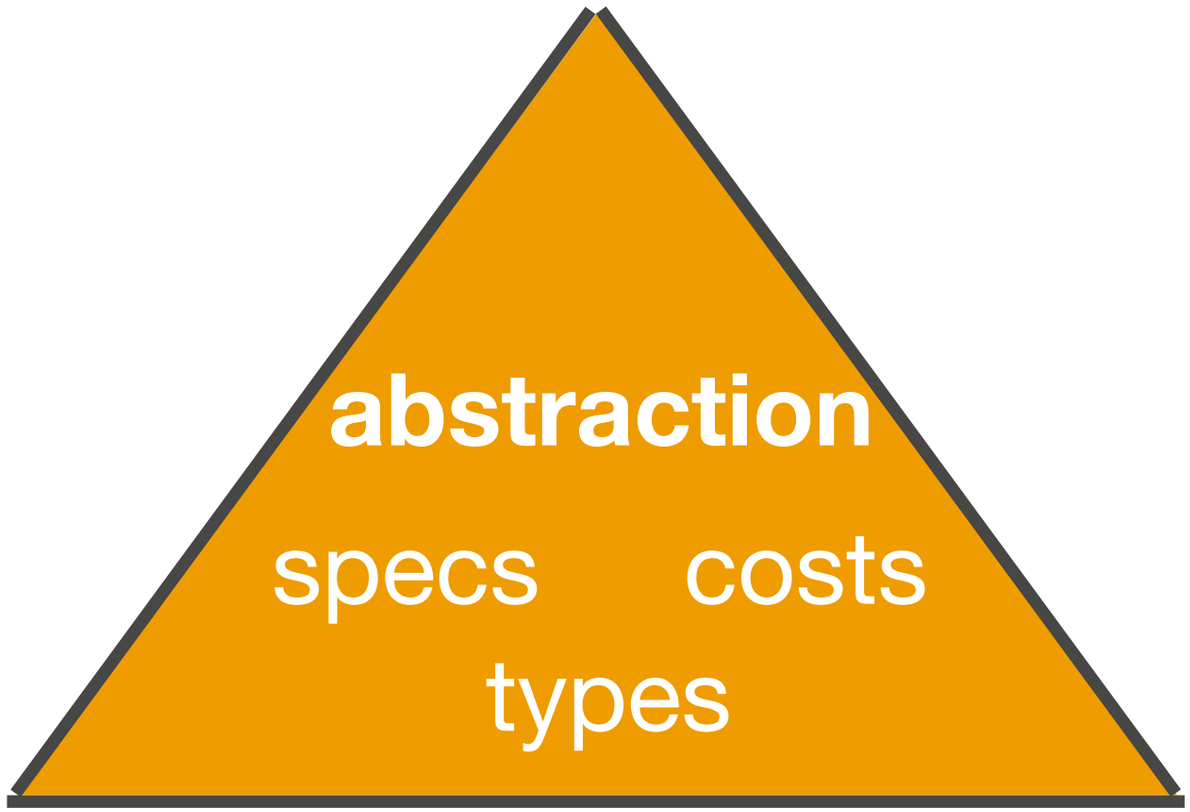
Abstract types

```
signature DICT =
sig
  structure Key : ORDERED
  type 'v dict

  val empty   : 'v dict
  val insert  : 'v dict -> (Key.t * 'v) -> 'v dict

  val lookup  : 'v dict -> Key.t -> 'v option

  val map     : ('a -> 'b) -> 'a dict -> 'b dict

  val merge  : ('v * 'v -> 'v) -> 'v dict * 'v dict -> 'v dict
end
```

Type classes

```
signature ORDERED =
sig
    type t
    val compare : t * t -> order
end
```

write code



reason about behavior

reason about efficiency

Abstract from the machine

Abstract from the machine

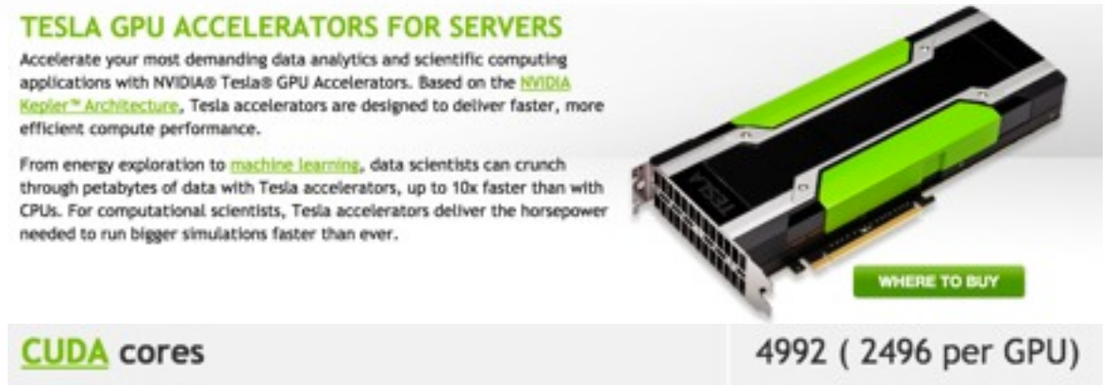
garbage collection :: RAM

Abstract from the machine

garbage collection :: RAM

scheduling :: CPUs

Andrew Chien, 2008



Parallel thinking

required for current and foreseeable platforms

Understand dependencies in an algorithm

Evaluate sequential and parallel efficiency

Parallel thinking

Parallel thinking

Say **what to compute** using
transformations of persistent data structures

Parallel thinking

Say **what to compute** using
transformations of persistent data structures

Analyze **how long it will take** using
language-based cost semantics [Blelloch et al.]

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost

Bounded implementation

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost **COMP 331**

Bounded implementation

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost **COMP 331**

time

Bounded implementation

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost **COMP 331**

time

space

Bounded implementation

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost **COMP 331**

time

space

I/O complexity

Bounded implementation

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost **COMP 331**

time

space

I/O complexity

parallelism

Bounded implementation

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost **COMP 331**

time

space

I/O complexity

parallelism

Bounded implementation

processor speed

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost **COMP 331**

time

space

I/O complexity

parallelism

Bounded implementation

processor speed

garbage collection

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost **COMP 331**

time

space

I/O complexity

parallelism

Bounded implementation

processor speed

garbage collection

cache policy

Cost abstraction [Blelloch and Greiner]

Code you write

- what to do
- abstract cost

Compiled code

- scheduled resources
- concrete cost **COMP 331**

time

space

I/O complexity

parallelism

Bounded implementation

processor speed

garbage collection

cache policy

number of processors and scheduling policy

Abstract Costs

Work is usual sequential complexity

Span is length of longest dependency

$$\frac{e_1 \Downarrow_{s_1}^{w_1} v_1 \quad e_2 \Downarrow_{s_2}^{w_2} v_2}{(e_1, e_2) \Downarrow_{\max(s_1,s_2)}^{w_1+w_2} (v_1, v_2)}$$

Abstract Costs

Work is usual sequential complexity

Span is length of longest dependency

$$\frac{e_1 \Downarrow_{s_1}^{w_1} v_1 \quad e_2 \Downarrow_{s_2}^{w_2} v_2}{(e_1, e_2) \Downarrow_{\max(s_1,s_2)}^{w_1+w_2} (v_1, v_2)}$$

COMP 321

Bounded Implementation

[Brent ’74]

work and *span* → time on *p* processors

$O(\max(work/p, span))$

- Ideally, each processor does work at each step
- Can’t beat the length of the longest dependency

Bounded Implementation

[Brent '74]

work and *span* → time on *p* processors

$O(\max(work/p, span))$

- Ideally, each processor does work at each step
- Can't beat the length of the longest dependency

Parallelizability ratio := *work*/*span*

```
fun mergesort (l : int list) : int list =
    case l of
        [] => []
      | [x] => [x]
      | _ => let val (pile1,pile2) = split l
             in
                 merge (mergesort pile1,
                        mergesort pile2)
             end
```

Work is O(n log n)

[7,1,3,6,8,4,2,5]

Work is O(n log n)

[7,1,3,6,8,4,2,5]

[7,1,3,6] [8,4,2,5]

Work is O(n log n)

[7,1,3,6,8,4,2,5]

[7,1,3,6] [8,4,2,5]

[7,1] [3,6] [8,4] [2,5]

Work is O(n log n)

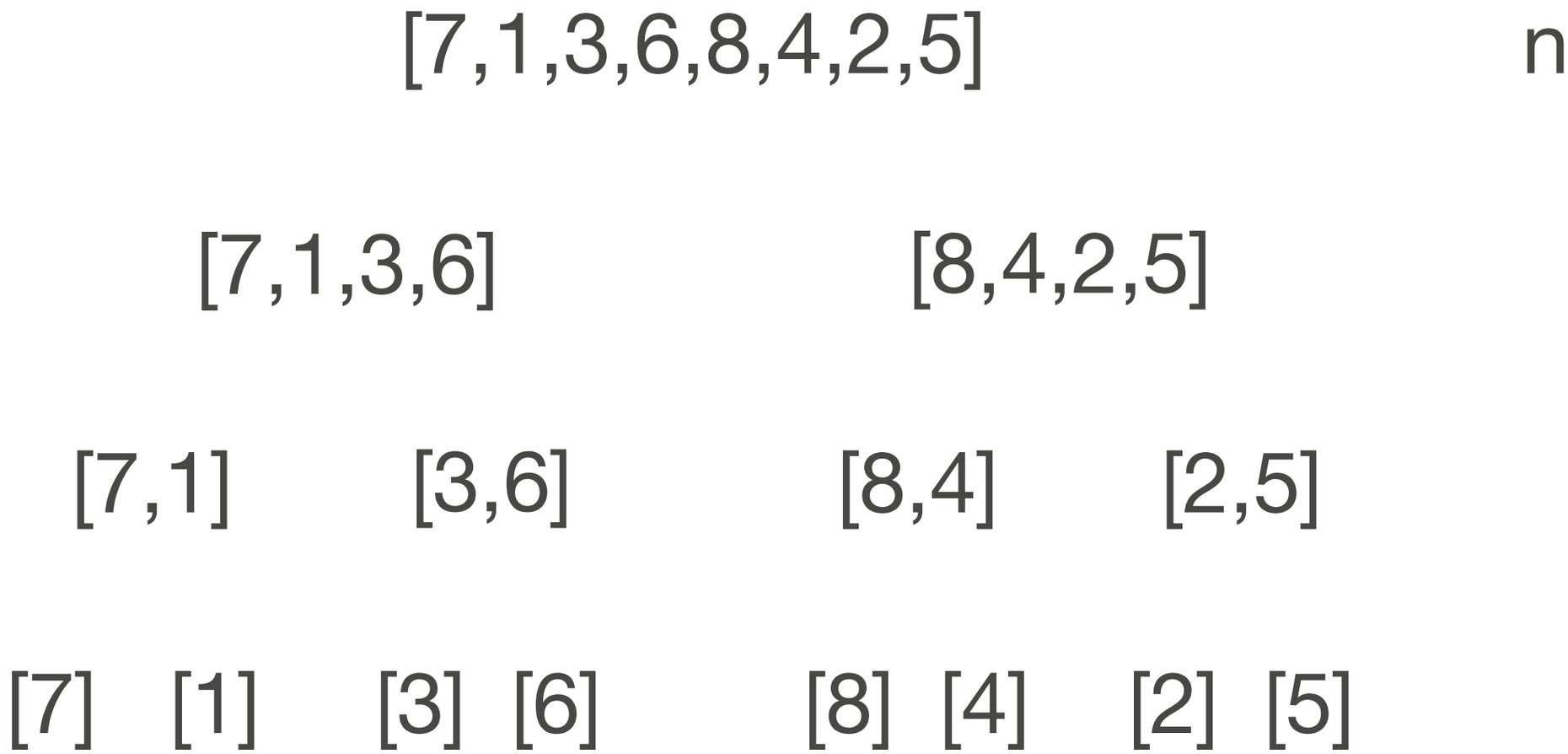
[7,1,3,6,8,4,2,5]

[7,1,3,6] [8,4,2,5]

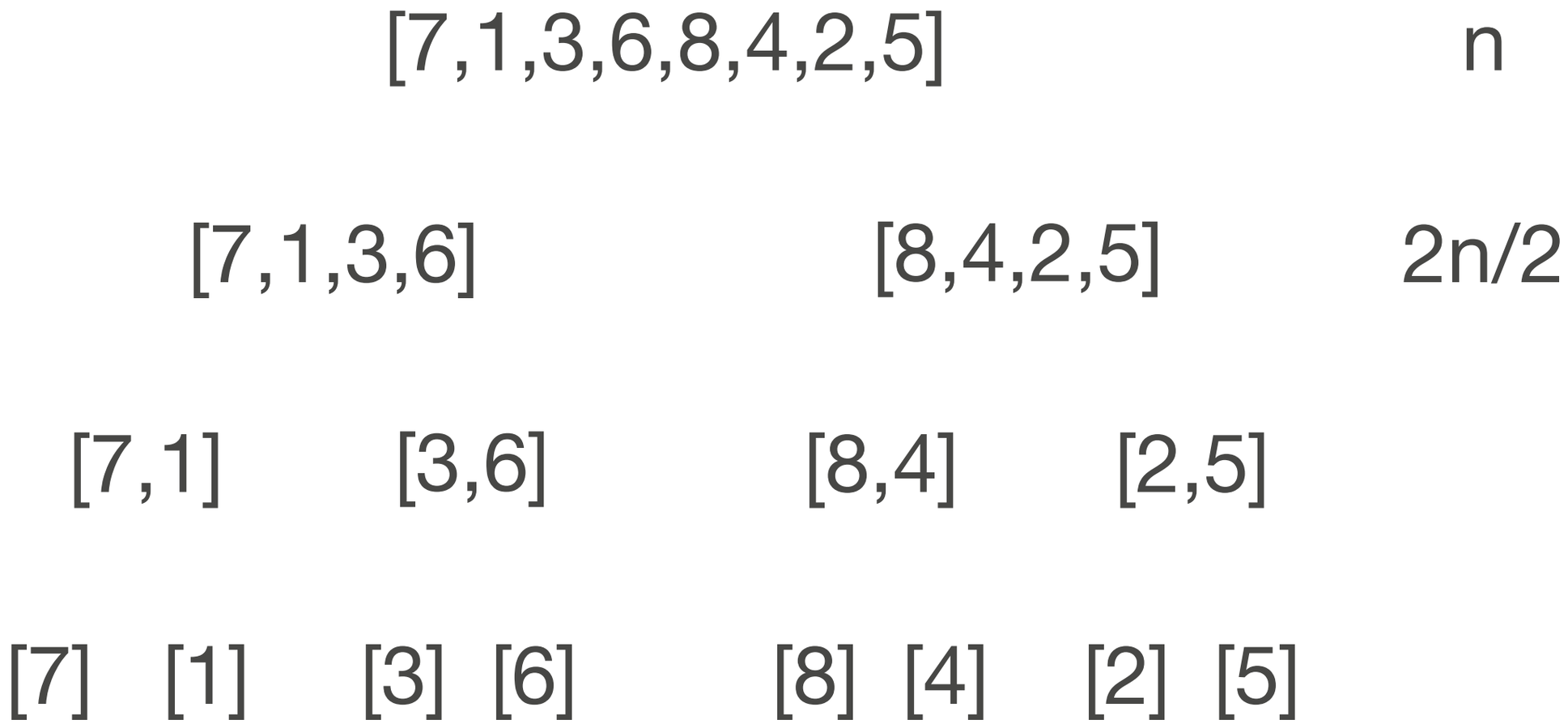
[7,1] [3,6] [8,4] [2,5]

[7] [1] [3] [6] [8] [4] [2] [5]

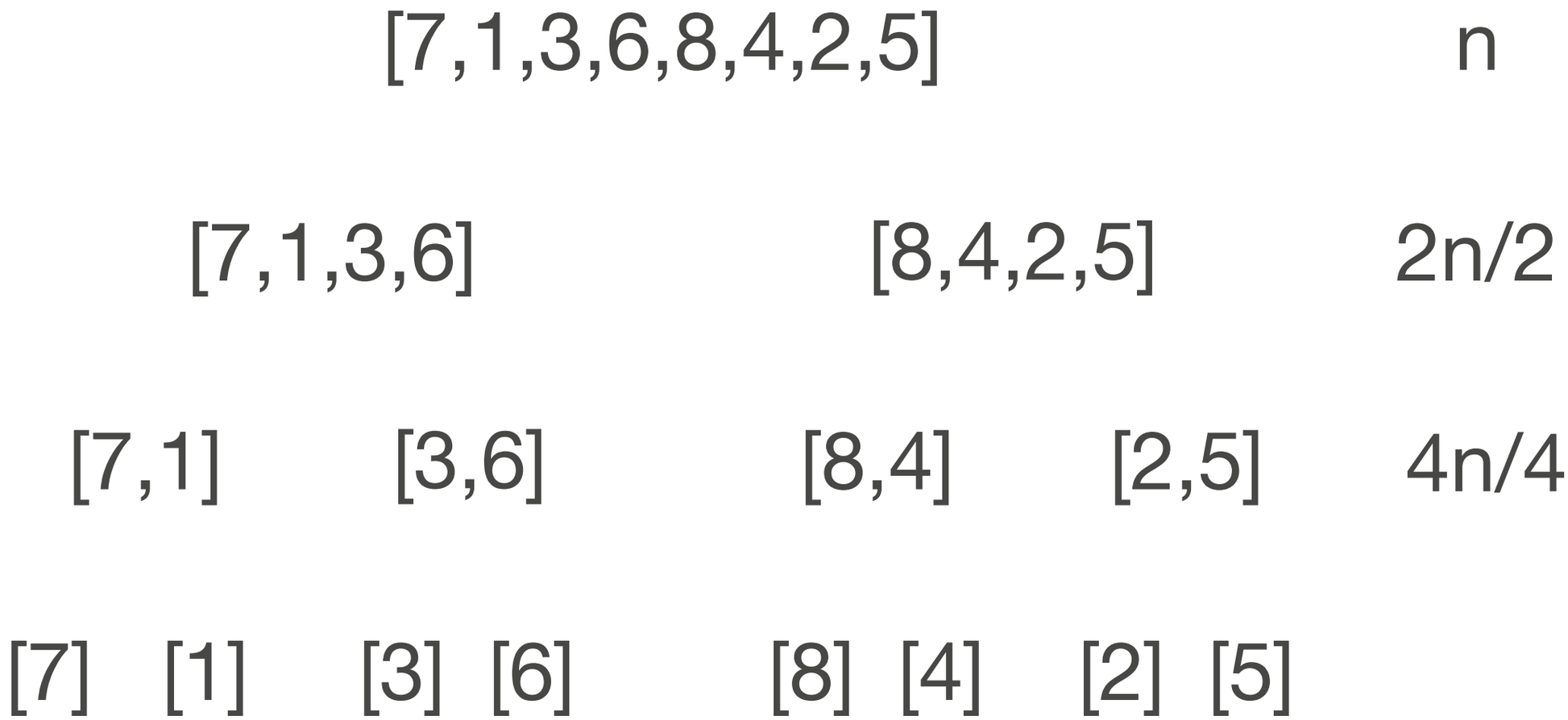
Work is O(n log n)



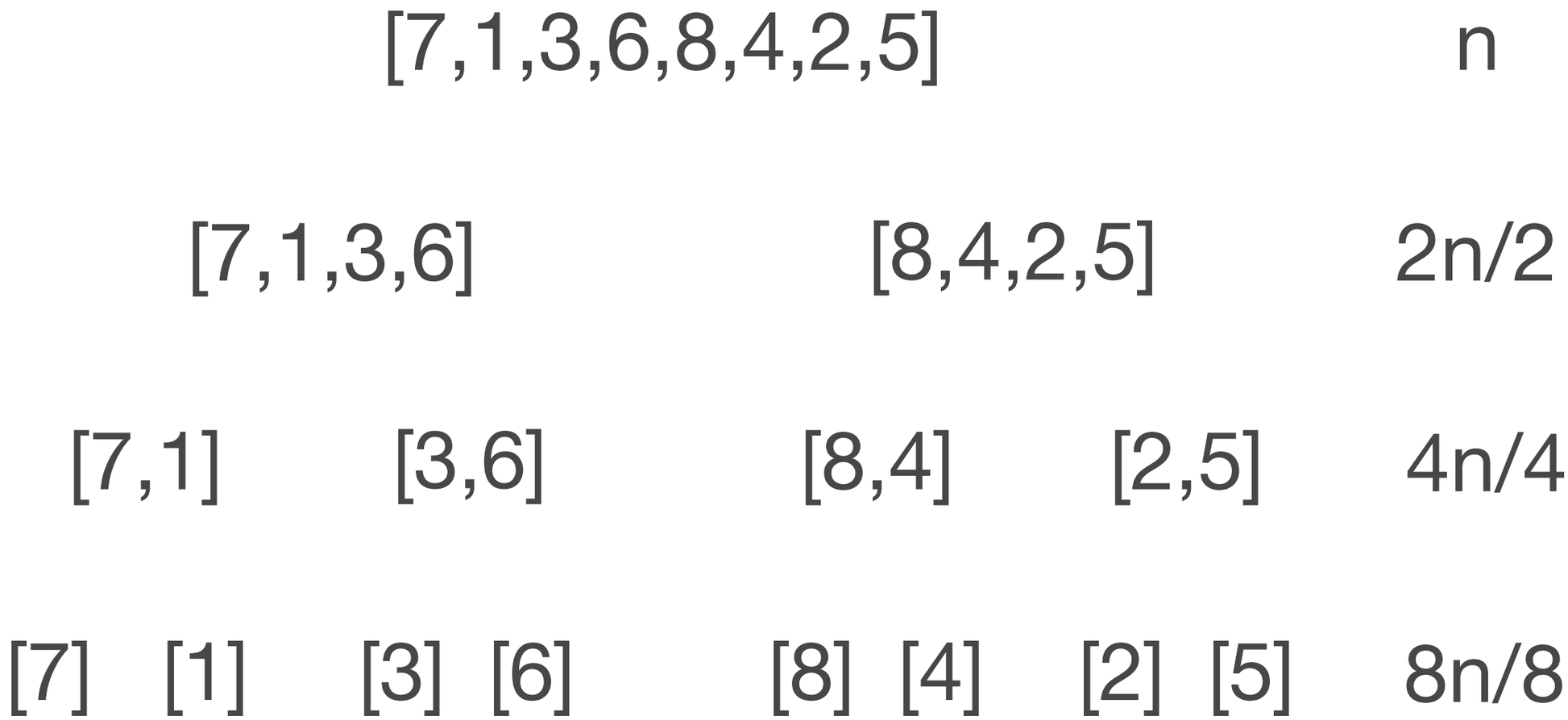
Work is O(n log n)



Work is O(n log n)



Work is O(n log n)



Span

```
fun mergesort (l : int list) : int list =
    case l of
        [] => []
      | [x] => [x]
      | _ => let val (pile1,pile2) = split l
             in
                 merge (mergesort pile1,
                        mergesort pile2)
             end
```

Span

```
fun mergesort (l : int list) : int list =
    case l of
        [] => []
      | [x] => [x]
      | _ => let val (pile1,pile2) = split l
             in
                 merge (mergesort pile1,
                        mergesort pile2)
             end
```

two recursive calls can be evaluated in parallel!

Span is

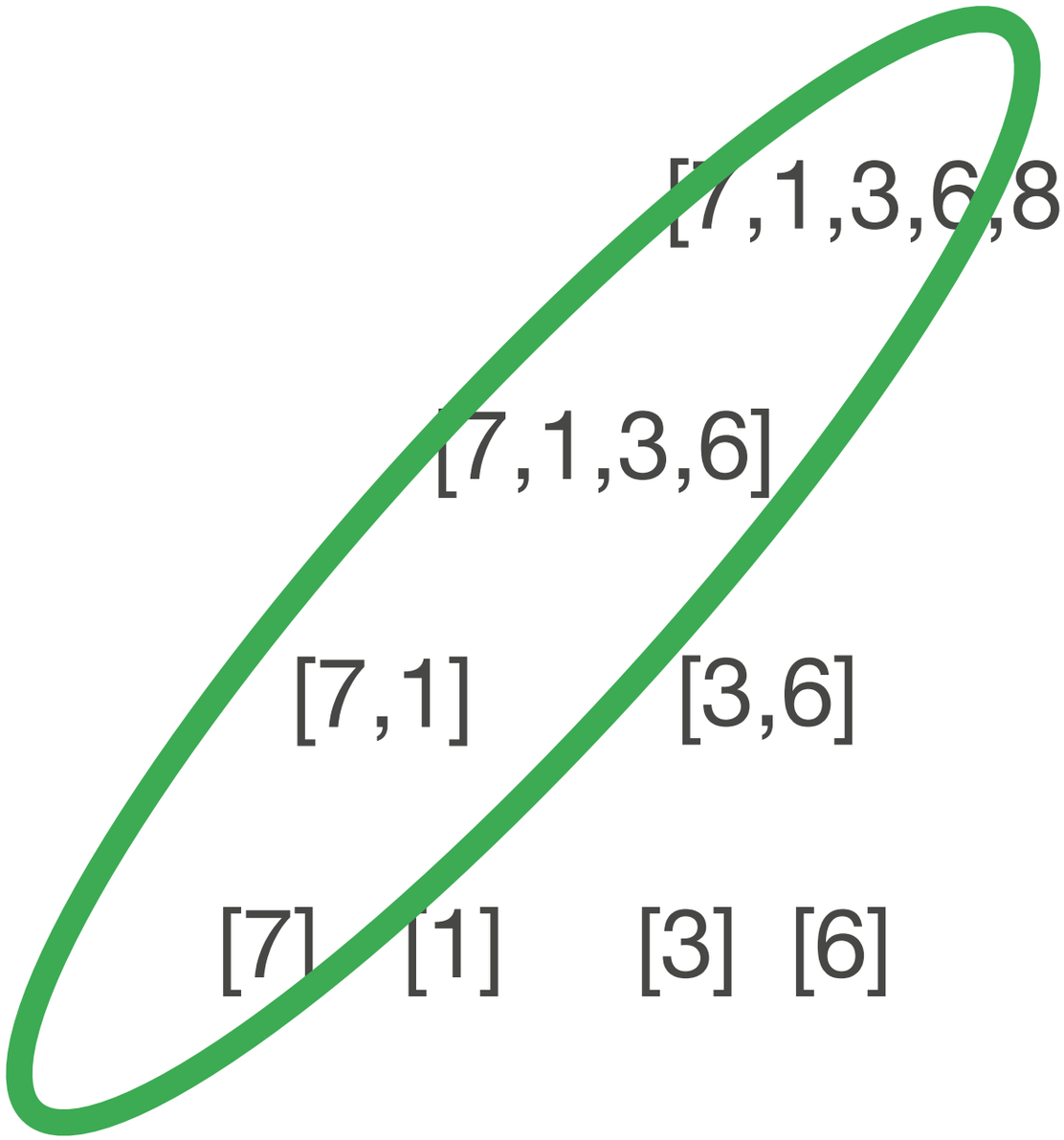
[7,1,3,6,8,4,2,5]

[7,1,3,6] [8,4,2,5]

[7,1] [3,6] [8,4] [2,5]

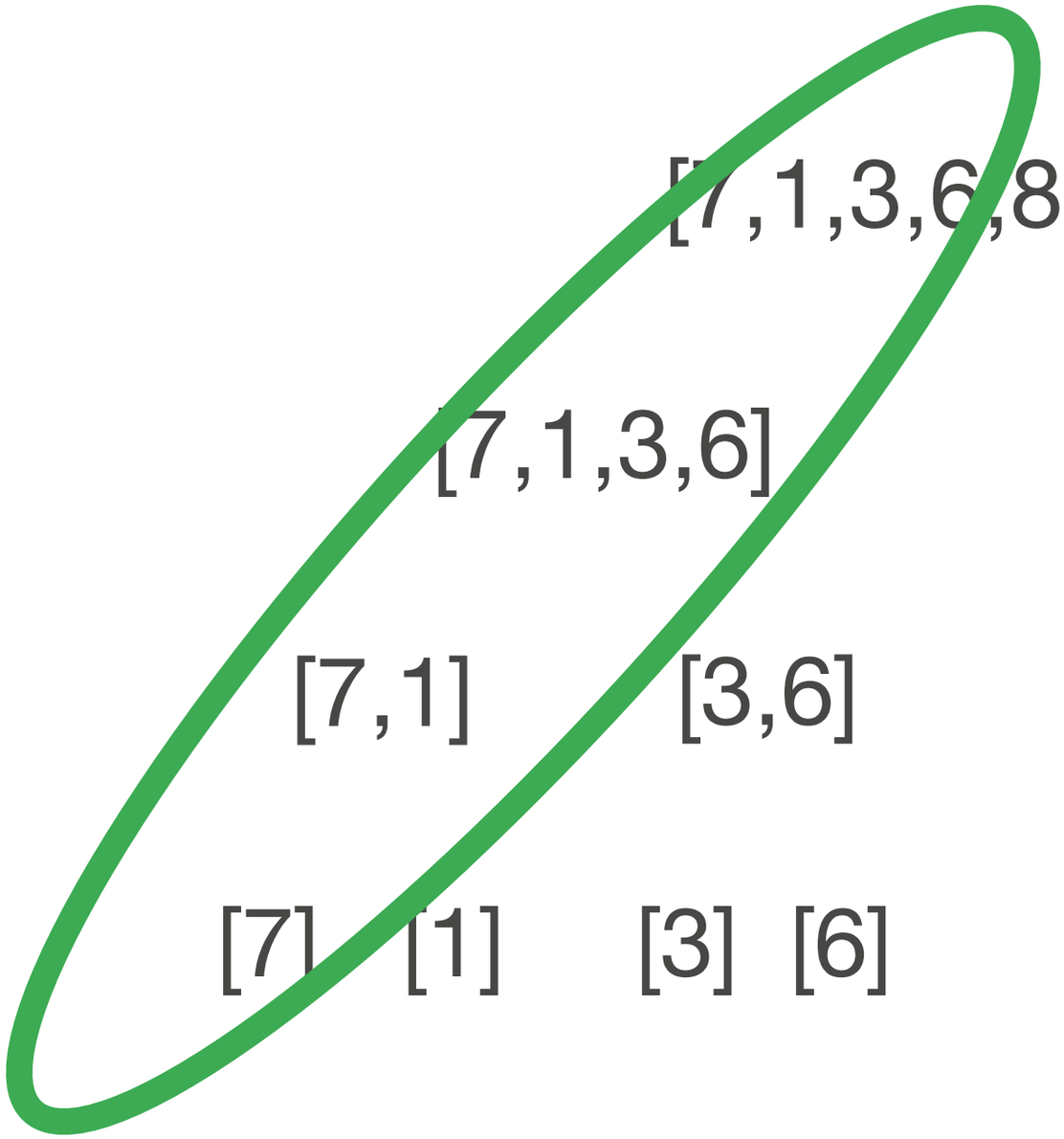
[7] [1] [3] [6] [8] [4] [2] [5]

Span is



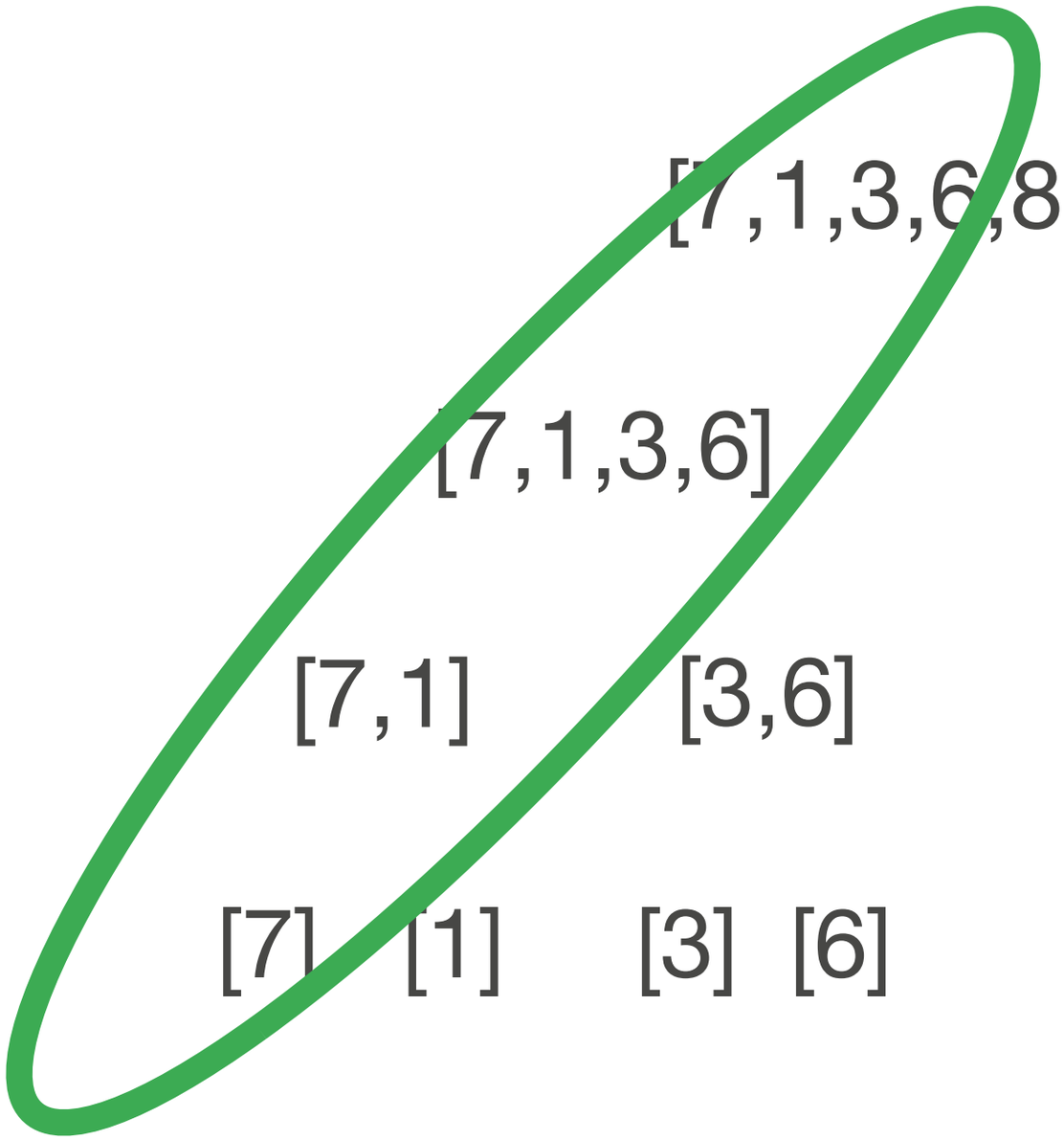
Span is

splitting and merging lists are sequential operations



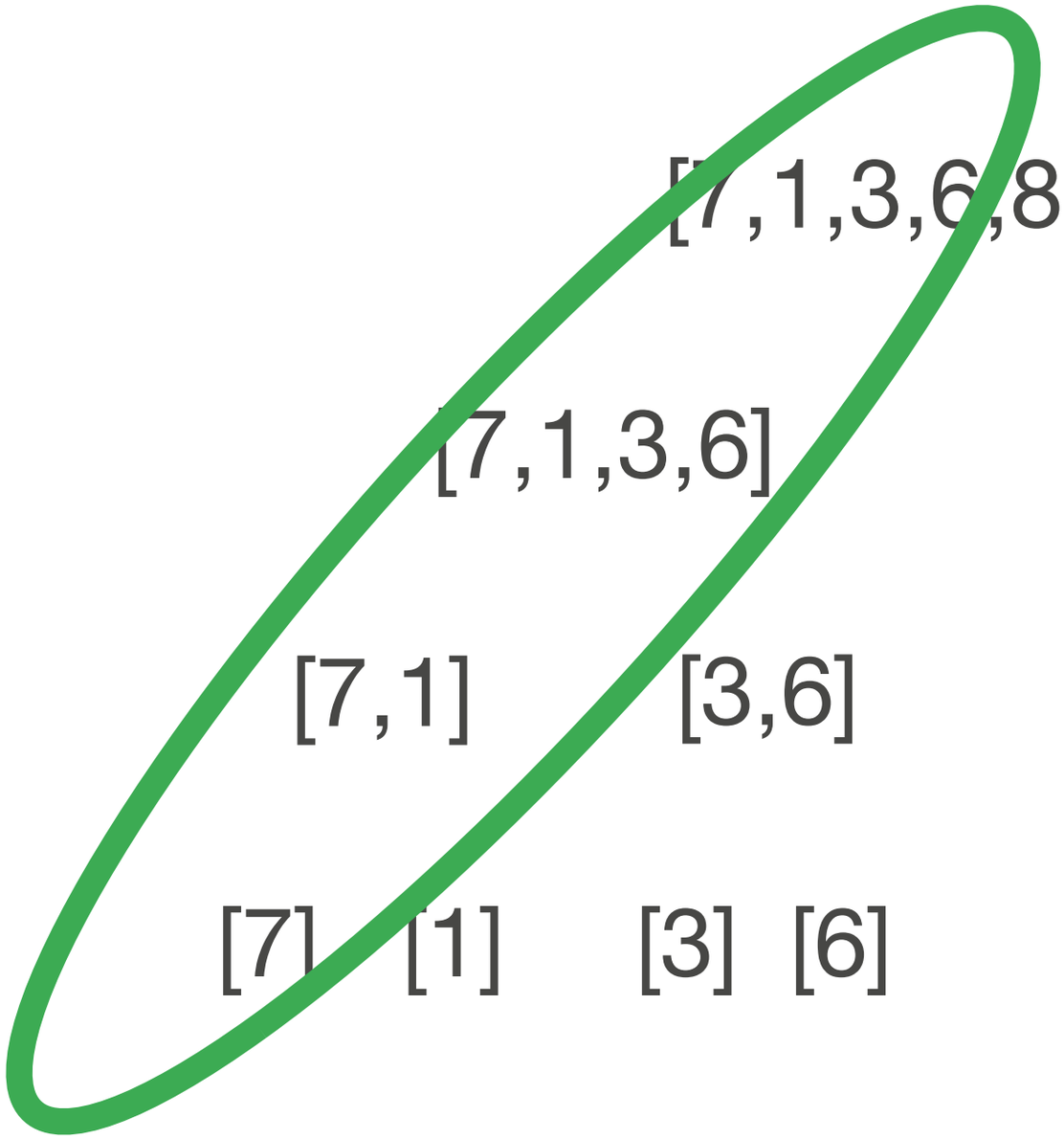
Span is

splitting and merging lists are sequential operations



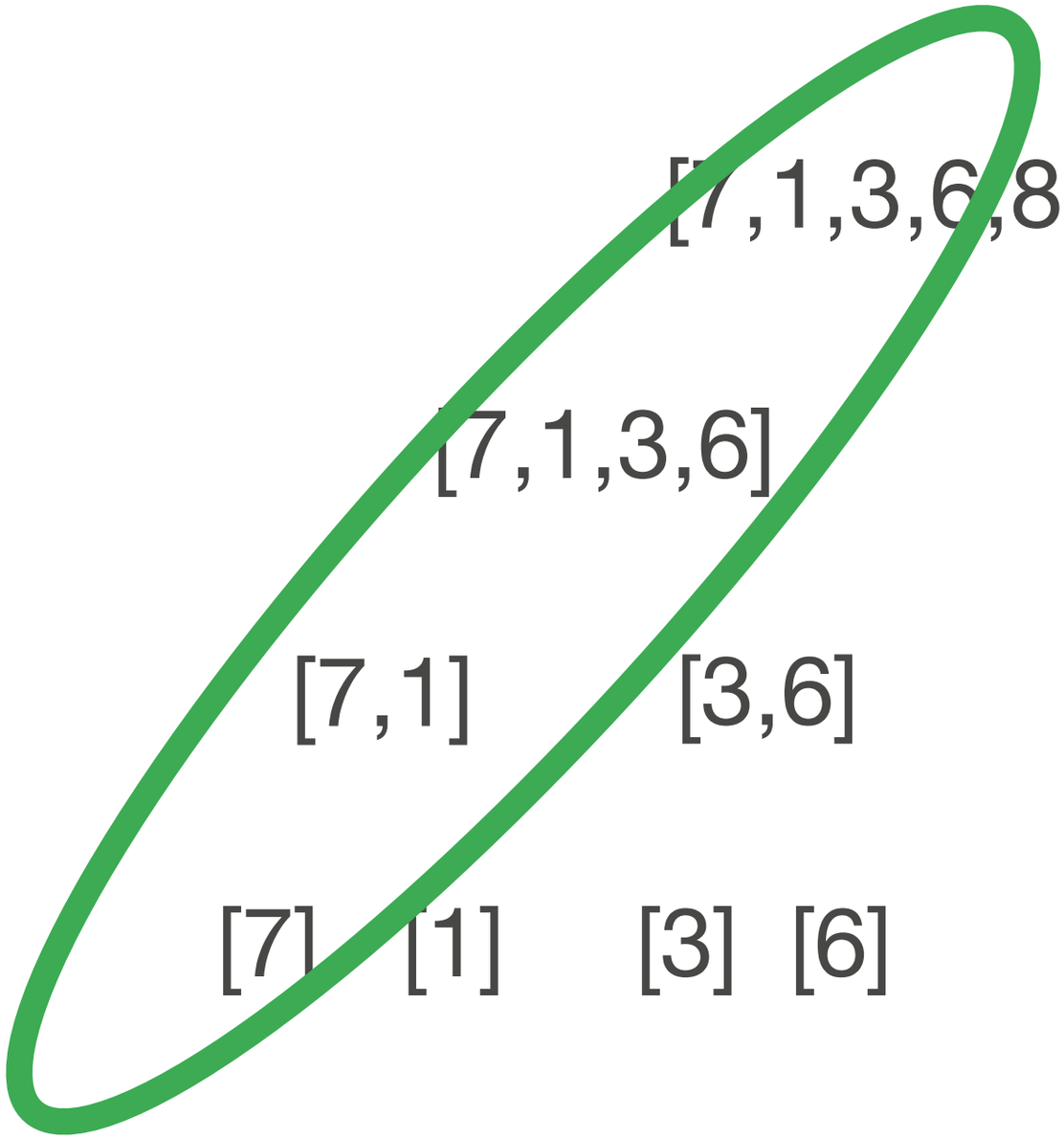
Span is

splitting and merging lists are sequential operations



Span is

splitting and merging lists are sequential operations



[7,1,3,6,8,4,2,5] n

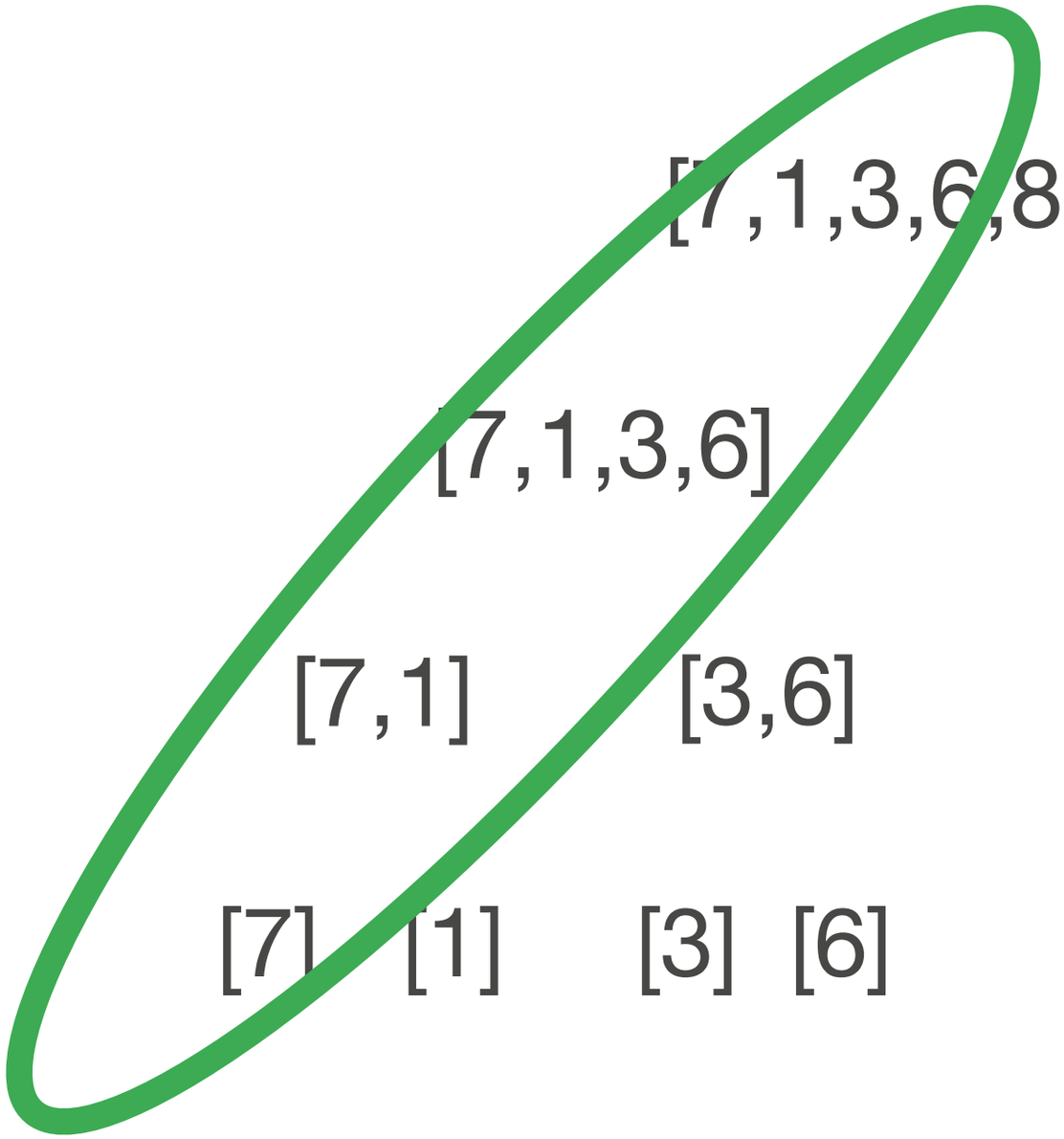
[7,1,3,6] [8,4,2,5] n/2

[7,1] [3,6] [8,4] [2,5] n/4

[7] [1] [3] [6] [8] [4] [2] [5]

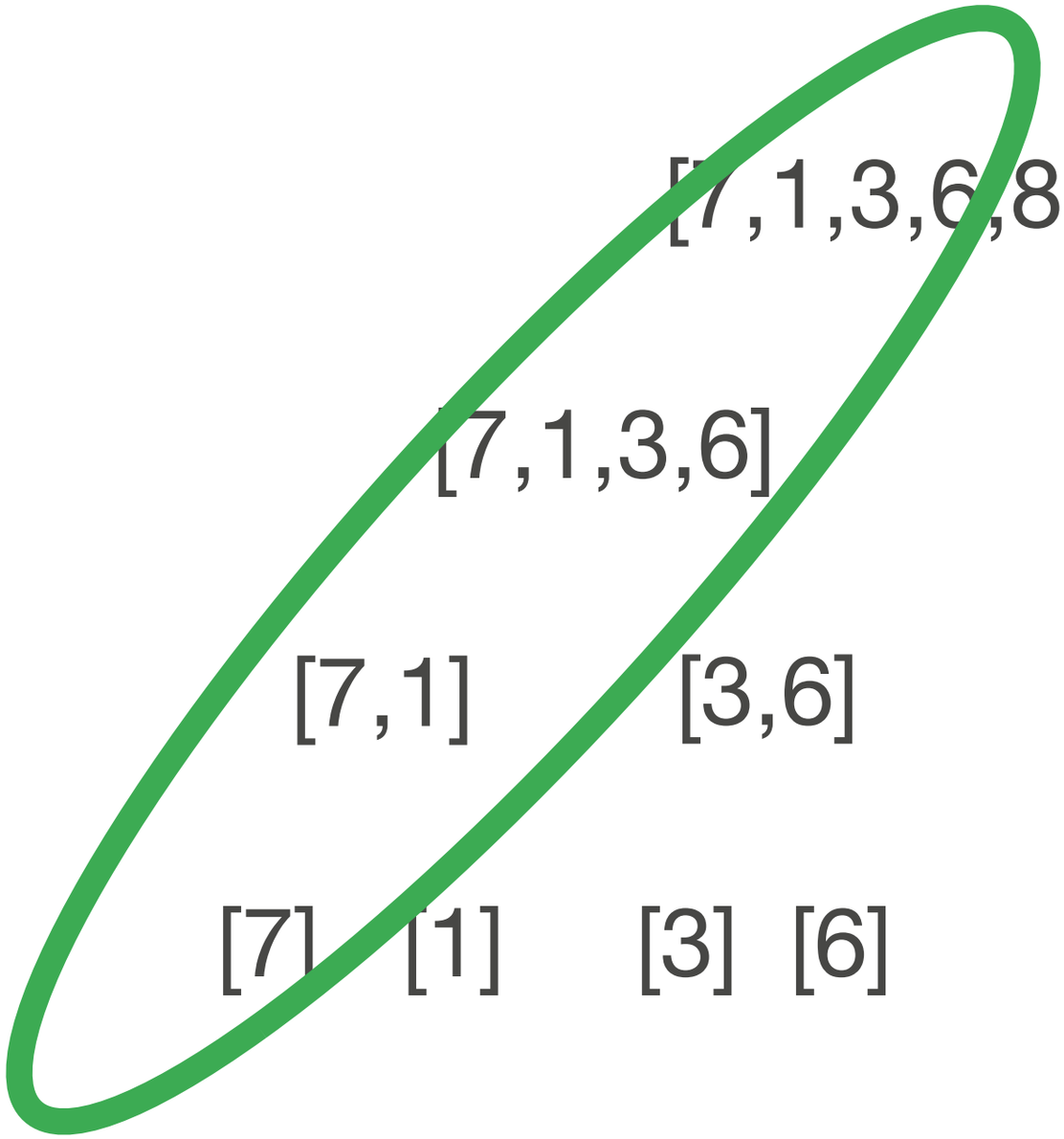
Span is

splitting and merging lists are sequential operations



Span is O(n)

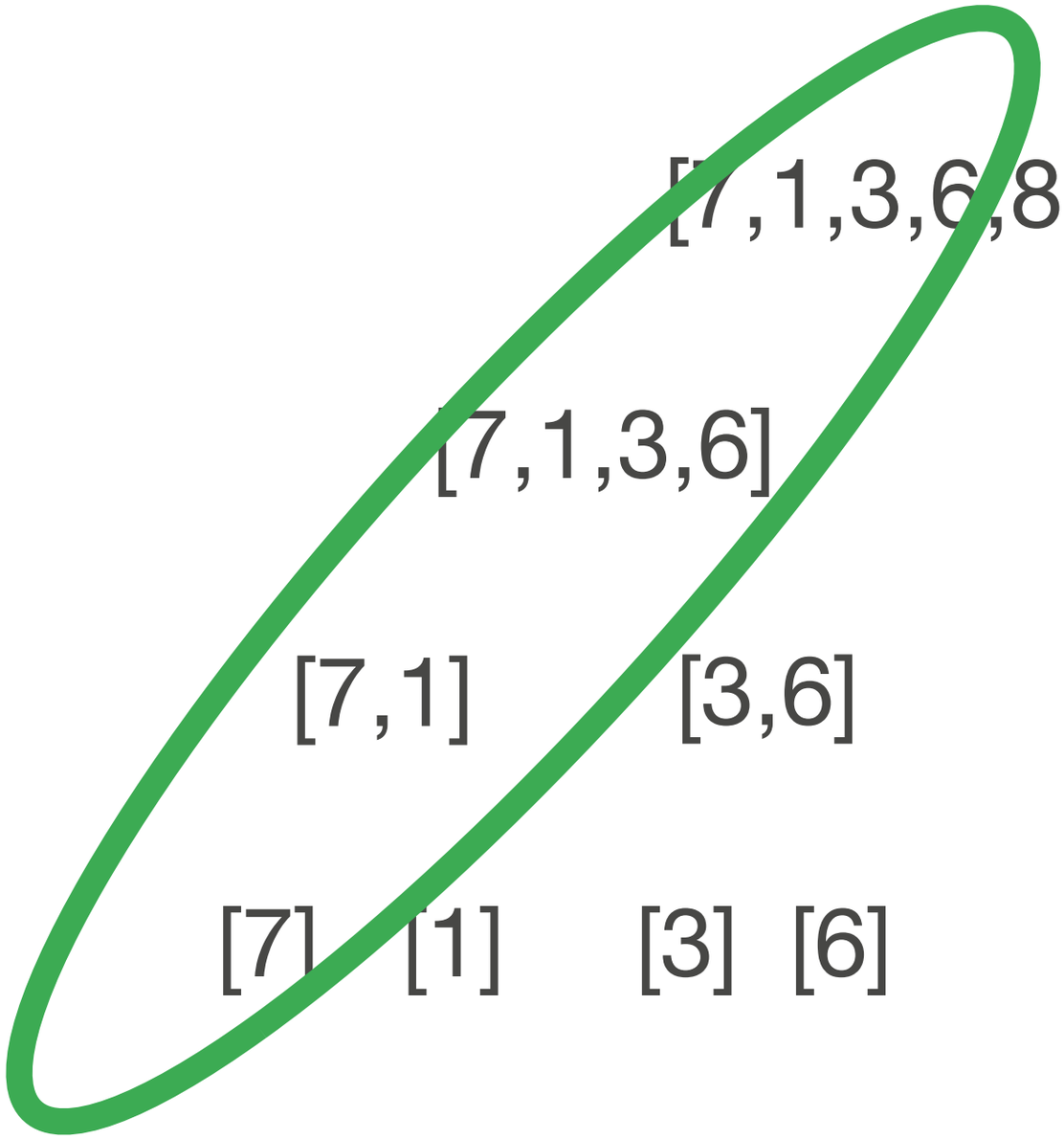
splitting and merging lists are sequential operations



[7,1,3,6,8,4,2,5]					n
[7,1,3,6]			[8,4,2,5]		n/2
[7,1]	[3,6]		[8,4]	[2,5]	n/4
[7] [1]	[3] [6]		[8] [4]	[2] [5]	n/8

Span is O(n)

splitting and merging lists are sequential operations



[7,1,3,6,8,4,2,5]				n
[7,1,3,6]		[8,4,2,5]		n/2
[7,1]	[3,6]	[8,4]	[2,5]	n/4
[7] [1]	[3] [6]	[8] [4]	[2] [5]	n/8

Parallelizability: log n

Data structures matter

Data structures matter

- Lists are bad for parallelism

Data structures matter

- Lists are bad for parallelism
- Trees are better

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
  | Node (l , x , r) =>
                  (insert(x,
                                merge (mergesort l ,
                                       mergesort r))
```

Span O((log n)^3) if balanced

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
  | Node (l , x , r) =>
                  (insert(x,
                                merge (mergesort l ,
                                       mergesort r))
```

Span O((log n)^3) if balanced

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
                  (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

splitting is given

Span O((log n)^3) if balanced

traversing entire tree multiplies span by log(n)

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
                  (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

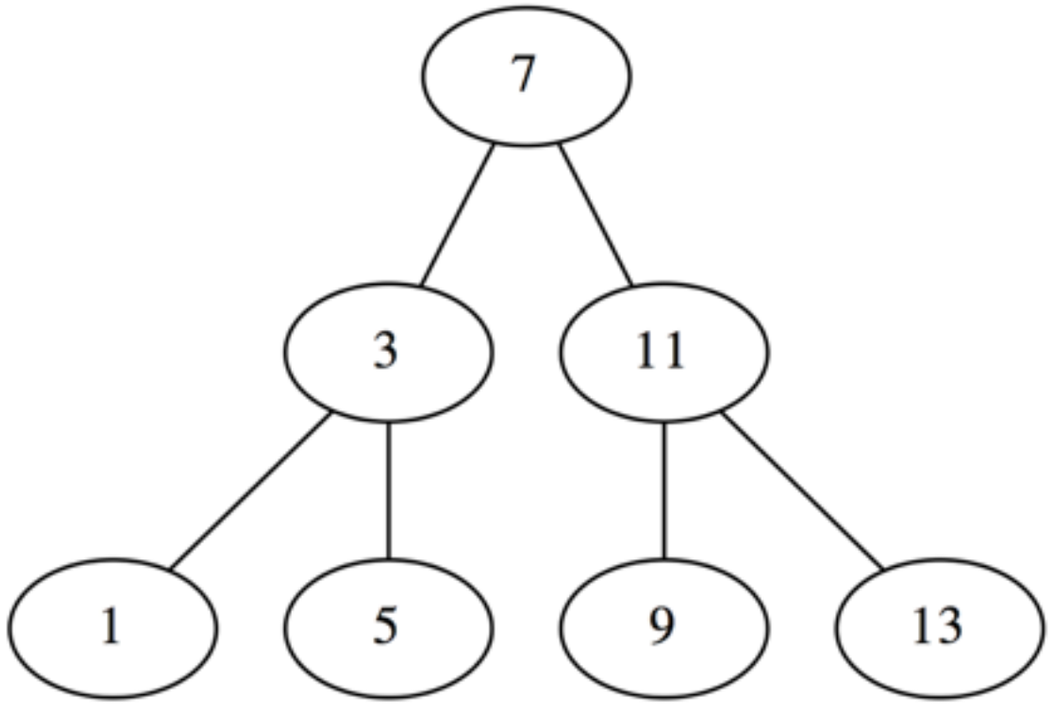
splitting is given

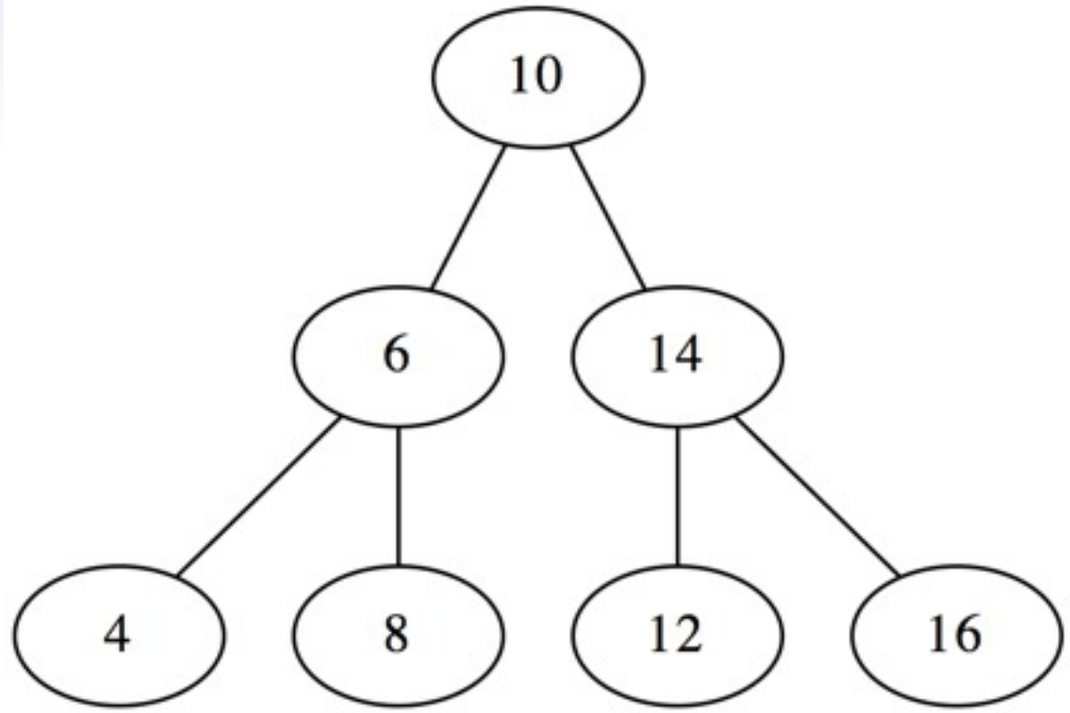
Span O((log n)^3) if balanced

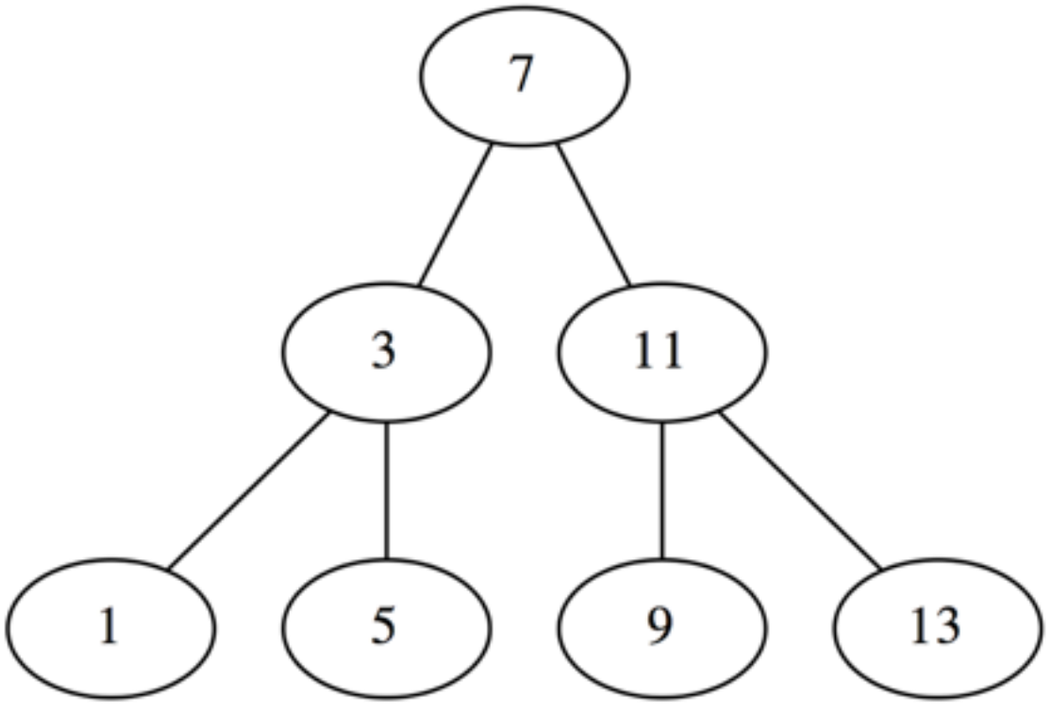
traversing entire tree multiplies span by log(n)

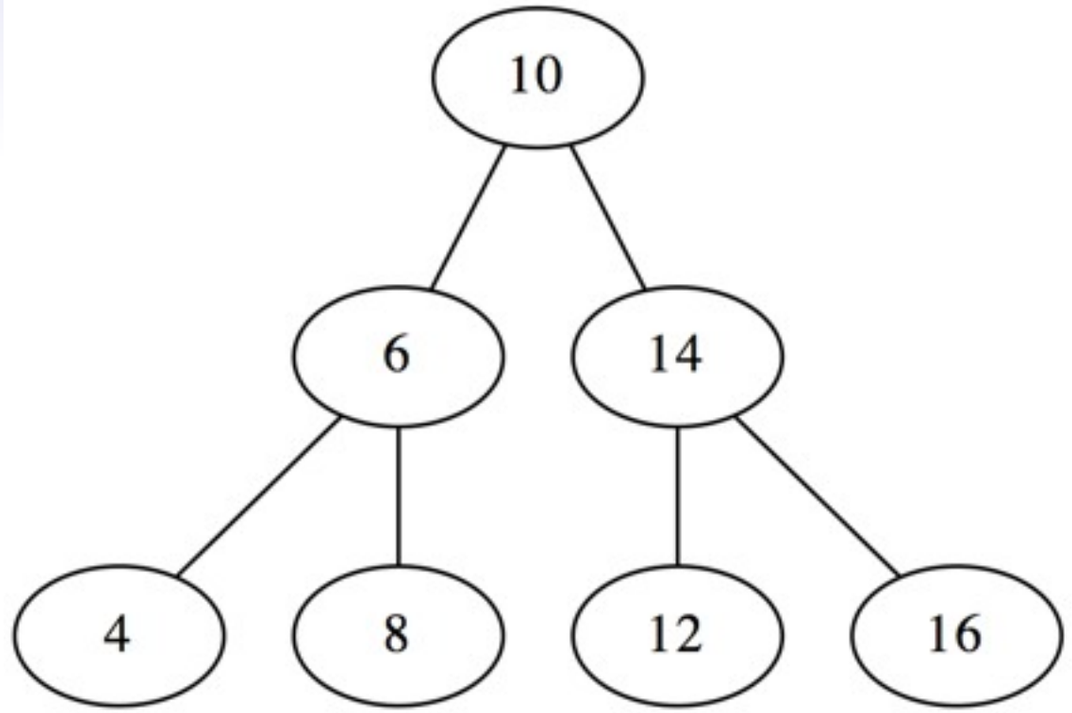
```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
                  (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

splitting is given

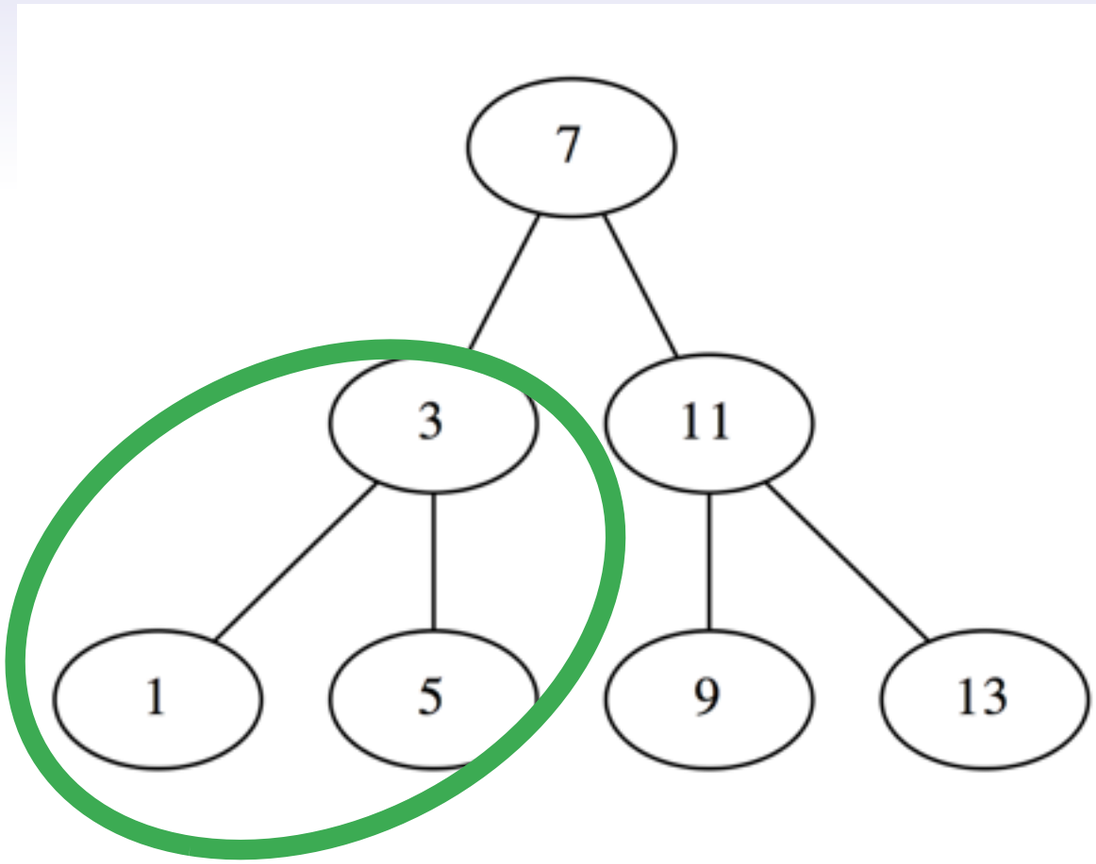

7
3
11
1
5
9
13

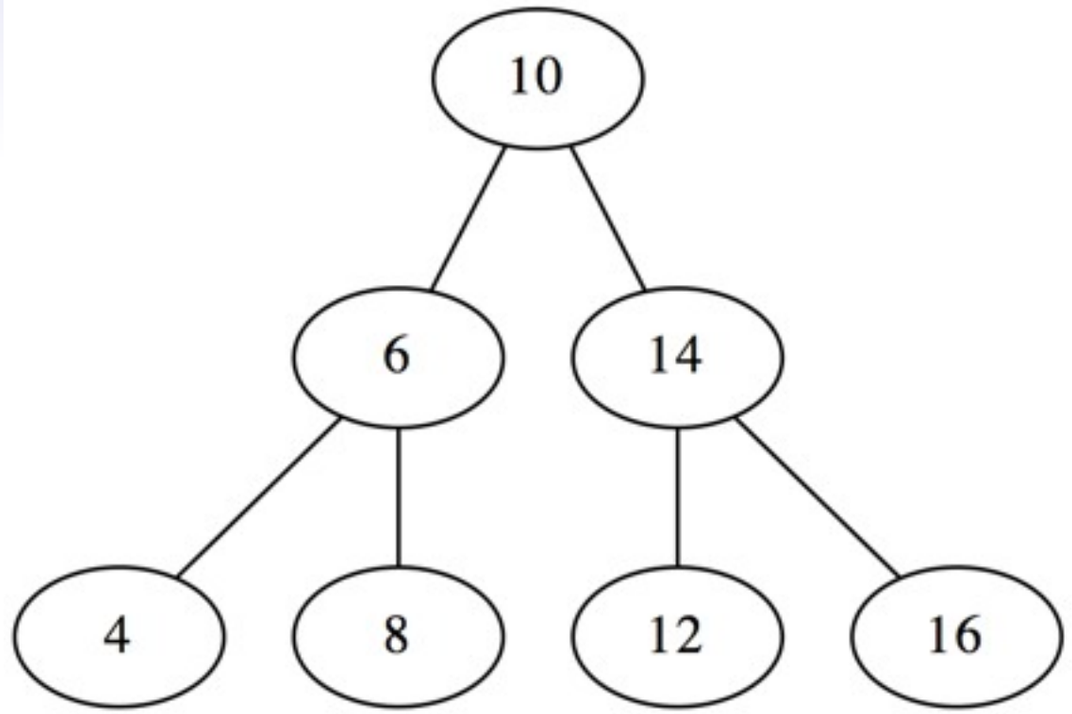

10
6
14
4
8
12
16


7
3
11
1
5
9
13

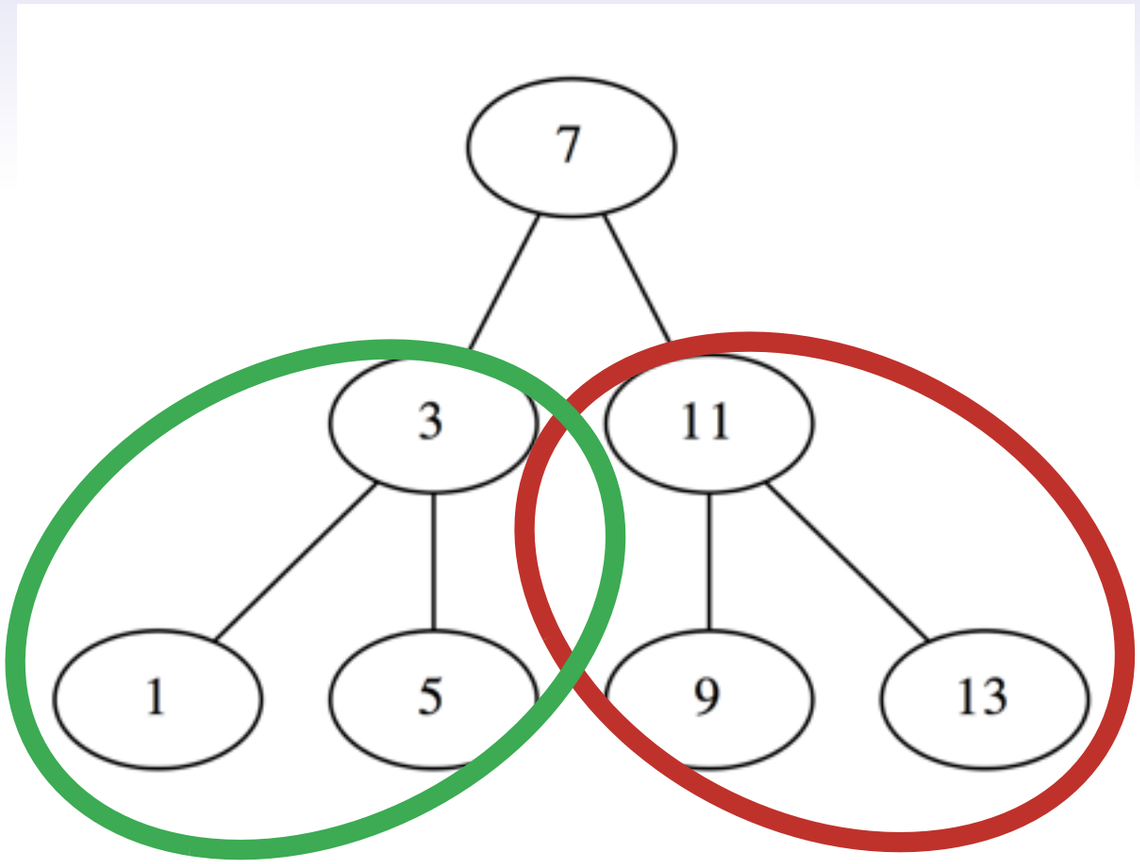

10
6
14
4
8
12
16

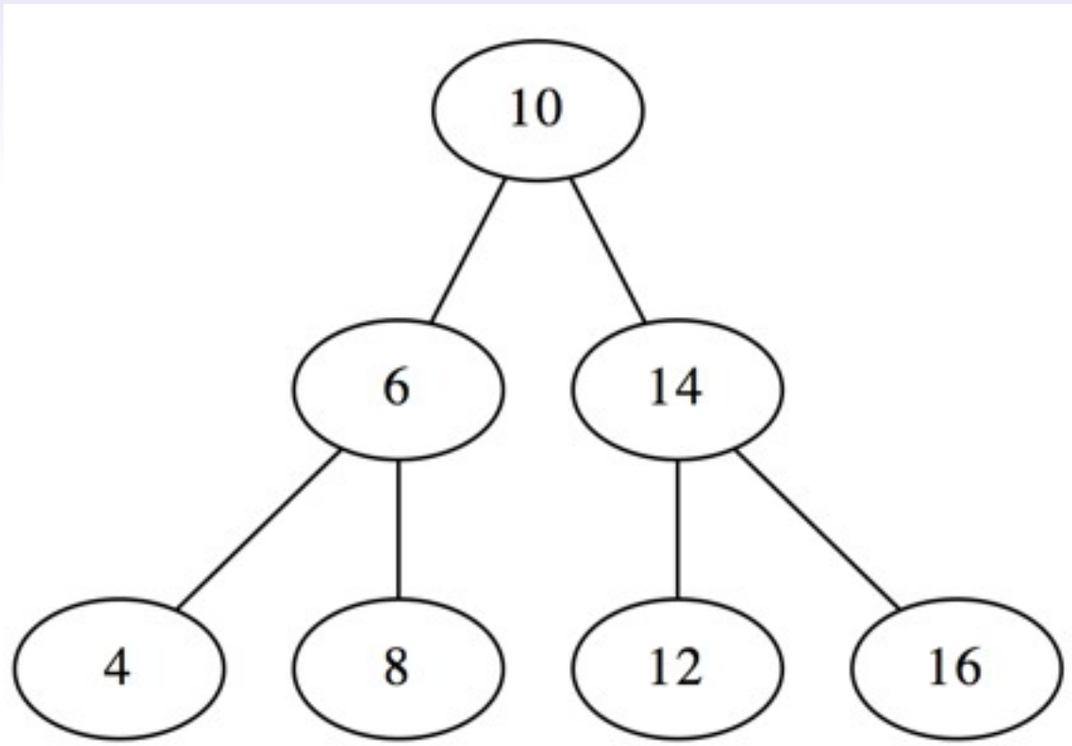

7


7
3
11
1
5
9
13

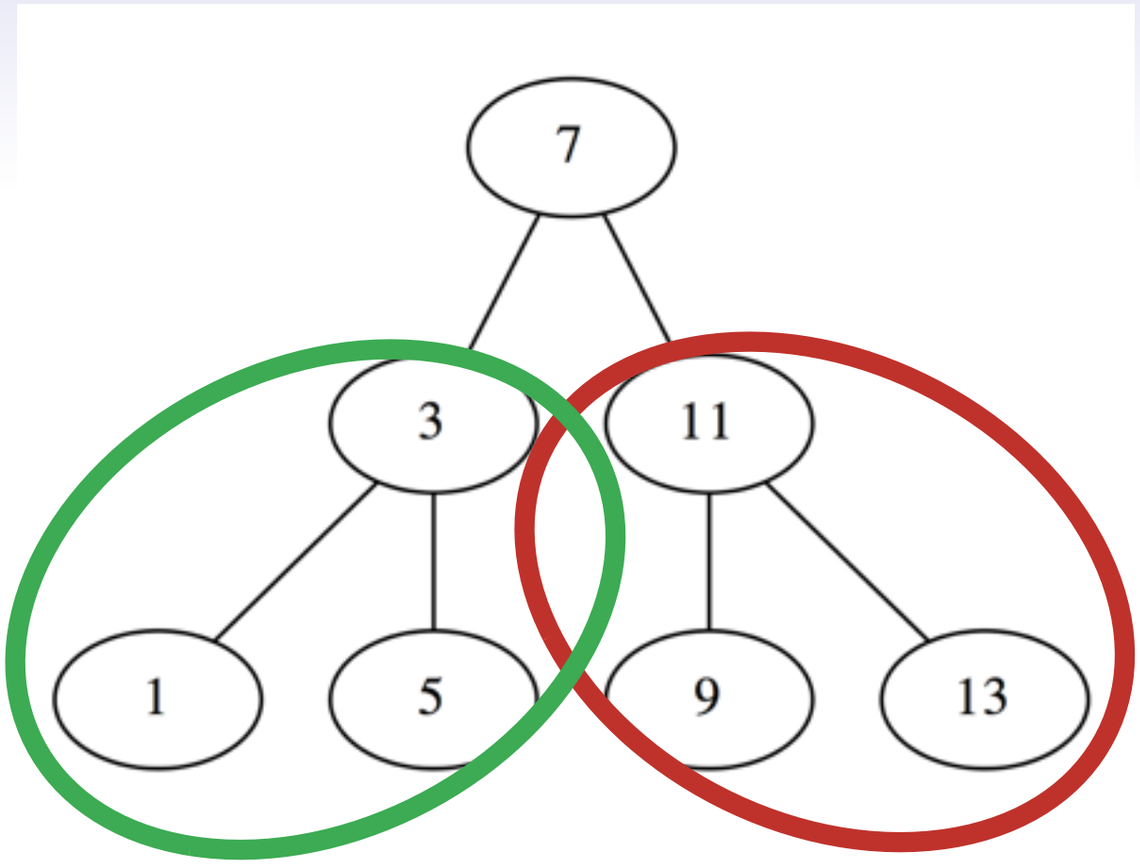

10
6
14
4
8
12
16

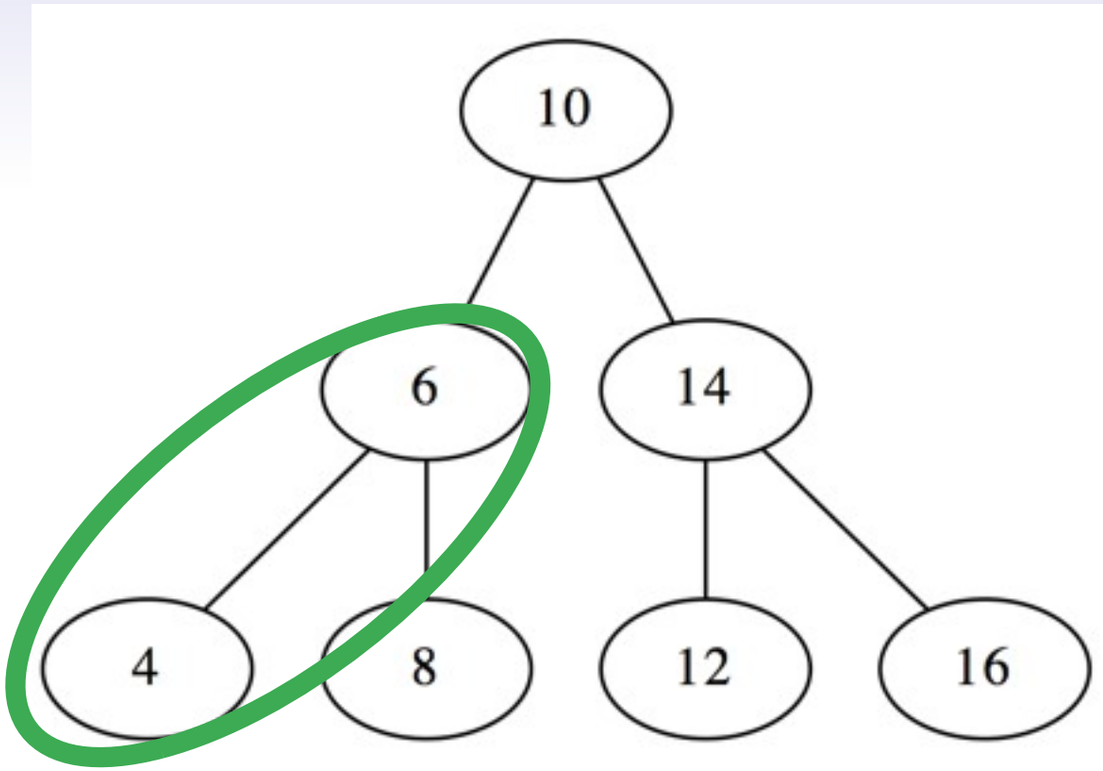

7


7
3
11
1
5
9
13

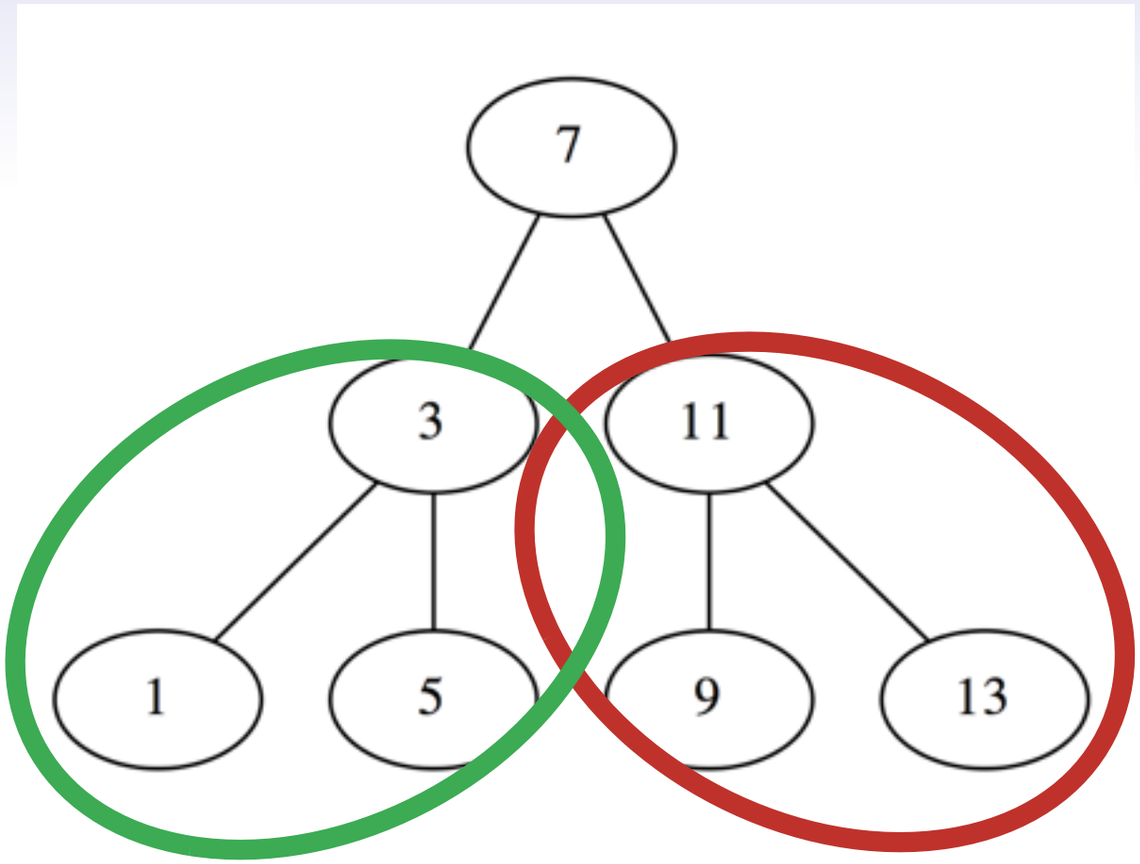

10
6
14
4
8
12
16

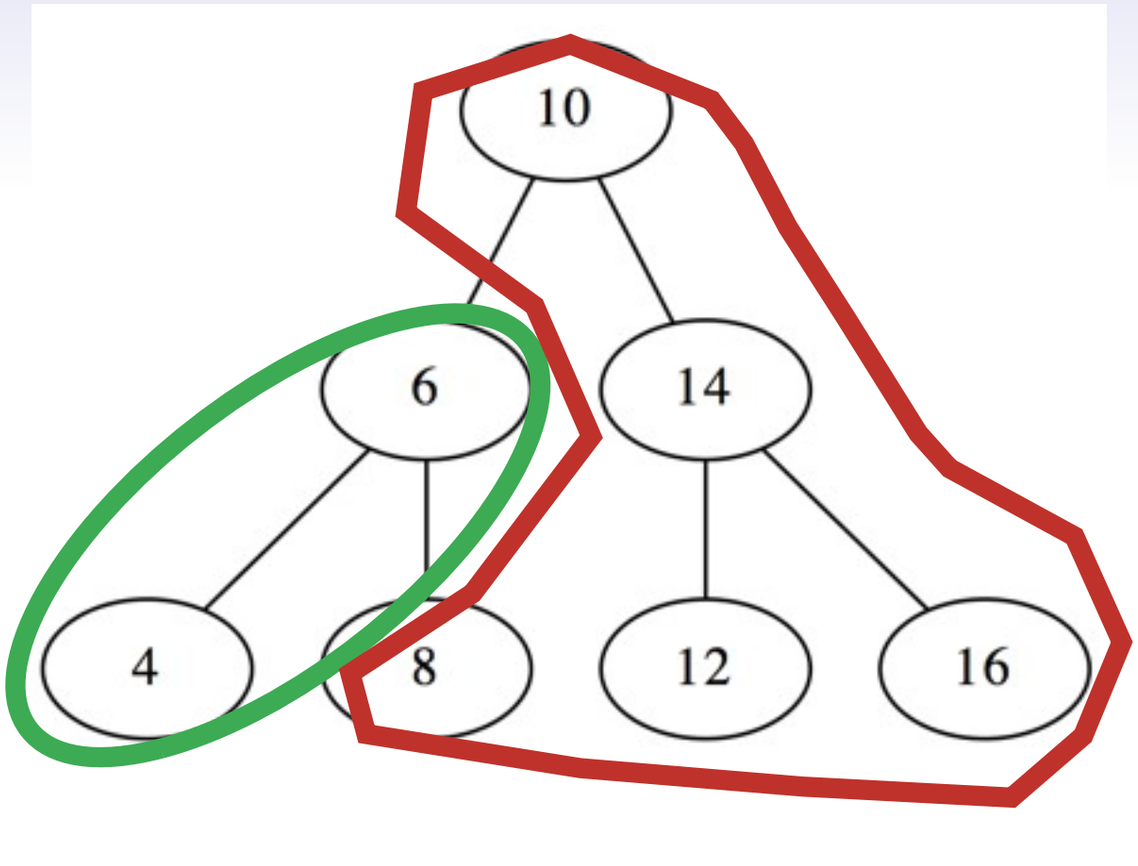

7


7
3
11
1
5
9
13

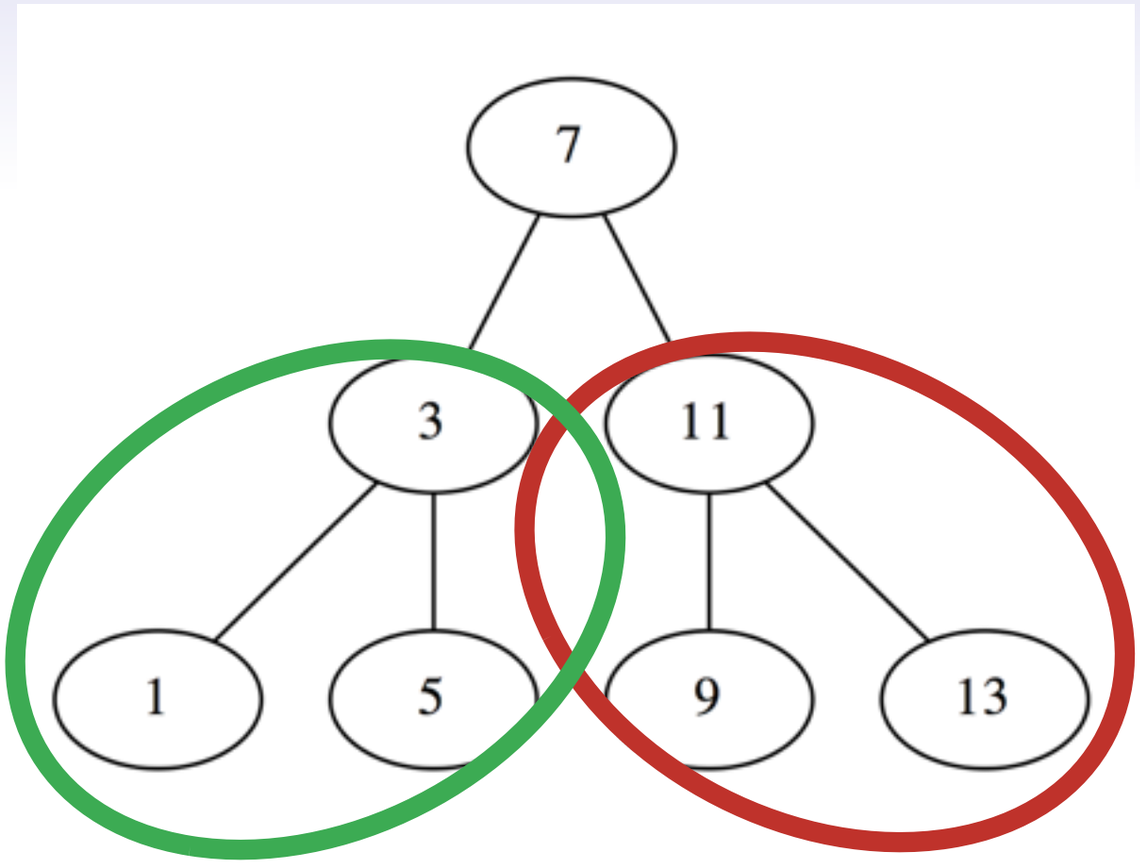

10
6
14
4
8
12
16

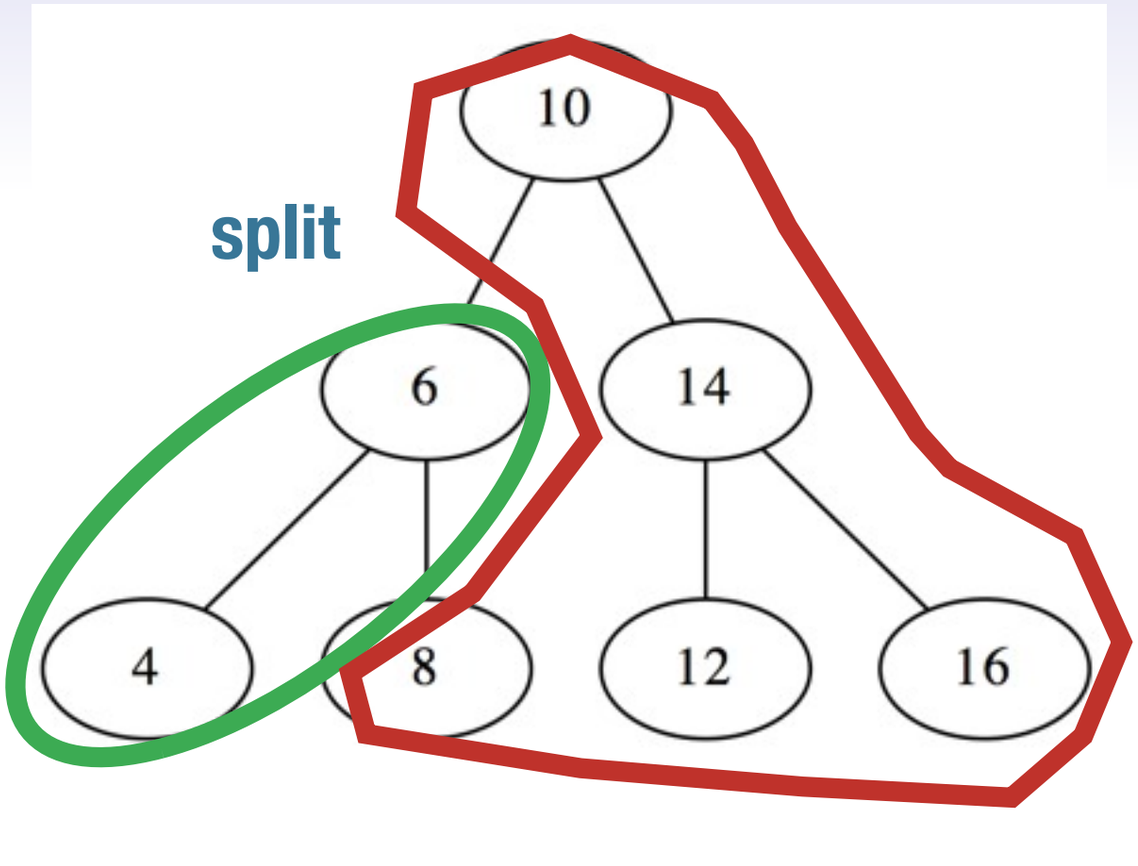

7


7
3
11
1
5
9
13

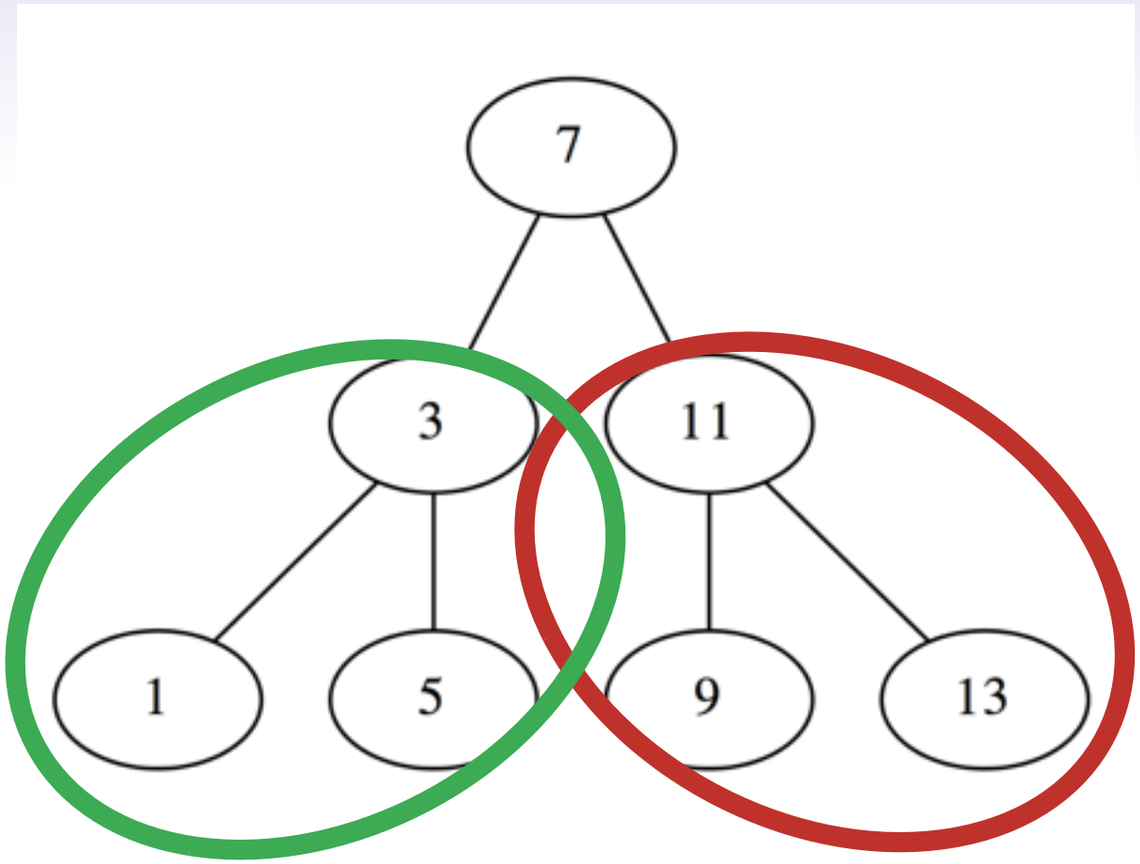

10
6
14
4
8
12
16

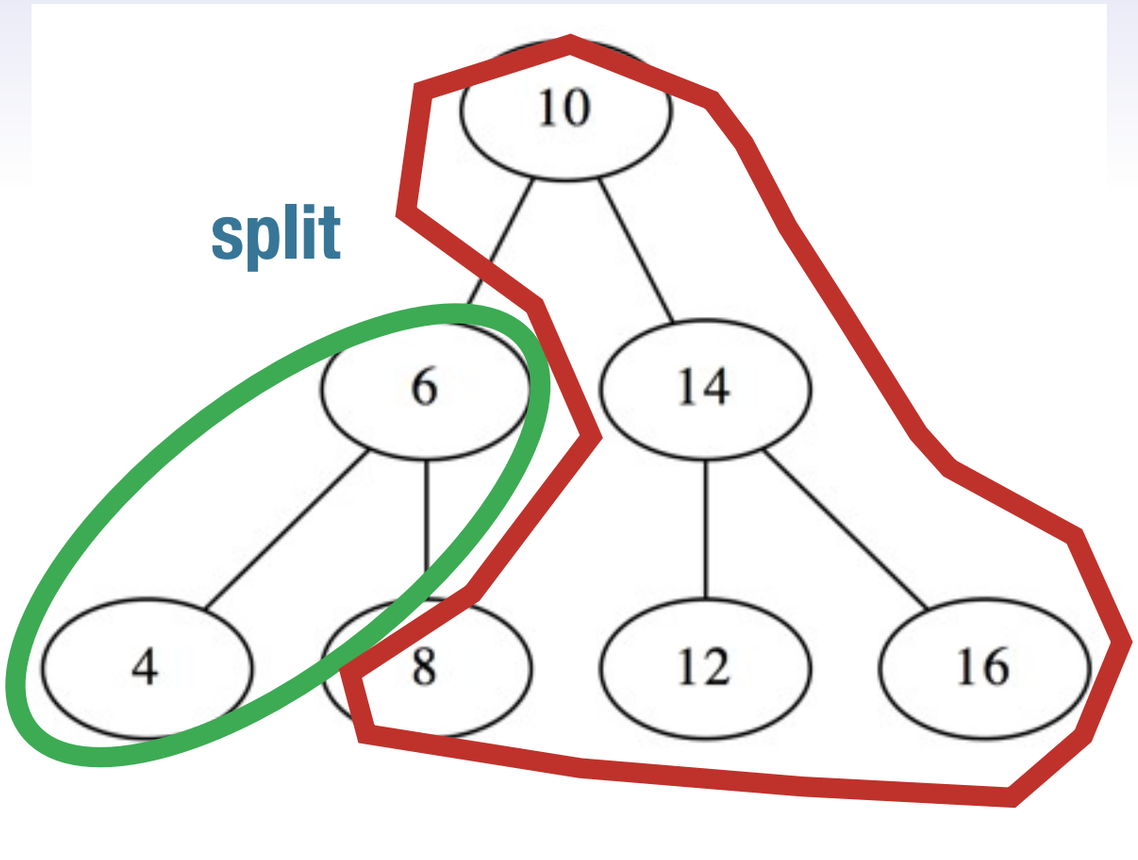

7

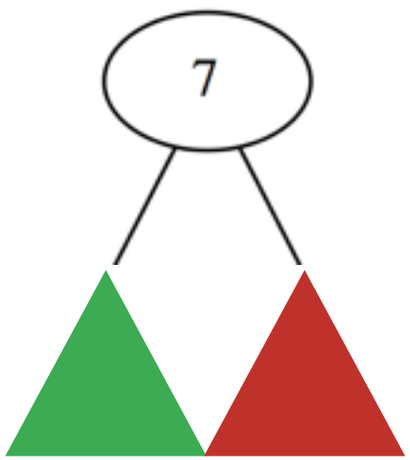

7
3
11
1
5
9
13

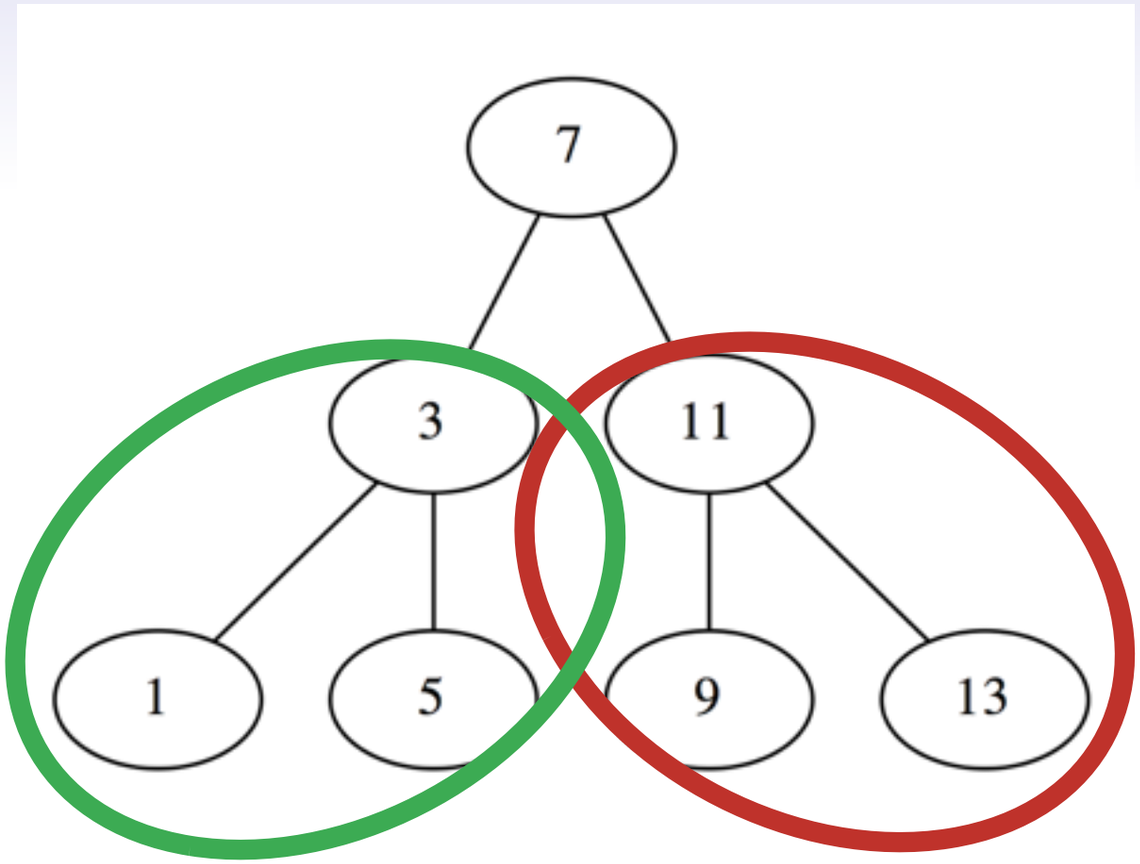

split
10
6
14
4
8
12
16

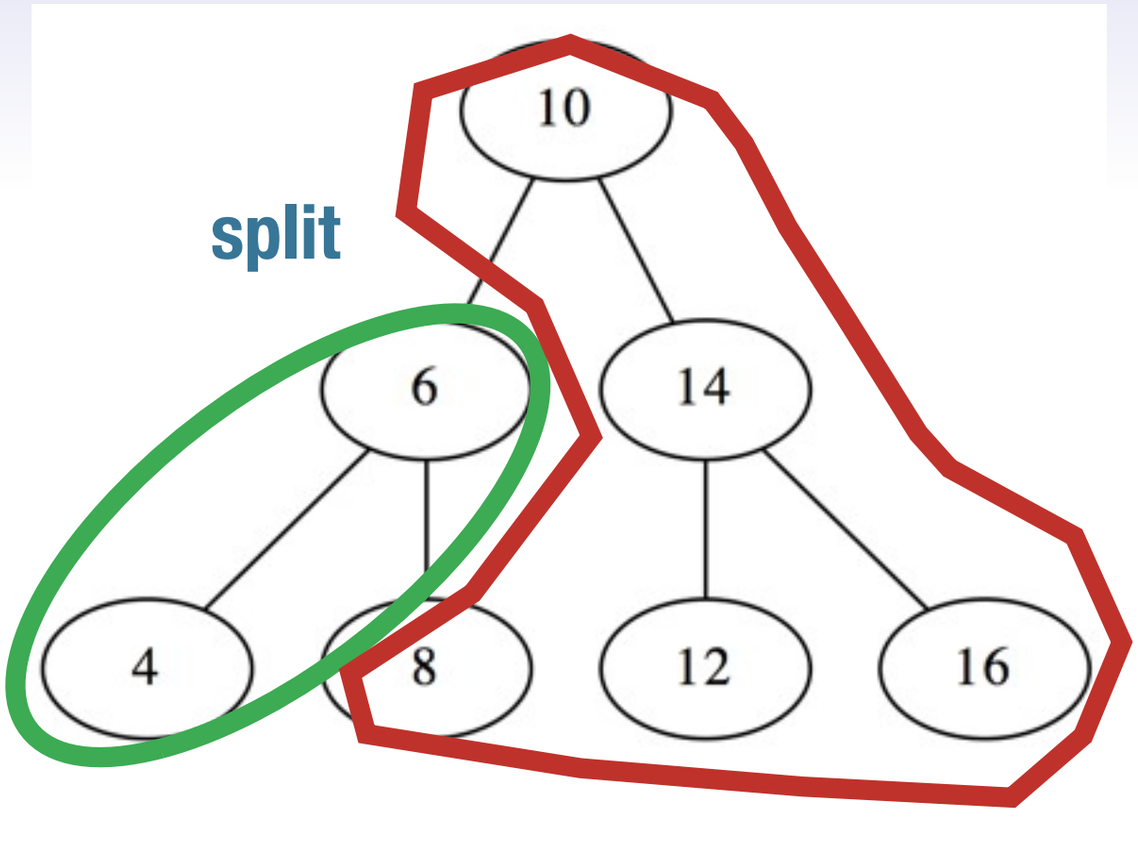

7

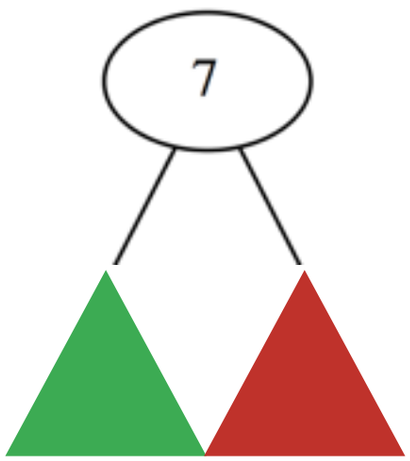

7
3
11
1
5
9
13


split
10
6
14
4
8
12
16


7







span O(depth t * depth t')

Span O((log n)^3) if balanced

traversing entire tree multiplies span by log(n)

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
                  (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

splitting is given

Span O((log n)^3) if balanced

traversing entire tree multiplies span by log(n)

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
                  (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

splitting is given

O((log n)^2) if recursive results are balanced

Span O((log n)^3) if balanced

traversing entire tree multiplies span by log(n)

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
        rebalance (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

splitting is given

O((log n)^2) if recursive results are balanced

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
        rebalance (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
        rebalance (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

recurs on an unbalanced tree!

why doesn't it ruin the work and span?

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
        rebalance (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

recurs on an unbalanced tree!
why doesn't it ruin the work and span?

rebalance: n work and $(\log n)^2$ span **if** depth(t) = c*log(n)

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
        rebalance (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

recurs on an unbalanced tree!
why doesn't it ruin the work and span?

rebalance: n work and $(\log n)^2$ span **if** depth(t) = c*log(n)

depth(merge(t,t')) ≤ depth(t) + depth(t')

```
fun mergesort (t : tree) : tree =
  case t of
       Empty => Empty
     | Node (l , x , r) =>
        rebalance (insert(x,
                          merge (mergesort l ,
                                 mergesort r))
```

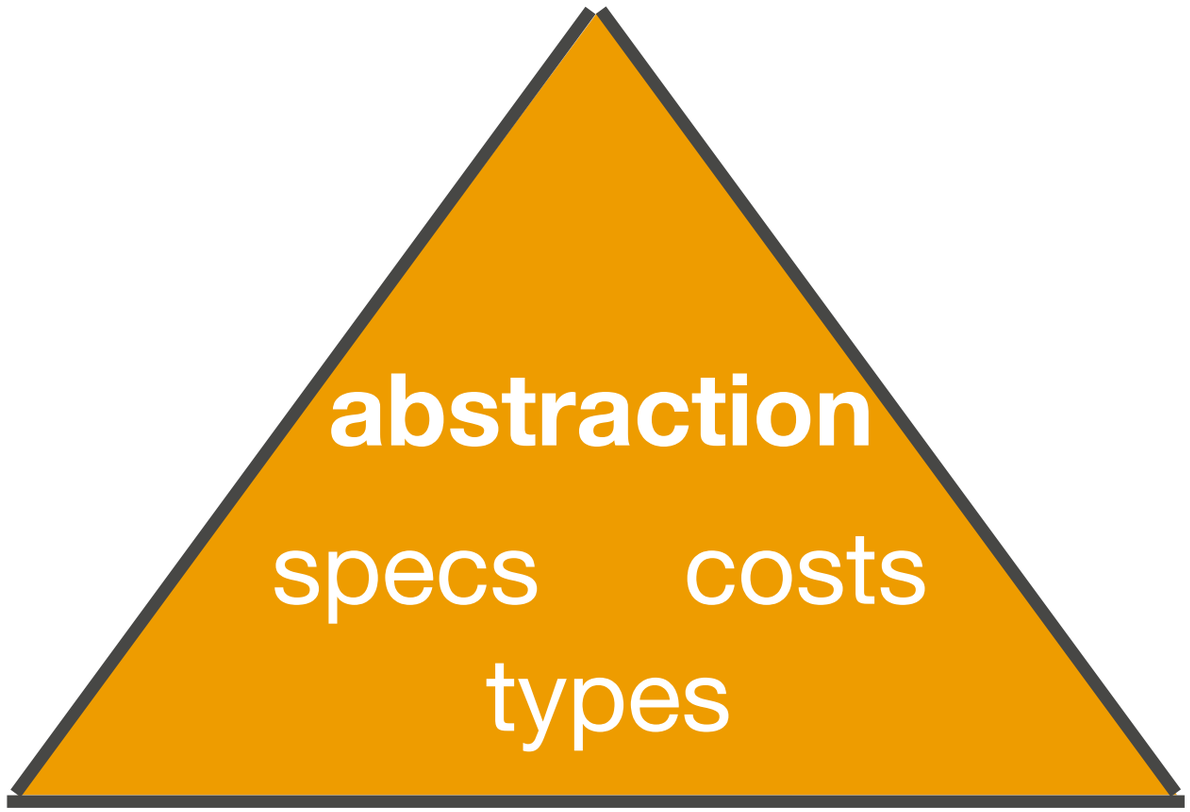
recurs on an unbalanced tree!
why doesn't it ruin the work and span?

rebalance: n work and $(\log n)^2$ span **if** depth(t) = c*log(n)

depth(merge(t,t')) ≤ depth(t) + depth(t')

Parallelizability: $n/(\log n)^2$ better than log n

write code



reason about behavior

reason about efficiency

Data structures matter

- Lists are bad for parallelism
- Trees are better
- Sequences are even better

n-way parallelism

`Seq.map f <x1,…,xn> = <f x1,…,f xn>`

Work is the sum of the works of `f xi`

Span is the max of the works of `f xi`

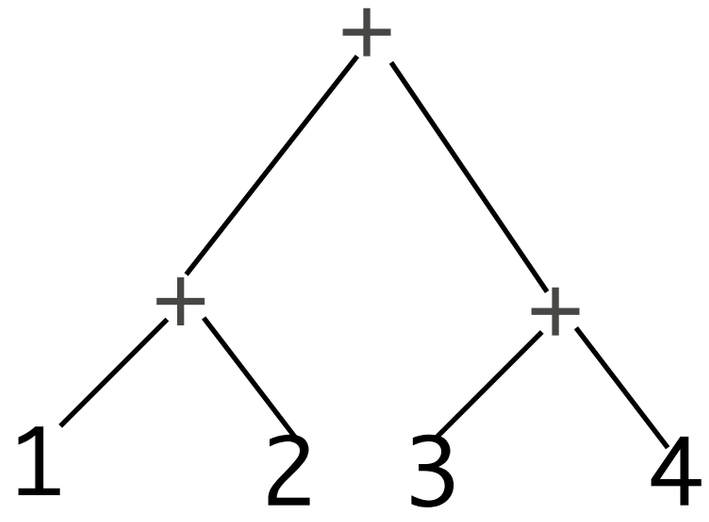
N-way parallelism

`Seq.reduce` $\oplus$ `0` $\langle x_1,\ldots,x_n\rangle = x_1 \oplus x_2 \oplus \ldots \oplus x_n$

Work is the sum of the works of $\oplus$'s

Span is given by balanced tree (log):

`reduce + 0 <1,2,3,4> =`



n-body simulation

[initial conditions from Princeton COS 126]

n-body simulation

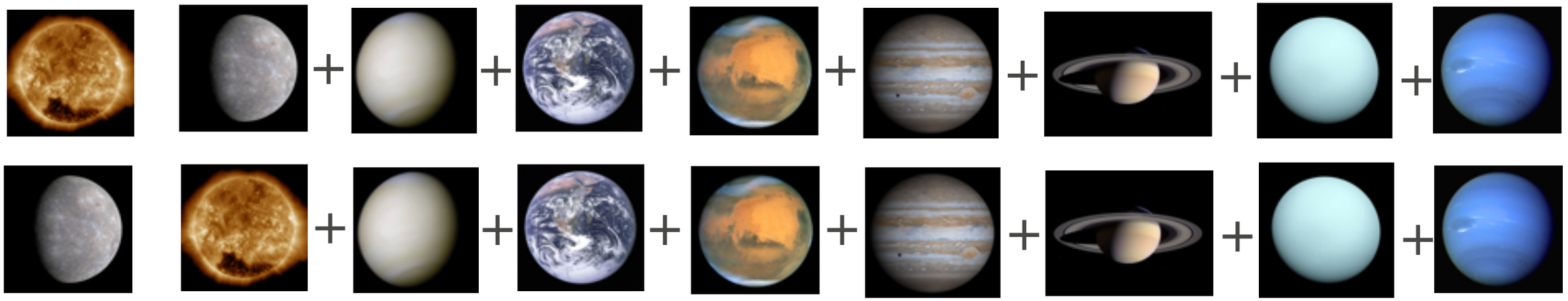
force on each body is sum of forces due to all other bodies

n-body simulation

force on each body is sum of forces due to all other bodies

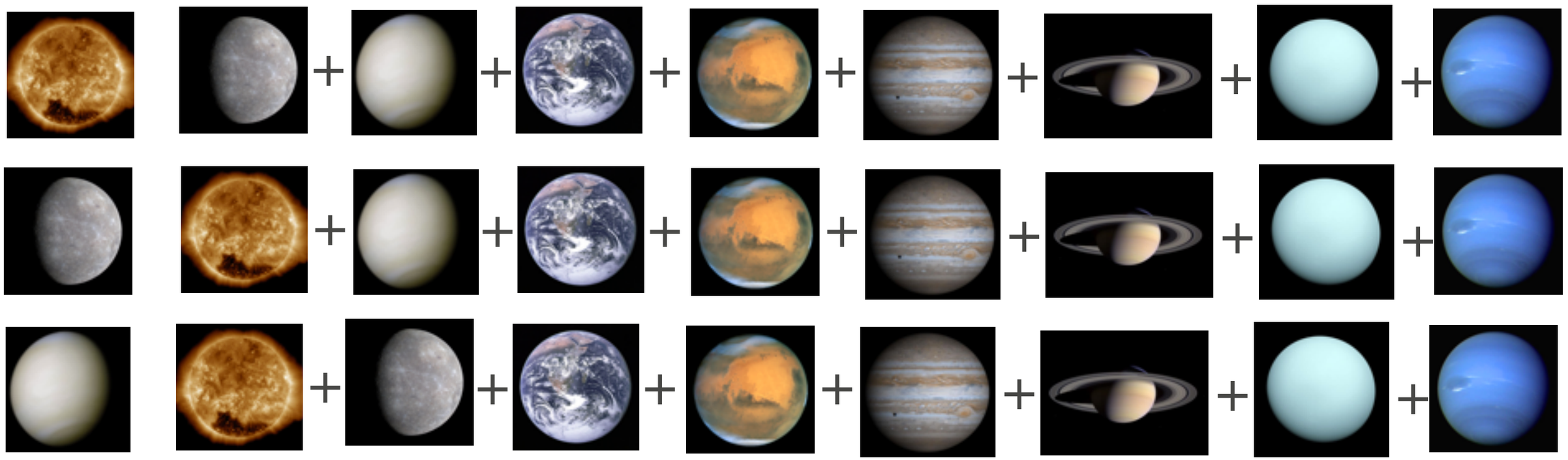
n-body simulation

force on each body is sum of forces due to all other bodies



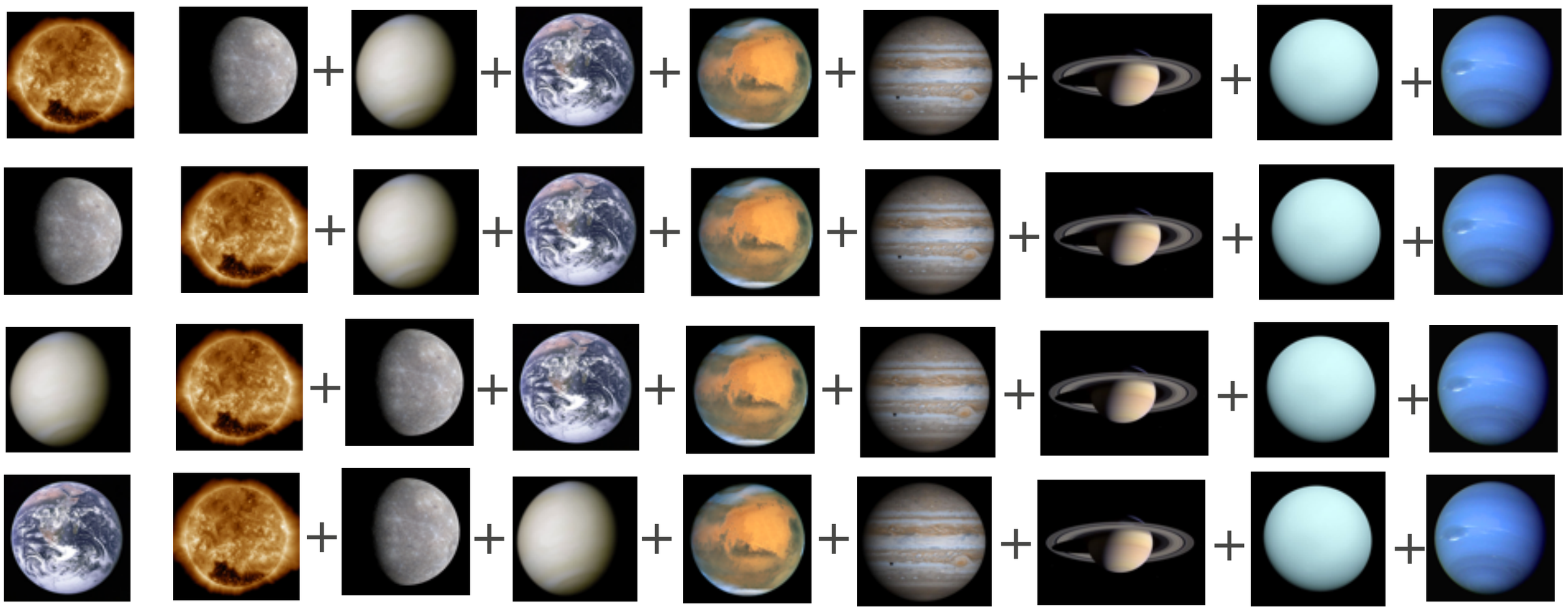
n-body simulation

force on each body is sum of forces due to all other bodies



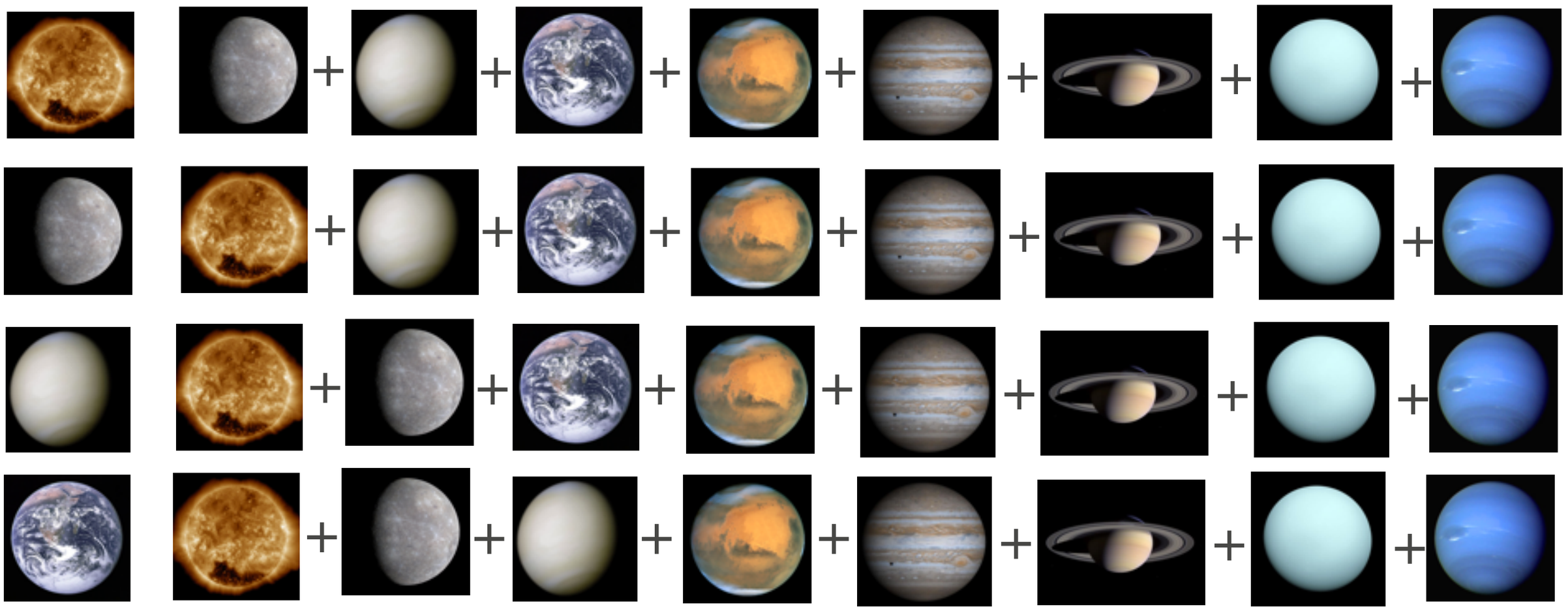
n-body simulation

force on each body is sum of forces due to all other bodies



n-body simulation

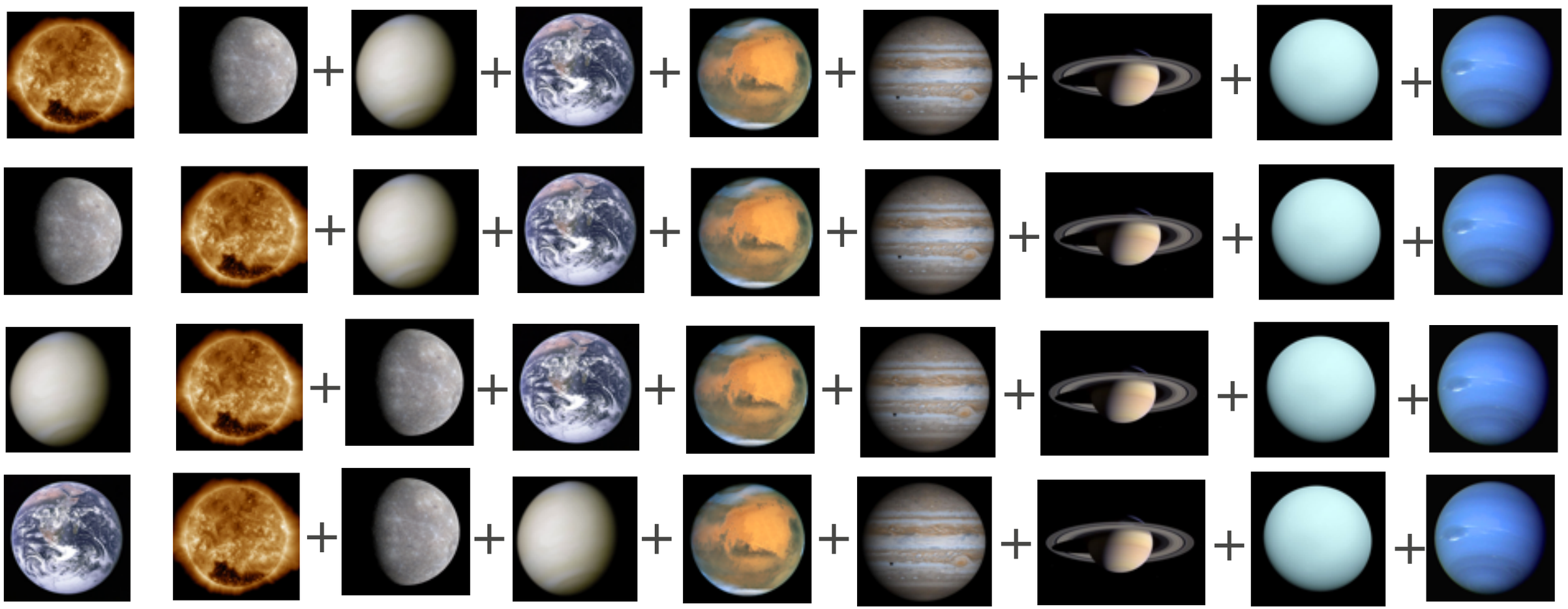
force on each body is sum of forces due to all other bodies



- .
- .
- .

n-body simulation

force on each body is sum of forces due to all other bodies



⋮

Parallelizability: $n^2/\log(n)$

Math is the code

```
fun accelerations (bodies : body Seq.seq) : vec Seq.seq =
    Seq.map (fn b1 =>
                 sum bodies (fn b2 => acc0n (b1 , b2)))
            bodies
```

Math is the code

```
fun accelerations (bodies : body Seq.seq) : vec Seq.seq =
    Seq.map (fn b1 =>
                sum bodies (fn b2 => acc0n (b1 , b2)))
            bodies
```

transform each body b1 into…

Math is the code

```
fun accelerations (bodies : body Seq.seq) : vec Seq.seq =
    Seq.map (fn b1 =>
                sum bodies (fn b2 => accOn (b1 , b2)))
            bodies
```

transform each body b1 into…

the sum over all bodies b2 of …

Math is the code

```
fun accelerations (bodies : body Seq.seq) : vec Seq.seq =
    Seq.map (fn b1 =>
                sum bodies (fn b2 => accOn (b1 , b2)))
            bodies
```

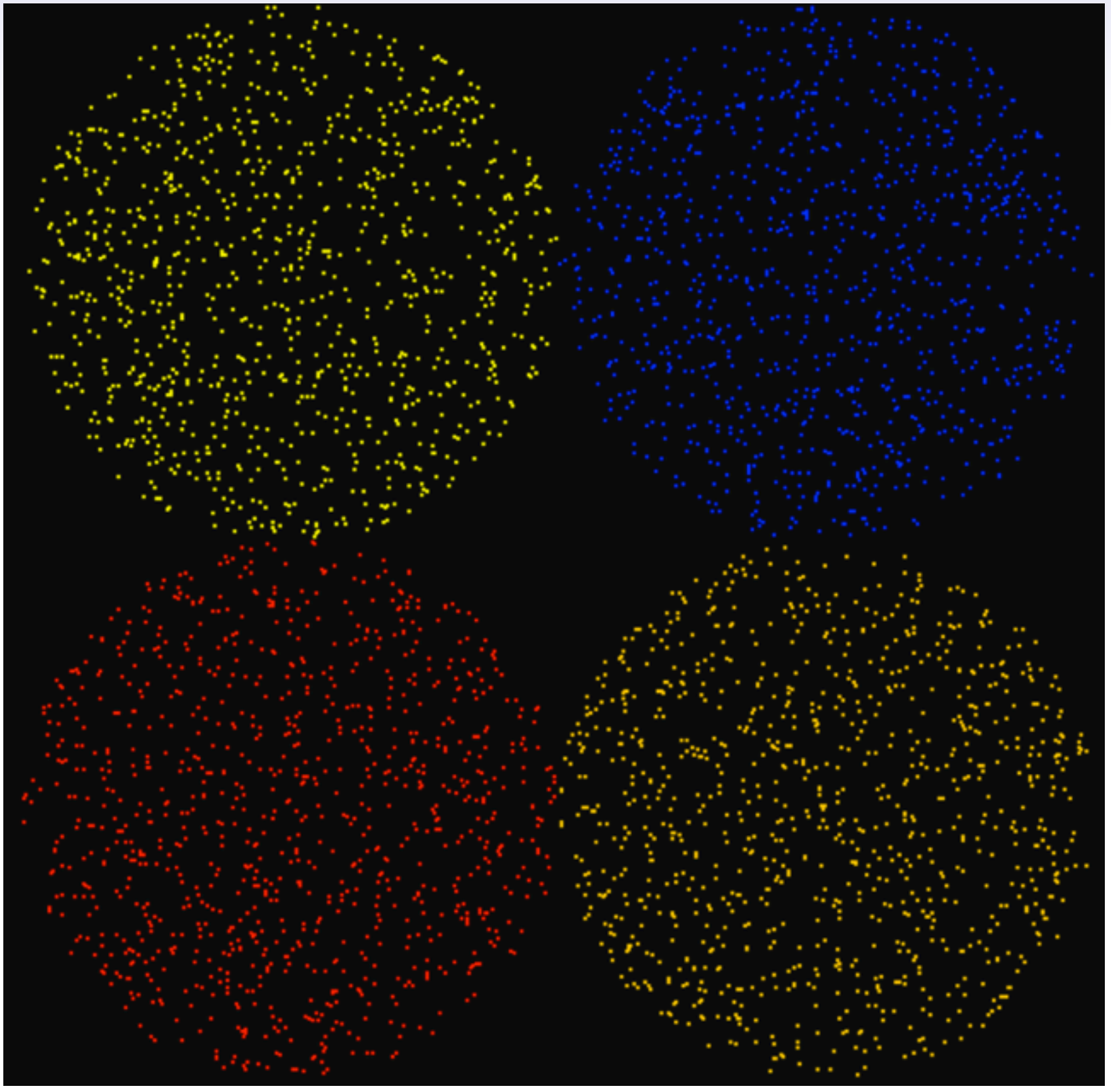
transform each body b1 into…

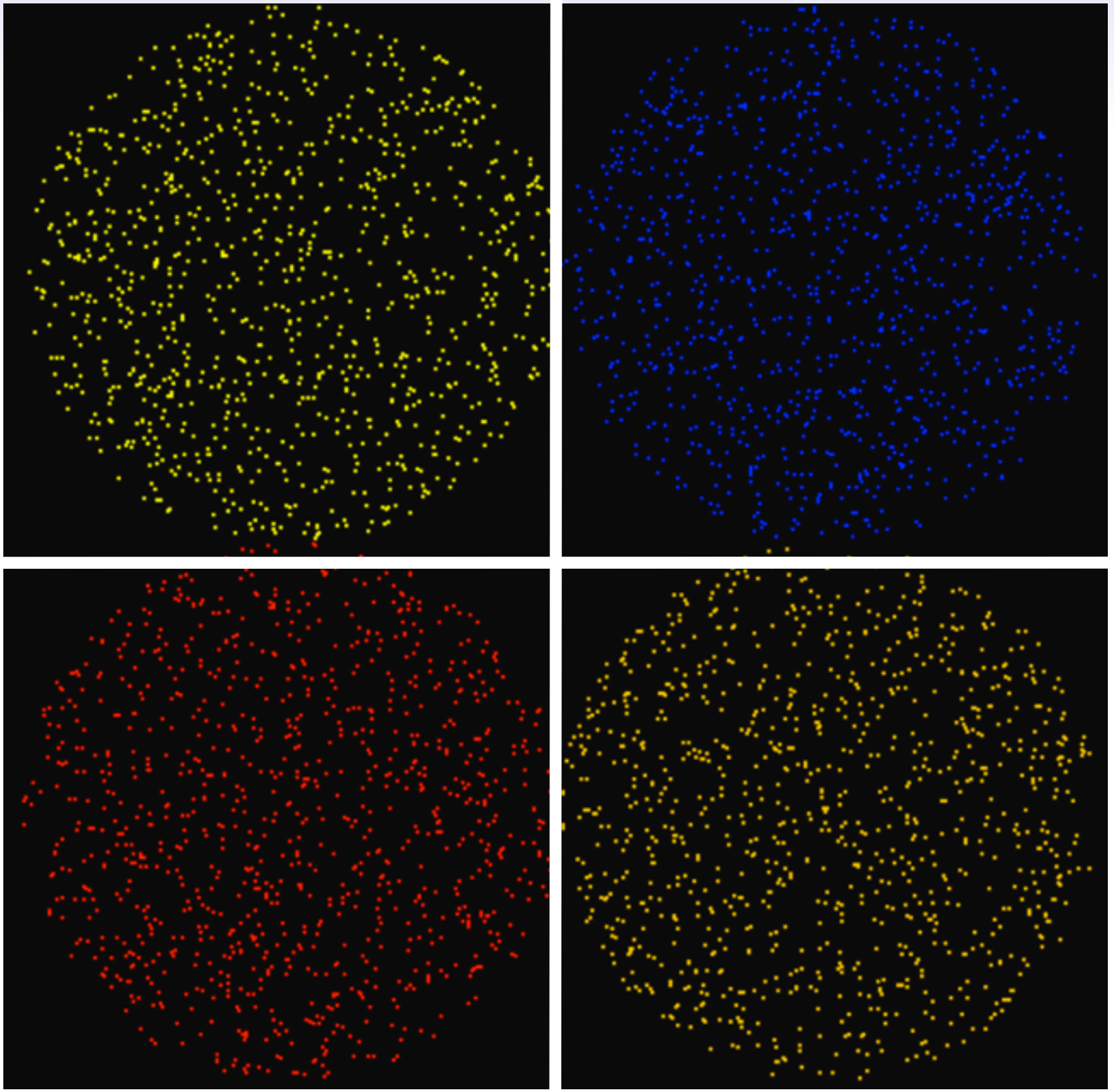
the sum over all bodies b2 of …

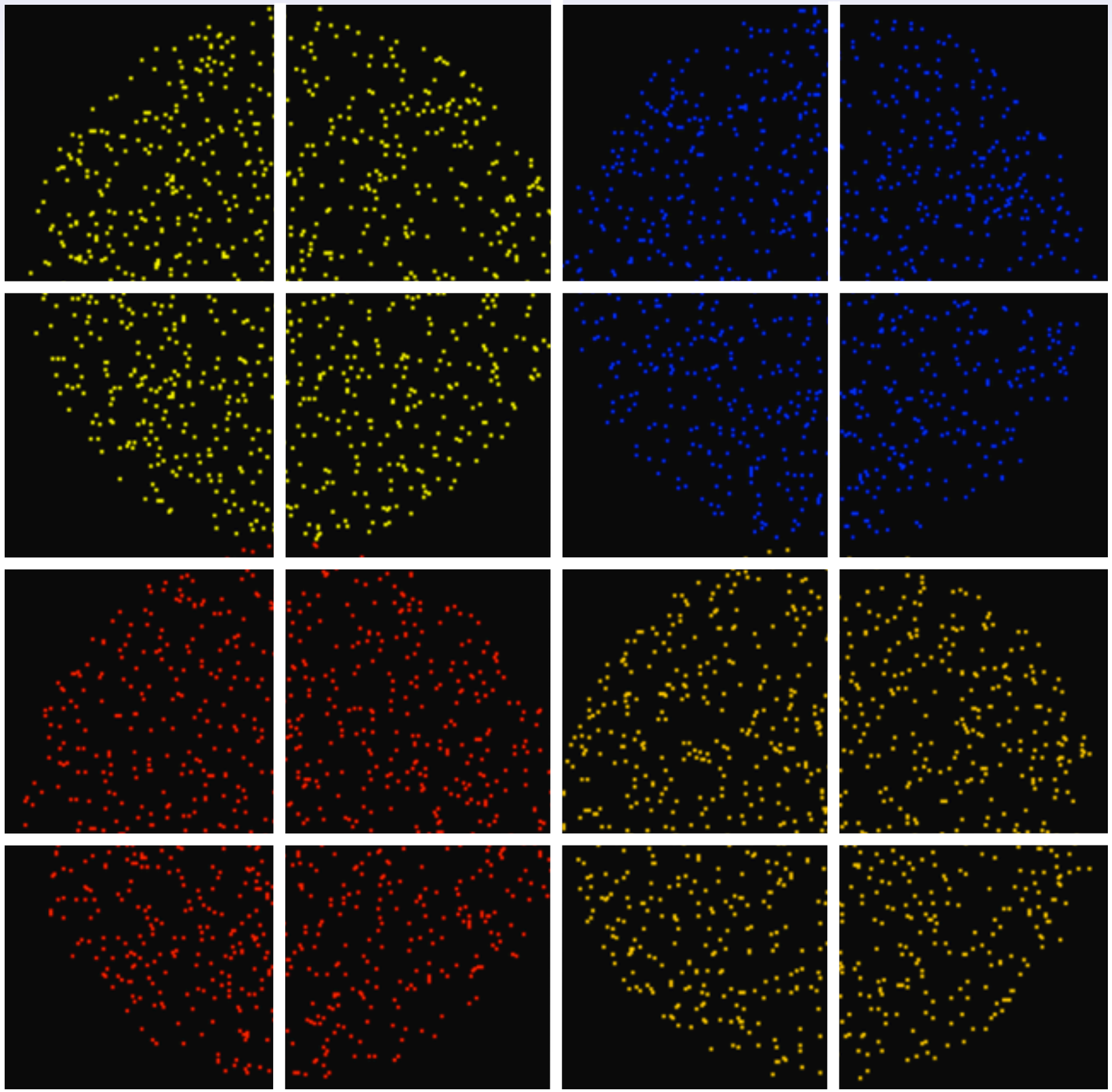
the acceleration on b1 due to b2 (Gm_2/d^2)

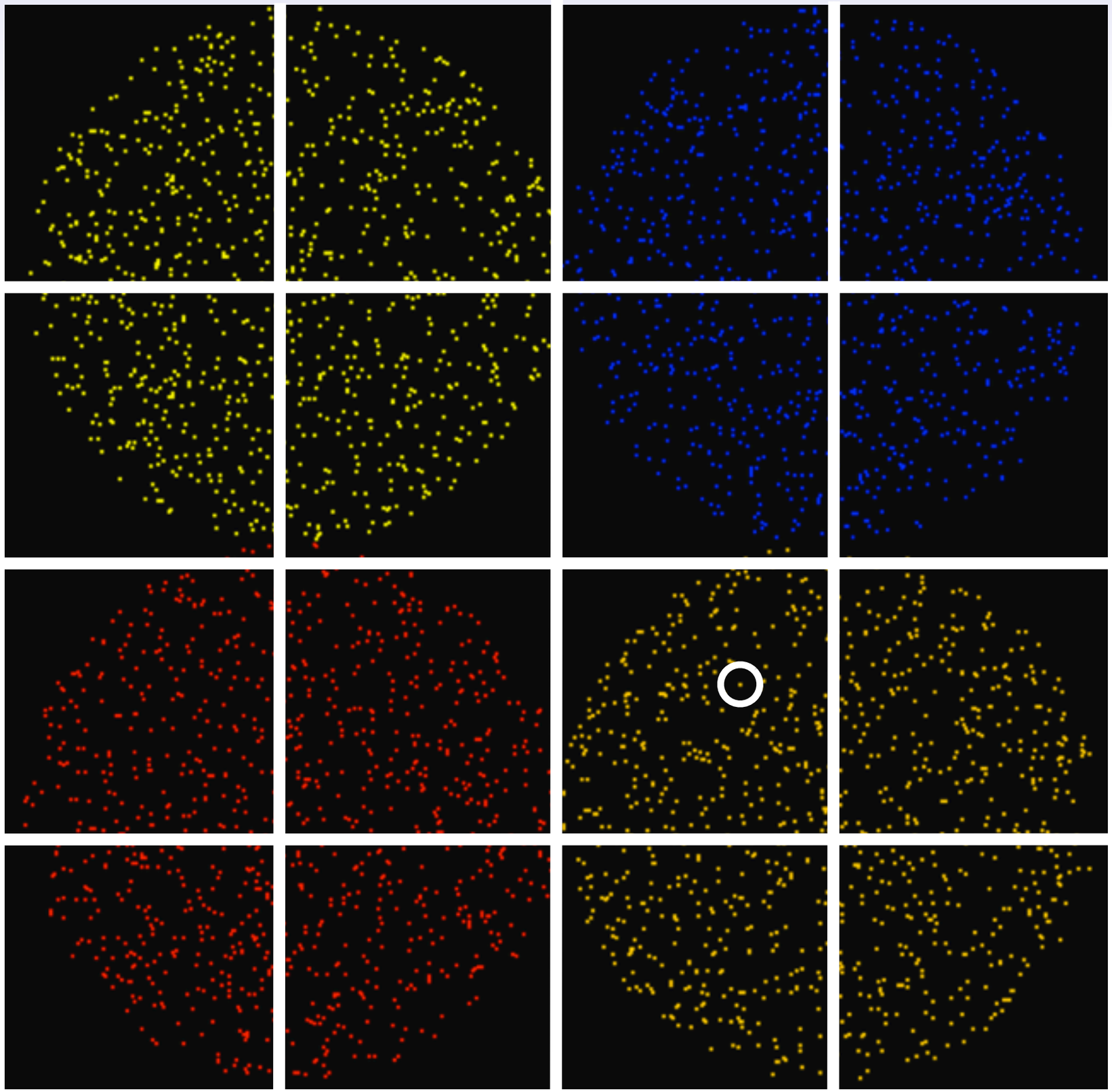
Barnes-Hut tree method

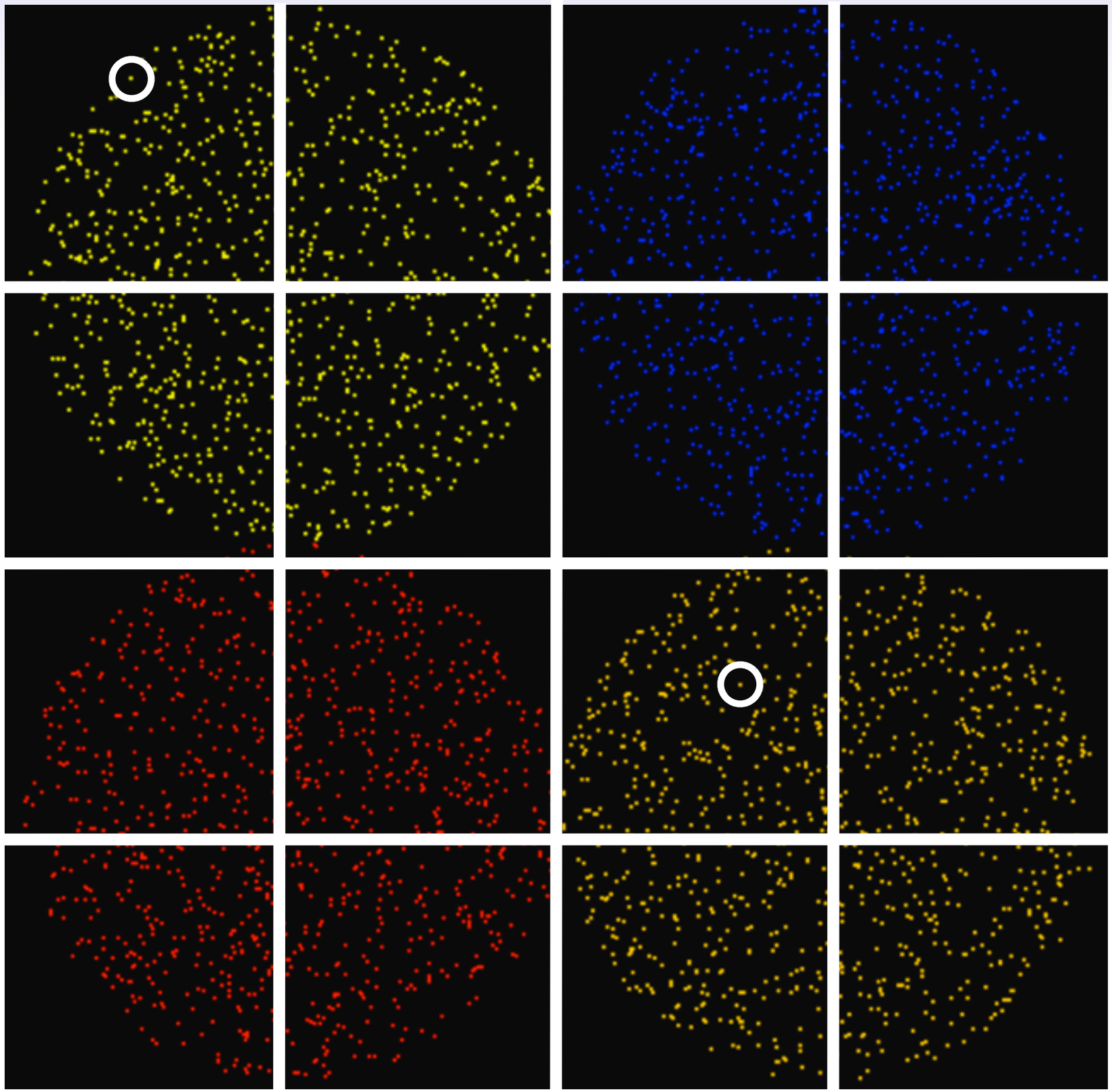
- $O(n^2)$ work is too high for some simulations
- Can approximate and still get useful results

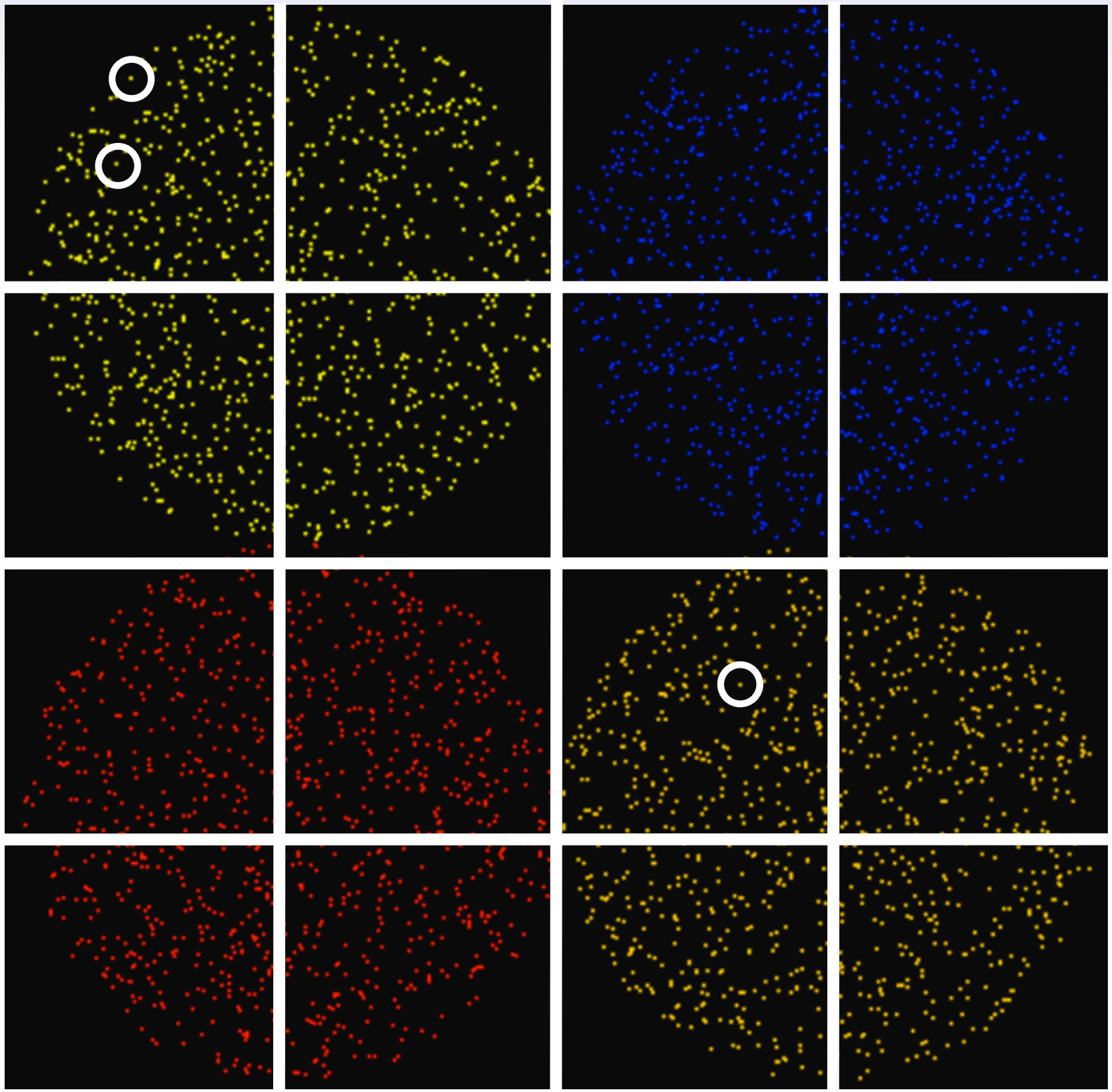


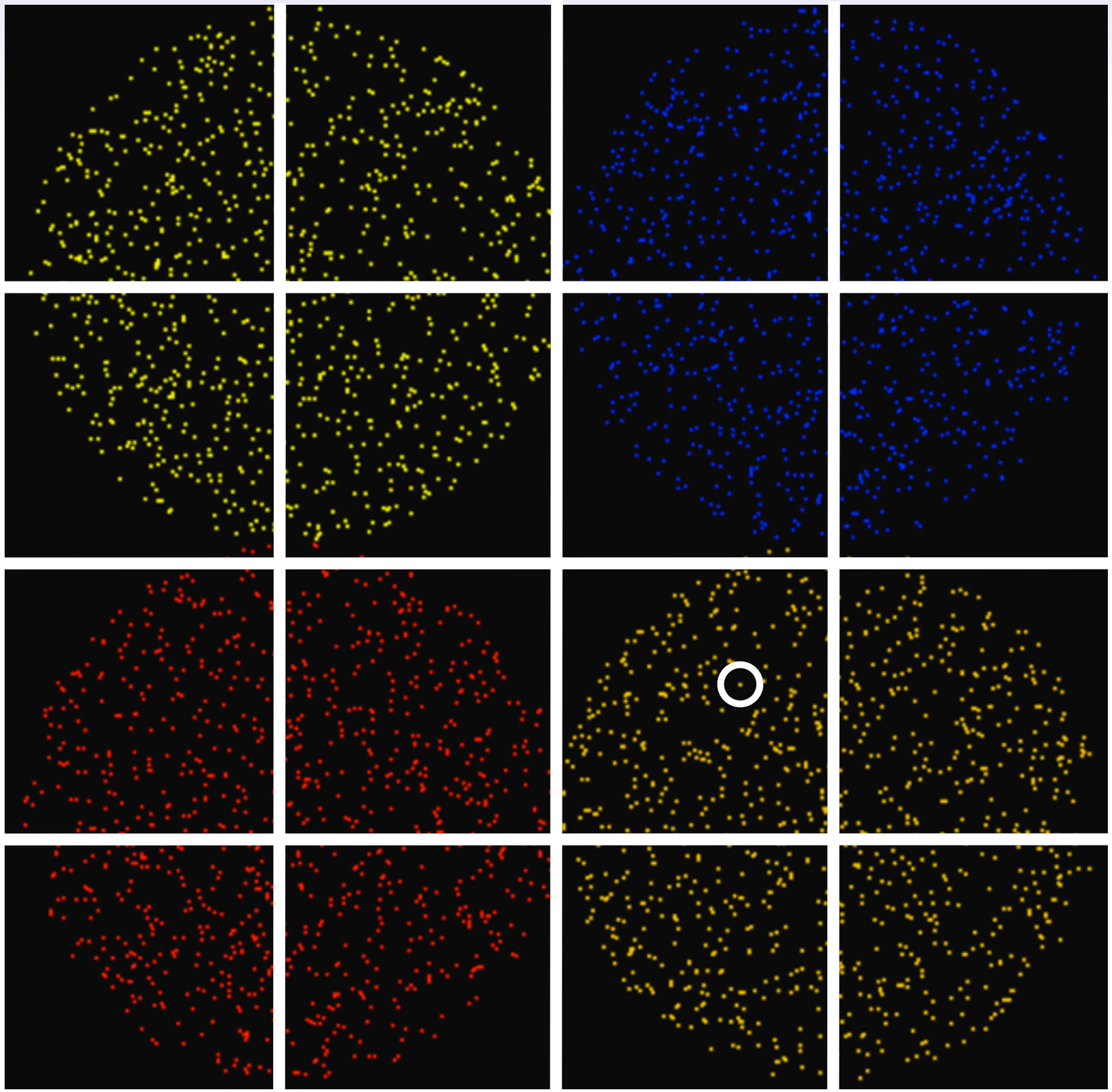


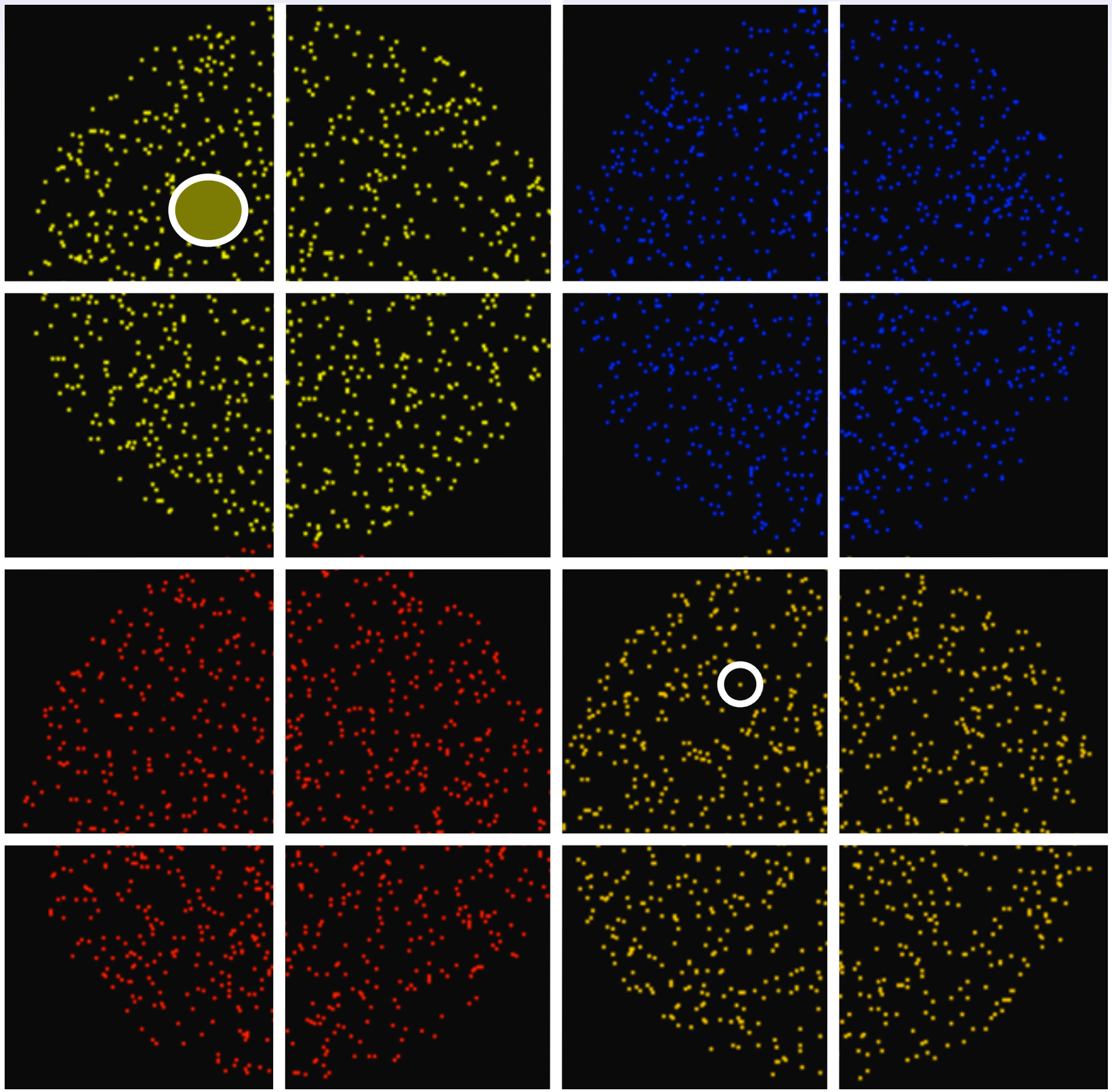




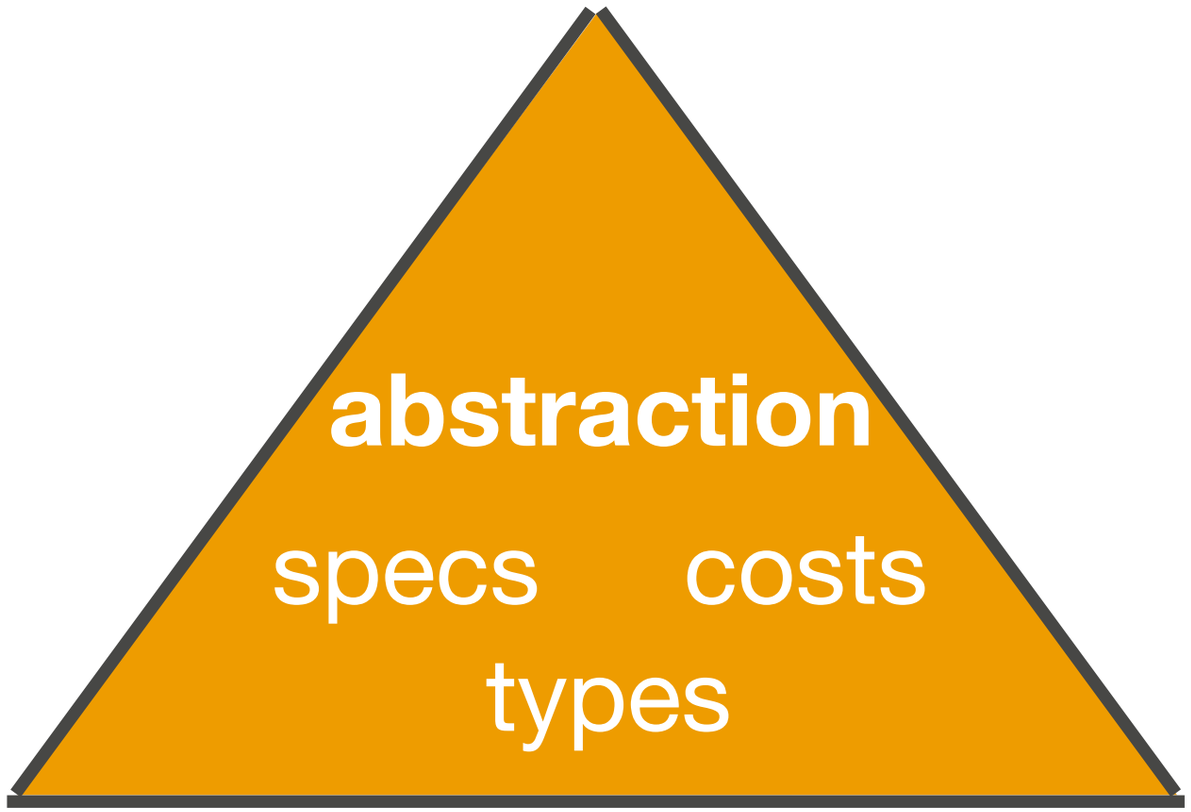








write code



reason about behavior

reason about efficiency

Conclusion

Now

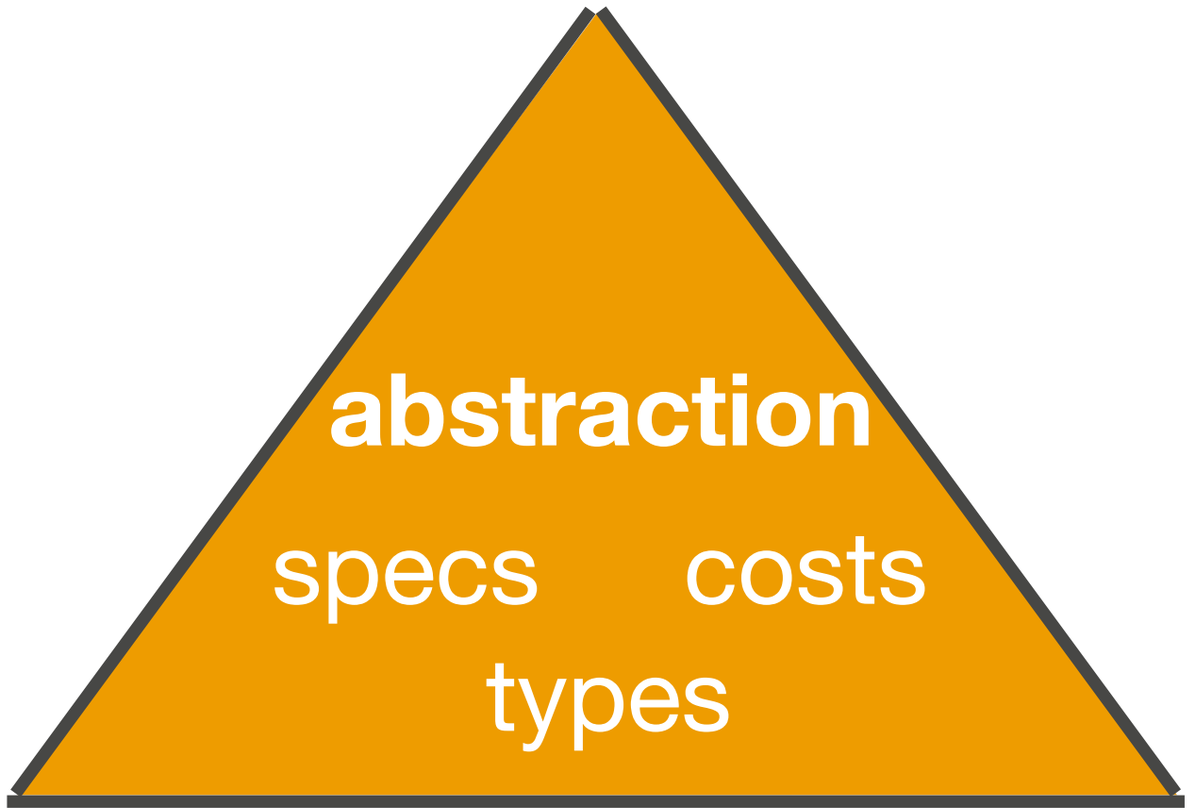
```
signature EXTRACT_COMBINE =
sig
    structure MR : MAP_REDUCE

    structure D : DICT

    val extractcombine : ('a -> (D.Key.t * 'v) Seq.seq)
                         -> ('v * 'v -> 'v)
                         -> 'a MR.mapreducable
                         -> 'v D.dict

end
```

```
signature CLASSIFIER =
sig
    structure Dataset : MAP_REDUCE

    type category

    type document = string Seq.seq
    type labeled_document = category Seq.seq * document

    val train_classifier : labeled_document Dataset.mapreducable
                           -> document -> (category * real)
end
```

8 months ago

```
string suffix(int x) {

  string s;

  if (x == 1) {
     return "st";
  }
  else if (x == 2) {
     return "nd";
  }
  else  if (x == 3) {
     return "rd";
  }
  else {
     return "th";
  }

}
```

write code



reason about behavior

reason about efficiency

Code is math

Code is math

Code is art