



Arhitecturi Paralele

Abordări probleme paralele

Prof. Florin Pop
As. Drd. Ing. Cristian Chilipirea
cristian.chilipirea@cs.pub.ro

Elemente preluate din cursul Prof. Ciprian Dobre



FACULTATEA DE
**AUTOMATICĂ ȘI
CALCULATOARE**





Merge algorithm

2	3	4	5	7
---	---	---	---	---

1	2	4	4	6
---	---	---	---	---

Take two **sorted** lists
merge them
Into one **sorted** list

1	2	2	3	4	4	4	5	6	7
---	---	---	---	---	---	---	---	---	---

Merge algorithm

2	3	4	5	7
---	---	---	---	---

1	2	4	4	6
---	---	---	---	---

Solution:

Always extract the smallest element
from the lists
(guaranteed to be first in one of them)

$O(n)$ complexity

1	2	2	3	4	4	4	5	6	7
---	---	---	---	---	---	---	---	---	---



Merge algorithm

2	3	4	5	7
1	2	4	4	6



Merge algorithm

2	3	4	5	7
	2	4	4	6

1



Merge algorithm

3	4	5	7
---	---	---	---

2	4	4	6
---	---	---	---

1	2
---	---



Merge algorithm

3	4	5	7
	4	4	6

1	2	2
---	---	---



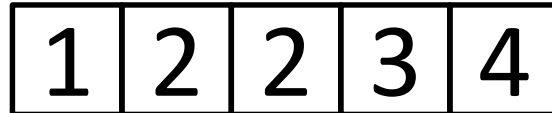
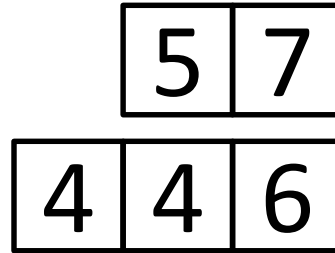
Merge algorithm

4	5	7
4	4	6

1	2	2	3
---	---	---	---



Merge algorithm





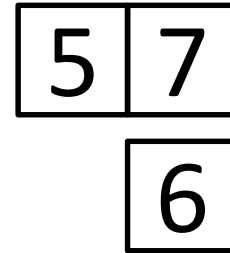
Merge algorithm

5	7
4	6

1	2	2	3	4	4
---	---	---	---	---	---



Merge algorithm





Merge algorithm

7

6

1	2	2	3	4	4	4	5
---	---	---	---	---	---	---	---



Merge algorithm

7

1	2	2	3	4	4	4	5	6
---	---	---	---	---	---	---	---	---



Merge algorithm

1	2	2	3	4	4	4	5	6	7
---	---	---	---	---	---	---	---	---	---

Merge algorithm

1	2	2	3	4	4	4	5	6	7
---	---	---	---	---	---	---	---	---	---

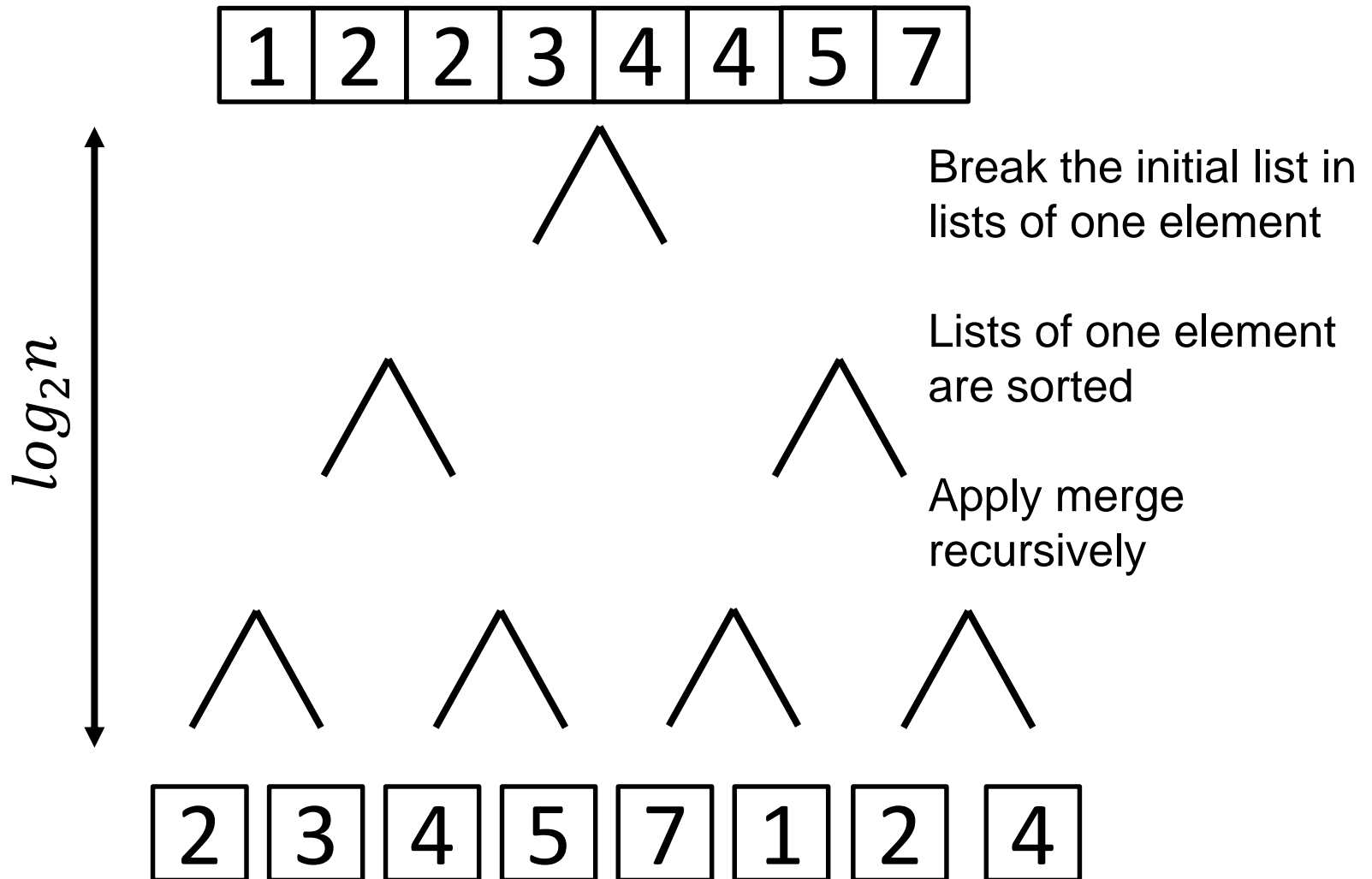
For next slides this
sign means merge



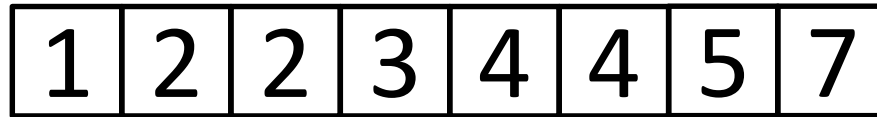
2	3	4	5	7
---	---	---	---	---

1	2	4	4	6
---	---	---	---	---

Merge sort



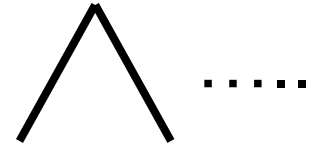
Merge sort



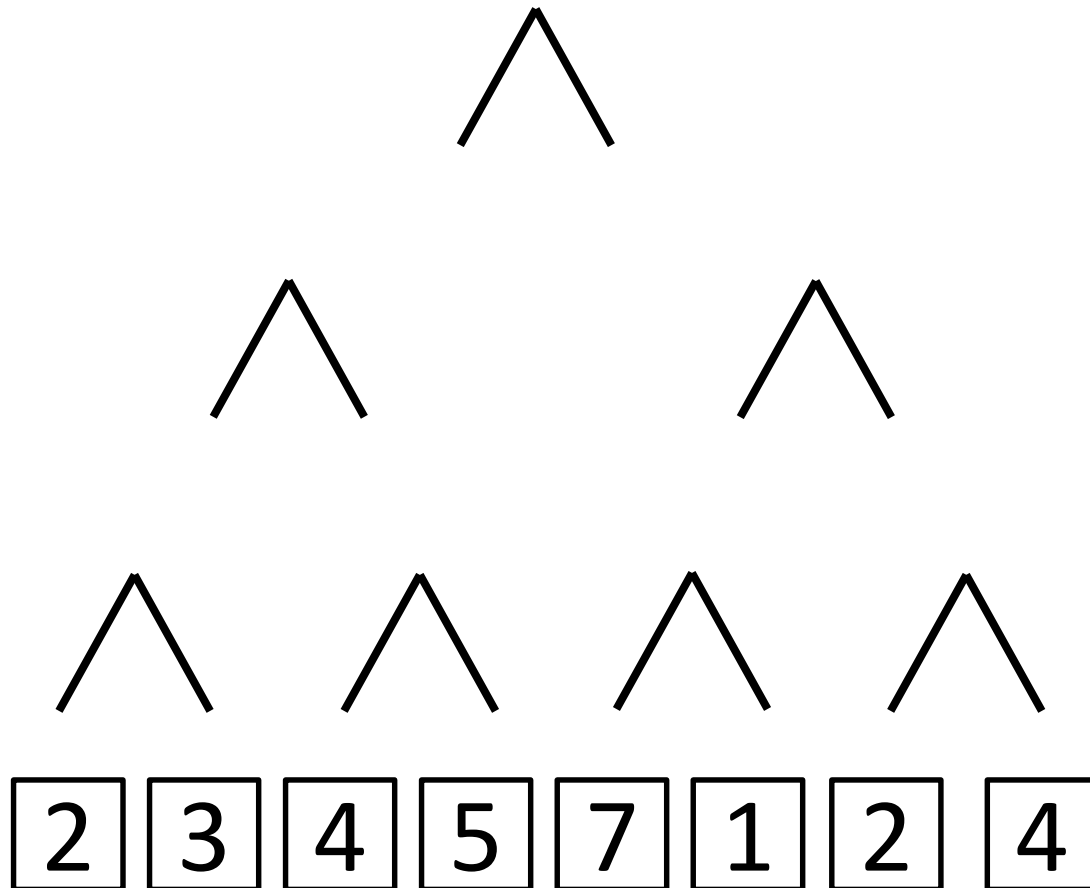
Sequential complexity:

.....

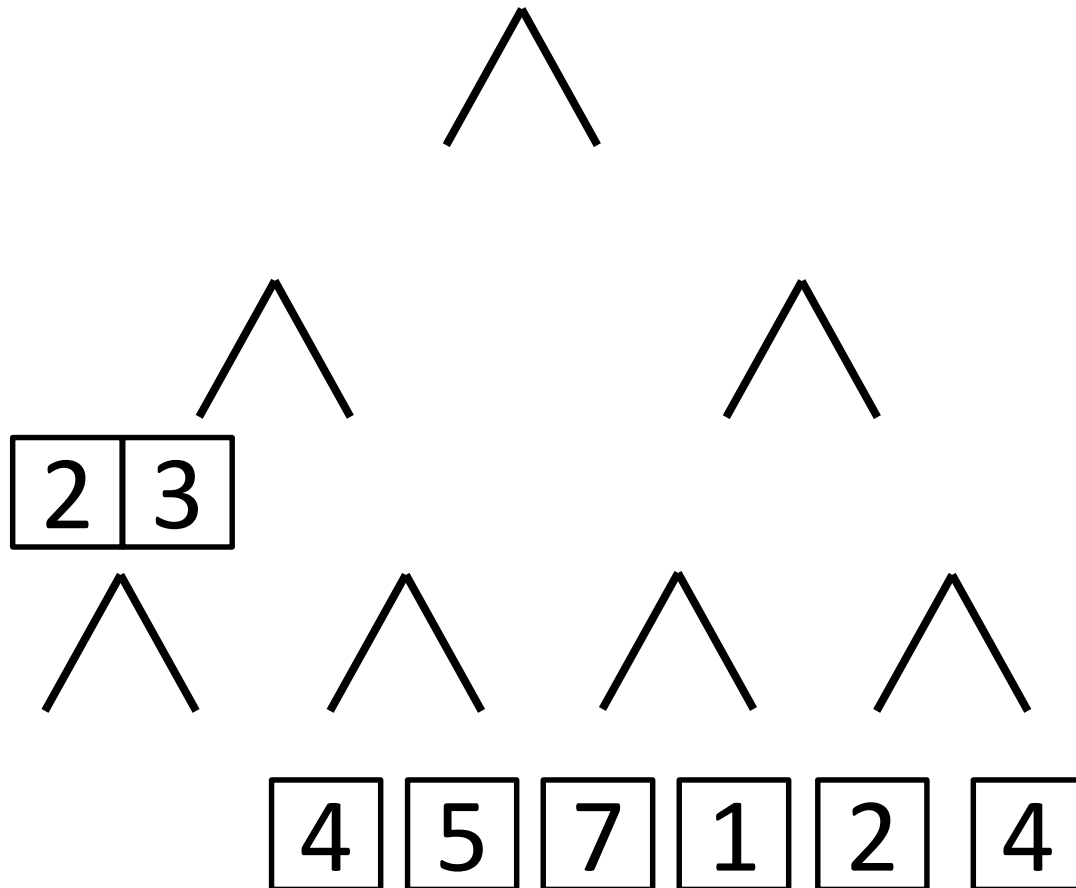
$$O(n * \log_2 n)$$



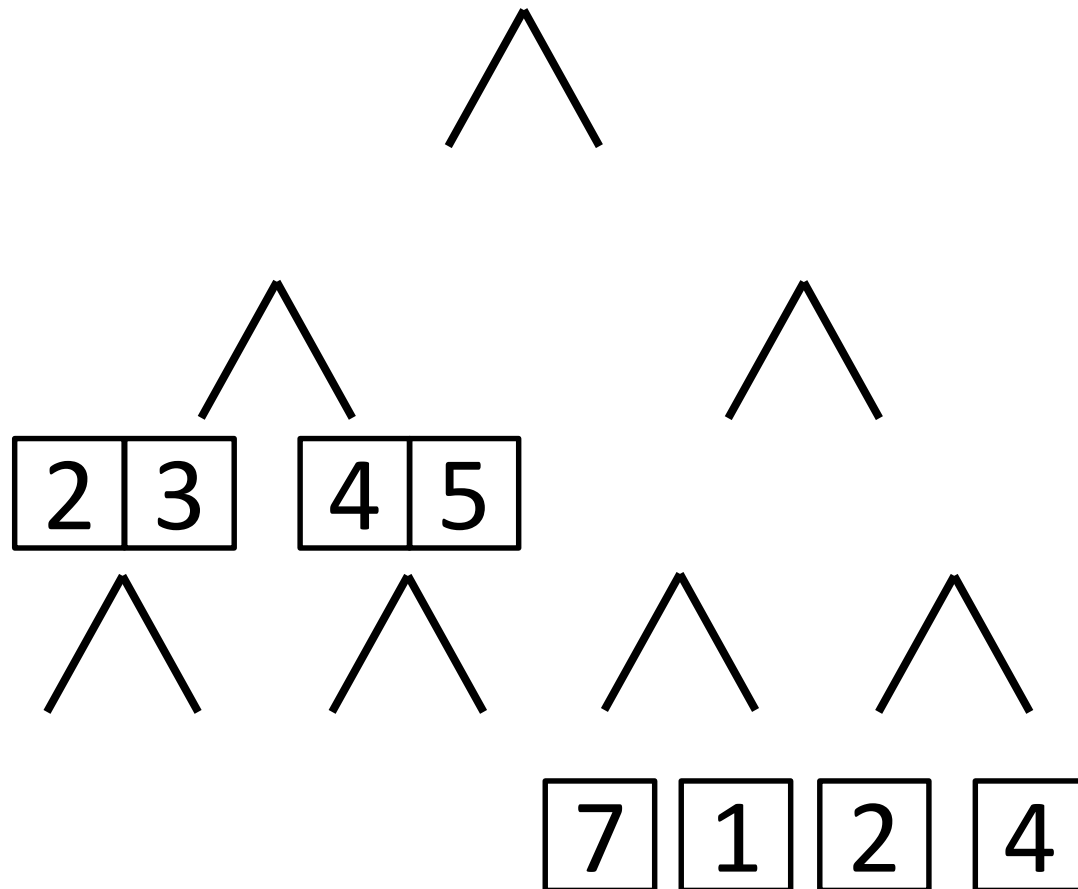
Merge sort



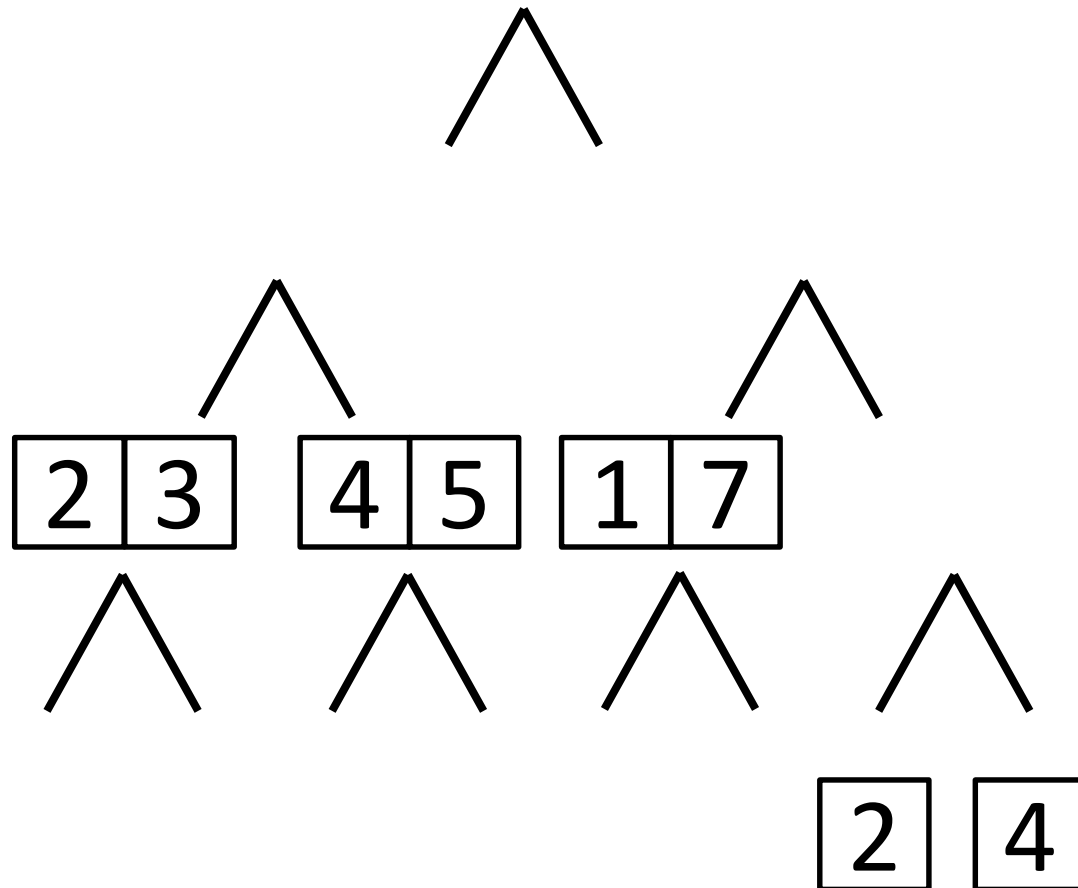
Merge sort



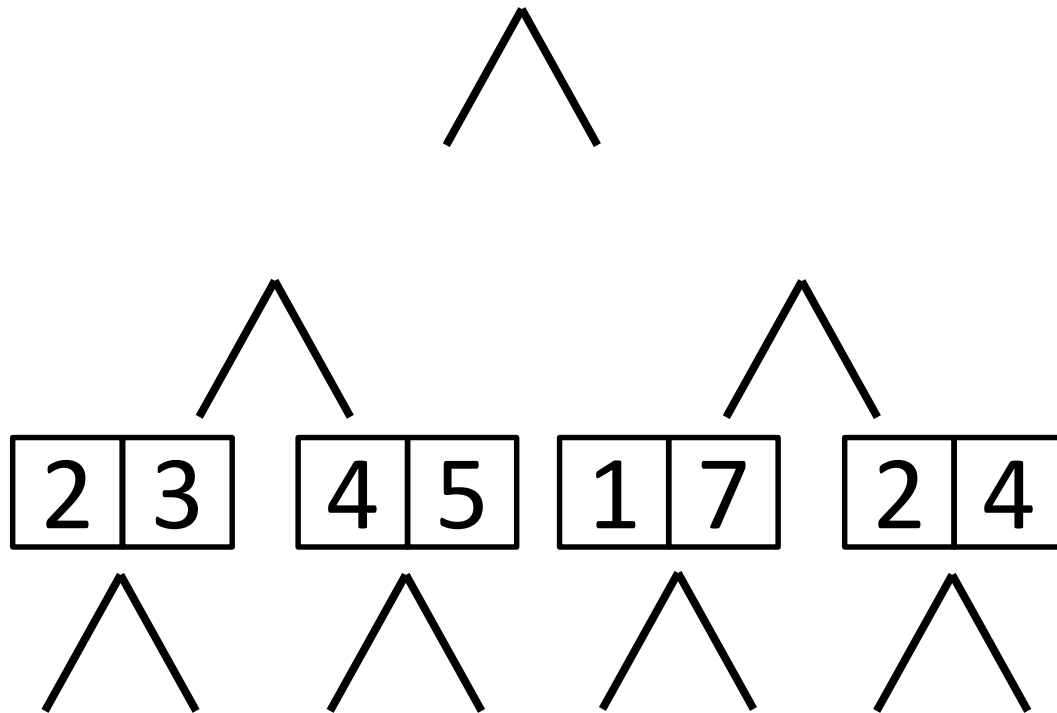
Merge sort



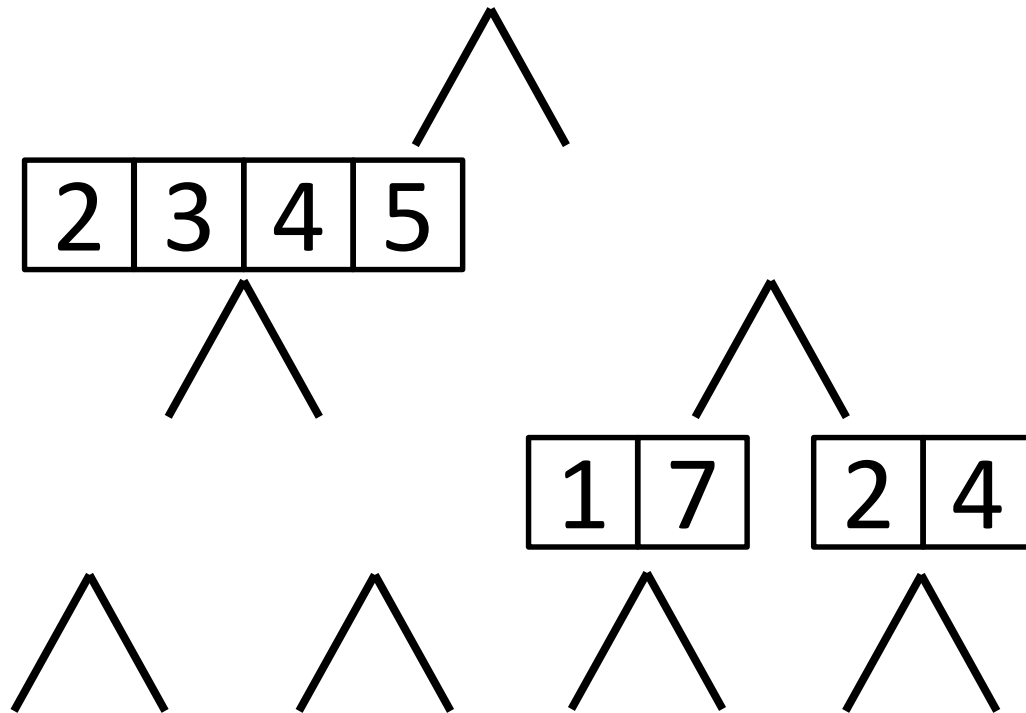
Merge sort



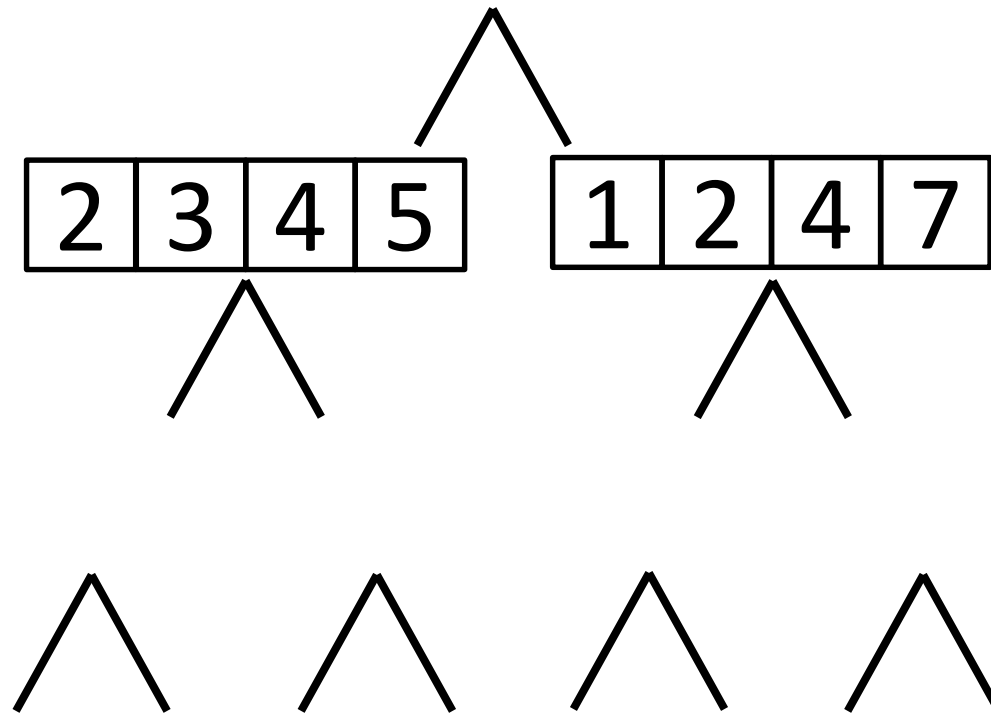
Merge sort



Merge sort



Merge sort

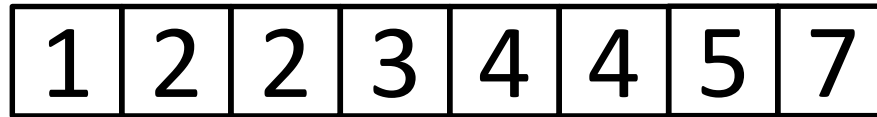


Merge sort

1	2	2	3	4	4	5	7
---	---	---	---	---	---	---	---



Parallel merge sort



Can be
executed
in parallel



Can be
executed
in parallel



Parallel merge sort



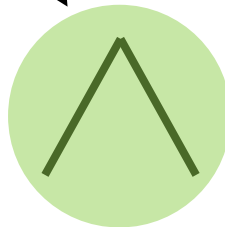
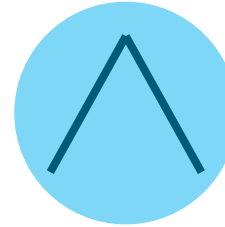
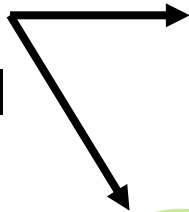
Result of



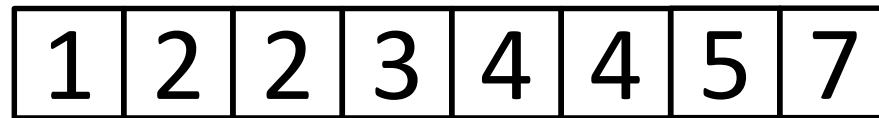
depends on
result of



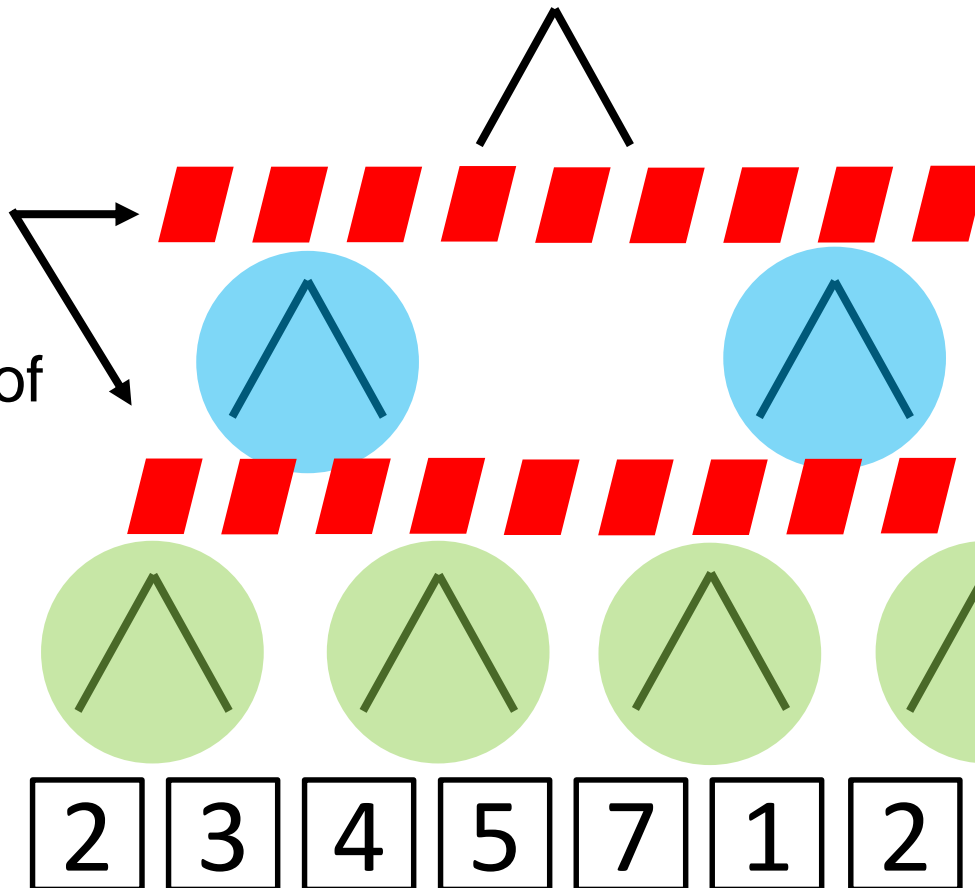
Can **NOT** be
executed
in parallel



Parallel merge sort



Solution:
Barrier
between
each level of
the tree



Each level of
the tree
needs to wait
for the
operations on
the previous
level to finish

Parallel merge sort

Parallel complexity:

p = processor cores

for $p=n$

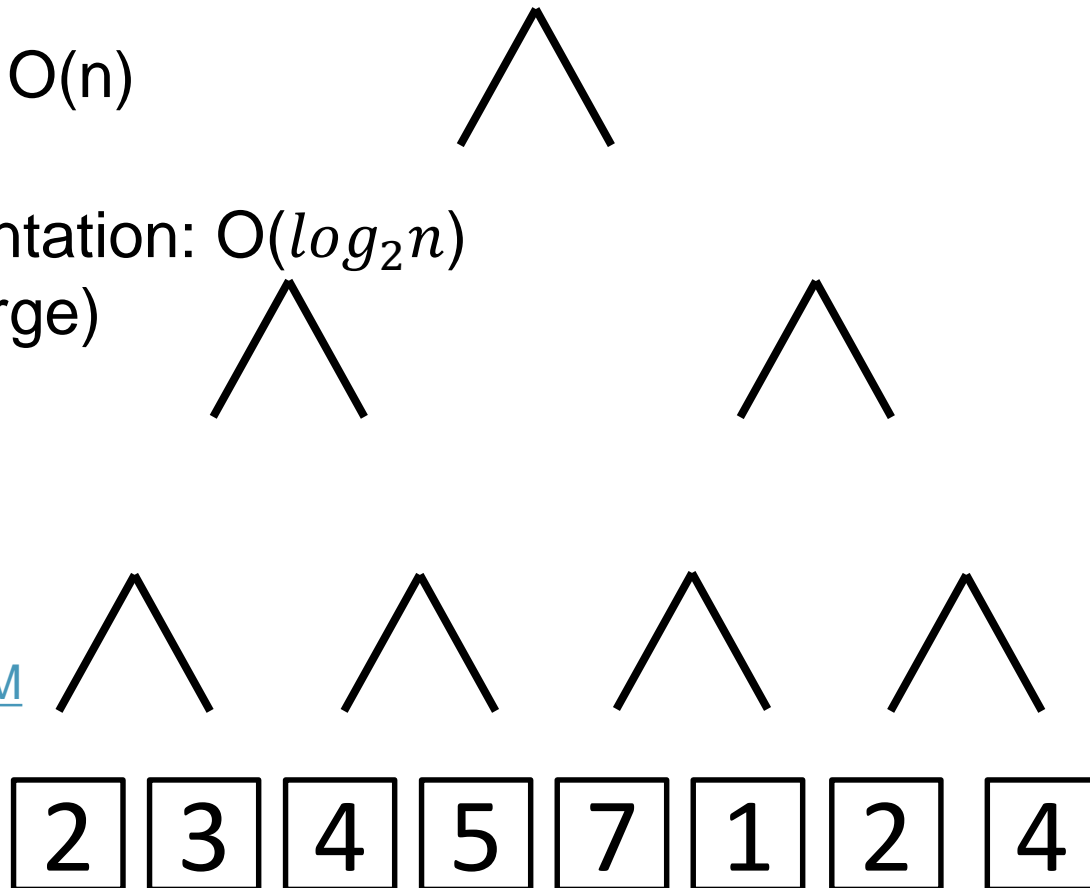
$$O(\sum_{i=1}^{\log_2 n} 2^i) = O(n)$$

Best implementation: $O(\log_2 n)$

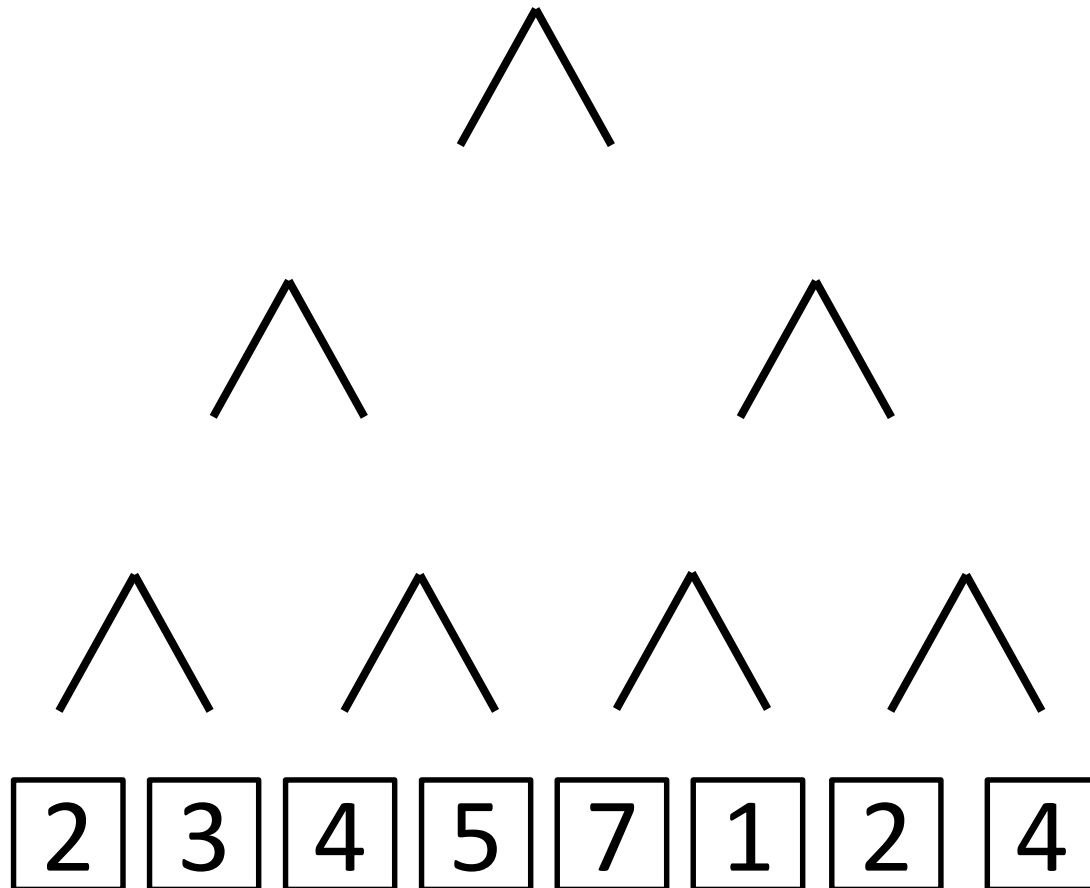
(with $O(1)$ merge)



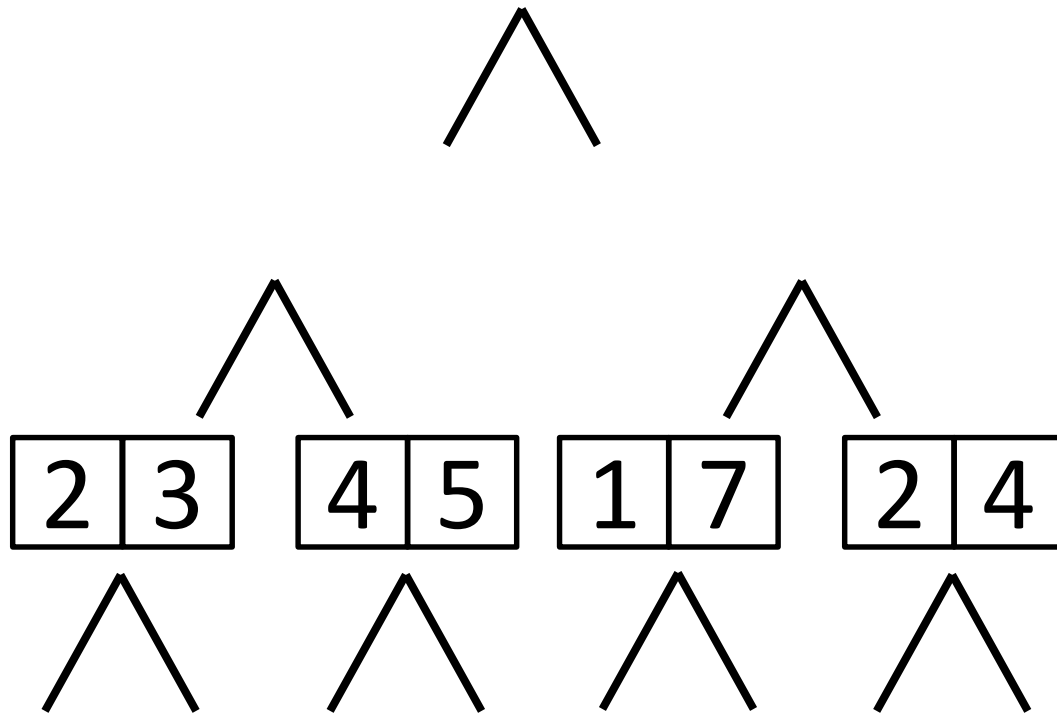
<http://goo.gl/okU3fM>



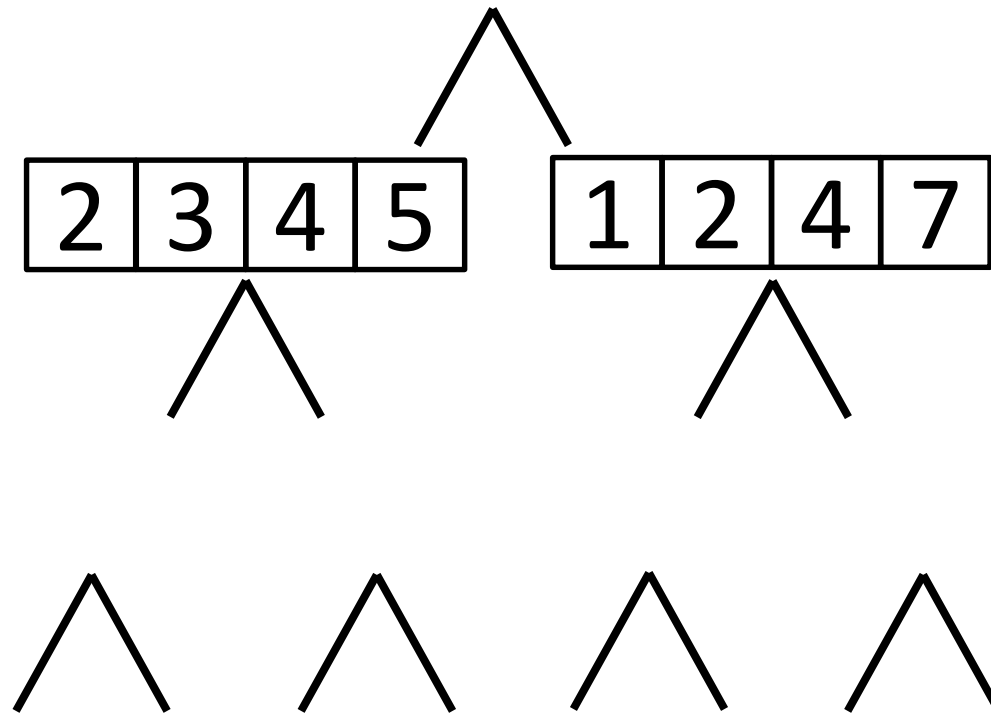
Parallel merge sort



Parallel merge sort



Parallel merge sort



Parallel merge sort

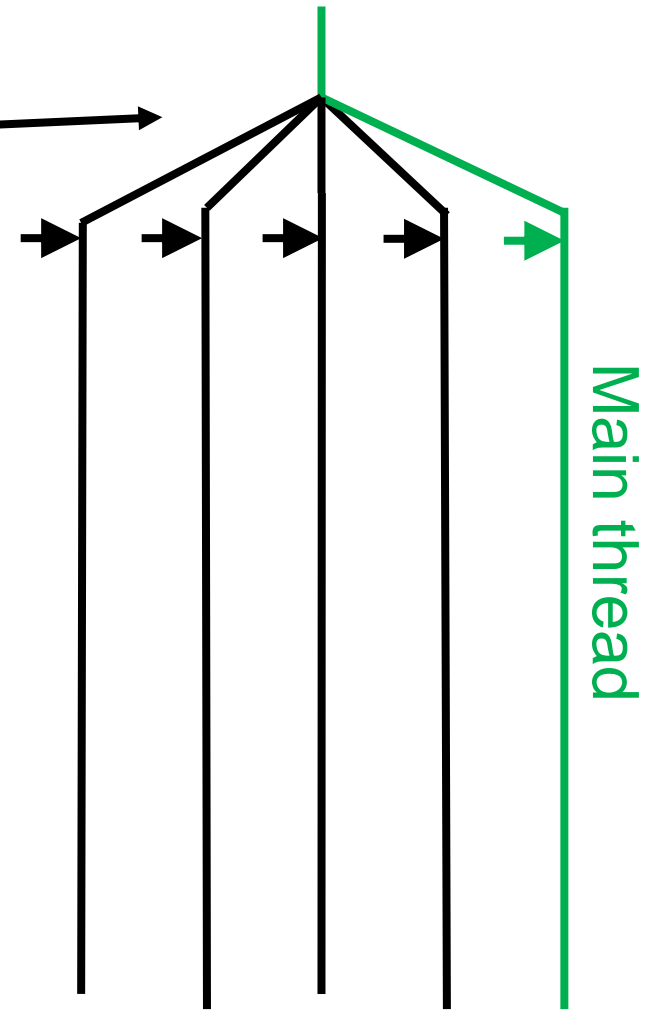
1	2	2	3	4	4	5	7
---	---	---	---	---	---	---	---





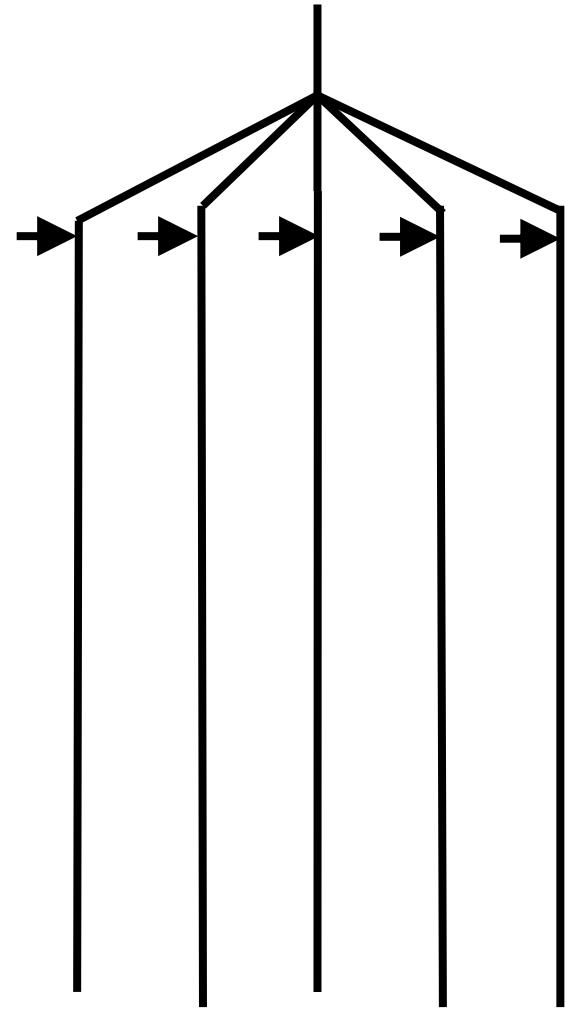
Executor Service

```
ExecutorService tpe =  
Executors.newFixedThreadPool(4);
```



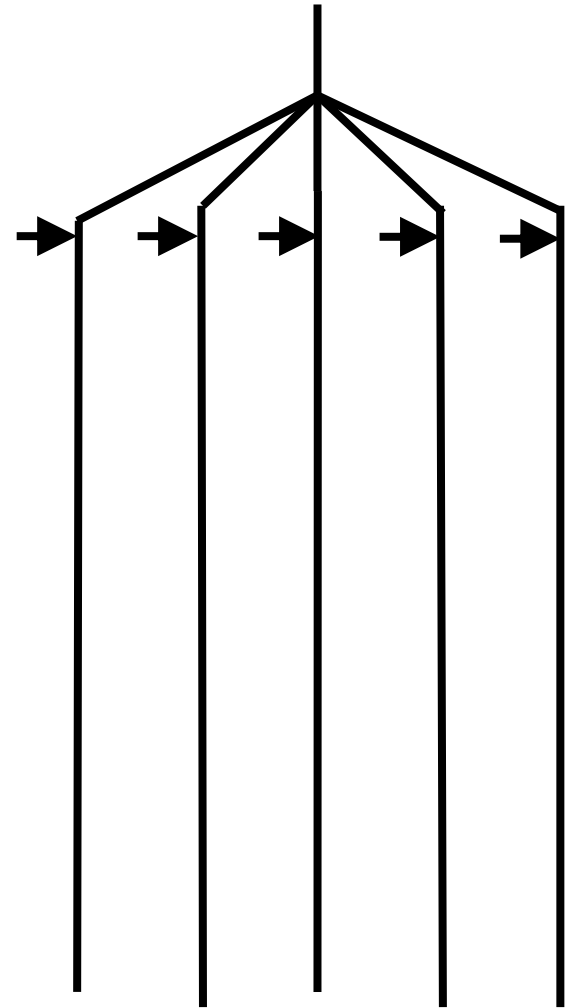
Executor Service

```
public class MyRunnable implements Runnable {  
    int a;  
    ExecutorService tpe;  
  
    public MyRunnable(ExecutorService tpe, int a) {  
        this.a = a;  
        this.tpe = tpe;  
    }  
  
    @Override  
    public void run() {  
        if (a > 10) {  
            tpe.shutdown();  
            return;  
        }  
        System.out.println(a);  
        tpe.submit(new MyRunnable(tpe, a + 3));  
    }  
}
```

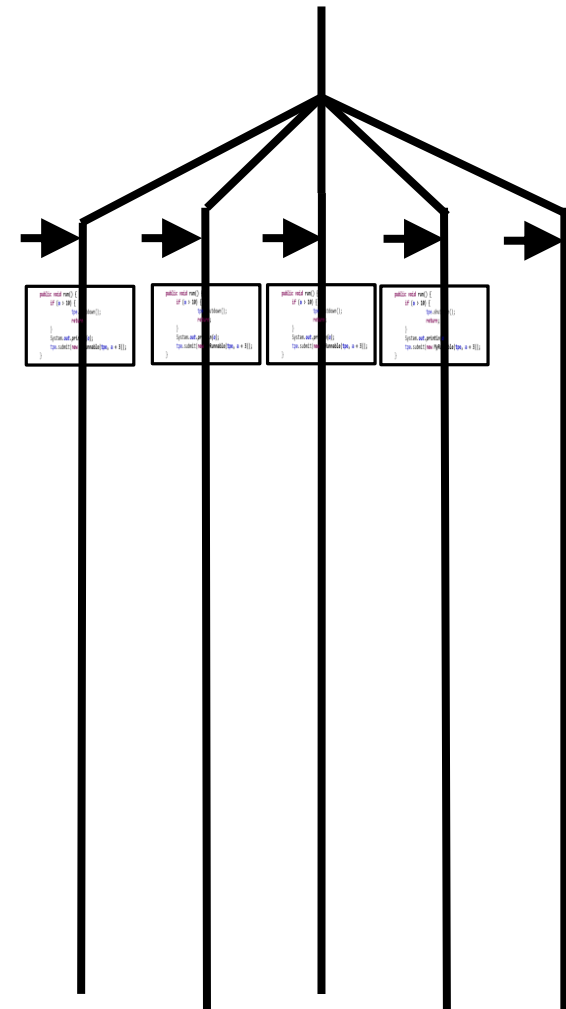


Executor Service

```
tpe.submit(new MyRunnable(tpe, 0));  
tpe.submit(new MyRunnable(tpe, 1));  
tpe.submit(new MyRunnable(tpe, 2));  
.....
```



Executor Service

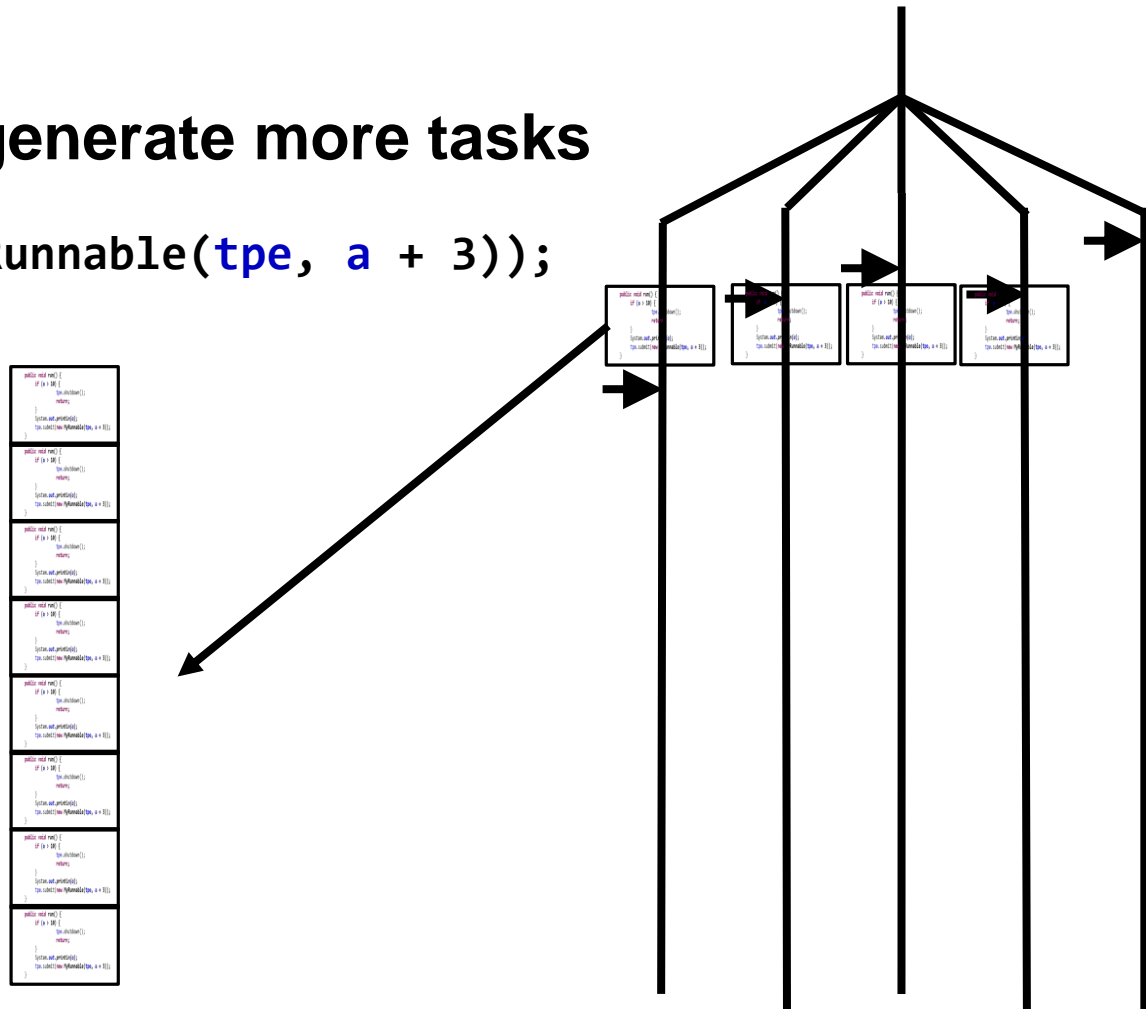


```
public void run() {
    if (isInterrupted())
        return;
    try {
        // ...
    } catch (InterruptedException e) {
        // ...
    }
}
```

Executor Service

Tasks can generate more tasks

```
tpe.submit(new MyRunnable(tpe, a + 3));
```



Executor Service

```

public void run() {
    if (isDone())
        return;
    System.out.println("Task " + task);
    try {
        task.run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void run() {
    if (isDone())
        return;
    System.out.println("Task " + task);
    try {
        task.run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void run() {
    if (isDone())
        return;
    System.out.println("Task " + task);
    try {
        task.run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

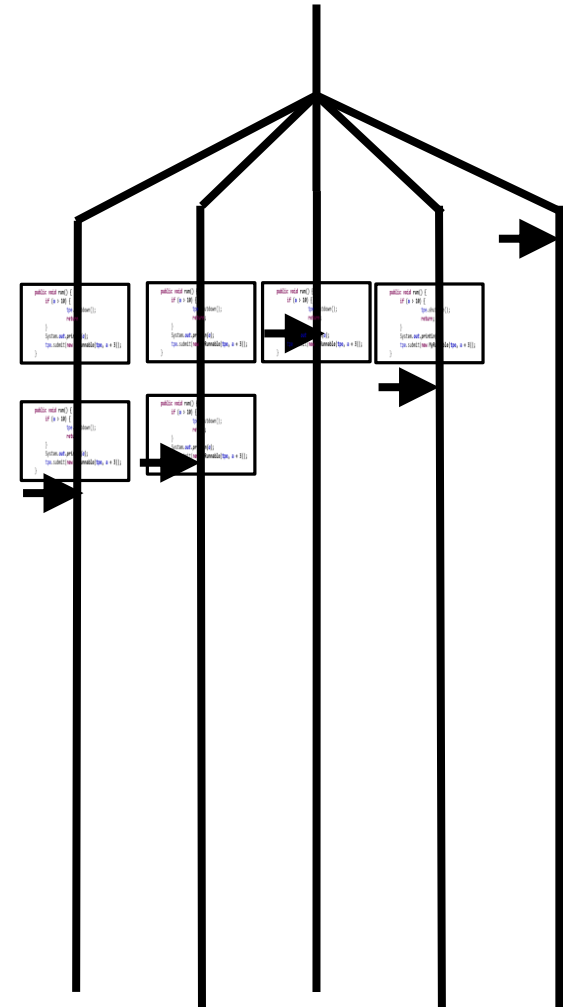
public void run() {
    if (isDone())
        return;
    System.out.println("Task " + task);
    try {
        task.run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void run() {
    if (isDone())
        return;
    System.out.println("Task " + task);
    try {
        task.run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void run() {
    if (isDone())
        return;
    System.out.println("Task " + task);
    try {
        task.run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

public void run() {
    if (isDone())
        return;
    System.out.println("Task " + task);
    try {
        task.run();
    } catch (Exception e) {
        e.printStackTrace();
    }
}

```



Executor Service

```

public void run() {
    if (isAlive())
        doWork();
}

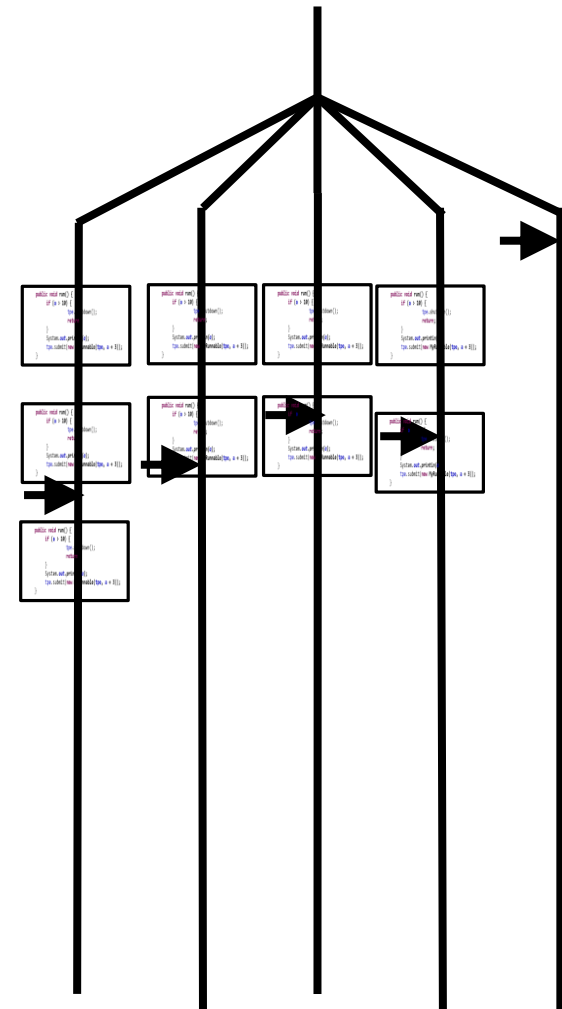
public void run() {
    if (isAlive())
        doWork();
}

public void run() {
    if (isAlive())
        doWork();
}

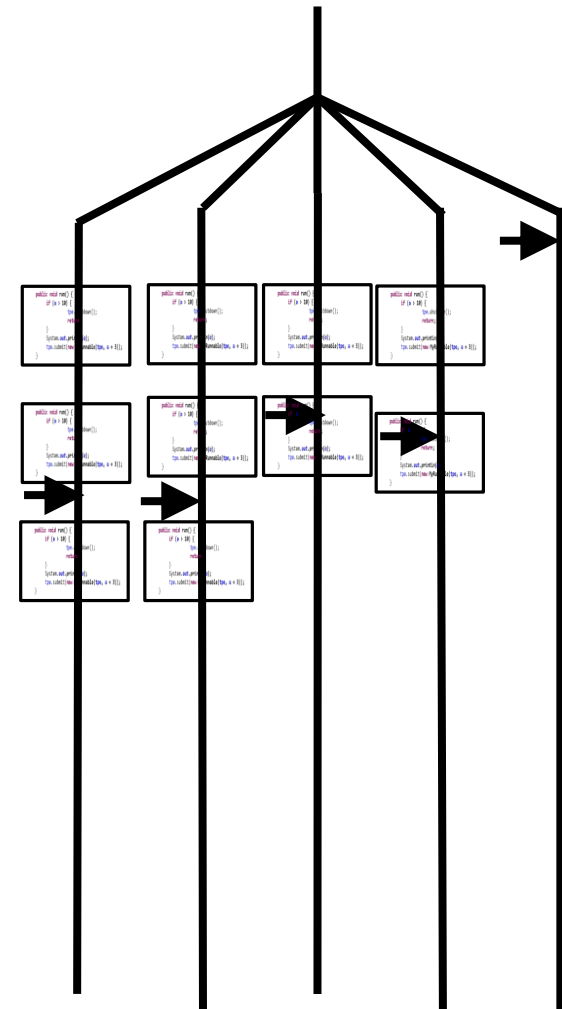
public void run() {
    if (isAlive())
        doWork();
}

public void run() {
    if (isAlive())
        doWork();
}

```



Executor Service



```

public void run() {
    if (isFirst()) {
        // do something
    }
    while (true) {
        // do something
    }
}

public void run() {
    if (isFirst()) {
        // do something
    }
    while (true) {
        // do something
    }
}

public void run() {
    if (isFirst()) {
        // do something
    }
    while (true) {
        // do something
    }
}

public void run() {
    if (isFirst()) {
        // do something
    }
    while (true) {
        // do something
    }
}

```

Executor Service

```

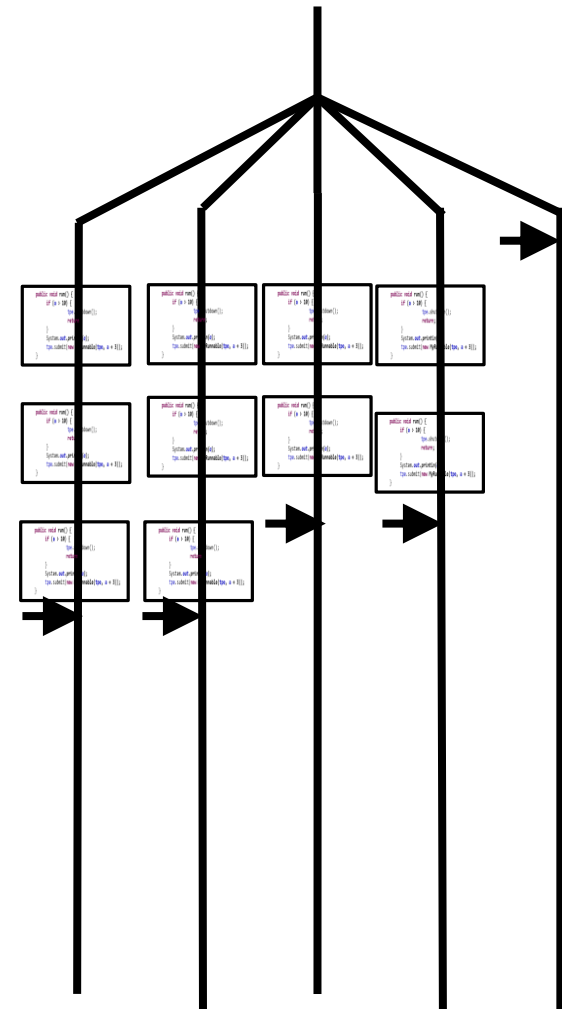
public void run() {
    if (isAlive()) {
        notify();
    }
    System.out.println(
        "Task " + Thread.currentThread().getName() + " is running."
    );
}

public void run() {
    if (isAlive()) {
        notify();
    }
    System.out.println(
        "Task " + Thread.currentThread().getName() + " is running."
    );
}

public void run() {
    if (isAlive()) {
        notify();
    }
    System.out.println(
        "Task " + Thread.currentThread().getName() + " is running."
    );
}

public void run() {
    if (isAlive()) {
        notify();
    }
    System.out.println(
        "Task " + Thread.currentThread().getName() + " is running."
    );
}

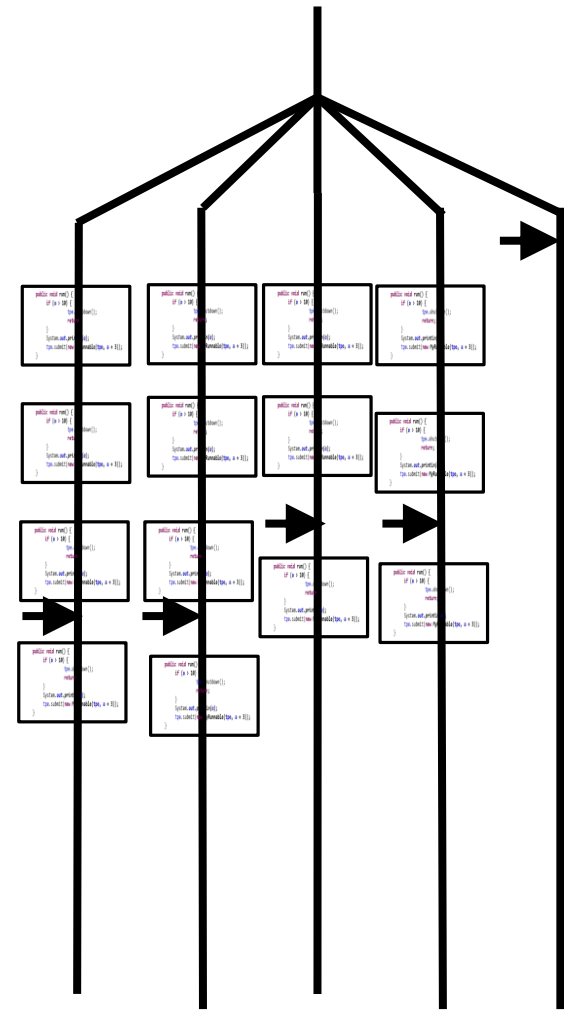
```



Executor Service

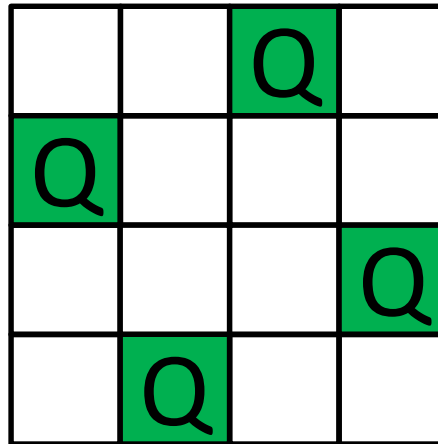
When do you stop the threads?
Depends on the problem.
Sometimes one solution is enough.
But some problems might not have
solutions.

`tpe.shutdown();`





N Queens Problem



N Queens Problem

No more than one queen per line

		Q	
Q			
			Q
	Q		

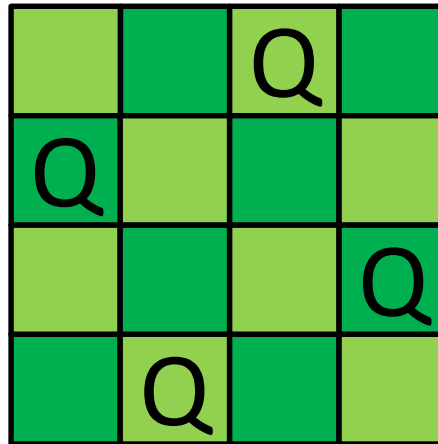
N Queens Problem

No more than one queen per column

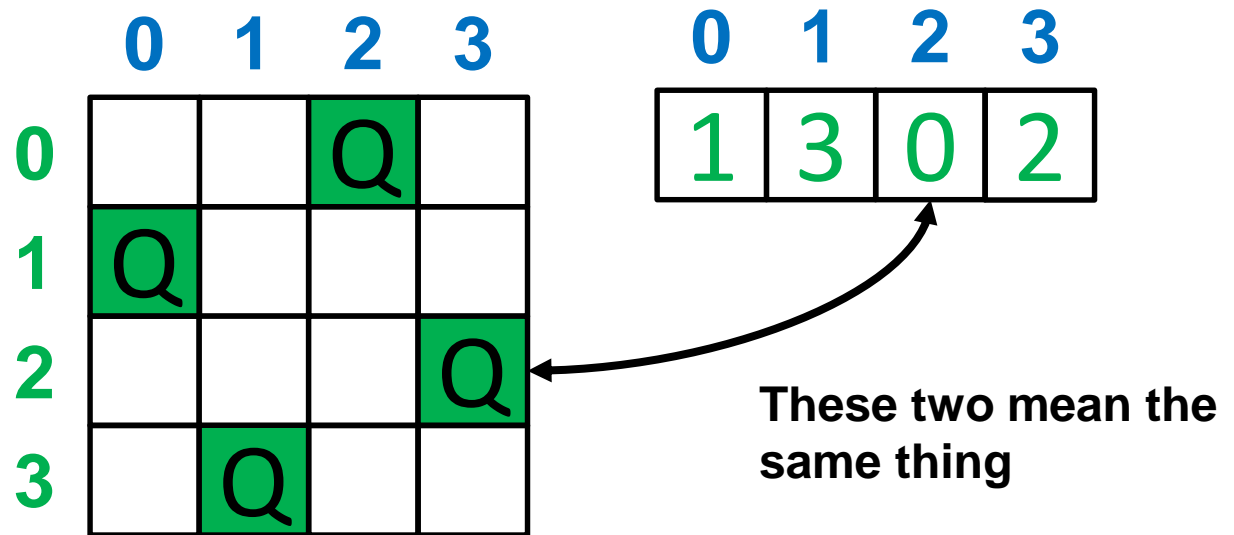
		Q	
Q			
			Q
	Q		

N Queens Problem

No more than one queen per diagonals



N Queens Problem



N Queens Problem - Solution

	0	1	2	3
0	Q			
1				
2				
3				

0	1	2	3
0			

N Queens Problem - Solution

	0	1	2	3
0	Q	Q		
1				
2				
3				

0	1	2	3
0	0		

Line conflict

N Queens Problem - Solution

	0	1	2	3
0	Q			
1		Q		
2				
3				

0	1	2	3
0	1		

Diagonal conflict

N Queens Problem - Solution

	0	1	2	3
0	Q			
1				
2		Q		
3				

0	1	2	3
0	2		

OK.

And so on...

N Queens Problem – Parallel Solution

0 1 2 3

0			
---	--	--	--

1			
---	--	--	--

2			
---	--	--	--

3			
---	--	--	--

N Queens Problem – Parallel Solution

0	1	2	3	
0	0			x
0	1			x
0	2			
0	3			

0	1	2	3	
2	0			
2	1			x
2	2			x
2	3			x

And so on...

0	1	2	3	
1	0			x
1	1			x
1	2			x
1	3			

0	1	2	3	
3	0			
3	1			
3	2			x
3	3			x



Abordarea log sau arbore



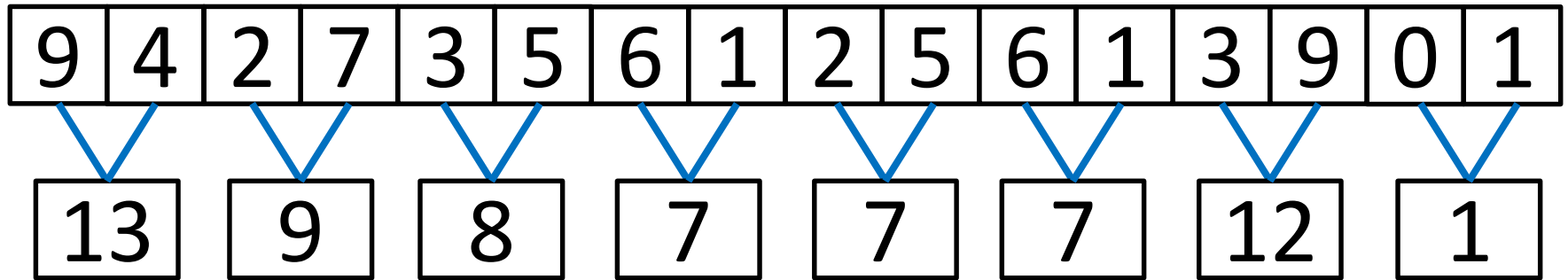
Reduce

Apply the same operation between all the elements of a vector.

Can be executed in $\log(n)$ time using a tree form.

The operation can take many forms (+, *, min, max, and, etc.)

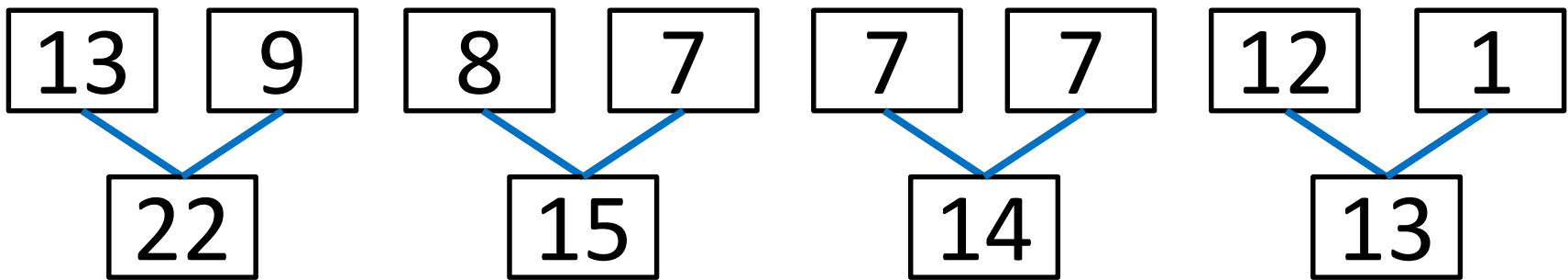
Reduce - sum



Can all be executed in parallel



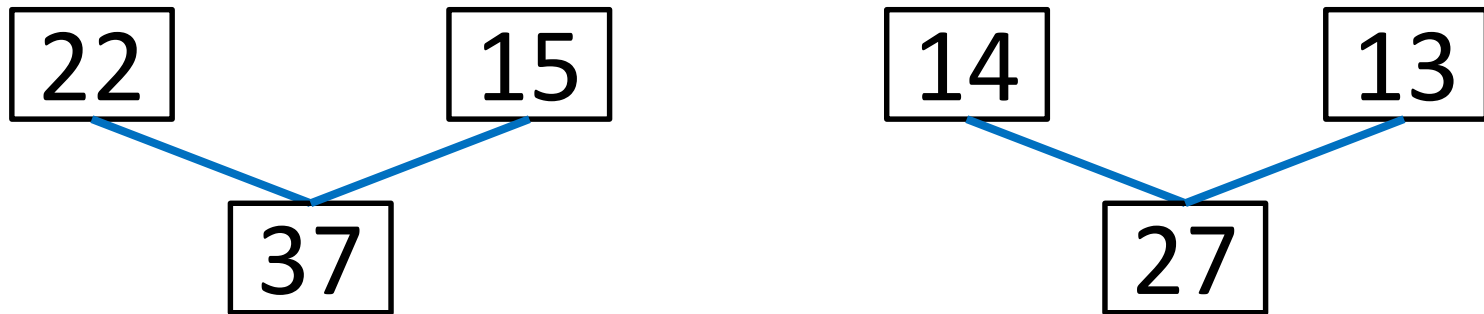
Reduce - sum



Can all be executed in parallel



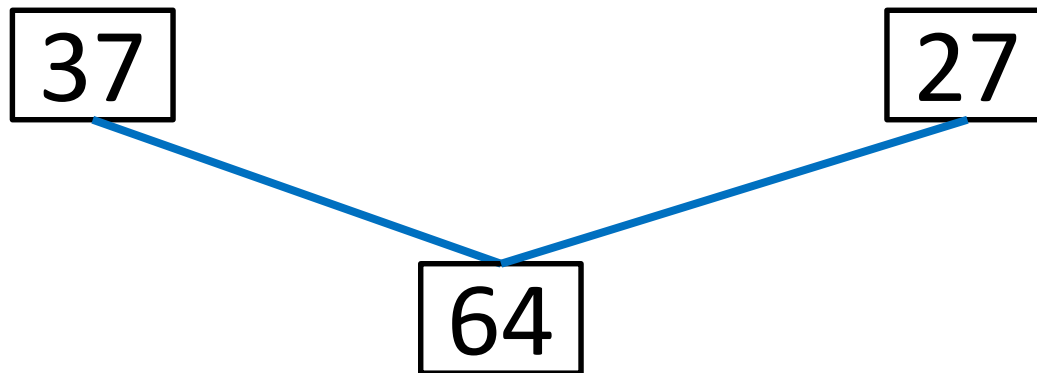
Reduce - sum



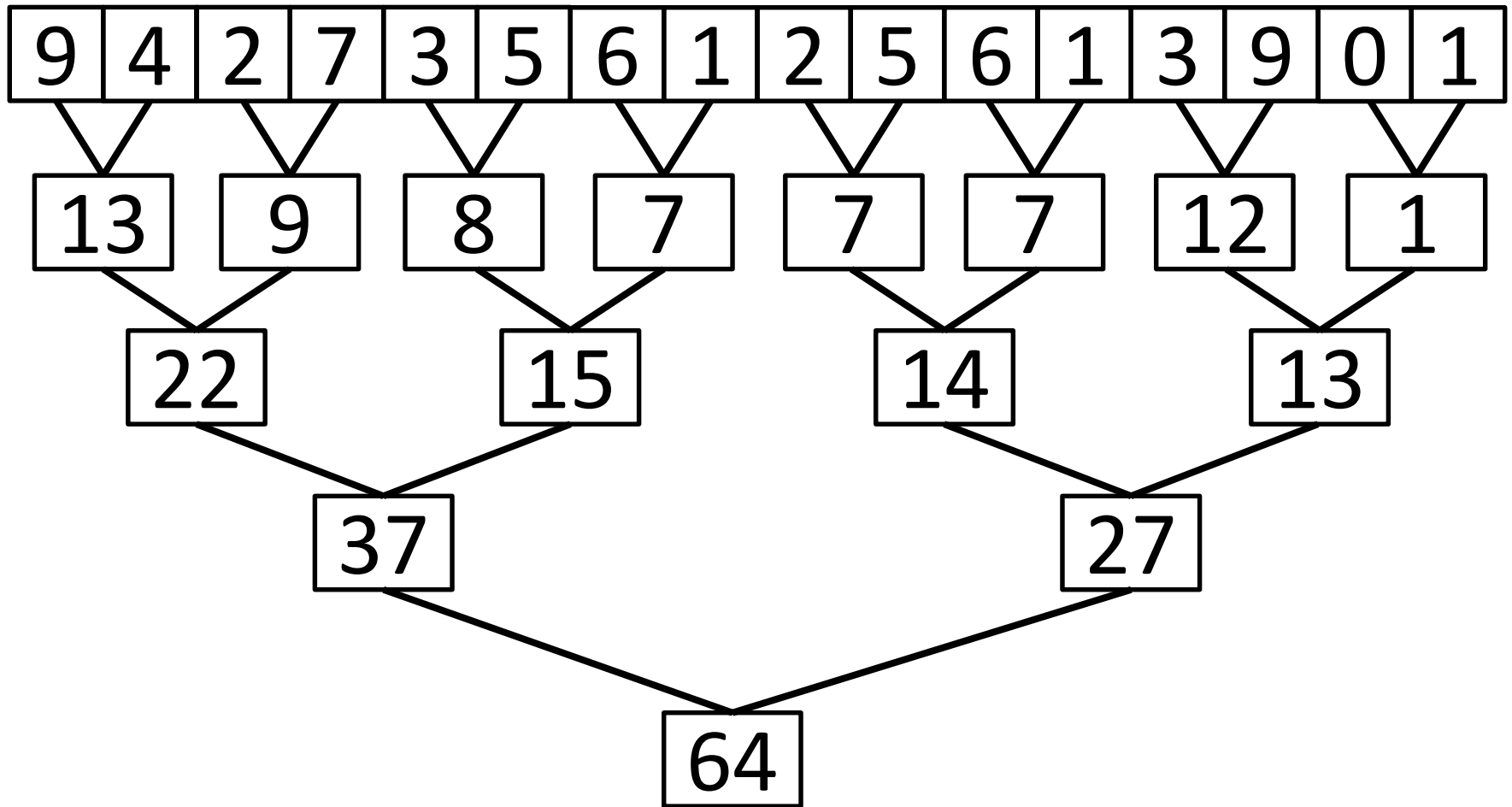
Can all be executed in parallel



Reduce - sum



Reduce - sum







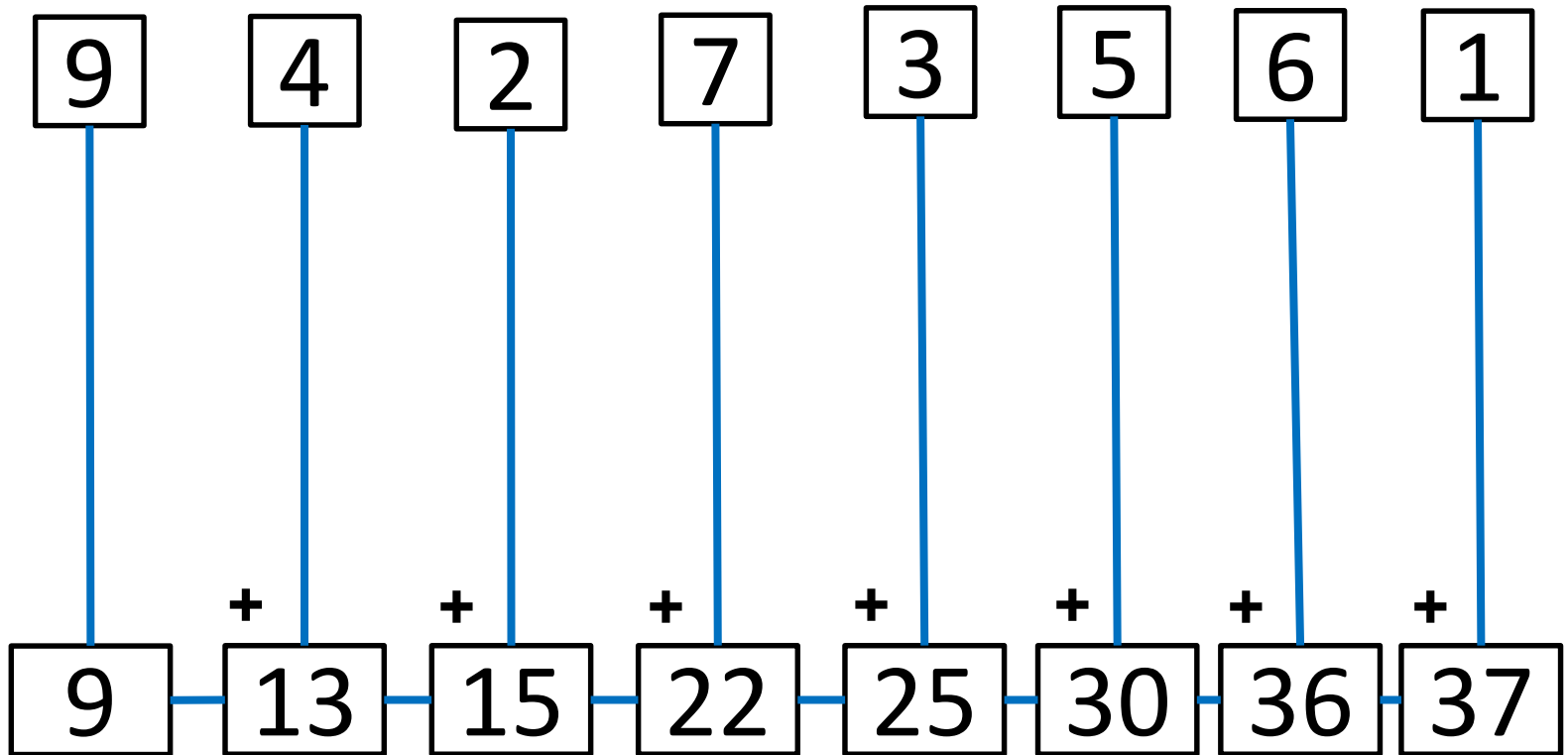
Scan

**Apply the same operation between all the elements of a vector.
Obtains all partial results.**

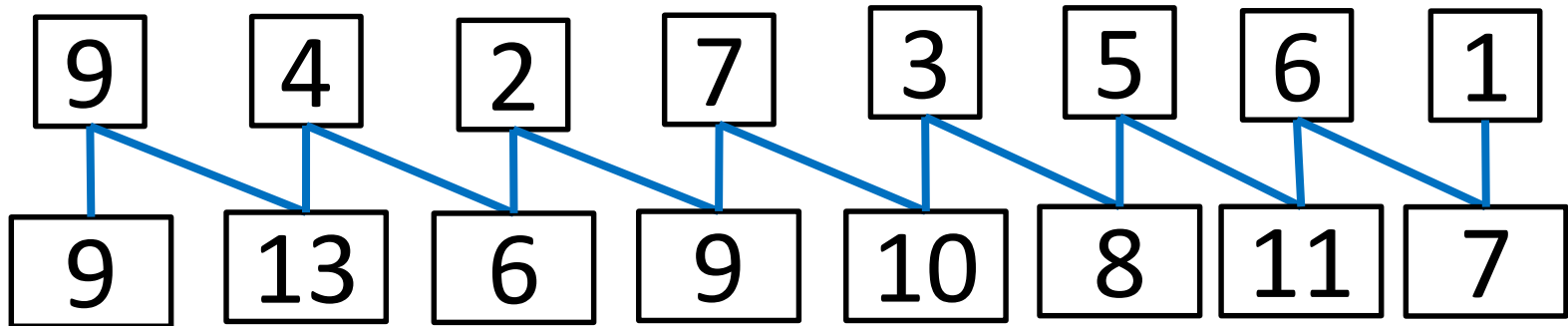
Can be executed in $\log(n)$ time using a special tree form.

The operation can take many forms (+, *, min, max, and, etc.)

Scan - sum



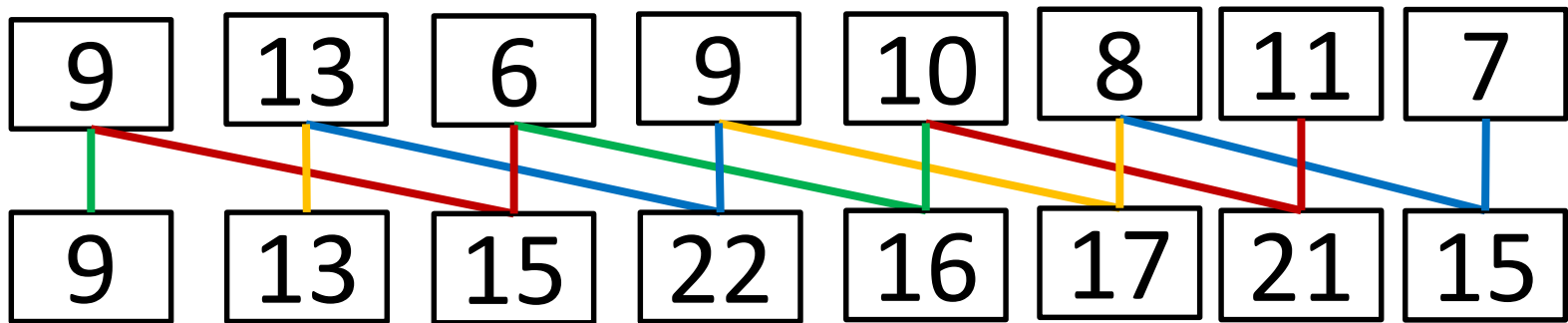
Scan - sum



Can all be executed in parallel



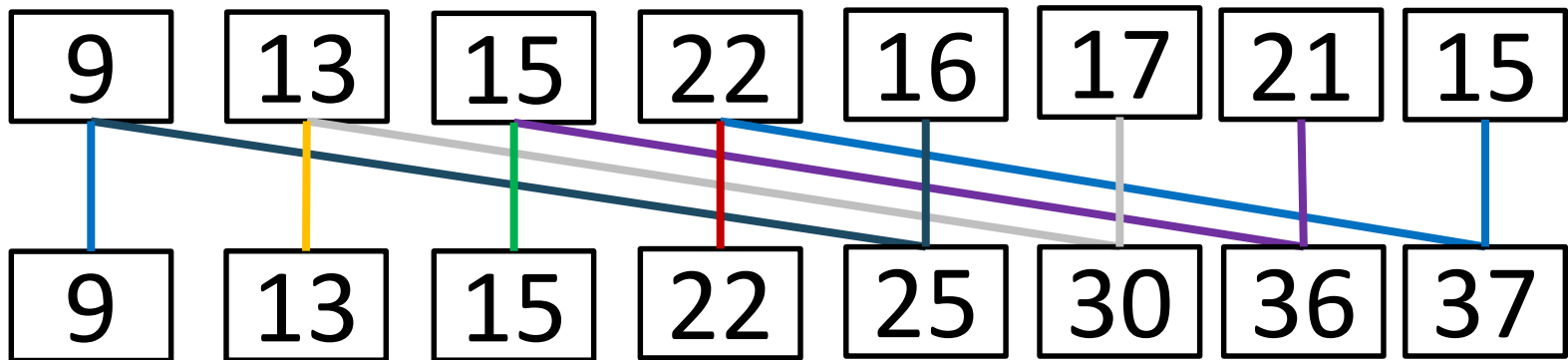
Scan - sum



Can all be executed in parallel



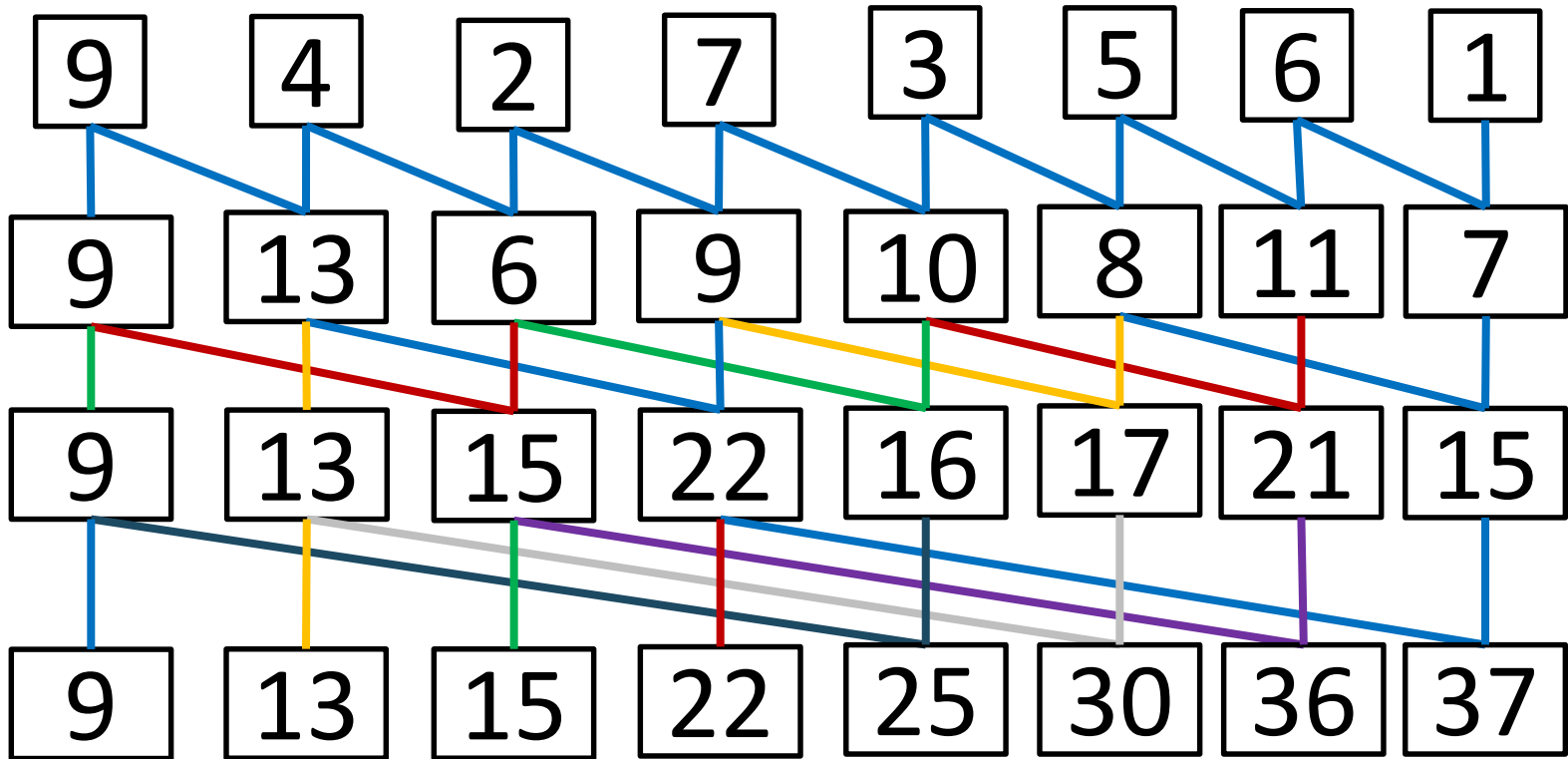
Scan - sum



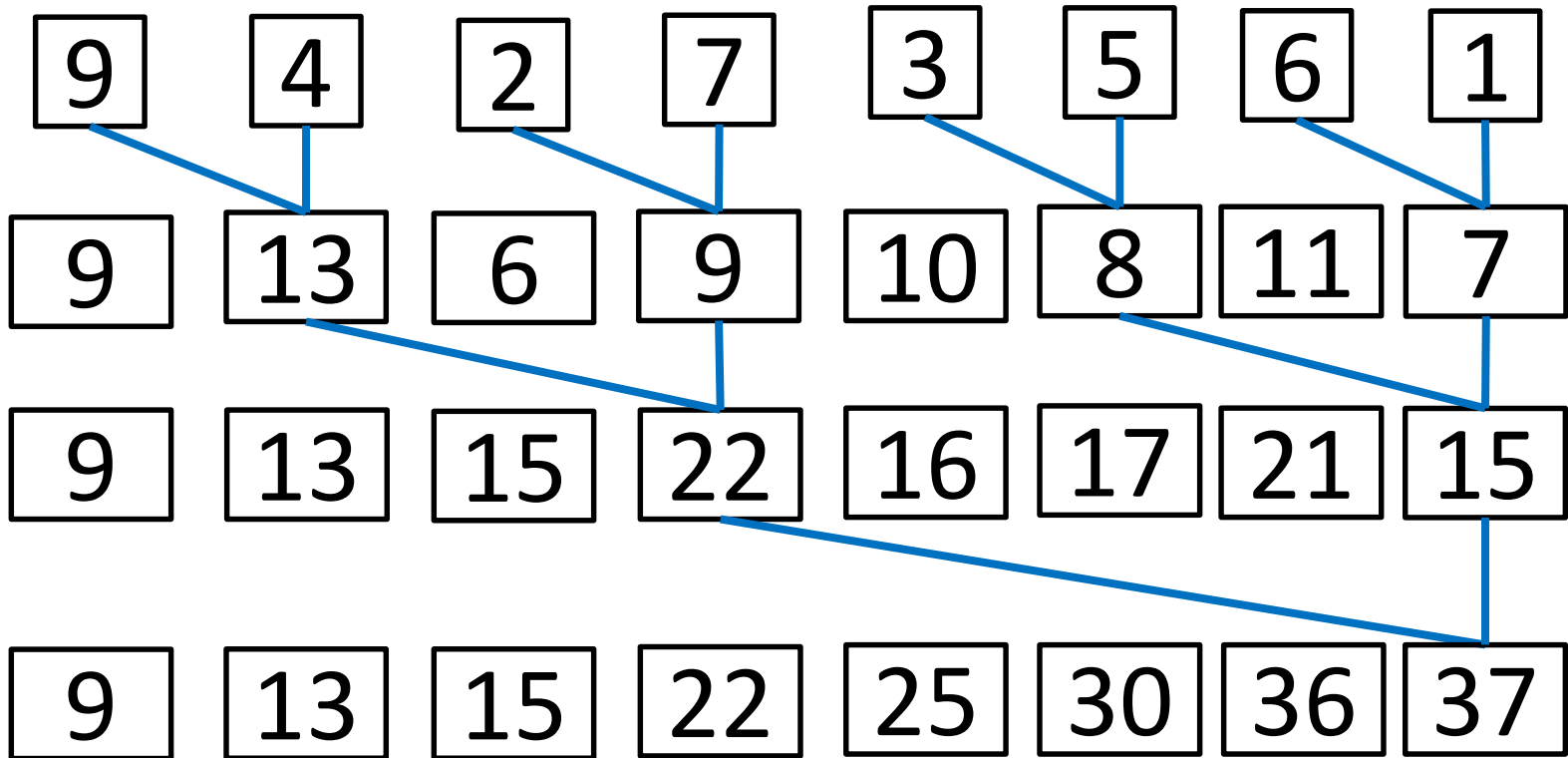
Can all be executed in parallel



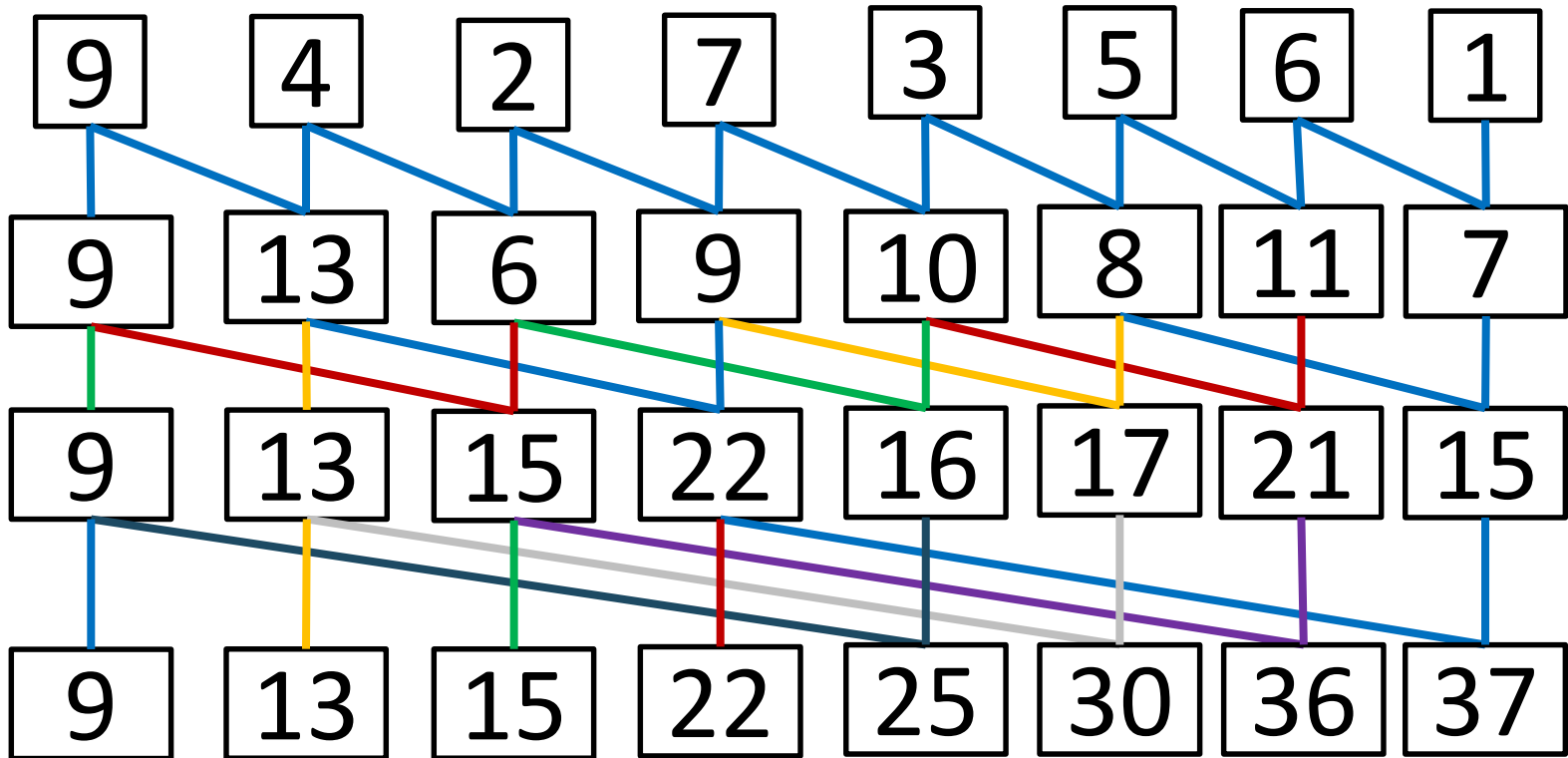
Scan – How it works?



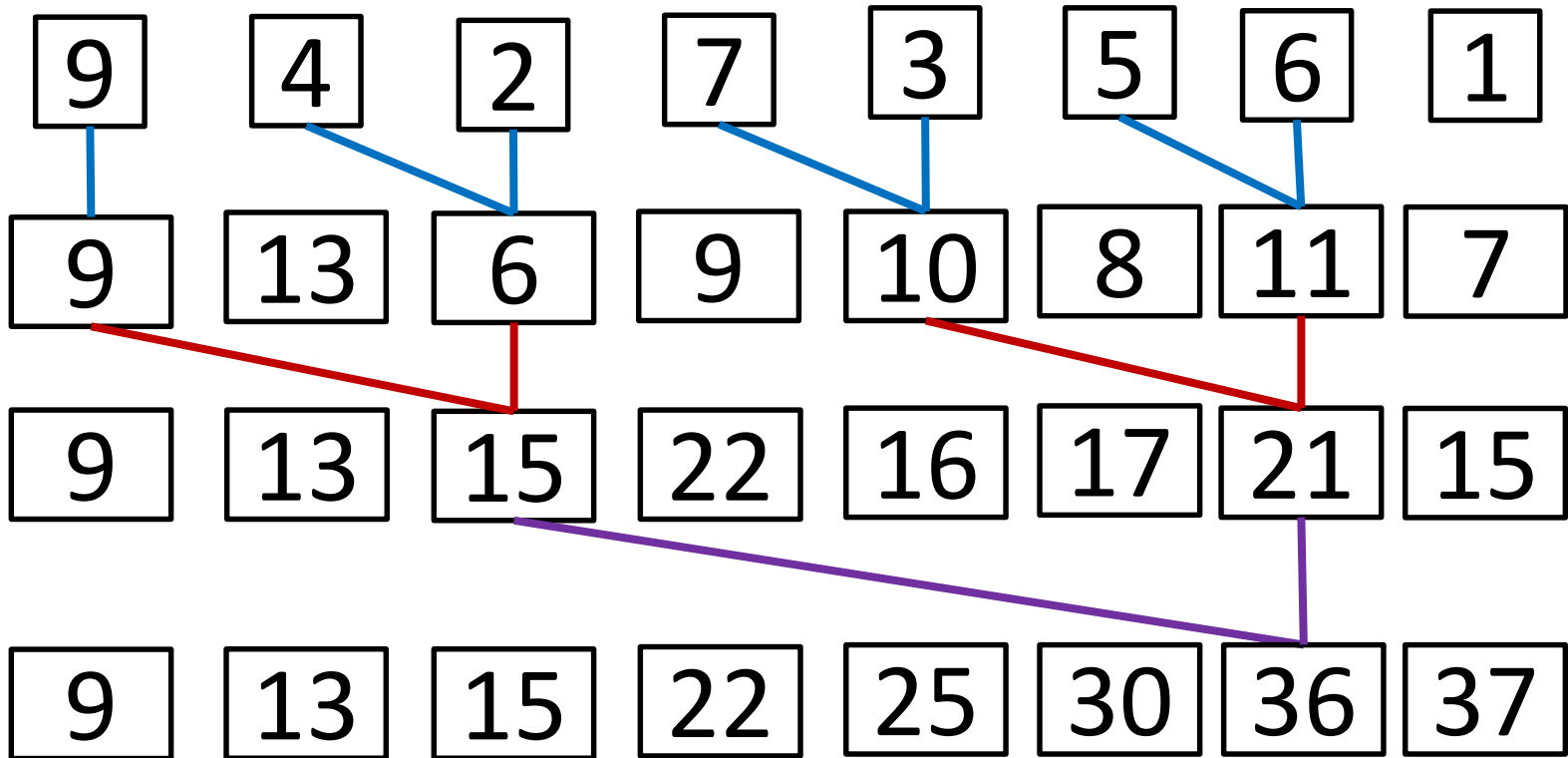
Scan – How it works?



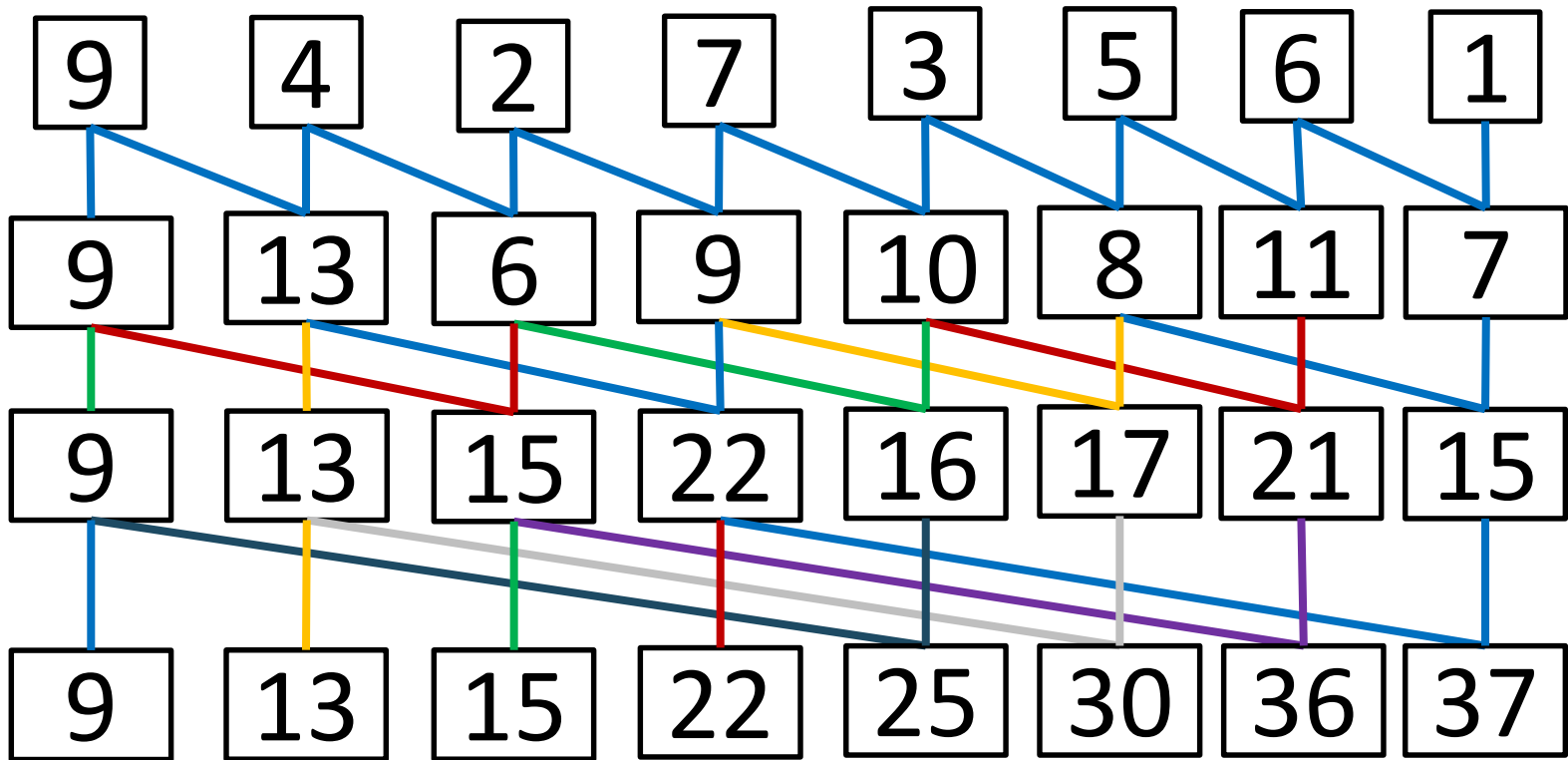
Scan – How it works?



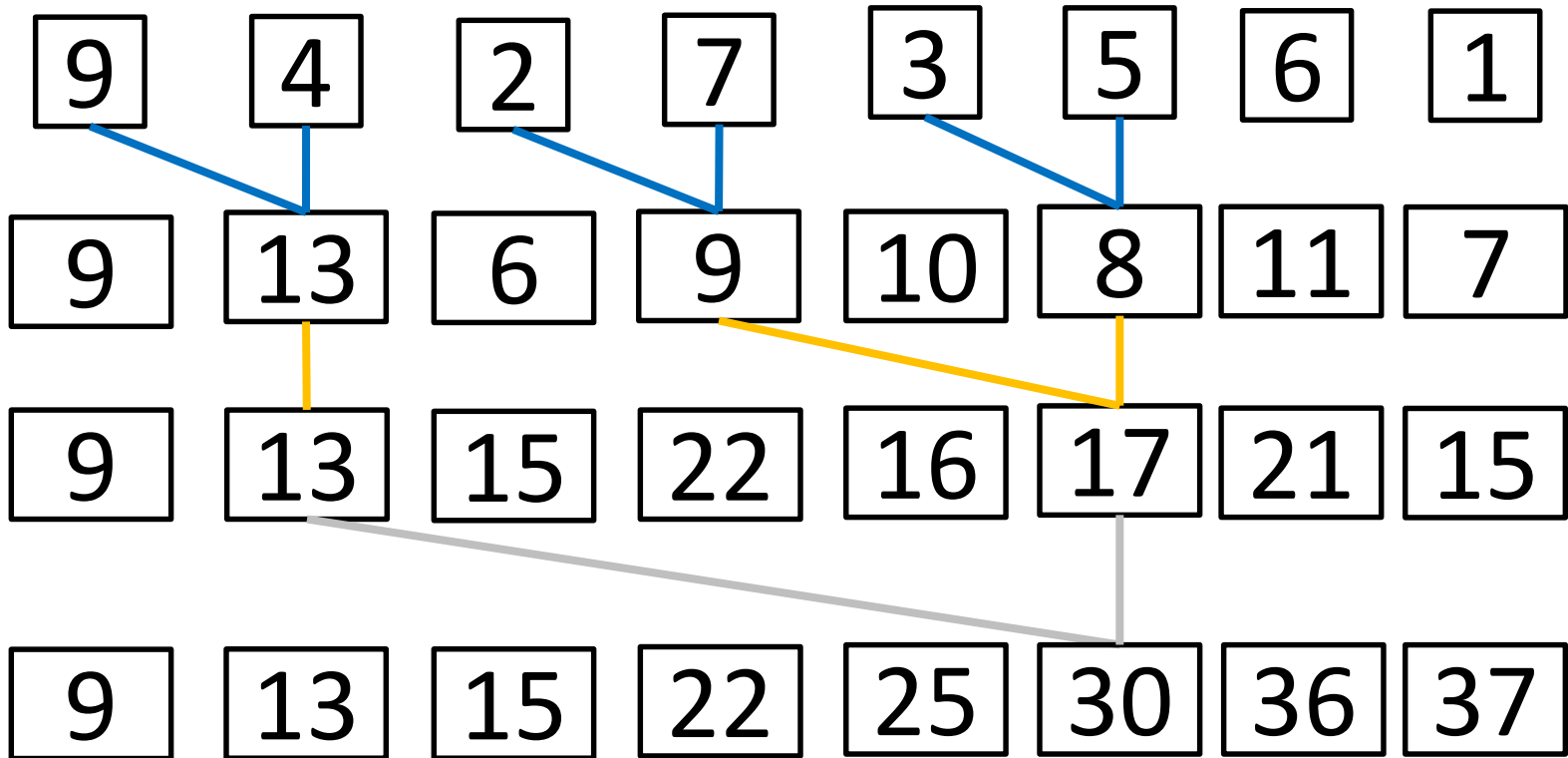
Scan – How it works?



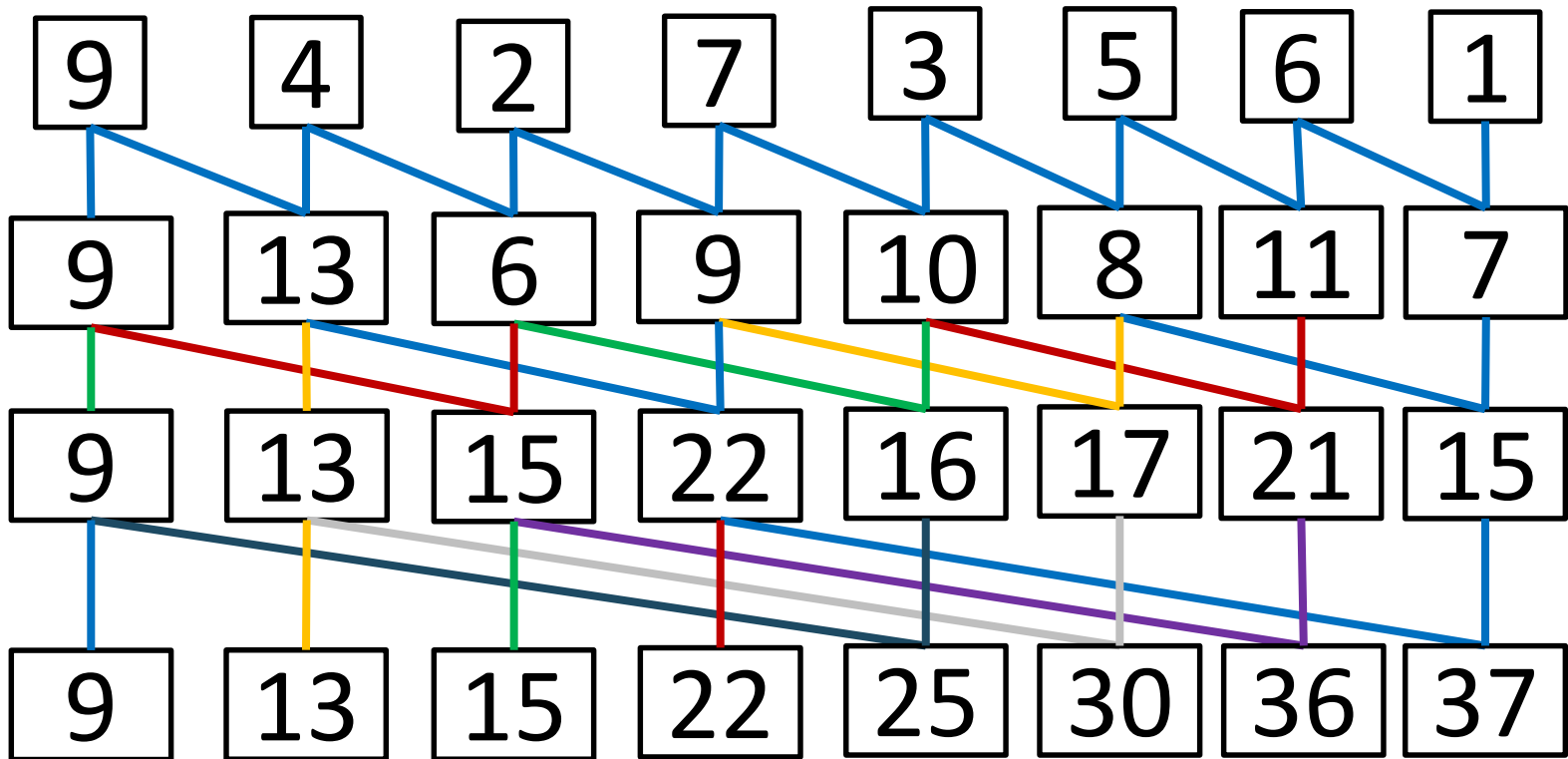
Scan – How it works?



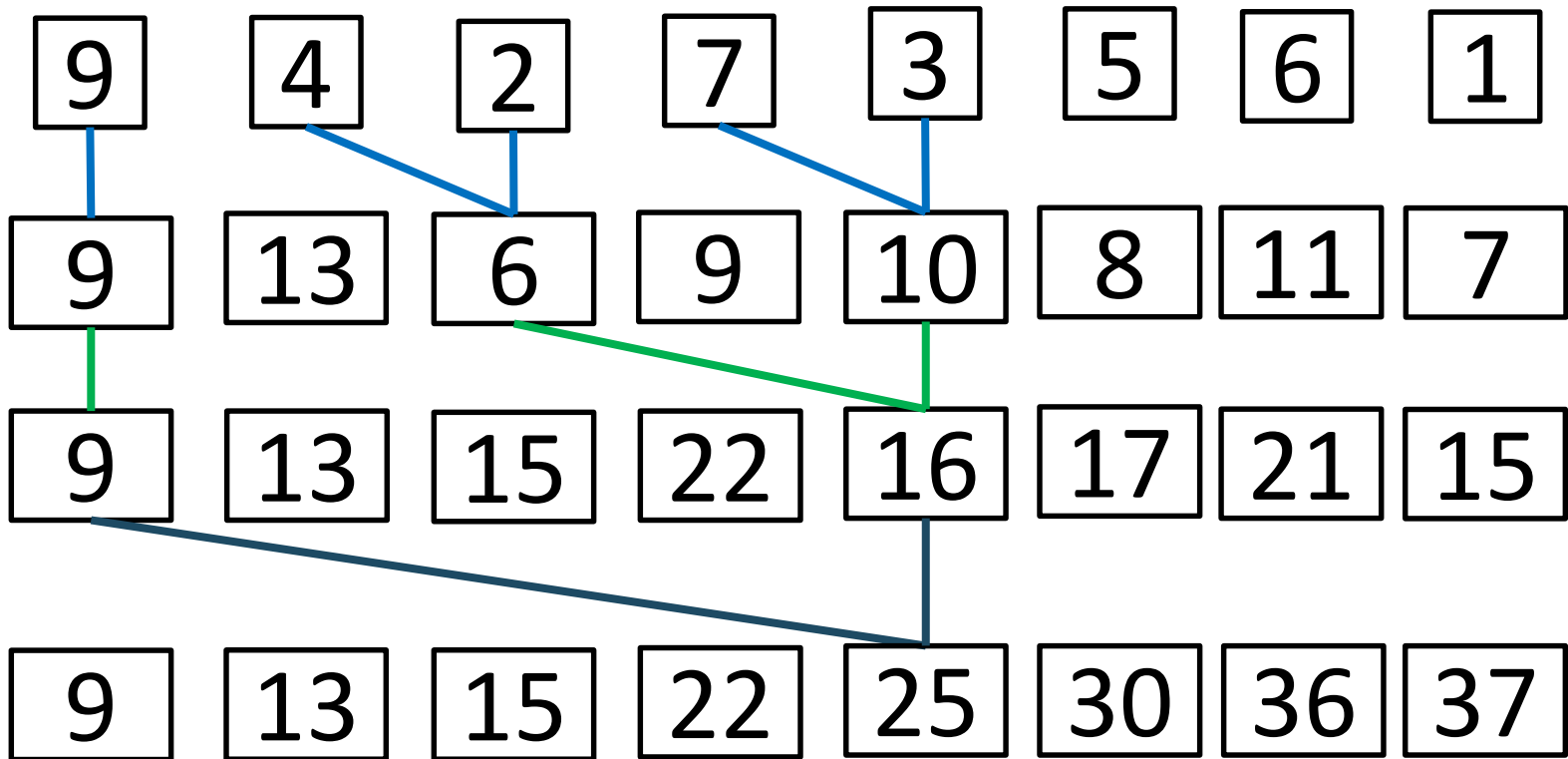
Scan – How it works?



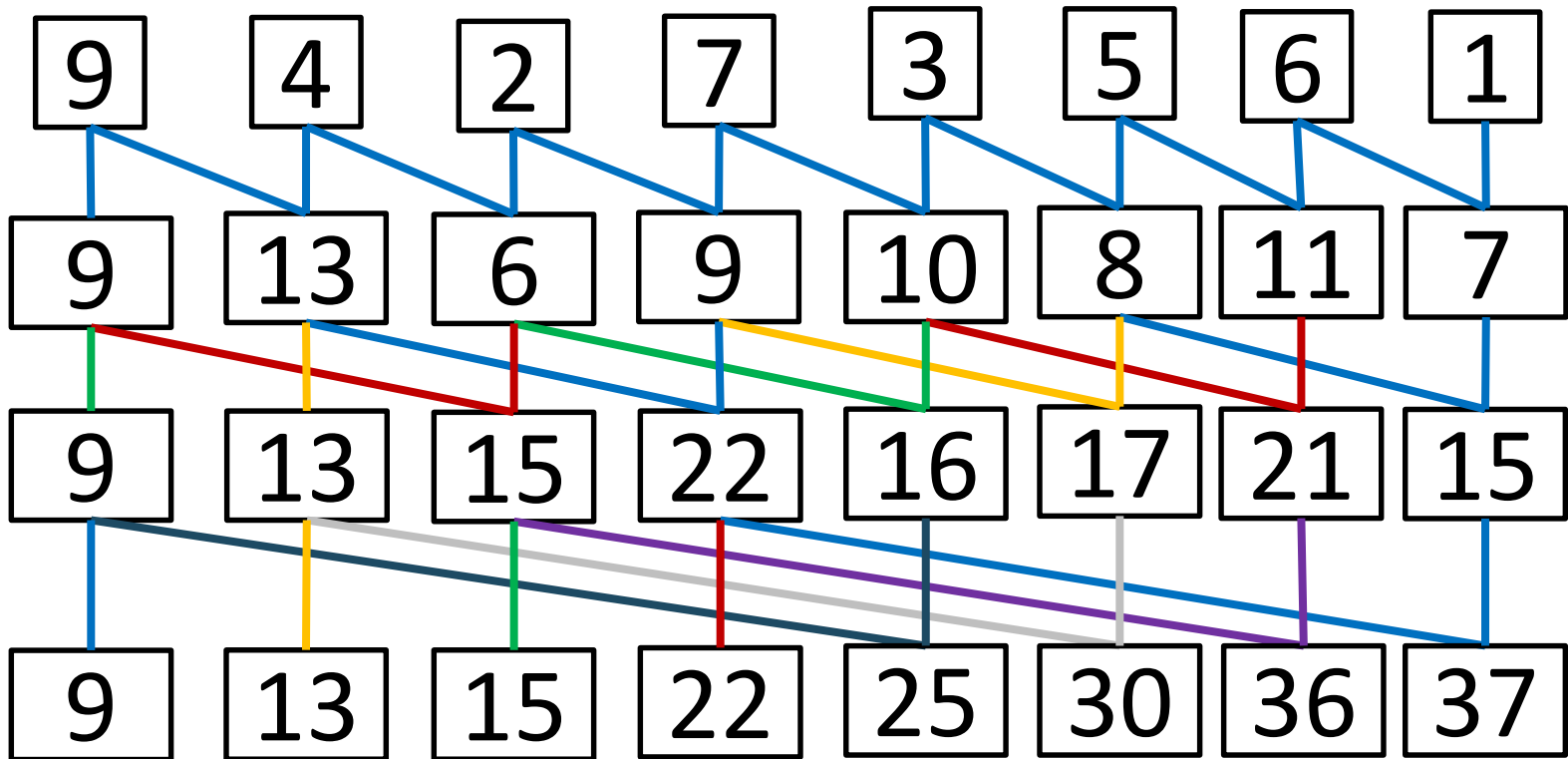
Scan – How it works?



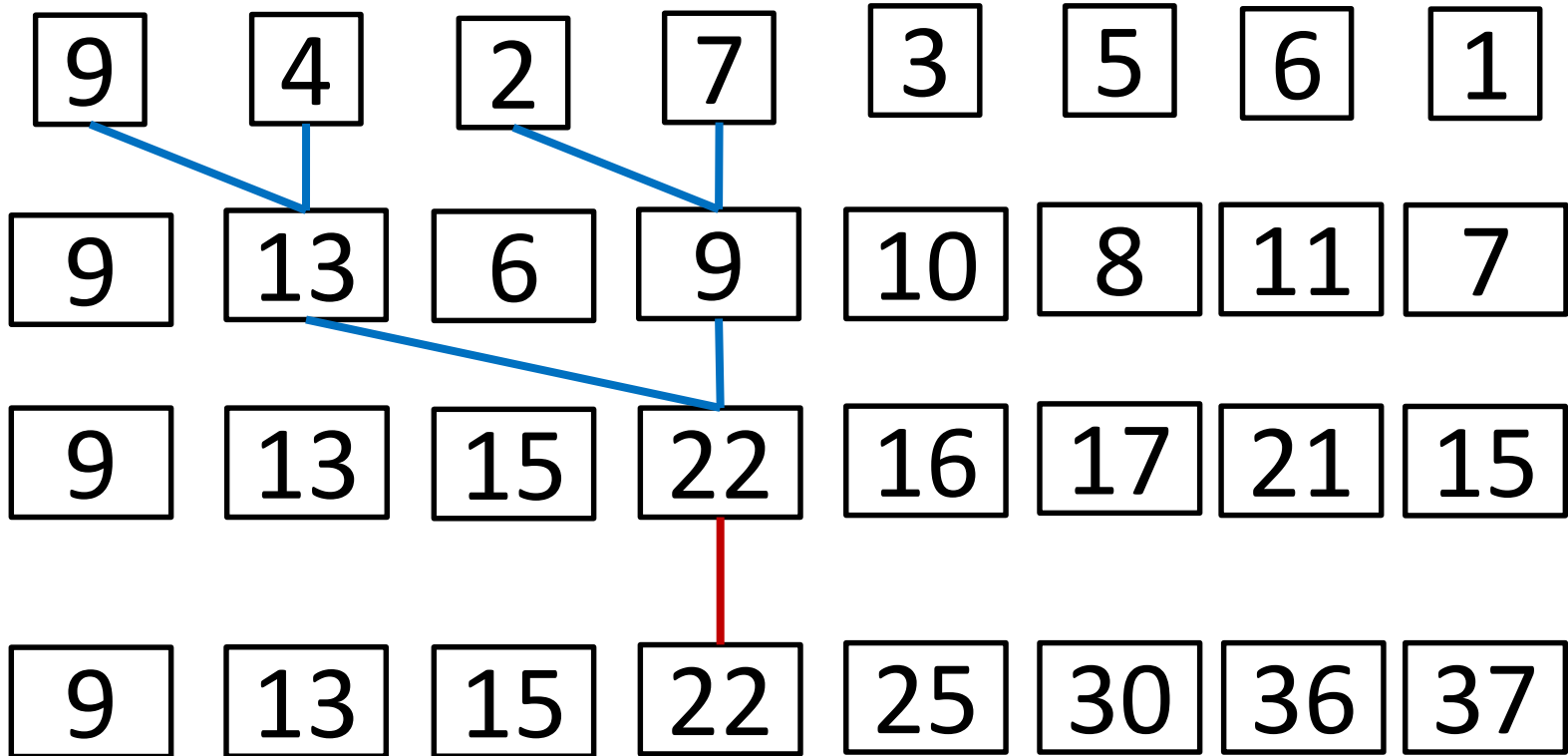
Scan – How it works?



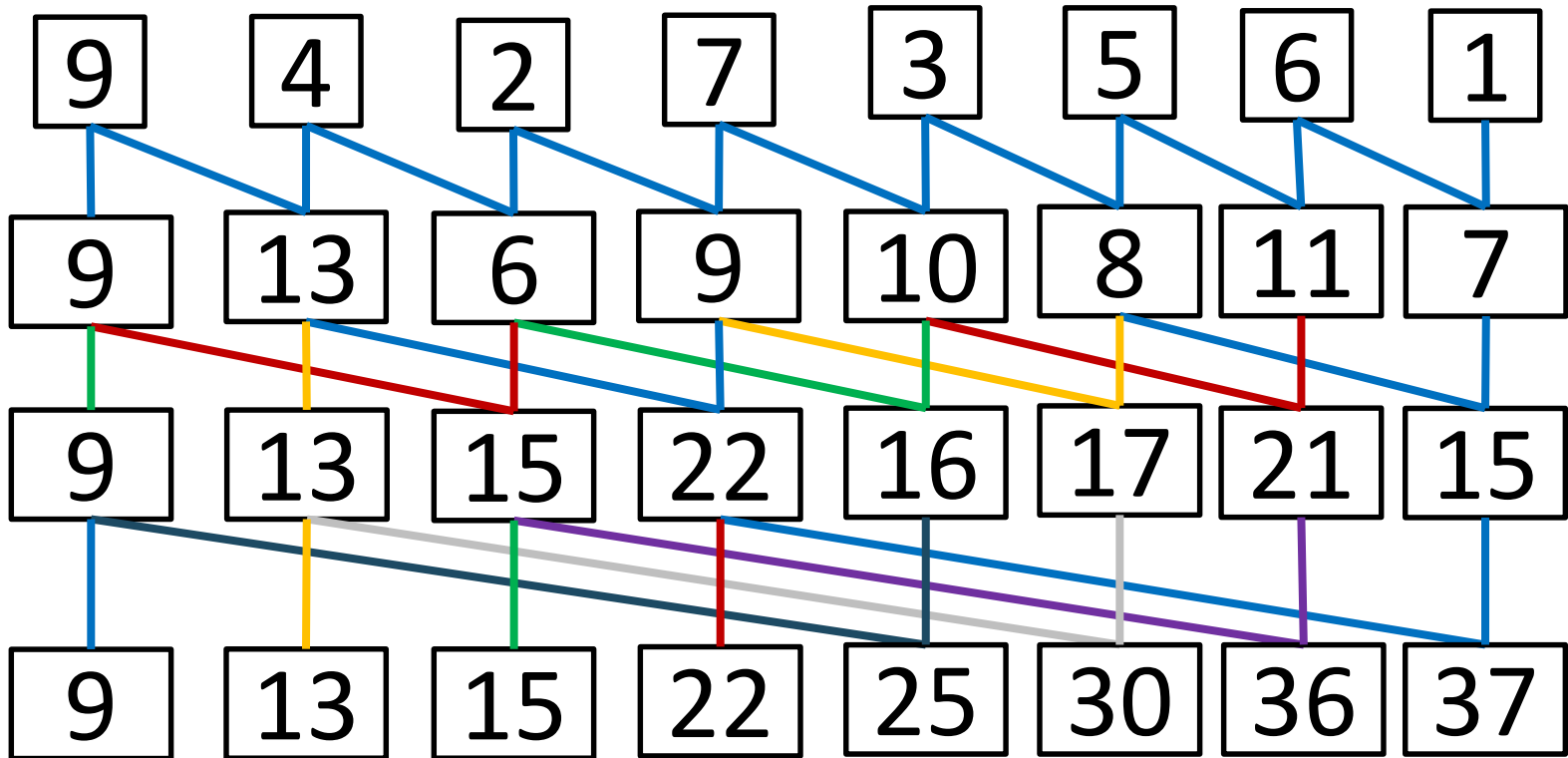
Scan – How it works?



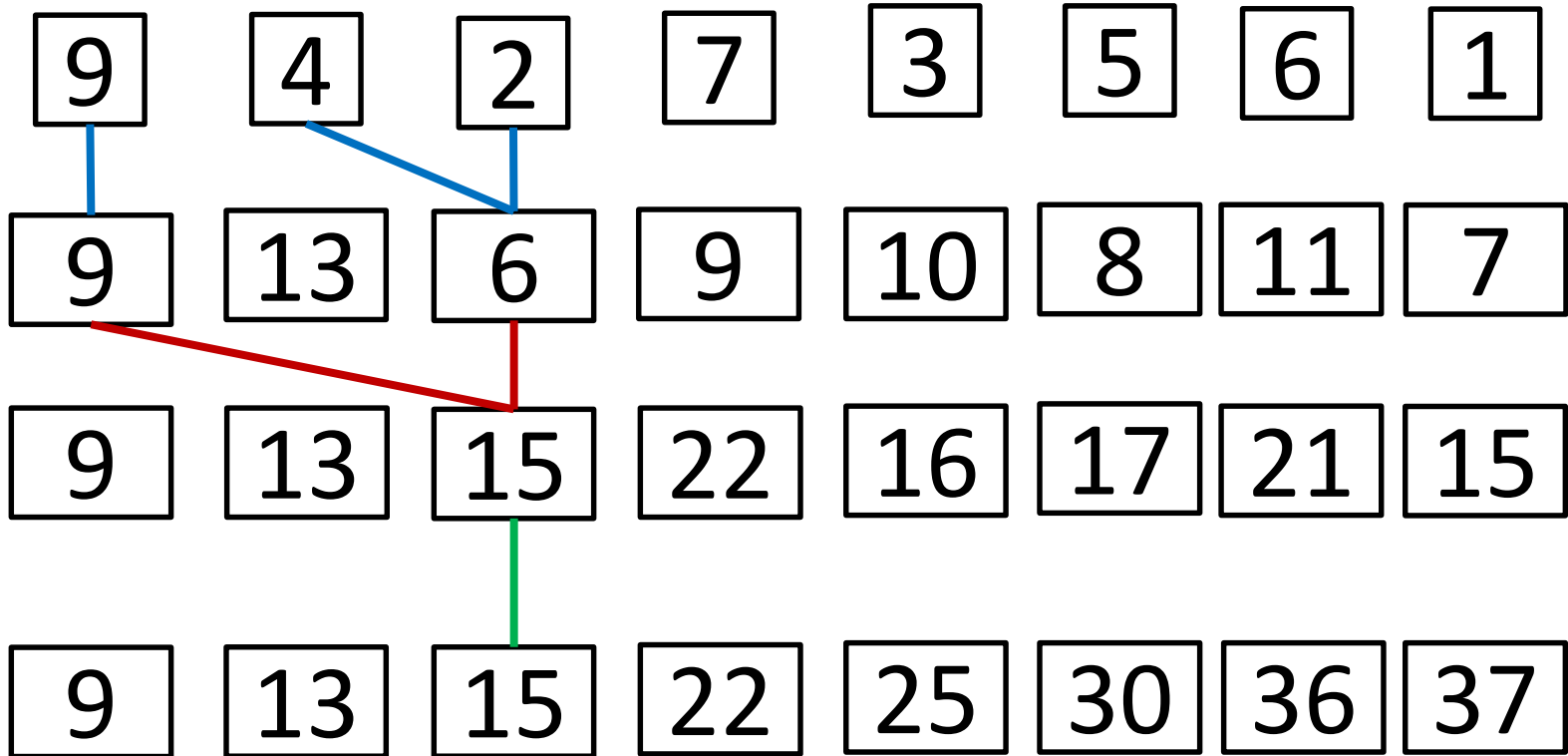
Scan – How it works?



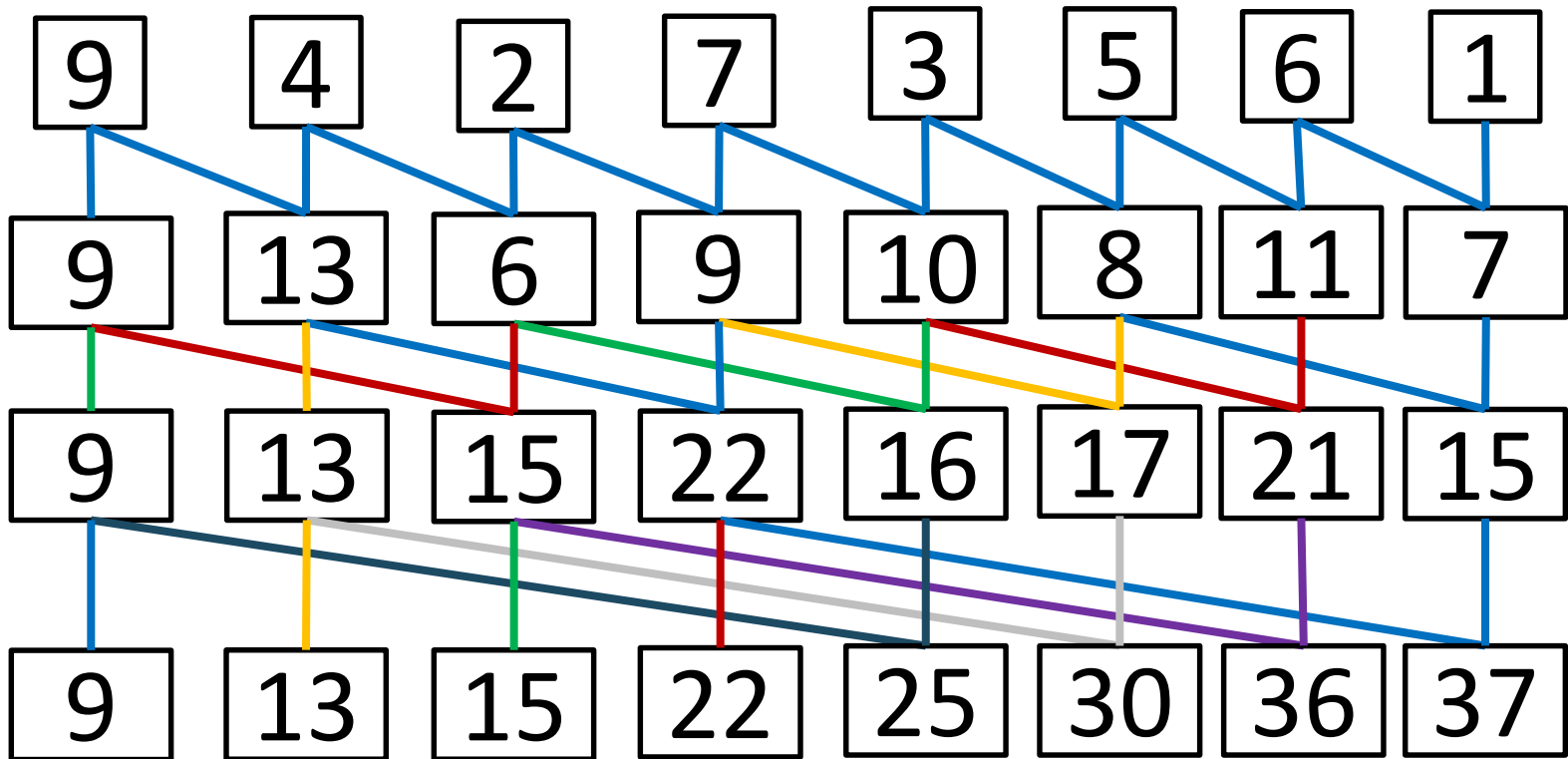
Scan – How it works?



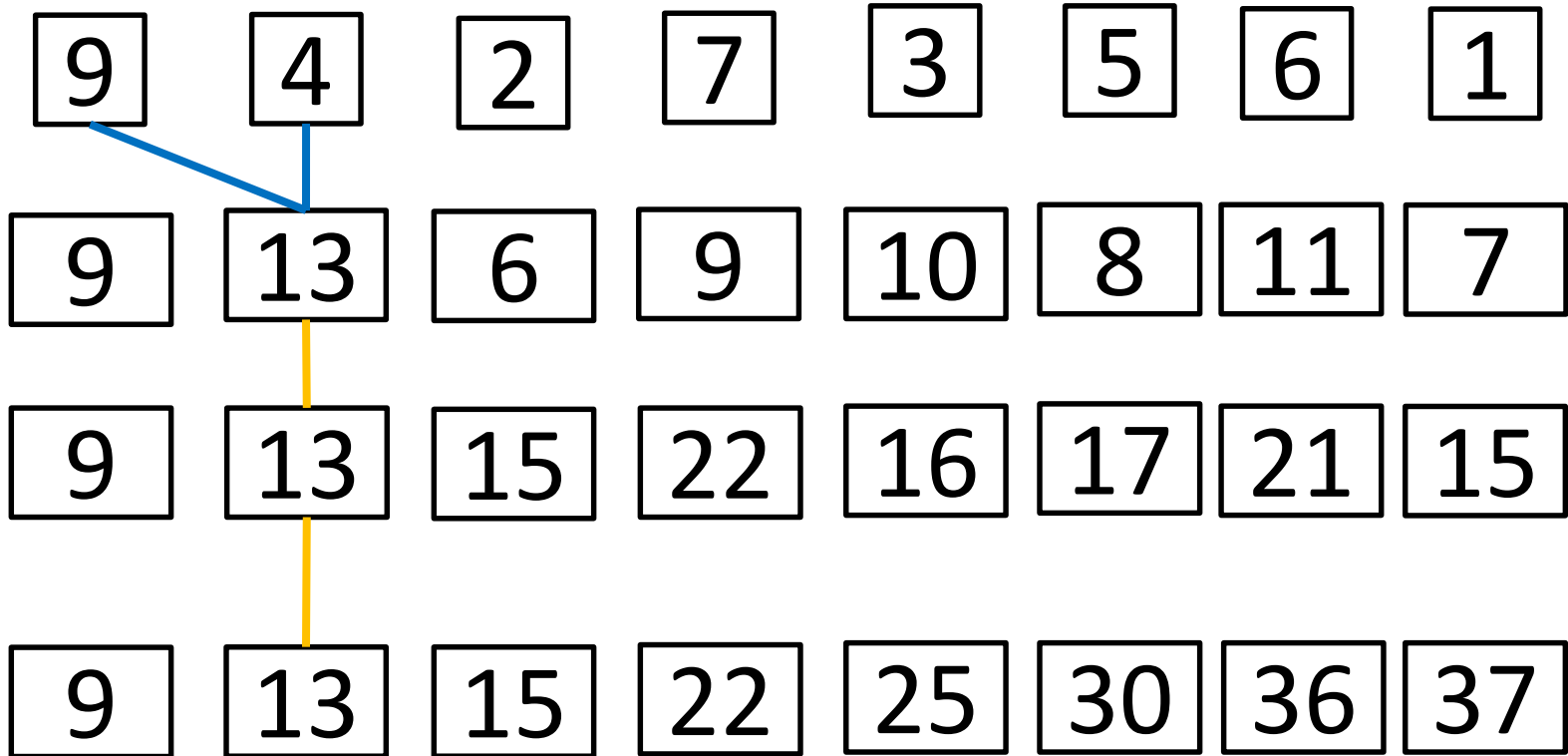
Scan – How it works?



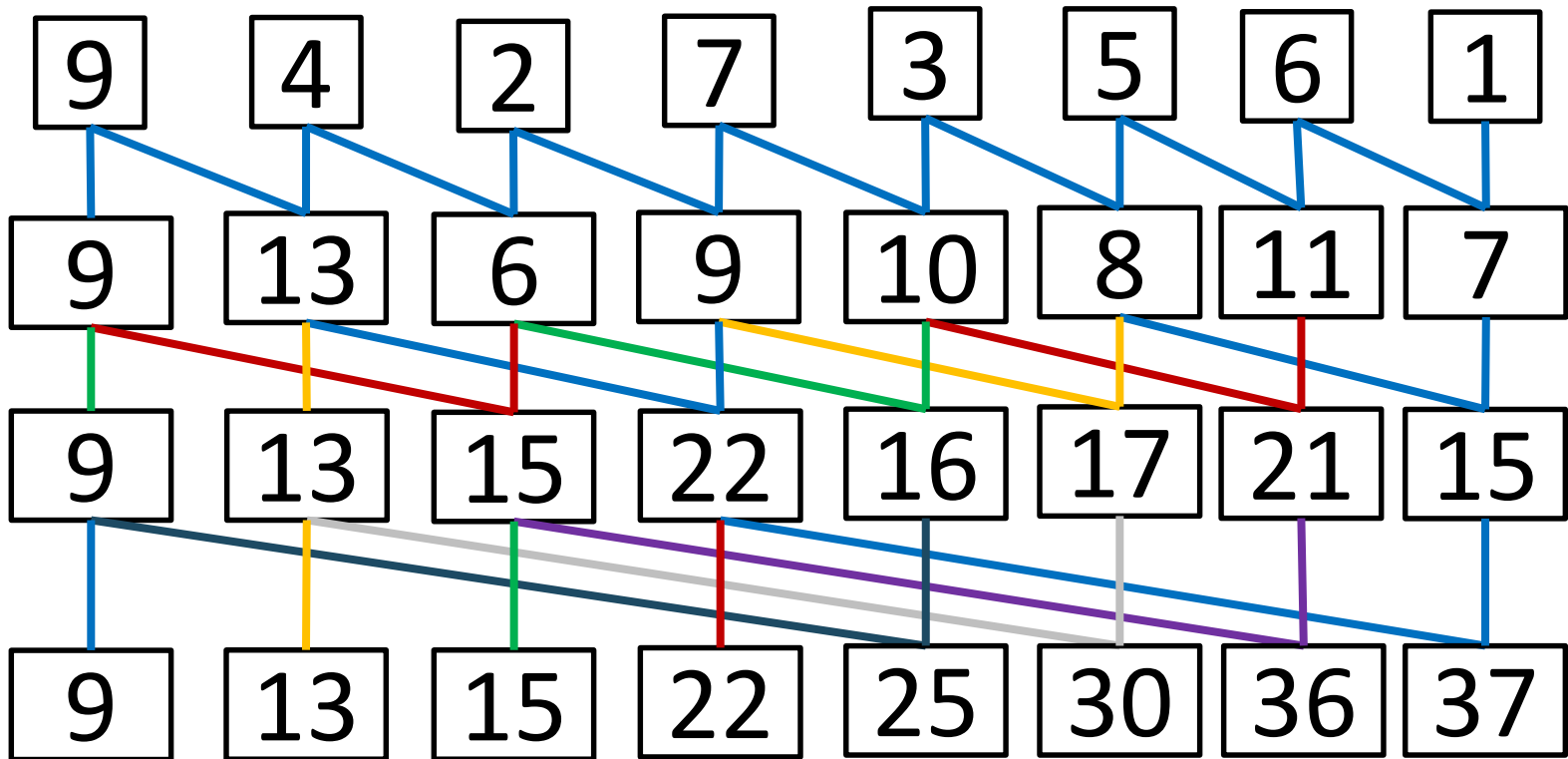
Scan – How it works?



Scan – How it works?



Scan – How it works?



Scan – How it works?

9	4	2	7	3	5	6	1
9	13	6	9	10	8	11	7
9	13	15	22	16	17	21	15
9	13	15	22	25	30	36	37





Value Broadcast

Start

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...

7																...
---	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	-----



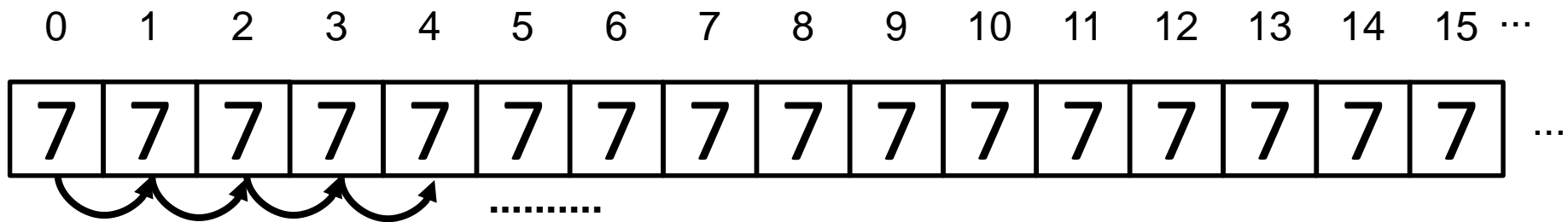
Value Broadcast

End

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	...

Inefficient Value Broadcast

$O(n)$ time

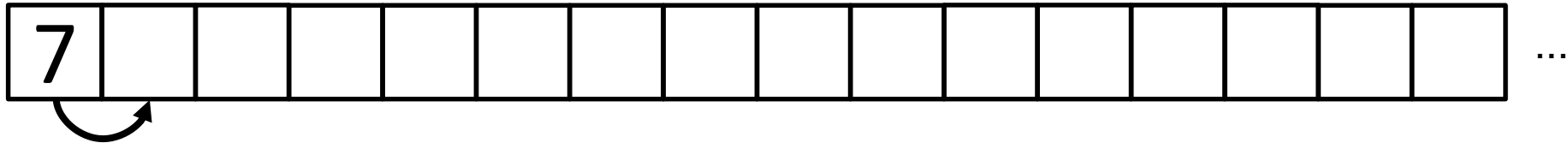


Efficient Value Broadcast

Every element that has the value
copies it to its current position + i

$i = 1$

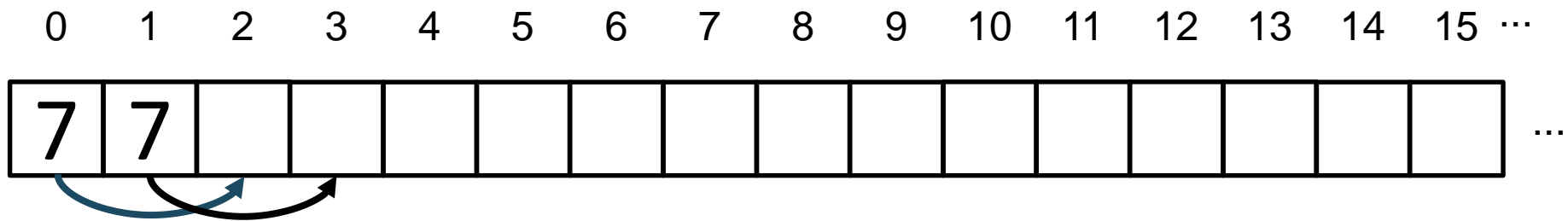
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...



Efficient Value Broadcast

Every element that has the value
copies it to its current position + i

$i = 2$

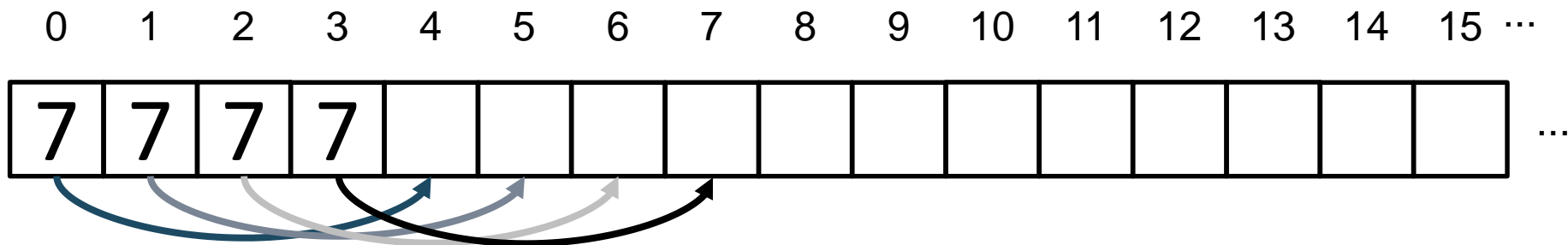


These operations can be executed in parallel

Efficient Value Broadcast

Every element that has the value
copies it to its current position + i

$i = 4$

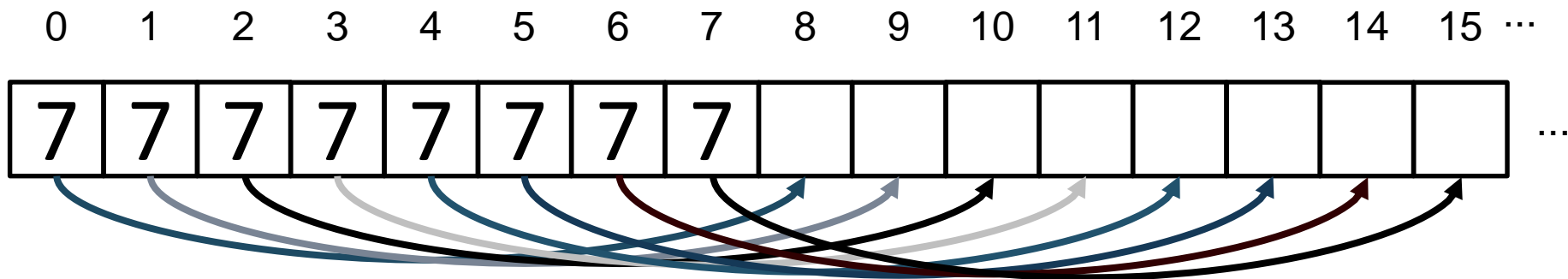


These operations can be executed in parallel

Efficient Value Broadcast

Every element that has the value
copies it to its current position + i

$i = 8$

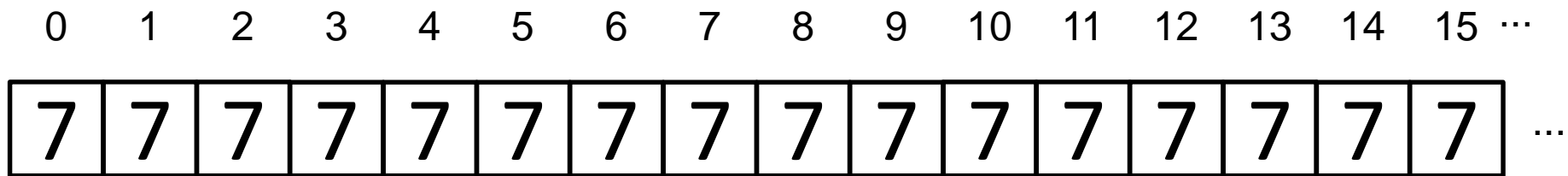


These operations can be executed in parallel



Efficient Value Broadcast

$O(\log_2(n))$ time with $p = n/2$ processors



Efficient Value Broadcast

$i = 2$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...

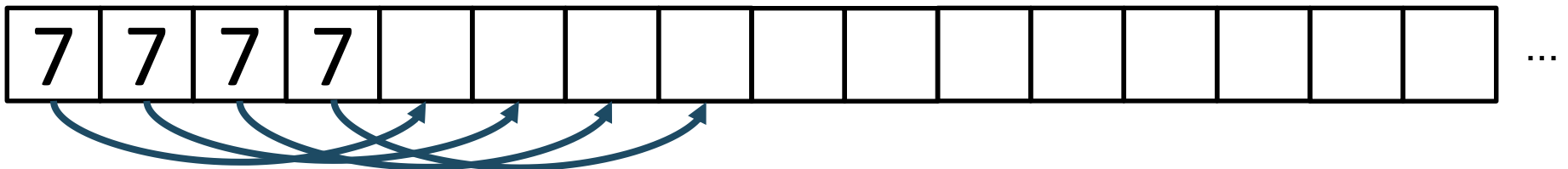


These operations can **NOT** be executed in parallel



$i = 4$

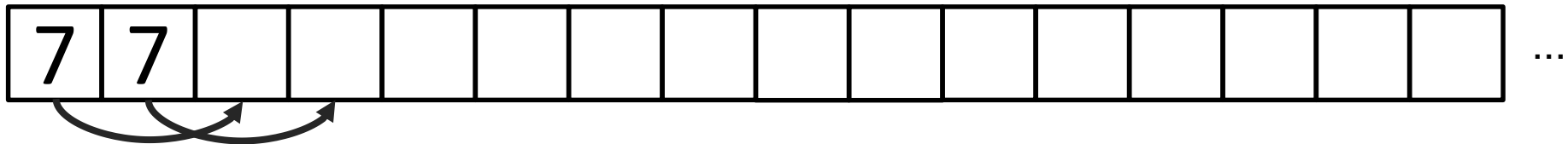
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...



Efficient Value Broadcast

$i = 2$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...



$i = 4$

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 ...

