Analysis of Algorthims
Big O Notation

Big O

- Big O describes how the time taken, or memory used, by a program scales with the amount of data it has to work on
- Big O describes the ‘complexity’ of a program
- Common sense tells us that a program takes longer when there is more data to work on… But not necessarily

Big O Complexities

- Linear search
- Stack
- Bubble sort
- Binary search
- Merge sort

Linear Search

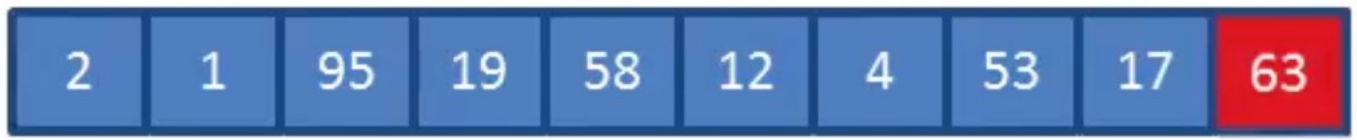
- Sometimes referred to as a sequential search
- An unordered list is searched for a particular value
- Each value in the list is compared with the target value
- Linear search implemented with a simple loop

Pseudocode

```
FOR i = 0 TO n - 1
    IF ArrayToSearch(i) = Target THEN
        bFound = True
        EXIT FOR
    END IF
NEXT i
```

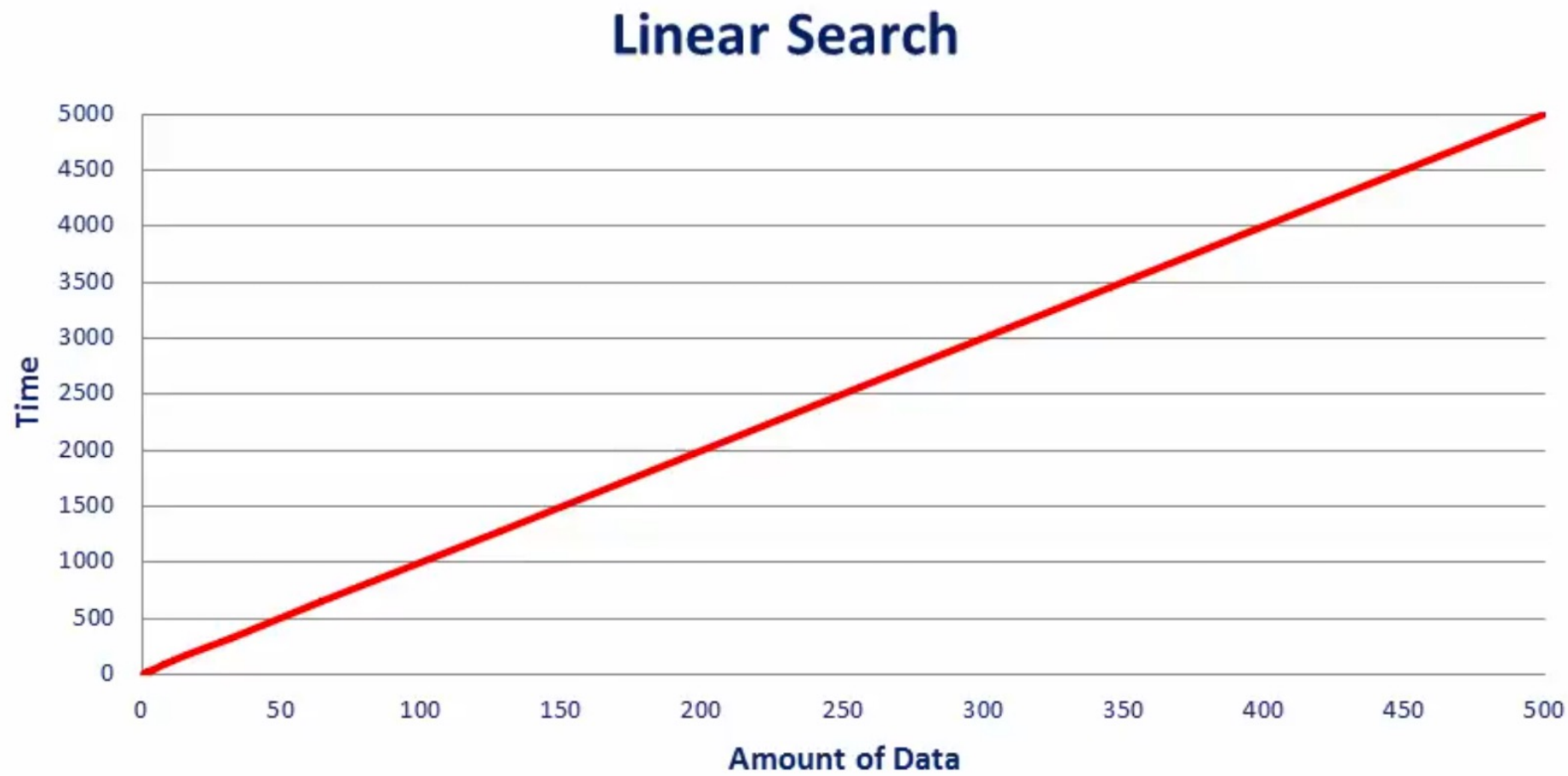

Target
63
2
1
95
19
63
Target
63
2
1
95
19
63

Target 63



Target 63

Data	Time
1	10
2	20
4	40
8	80
16	160
32	320
64	640
128	1280
256	2560
512	5120

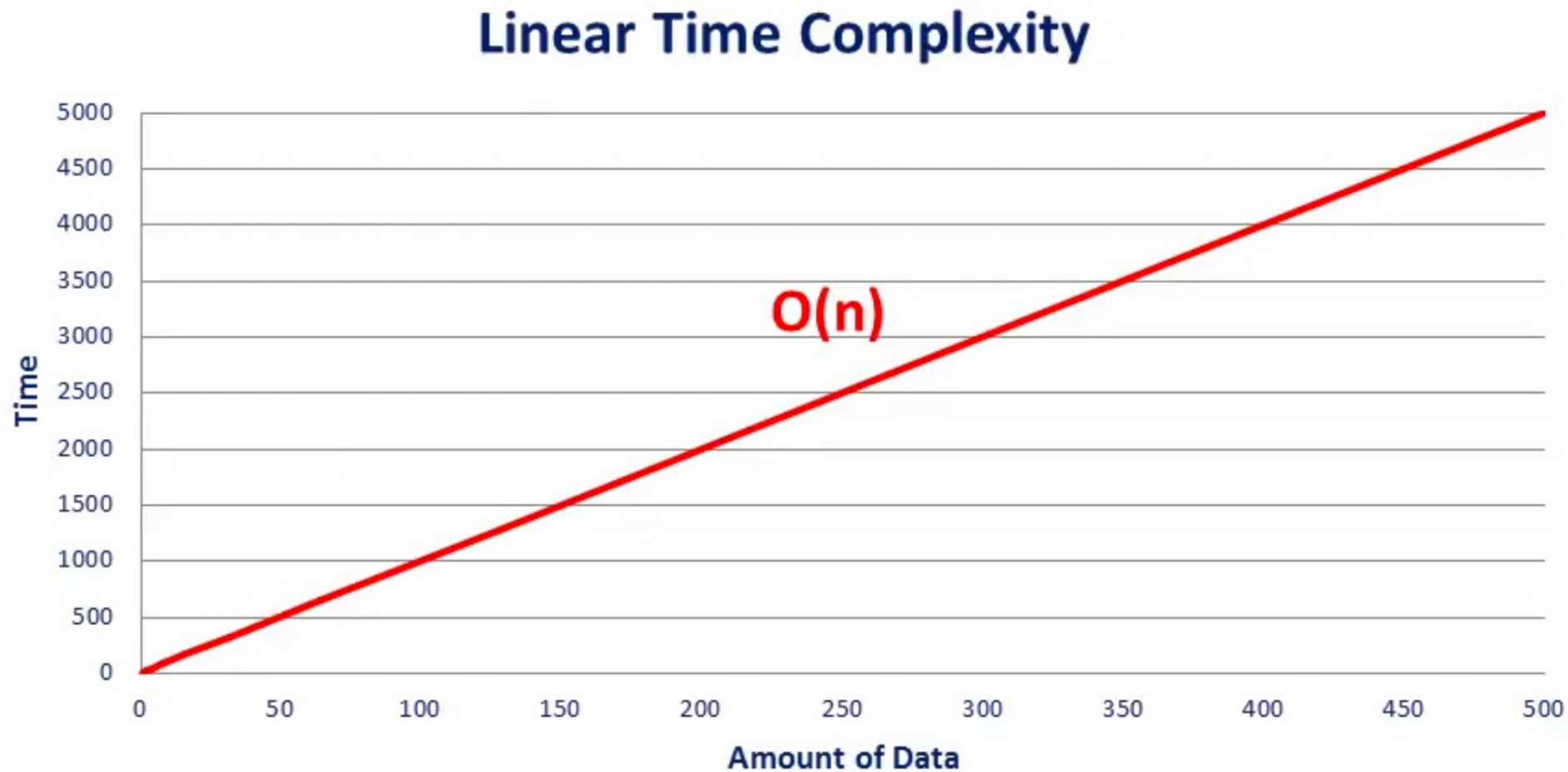


Linear Search Complexity

- For n data items, the time taken is equal to some constant multiplied by n
- The Big O time complexity is **Linear**

O(n)

Data	Time
1	10
2	20
4	40
8	80
16	160
32	320
64	640
128	1280
256	2560
512	5120



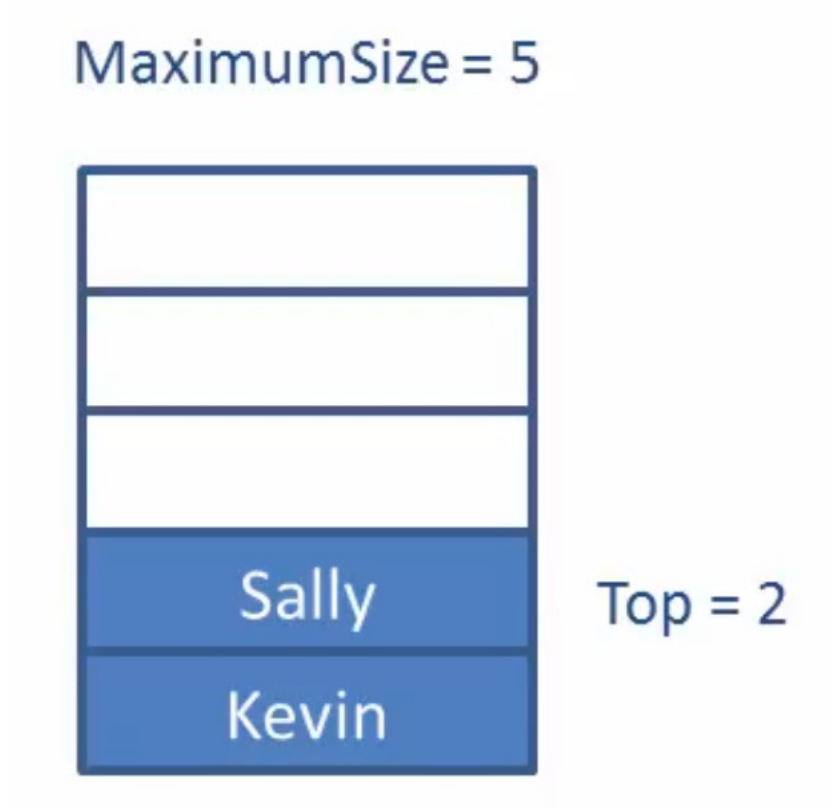
Stack

- Items are pushed onto and popped off the top of a stack
- Peek examines top item without removing it
- Last in first out data structure (LIFO)
- Implemented with an array and a pointer to the top item

```
Procedure Push
    IF Top = MaximumSize THEN
        OUTPUT “Stack overflow”
    ELSE
        Top = Top + 1
        ArrayStack(Top) = new item
    END IF
END Procedure
```

```
Procedure Push
    IF Top = MaximumSize THEN
        OUTPUT “Stack overflow”
    ELSE
        Top = Top + 1
        ArrayStack(Top) = new item
    END IF
END Procedure
```

```
Procedure Pop
    IF Top = 0 THEN
        OUTPUT “Stack is empty”
    ELSE
        copy item = ArrayStack(Top)
        Top = Top - 1
    END IF
END Procedure
```


MaximumSize = 5
Sally
Kevin
Top = 2


Push
Beatrix
MaximumSize = 5
Sally
Kevin
Top = 2

Push

MaximumSize = 5

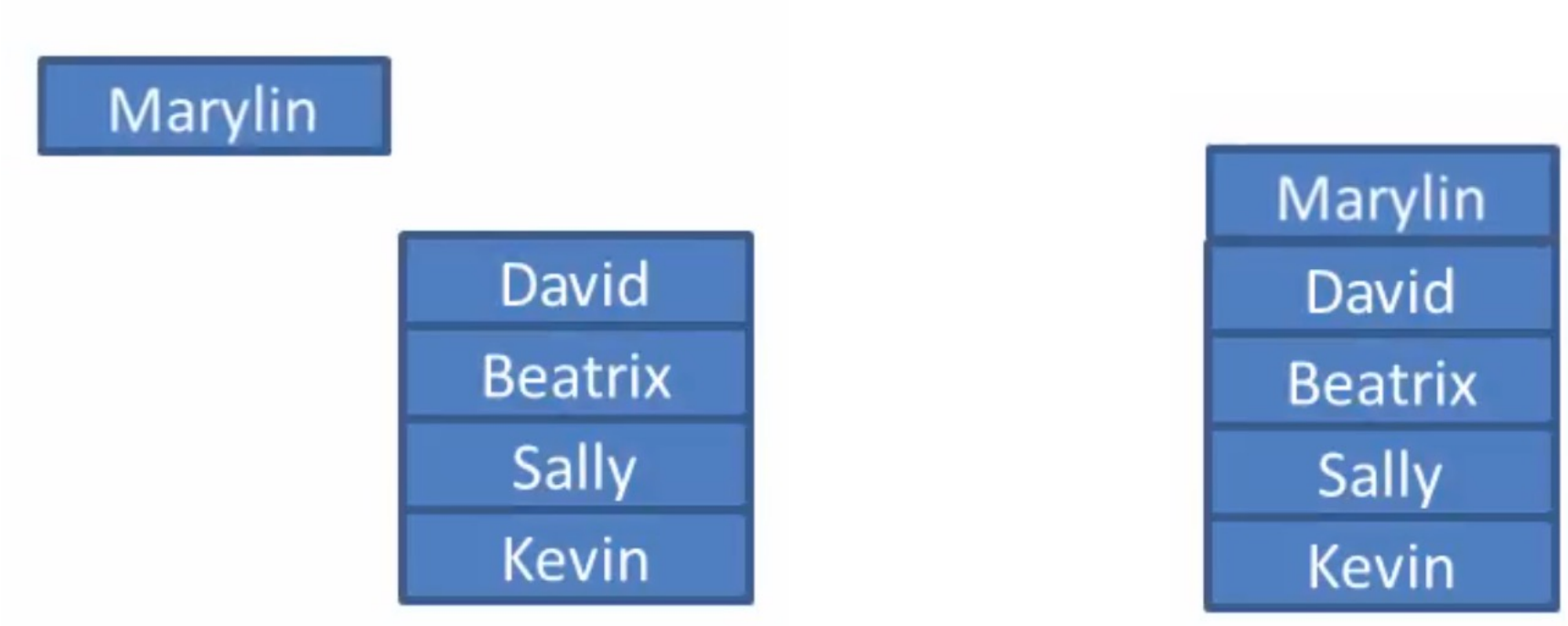
Top = 3

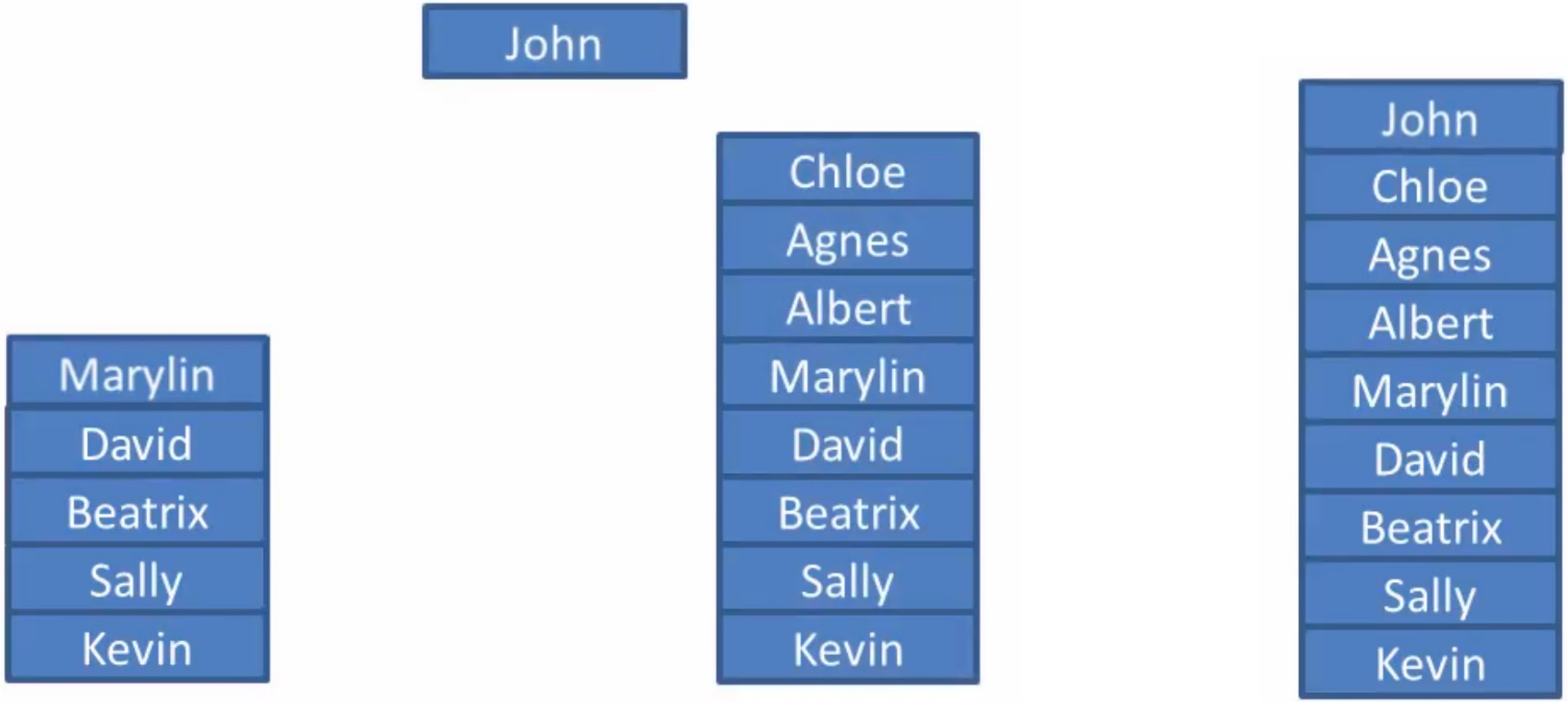
Pop

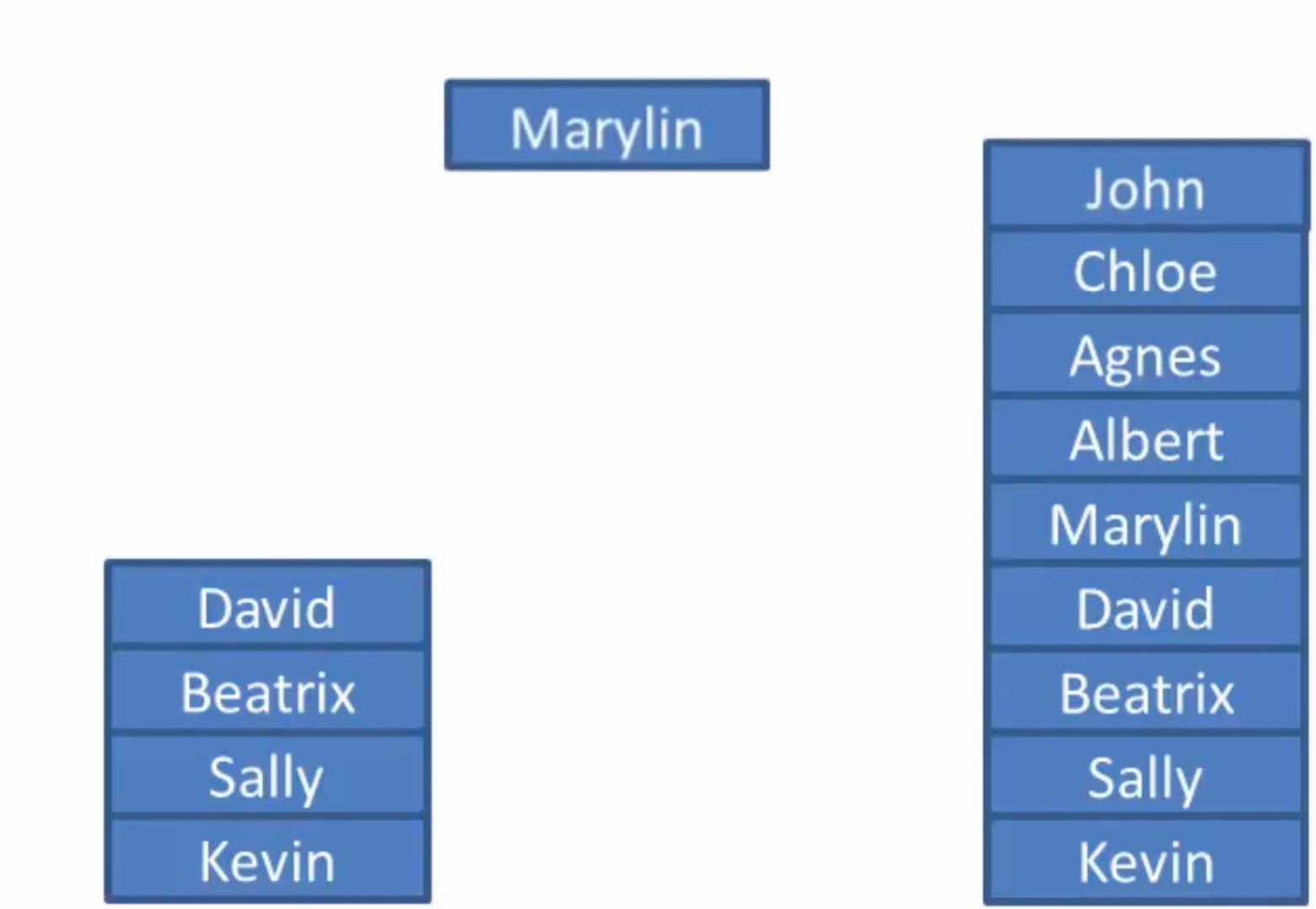
MaximumSize = 5

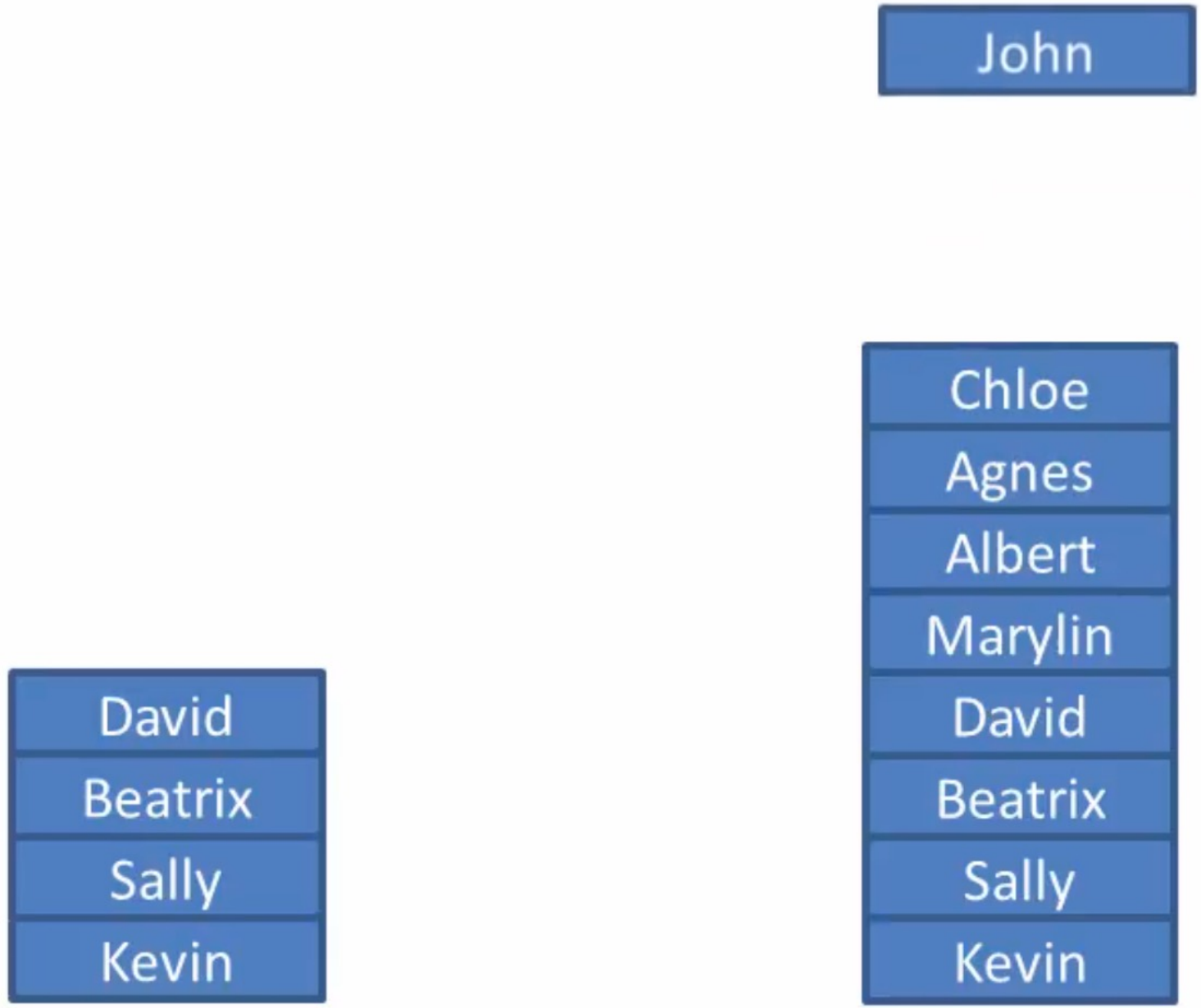
Top = 3


Pop
Beatrix
MaximumSize = 5
Beatrix
Sally
Kevin
Top = 2

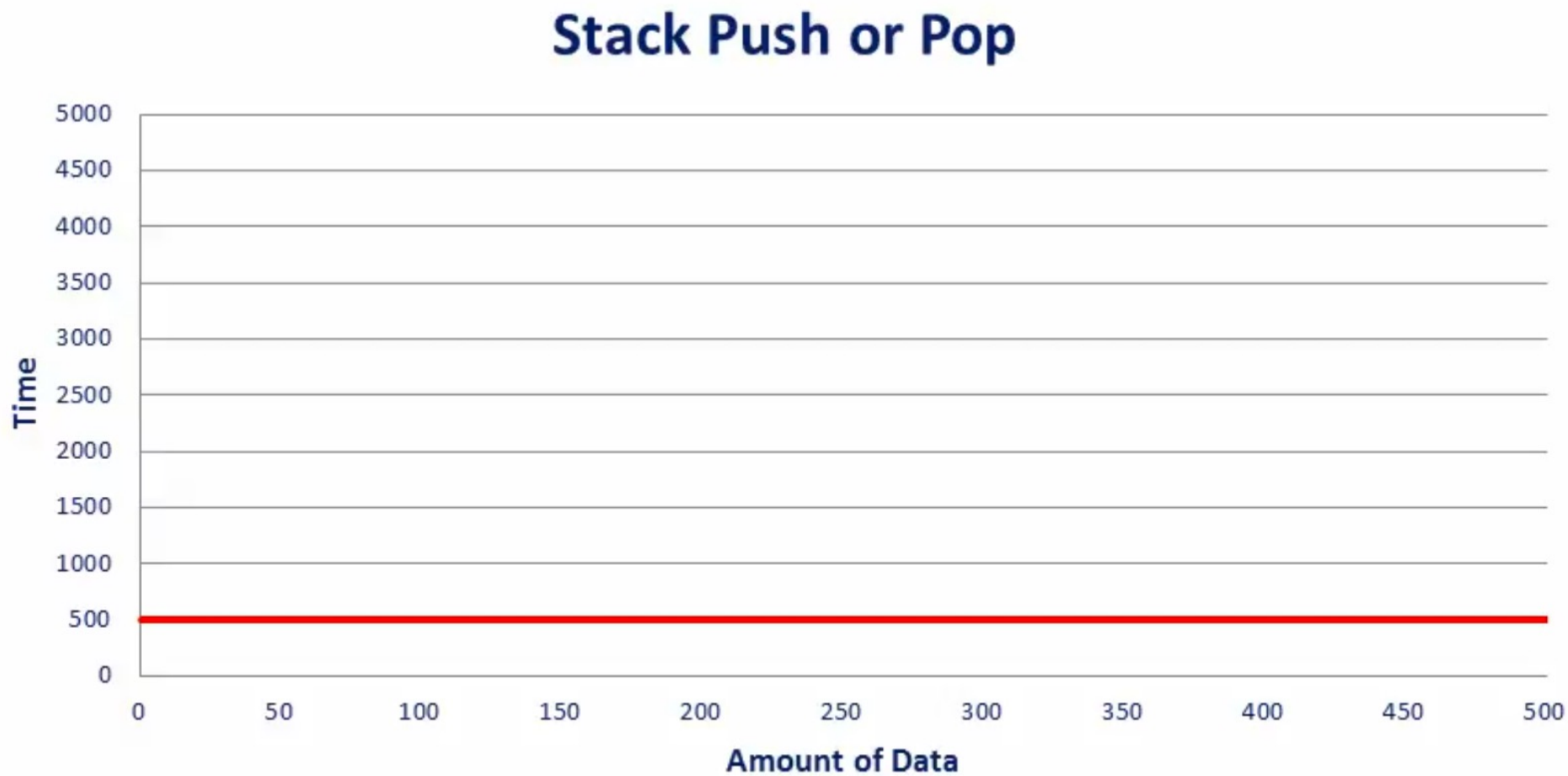

Marylin
David
Beatrix
Sally
Kevin
Marylin
David
Beatrix
Sally
Kevin


John
Marylin
David
Beatrix
Sally
Kevin
Chloe
Agnes
Albert
Marylin
David
Beatrix
Sally
Kevin
John
Chloe
Agnes
Albert
Marylin
David
Beatrix
Sally
Kevin


Marylin
David
Beatrix
Sally
Kevin
John
Chloe
Agnes
Albert
Marylin
David
Beatrix
Sally
Kevin


John
Chloe
Agnes
Albert
Marylin
David
Beatrix
Sally
Kevin
David
Beatrix
Sally
Kevin

Data	Time
1	500
2	500
4	500
8	500
16	500
32	500
64	500
128	500
256	500
512	500

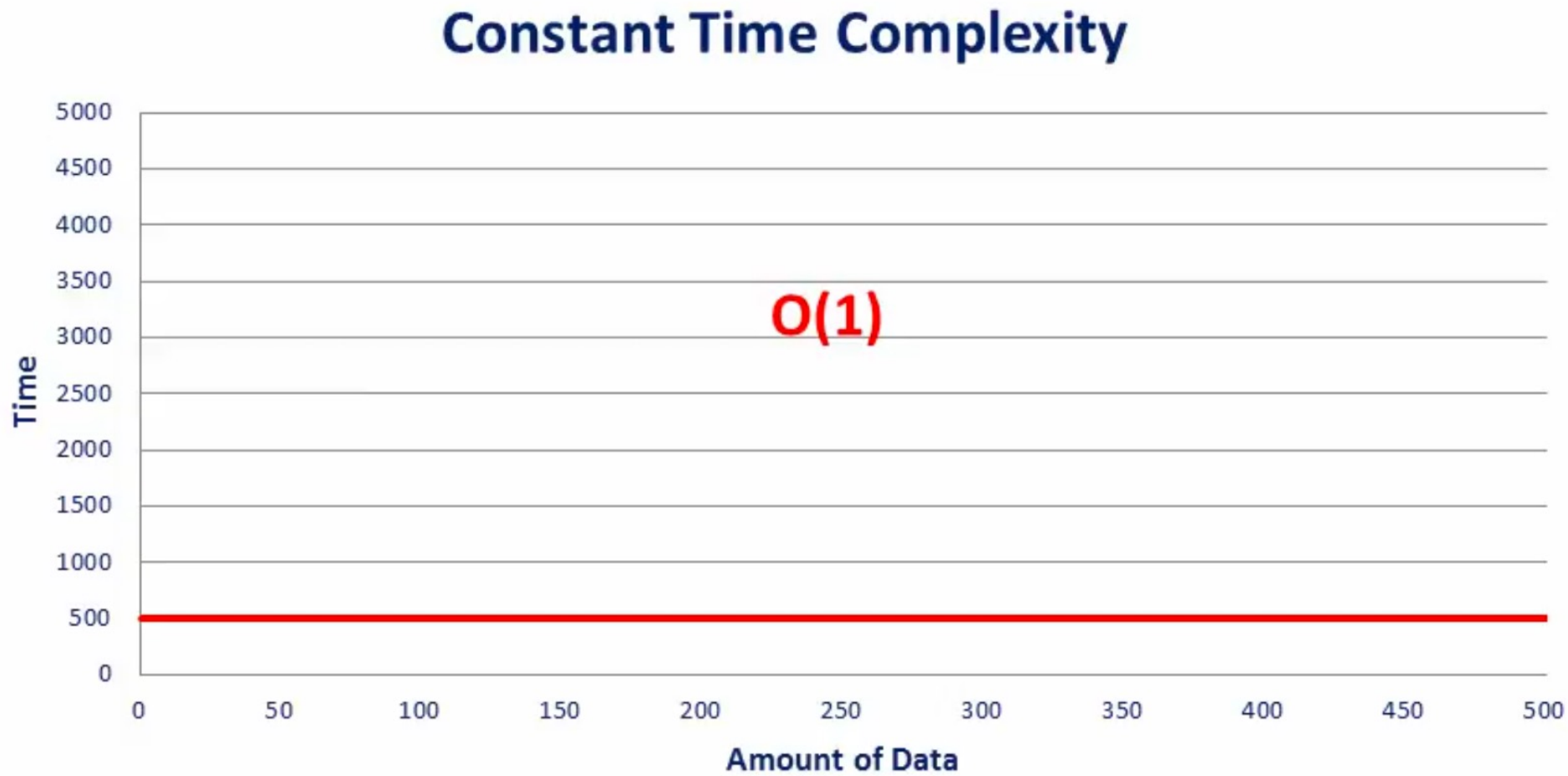


Stack Operations Complexity

- Increasing the amount of data makes no difference to the time taken by push or pop
- The Big O time complexity is **Constant**

O(1)

Data	Time
1	500
2	500
4	500
8	500
16	500
32	500
64	500
128	500
256	500
512	500



The Dominant Term

- An algorithm working on a data structure of size n might take $\mathbf{5n^3 + n^2 + 4n + 3}$ steps

The Dominant Term

- An algorithm working on a data structure of size n might take $\mathbf{5n^3} + n^2 + 4n + 3$ steps
- The larger n becomes, the less significant the smaller terms become, so we ignore everything except $5n^3$

The Dominant Term

- An algorithm working on a data structure of size n might take $5n^3 + n^2 + 4n + 3$ steps
- The larger n becomes, the less significant the smaller terms become, so we ignore everything except $5n^3$
- We can also ignore any constants, so the Big O time complexity of this algorithm is **$O(n^3)$**

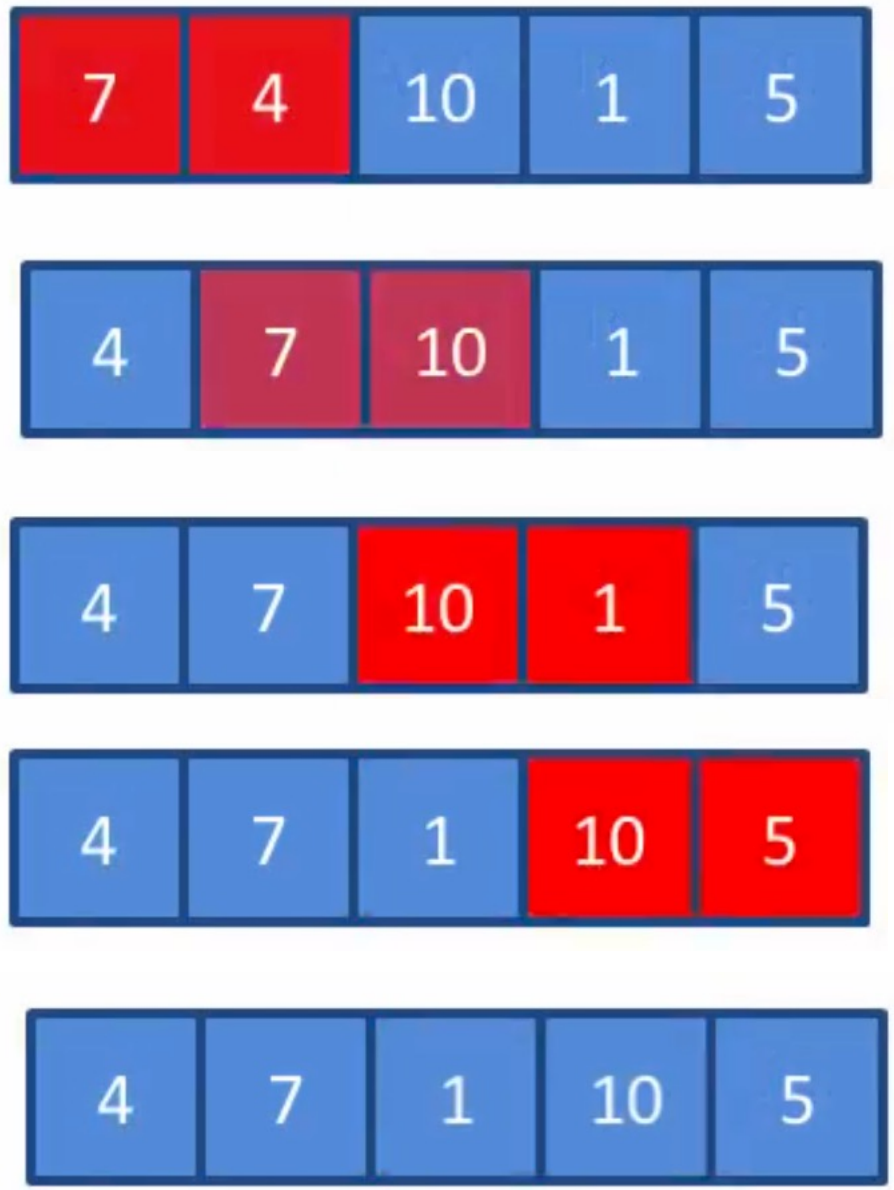
Bubble Sort

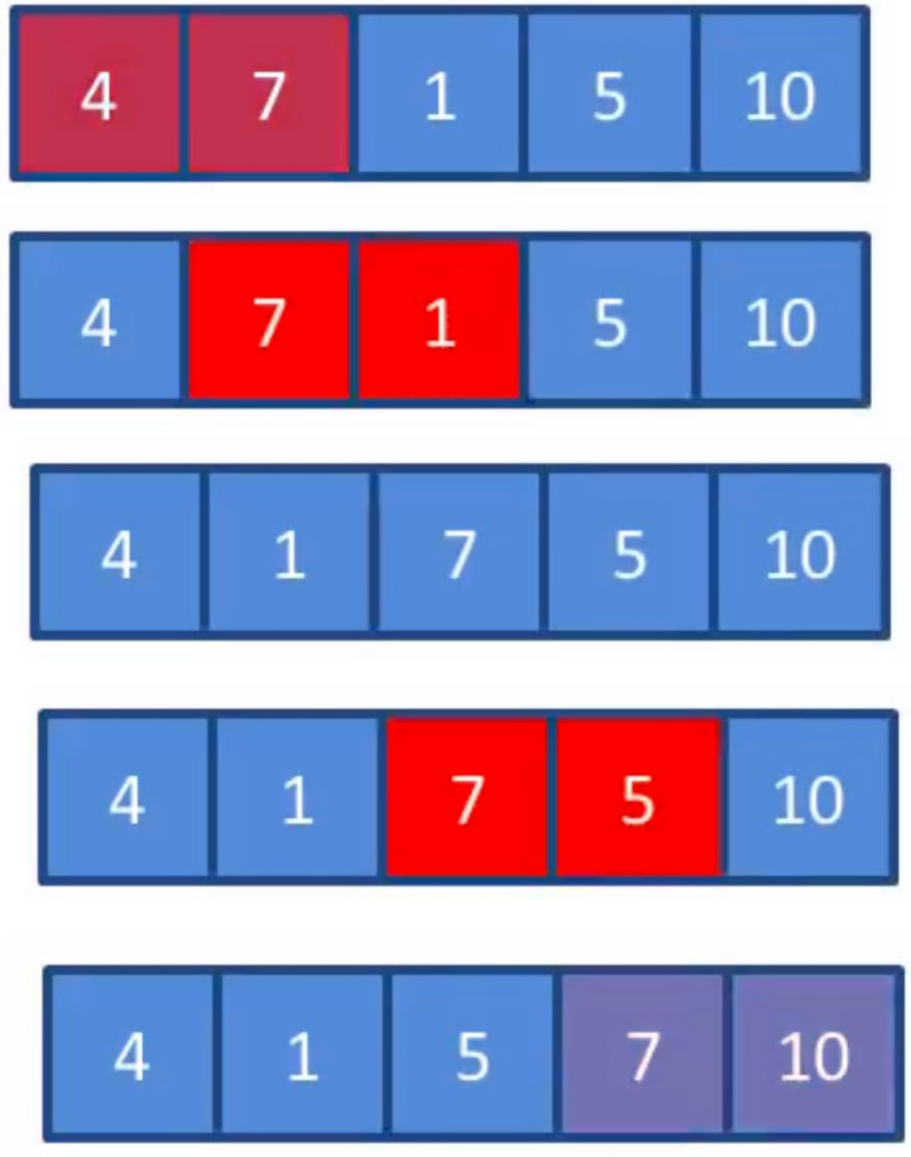
Bubble Sort

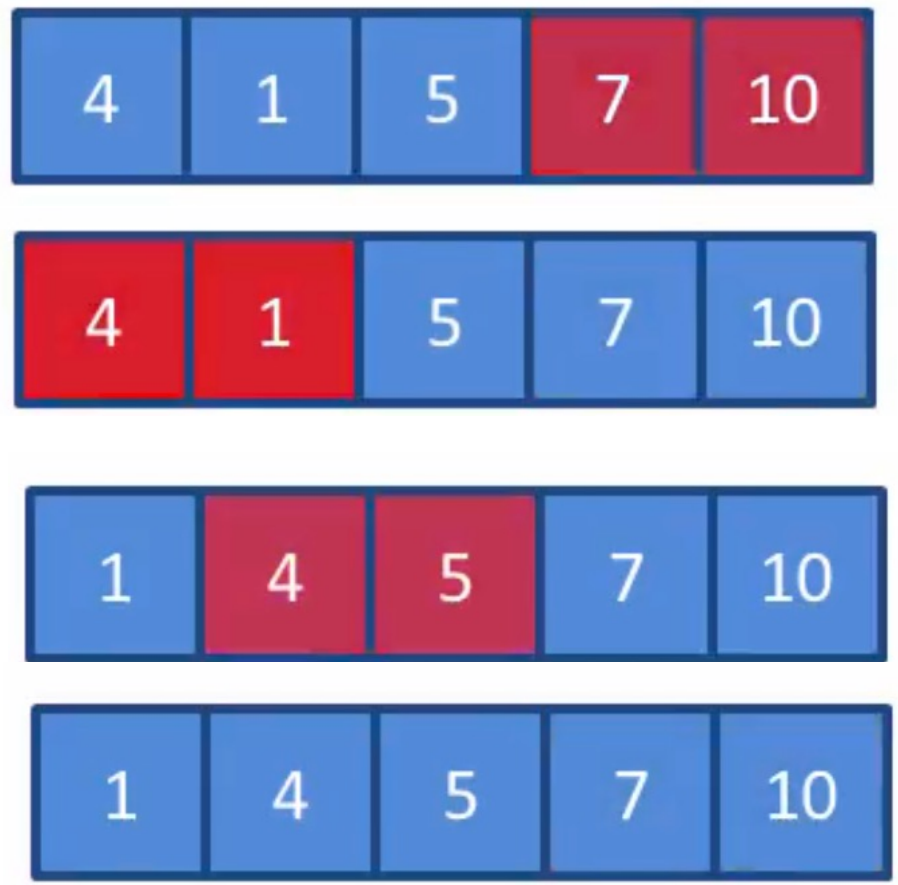
- Sorts a list of items into numeric or alphabetical order
- Scans a list comparing pairs of values and swapping their positions if necessary
- For n data items, the list is scanned like this n-1 times
- Various enhancements possible

Pseudocode

```
FOR iPass = 1 to n - 1
    FOR i = 0 TO n - 2
        IF ArrayToSort(i) > ArrayToSort(i + 1) THEN
            Swap ArrayToSort(i) with ArrayToSort(i + 1)
        END IF
    NEXT i
NEXT iPass
```

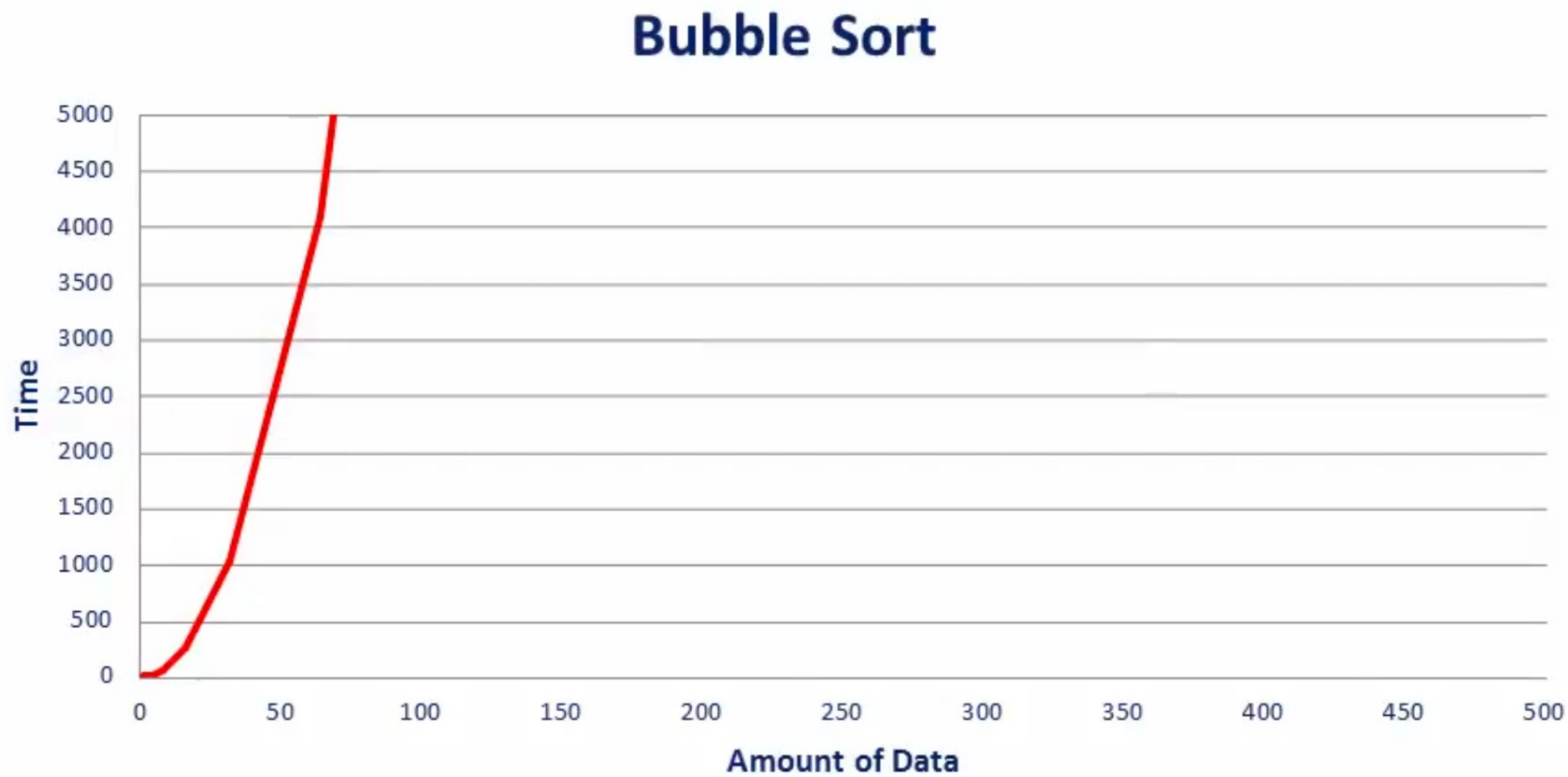

7 4 10 1 5
4 7 10 1 5
4 7 10 1 5
4 7 1 10 5
4 7 1 10 5


4 7 1 5 10
4 7 1 5 10
4 1 7 5 10
4 1 7 5 10
4 1 5 7 10


4 1 5 7 10
4 1 5 7 10
1 4 5 7 10
1 4 5 7 10


10 9 8 7 6 5 4 3 2 1
9 8 7 6 5 4 3 2 1 10
8 7 6 5 4 3 2 1 9 10
1 2 3 4 5 6 7 8 9 10

Data	Time
1	1
2	4
4	16
8	64
16	256
32	1024
64	4096
128	16384
256	65536
512	262144



Bubble Sort Complexity

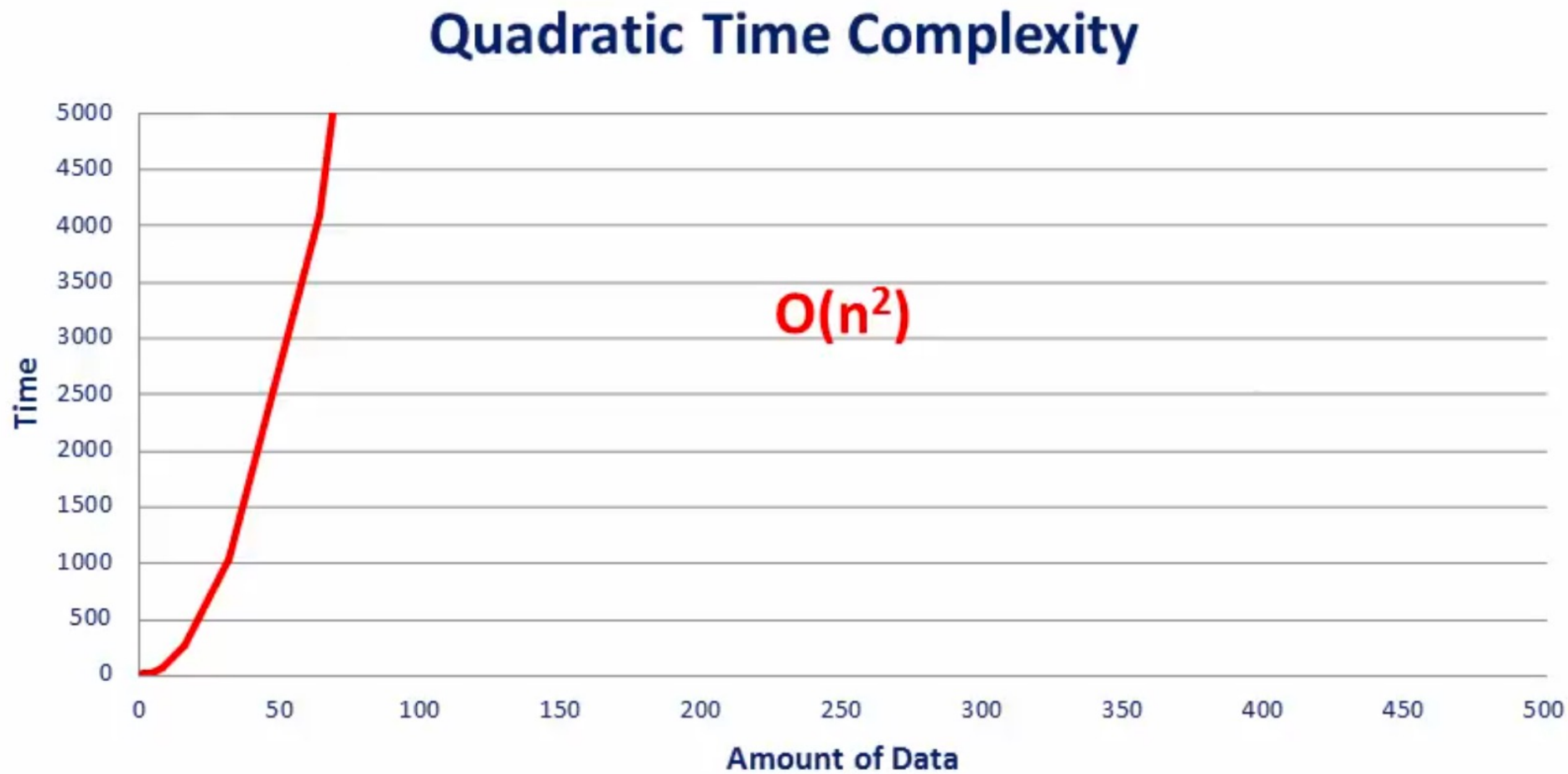
- For n data items, a simple implementation performs (n - 1) * (n - 1) operations
- This can be written n^2 - 2n + 1, and the dominant term is n^2
- The Big O time complexity is **Quadratic**

Bubble Sort Complexity

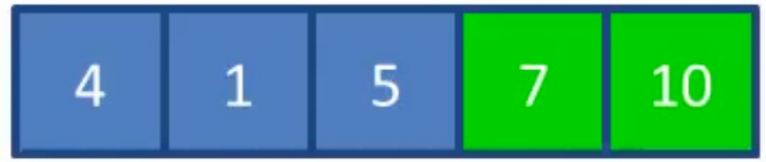
- For n data items, a simple implementation performs (n - 1) * (n - 1) operations
- This can be written n^2 - 2n + 1, and the dominant term is n^2
- The Big O time complexity is **Quadratic**

$O(n^2)$

Data	Time
1	1
2	4
4	16
8	64
16	256
32	1024
64	4096
128	16384
256	65536
512	262144



Enhanced Bubble Sort



- Largest item is in the correct position after the first pass
- Second largest item is in the correct place after the next pass
- and so on…
- The inner loop can run one less time with each pass of the outer loop

```
FOR iPass = 1 to n - 1
    FOR i = 0 to n - 1 - iPass
        IF ArrayToSort(i) > ArrayToSort(i + 1) THEN
            Temp = ArrayToSort(i)
            ArrayToSort(i) = ArrayToSort(i + 1)
            ArrayToSort(i + 1) = Temp
        END IF
    NEXT i
NEXT iPass
```

Enhanced Algorithm Complexity

- For n items of data, the enhanced bubble sort algorithm performs $(n - 1) + (n - 2) + (n - 3) + \ldots + 3 + 2 + 1$ operations
- This can be shown to be $(n^2 – n)/2$
- This is a 50% reduction in the time taken, but…
- The dominant term is still n^2
- The complexity is still **Quadratic** **$O(n^2)$**

Alternative Enhanced Bubble Sort

- If the inner loop performs no swaps, the data must now be in the correct order
- Check for swaps with a Boolean variable
- Force an early exit when there's no more work to do

Pseudocode

```
REPEAT
    Swapped = False
    FOR i = 0 TO Length(ArrayToSort) – 2
        IF ArrayToSort(i) > ArrayToSort(i + 1) THEN
            Swap ArrayToSort(i) with  ArrayToSort(i + 1)
            Swapped = True
        END IF
    NEXT i
UNTIL Swapped = False
```

Best versus Worst Case Scenario

- Best case scenario
  - Data is already sorted, the inner loop will run only once

    Linear O(n)

- Worst case scenario
  - Data is in reverse order, every item has to be moved

    Quadratic $O(n^2)$

Logarithms

- The inverse of exponentiation

$2^3 = 8$ $\log_2 8 = 3$

$10^4 = 10000$ $\log_{10} 10000 = 4$

- Generally…

$x^z = y$ $\log_x y = z$

Logarithms

$2^0 = 1$	$\log_2 1 = 0$
$2^1 = 2$	$\log_2 2 = 1$
$2^2 = 4$	$\log_2 4 = 2$
$2^3 = 8$	$\log_2 8 = 3$
$2^4 = 16$	$\log_2 16 = 4$
$2^5 = 32$	$\log_2 32 = 5$

Logarithms

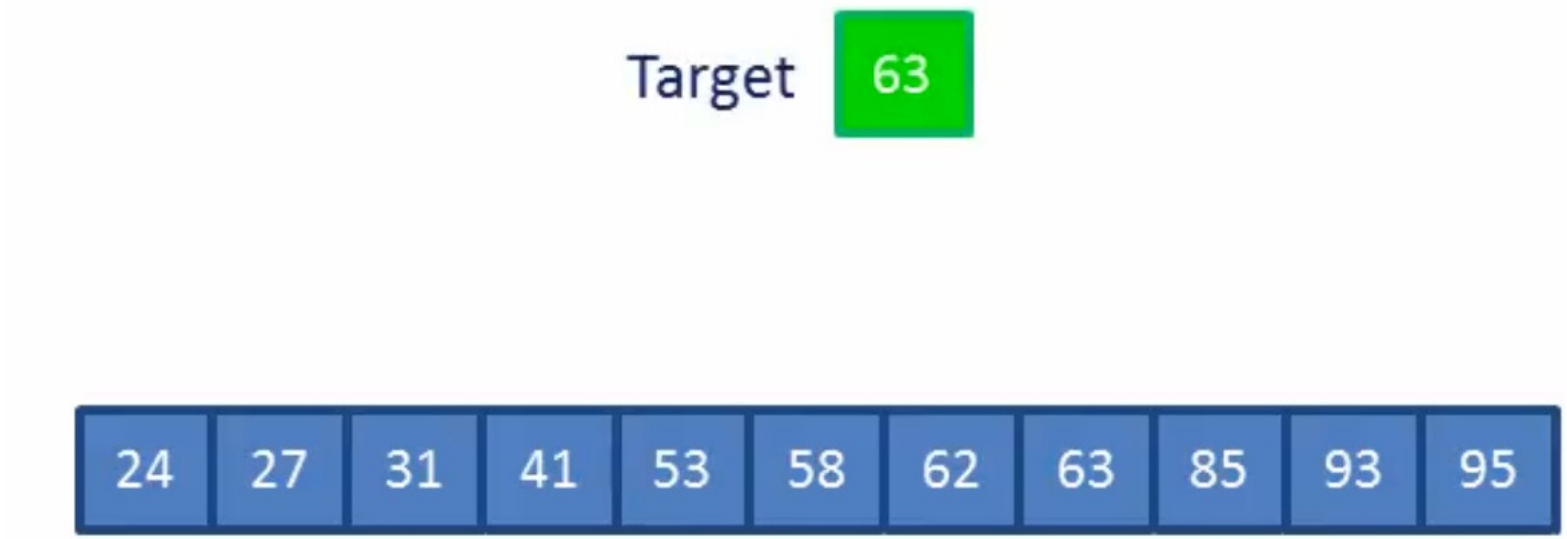
$2^0 = 1$	$\log_2 1 = 0$
$2^1 = 2$	$\log_2 2 = 1$
$2^2 = 4$	$\log_2 4 = 2$
$2^3 = 8$	$\log_2 8 = 3$
$2^4 = 16$	$\log_2 16 = 4$
$2^5 = 32$	$\log_2 32 = 5$

Binary Search

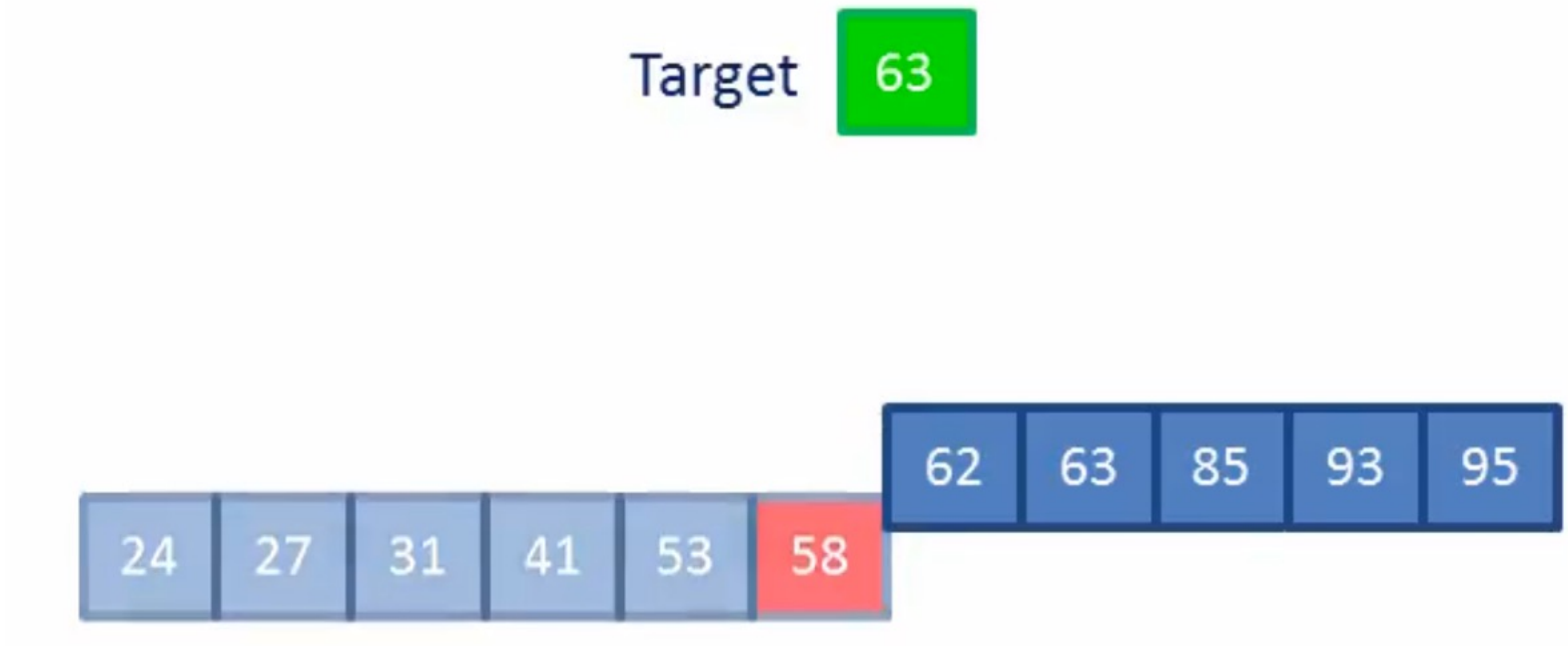
Binary Search

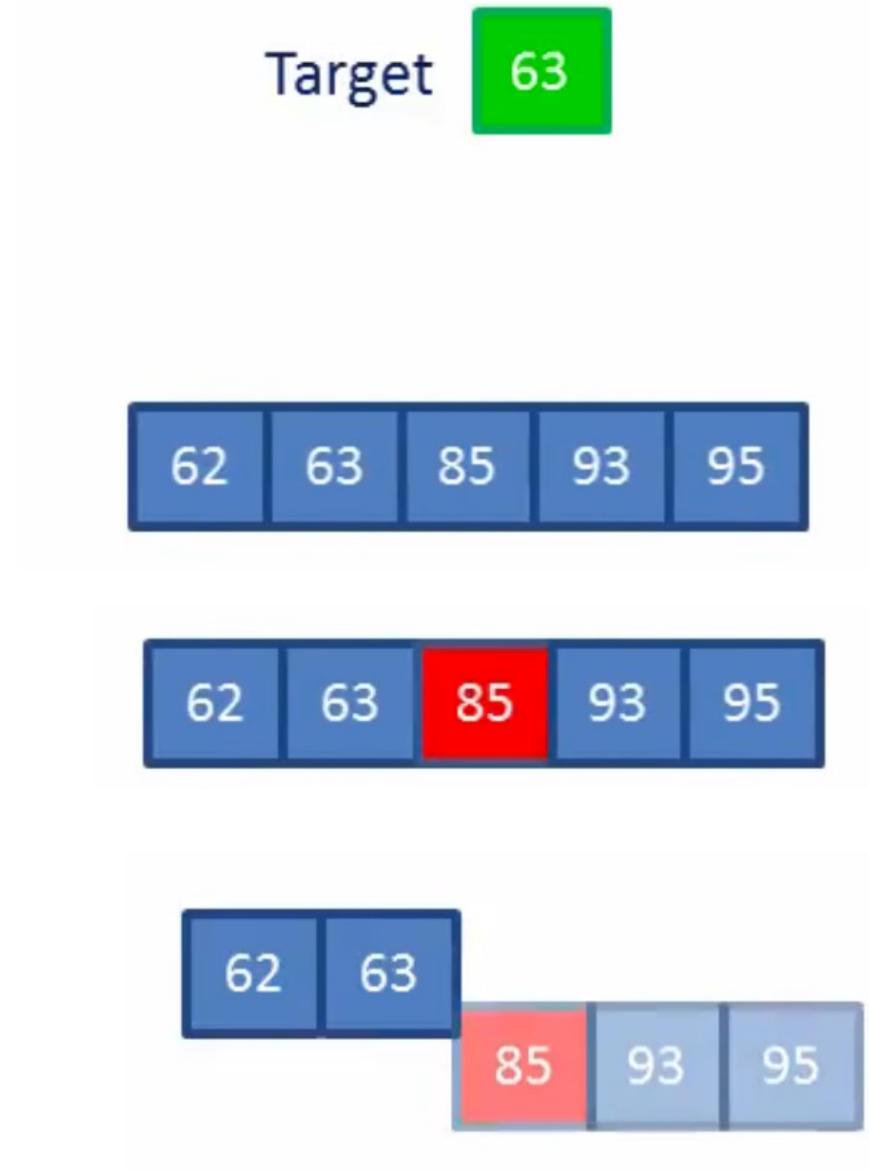
- Used to search an ordered list for a particular value
- Divide and conquer approach
- Target compared with middle value, then half of the list is discarded, repeatedly, until the target is found
- Very efficient for large sorted lists

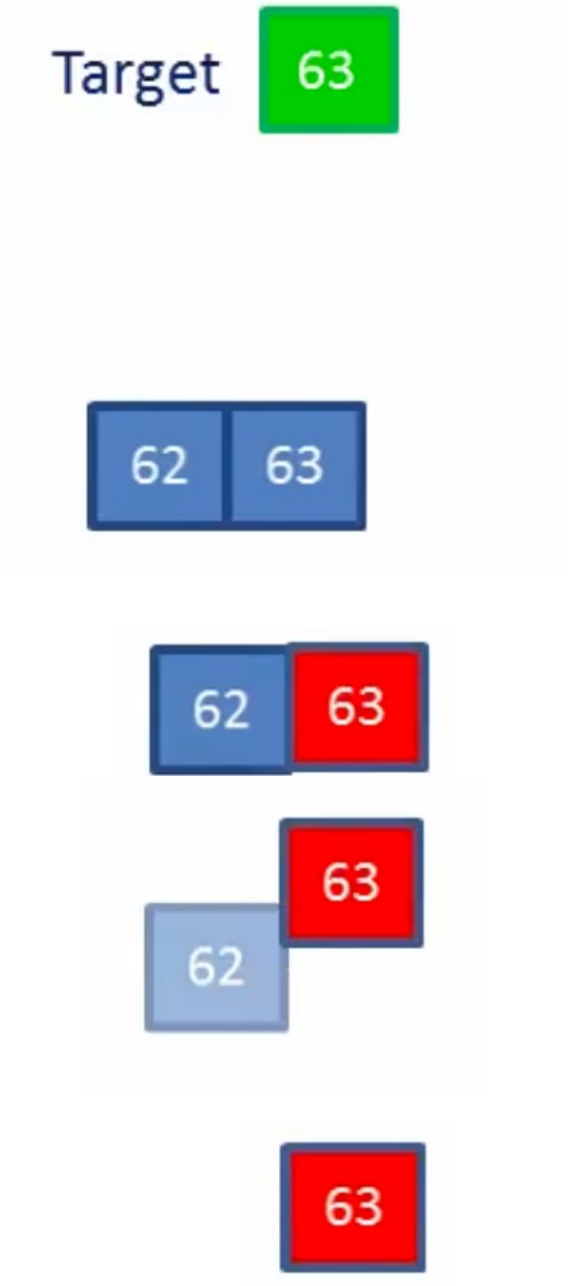
```
iLow = LBound(DataArray)
iHigh = UBound(DataArray)

Do While iLow <= iHigh
        iMiddle = (iLow + iHigh) / 2
        If Target = DataArray(iMiddle) Then
                bFound = True
                Exit Do
        ElseIf Target < DataArray(iMiddle) Then
                iHigh = (iMiddle - 1)
        Else
                iLow = (iMiddle + 1)
        End If
Loop
```


Target
63
24
27
31
41
53
58
62
63
85
93
95

Target 63

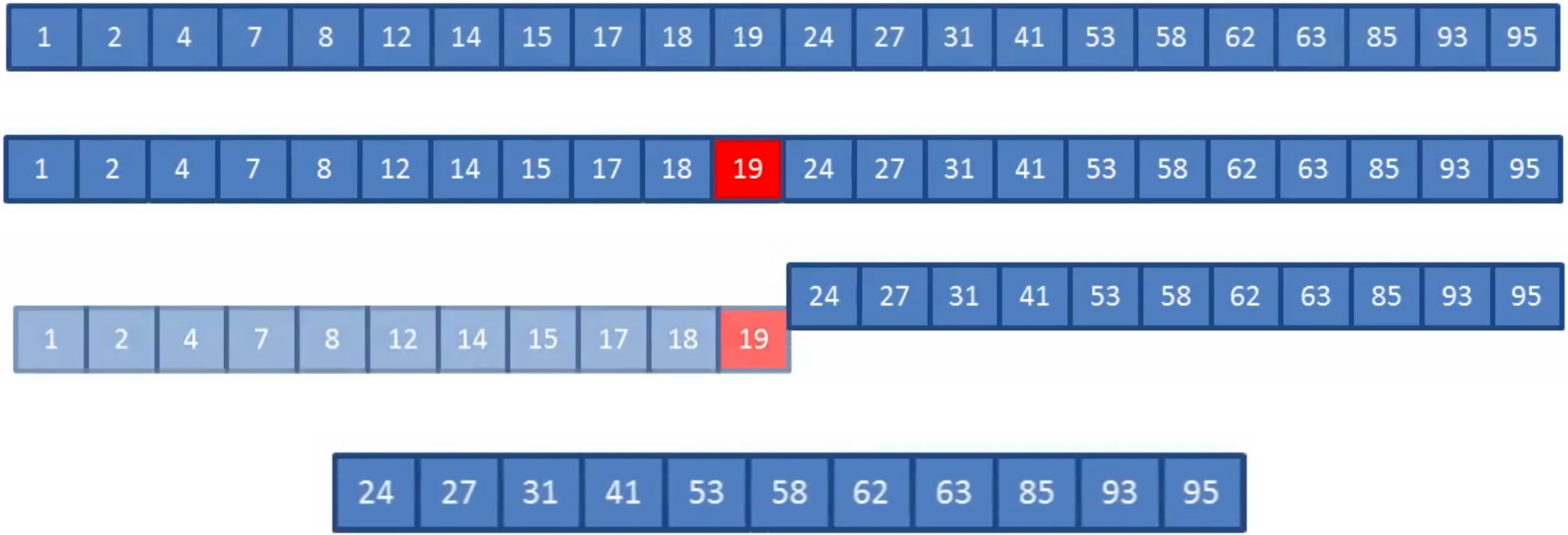

Target
63
62
63
85
93
95
24
27
31
41
53
58


Target
63
62
63
85
93
95
62
63
85
93
95
62
63
85
93
95



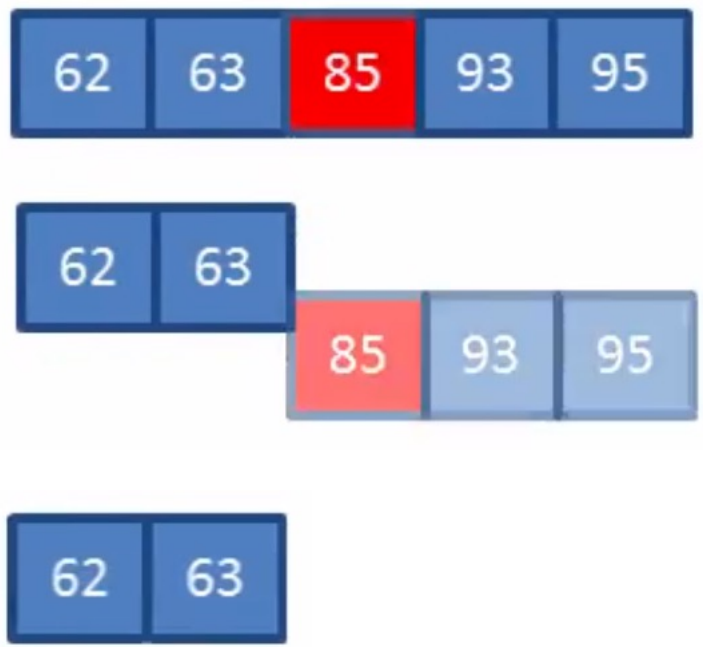
We choped the list 3 times. Sometimes also known as Binary Cop

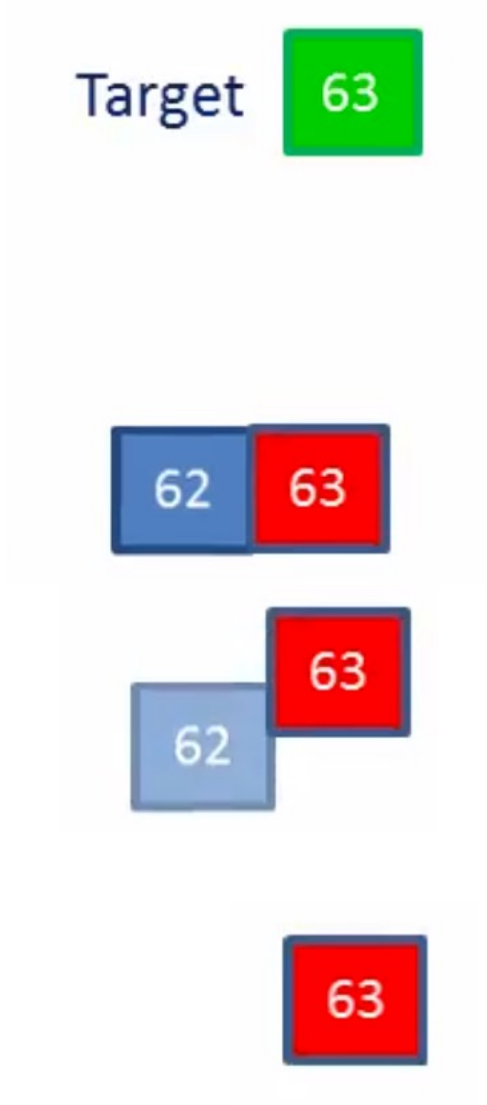
Target 63


1 2 4 7 8 12 14 15 17 18 19 24 27 31 41 53 58 62 63 85 93 95
1 2 4 7 8 12 14 15 17 18 19 24 27 31 41 53 58 62 63 85 93 95
1 2 4 7 8 12 14 15 17 18 19
24 27 31 41 53 58 62 63 85 93 95
24 27 31 41 53 58 62 63 85 93 95


Target
63
24
27
31
41
53
58
62
63
85
93
95
62
63
85
93
95
24
27
31
41
53
58
62
63
85
93
95

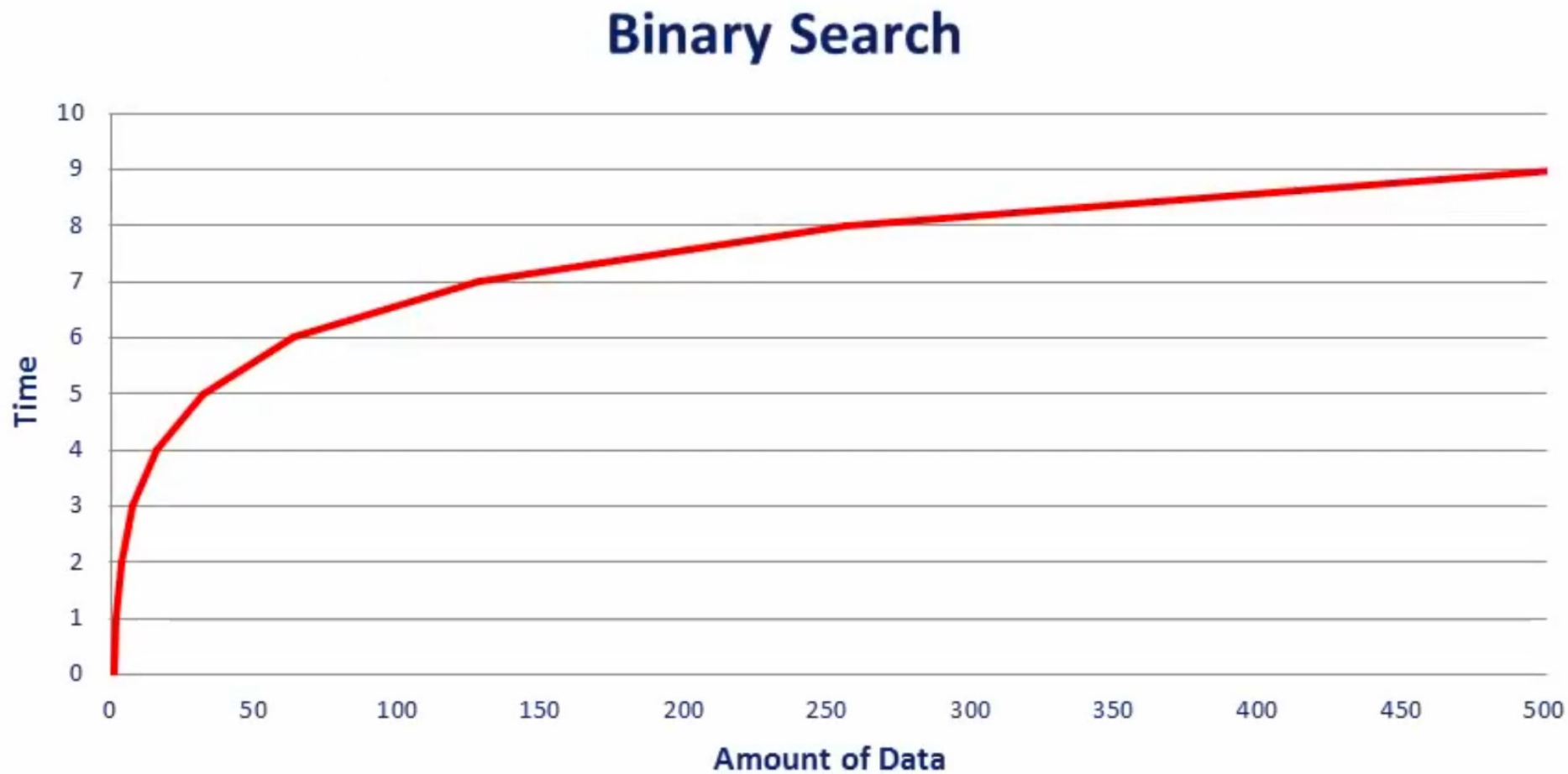
Target 63





In 4 chops – we reach at the target.

Data	Time
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9

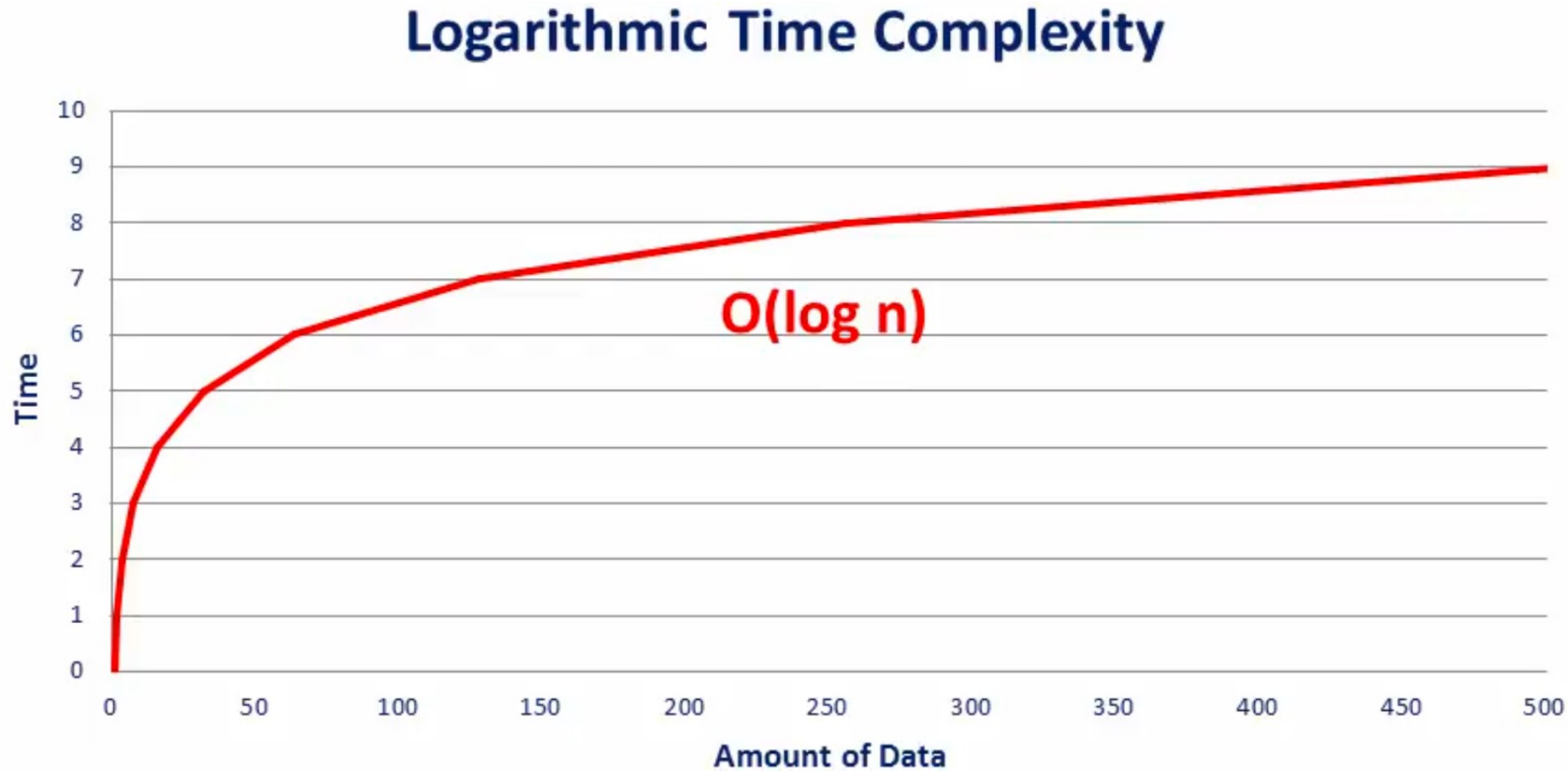


Binary Search Complexity

- Double the data requires only one extra chop
- This makes the binary search very efficient for very large data sets, if the data is already sorted
- The Big O time complexity is **Logarithmic**

O(log n)

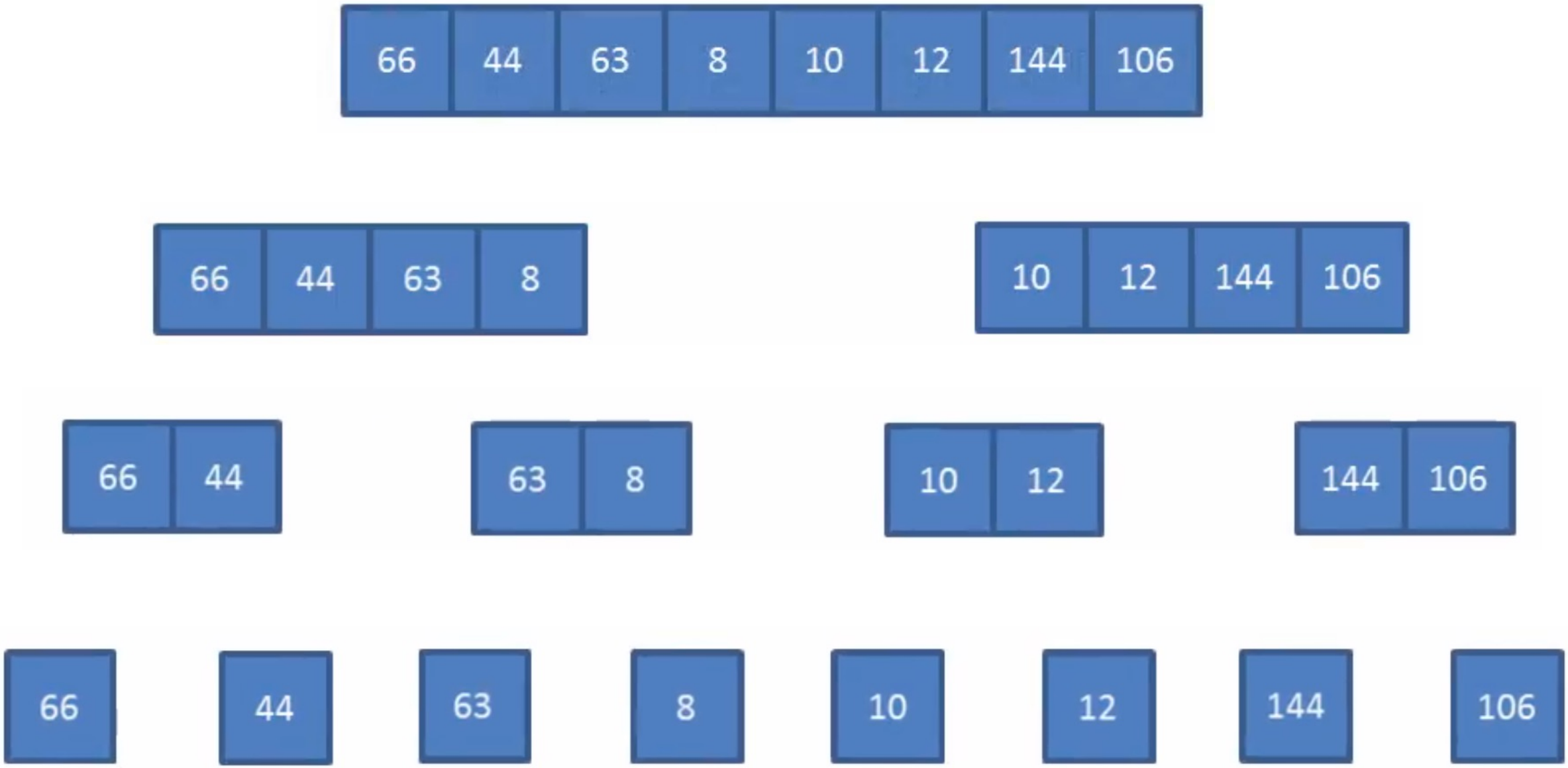
Data	Time
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9

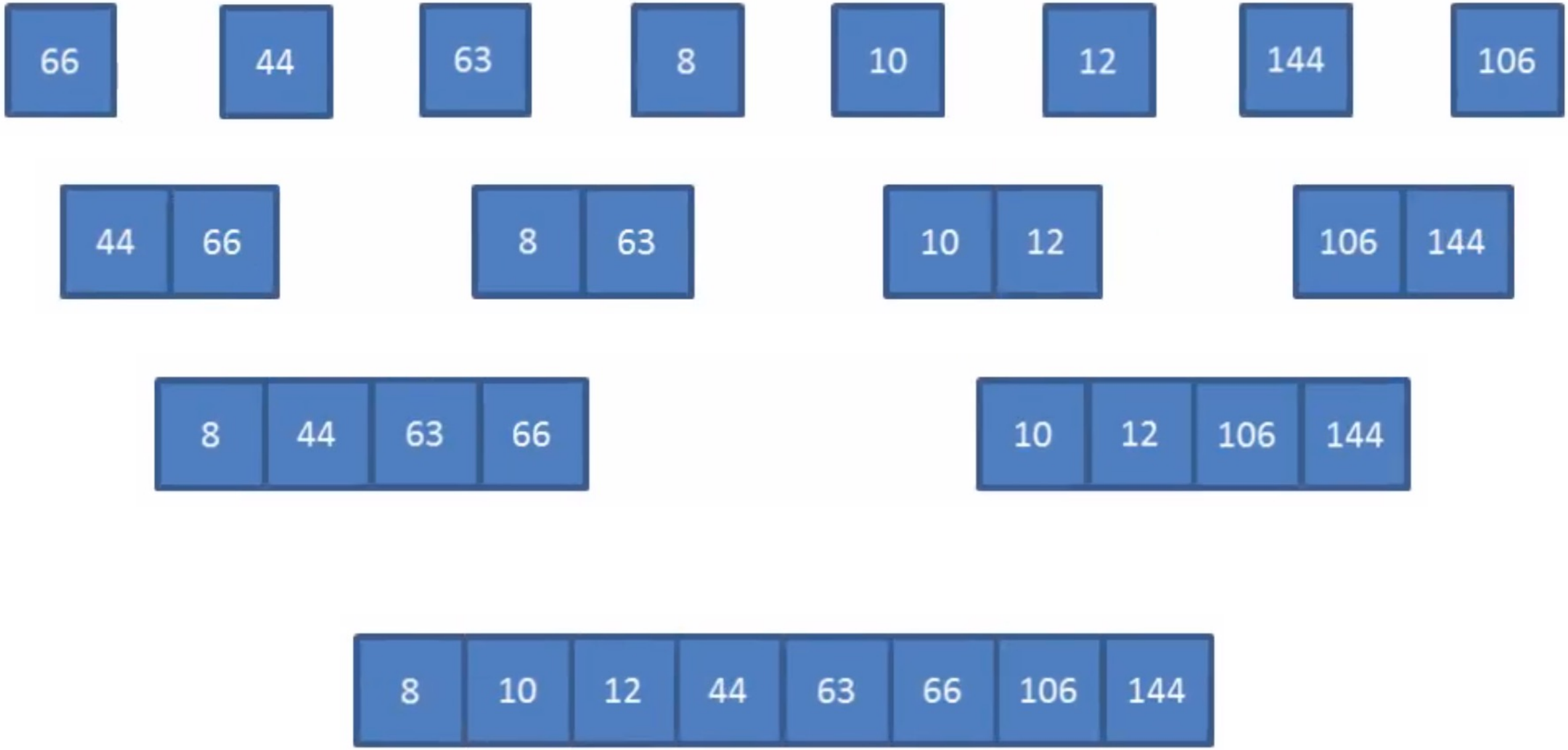


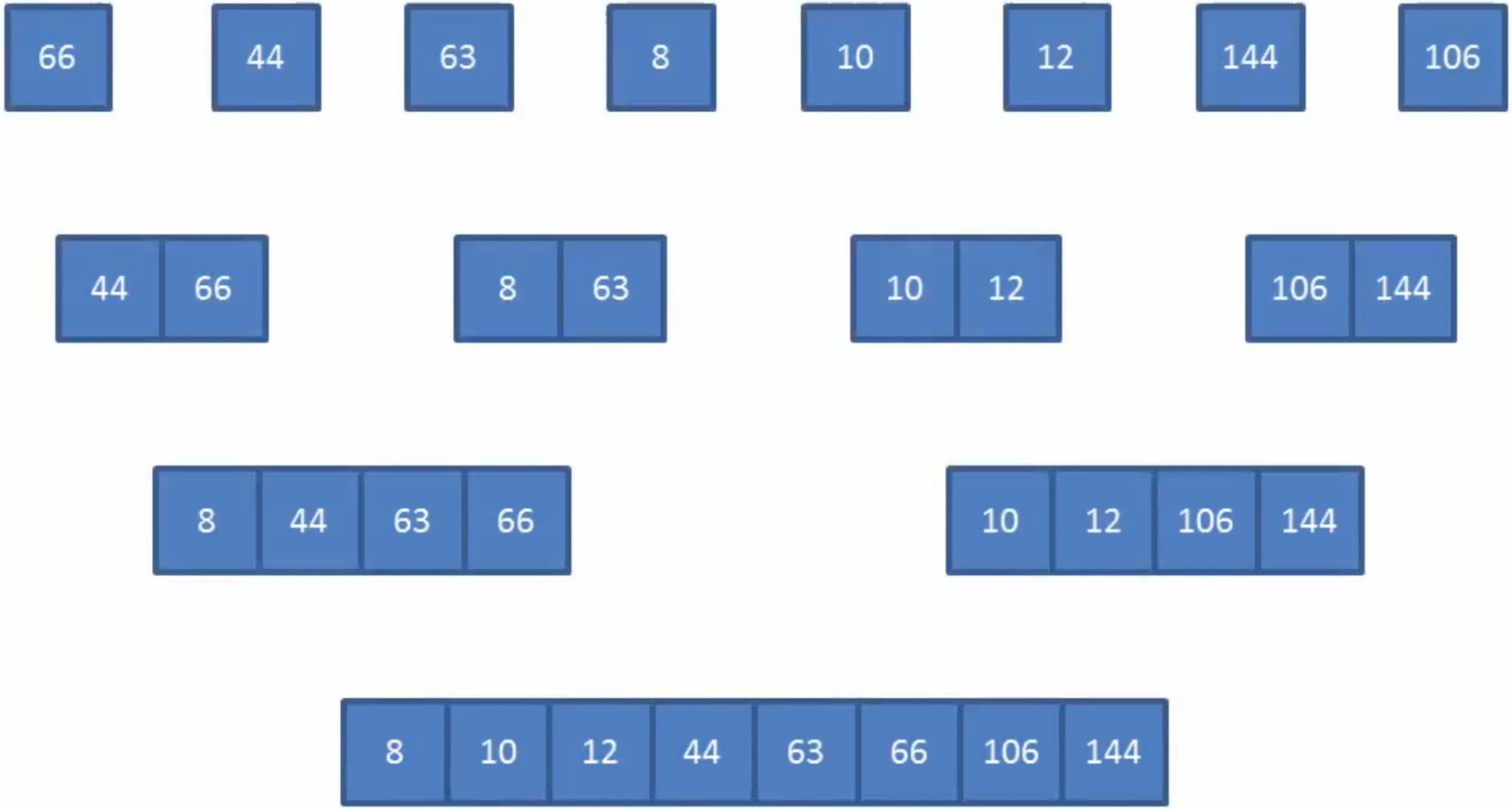
Merge Sort

Merge Sort

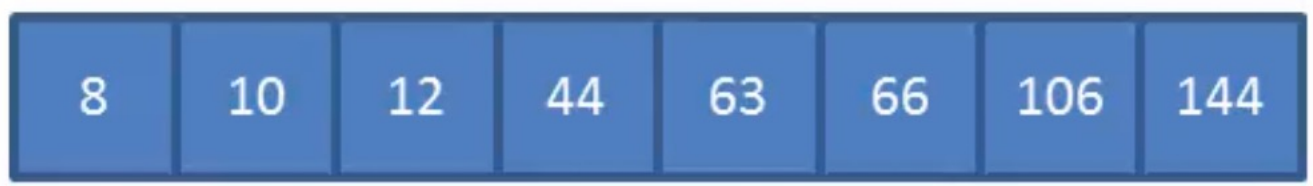
- Sorts the data in a list
- Divide and conquer approach
- Splits a list into several sub lists each of which contains only one item, and is therefore by definition sorted
- Pairs of sub lists merged together, sorting as it goes


66 44 63 8 10 12 144 106
66 44 63 8
10 12 144 106
66 44
63 8
10 12
144 106
66
44
63
8
10
12
144
106


66
44
63
8
10
12
144
106
44
66
8
63
10
12
106
144
8
44
63
66
10
12
106
144
8
10
12
44
63
66
106
144

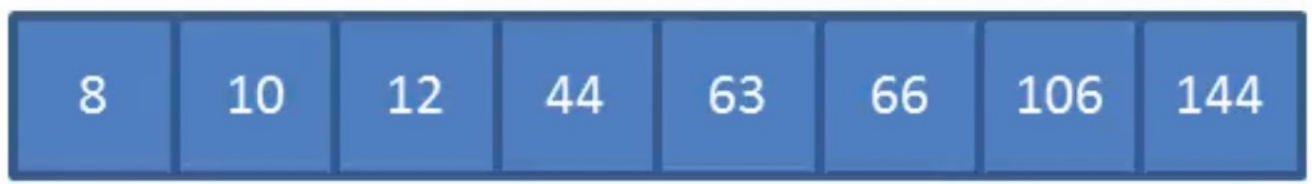

66
44
63
8
10
12
144
106
44
66
8
63
10
12
106
144
8
44
63
66
10
12
106
144
8
10
12
44
63
66
106
144

8 append operations



8 append operations

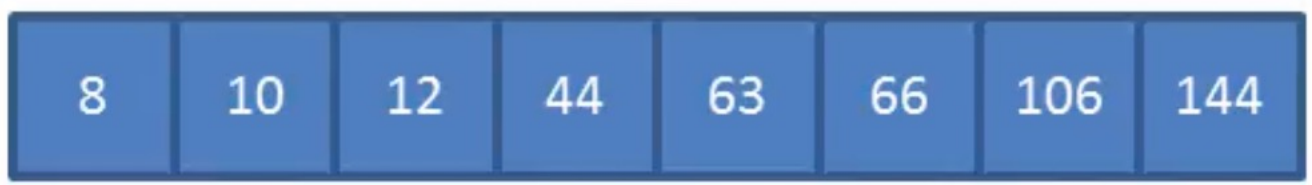
8 append operations



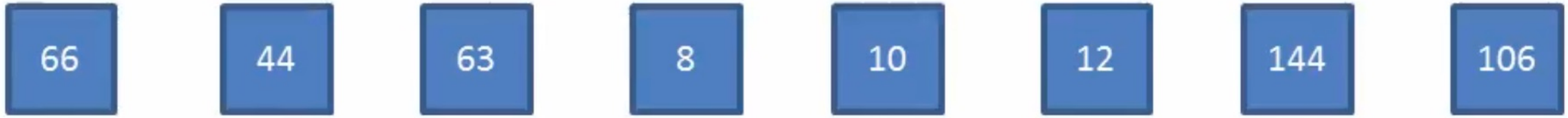
8 append operations

8 append operations

8 append operations



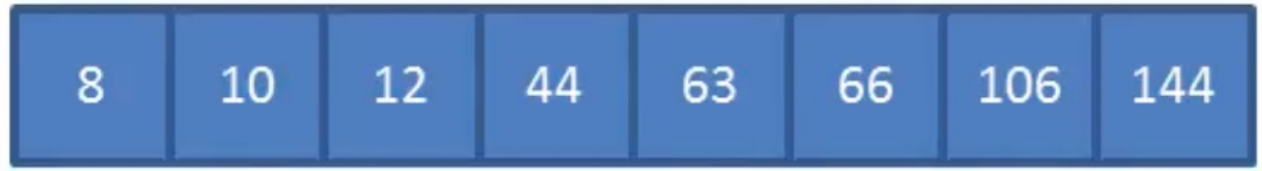
8 * 3 append operations in total



8 append operations

8 append operations

8 append operations



Merge Sort Complexity

- If n = 8, merge sort does n * 3 append operations

Merge Sort Complexity

- If n = 8, merge sort does n * 3 append operations
- $\log_2 n = 3$, so this is n * $\log_2 n$ append operations

Merge Sort Complexity

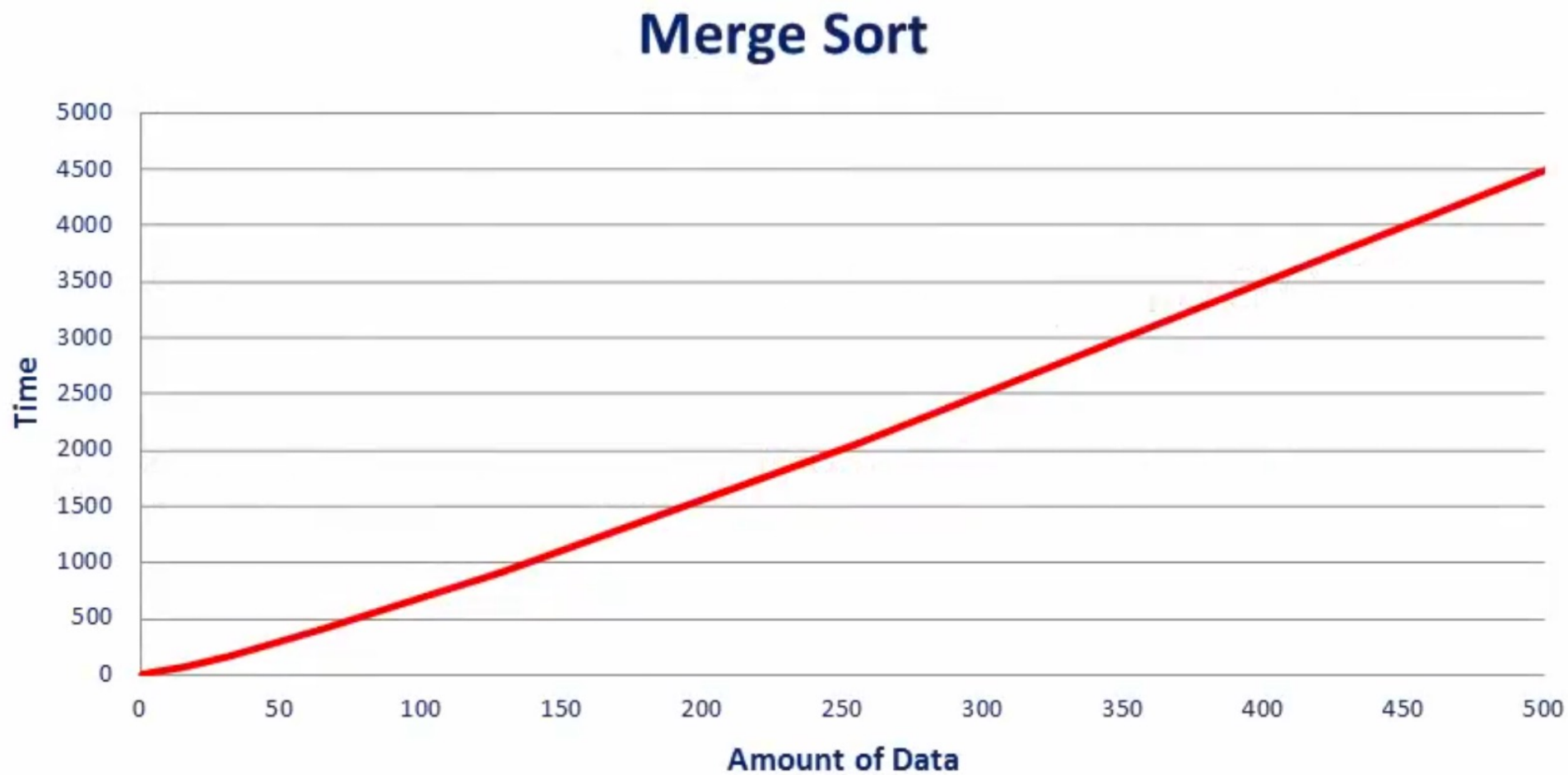
- If n = 8, merge sort does n * 3 append operations
- $\log_2 n = 3$, so this is n * $\log_2 n$ append operations
- The Big O time complexity is **'Linearithmic'**

Merge Sort Complexity

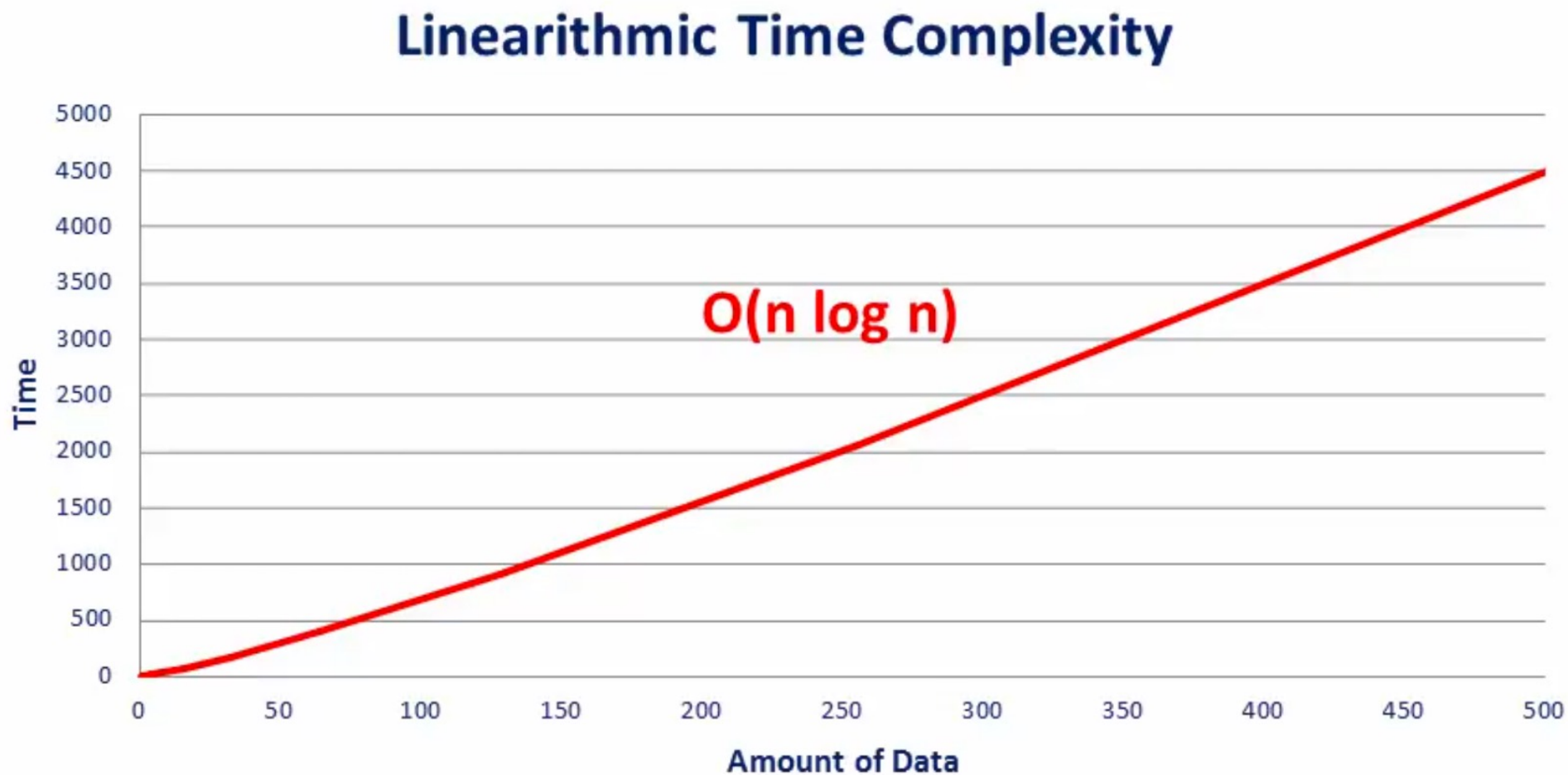
- If n = 8, merge sort does n * 3 append operations
- $\log_2 n = 3$, so this is n * $\log_2 n$ append operations
- The Big O time complexity is **'Linearithmic'**

O(n log n)

Data	Time
1	0
2	2
4	8
8	24
16	64
32	160
64	384
128	896
256	2048
512	4608



Data	Time
1	0
2	2
4	8
8	24
16	64
32	160
64	384
128	896
256	2048
512	4608



Big O

- Big O describes how the time taken, or memory used, by a program scales with the amount of data it has to work on

- Constant $O(1)$
  - Time taken is independent of the amount of data
  - Stack push, pop and peek; Queue enqueue and dequeue; Insert a node into a linked list
- Linear $O(n)$
  - Time taken is directly proportional to the amount of data
  - Linear search; Count items in a list; Compare a pair of strings
- Quadratic $O(n^2)$
  - Time taken is proportional to the amount of data squared
  - Bubble sort; Selection sort; Insertion sort, Traverse a 2D array
- Polynomial $O(n^k)$
  - Time taken is proportional to the amount of data raised to the power of a constant
- Logarithmic $O(\log n)$
  - Time taken is proportional to the logarithm of the amount of data
  - Binary search a sorted list; Search a binary tree
- Linearithmic $O(n \log n)$
  - Time taken is proportional to the logarithm of the amount of data , multiplied by the amount of data
  - Merge sort; Quicksort
- Exponential $O(k^n)$
  - Time taken is proportional to a constant raised to the power of the amount of data
  - n-Queens problem; Travelling salesman

Big O

Big O complexitiy is not about the real perforence of the program.

Its about how well a program scale

Or How well a program mainatian its perforemcne when given more data to work with.

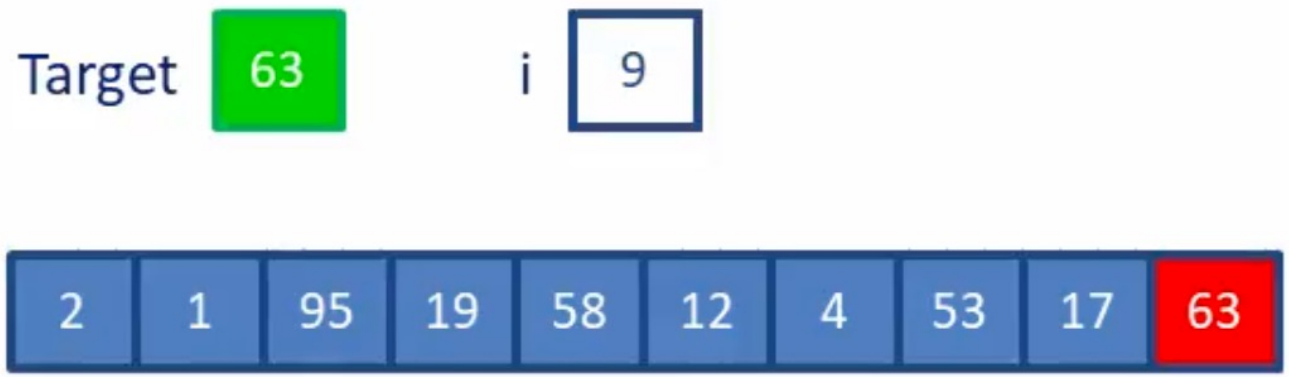
Big O

Space Complexity

Time complexity of a program i.e., impact of more input data on the time it take to complete.

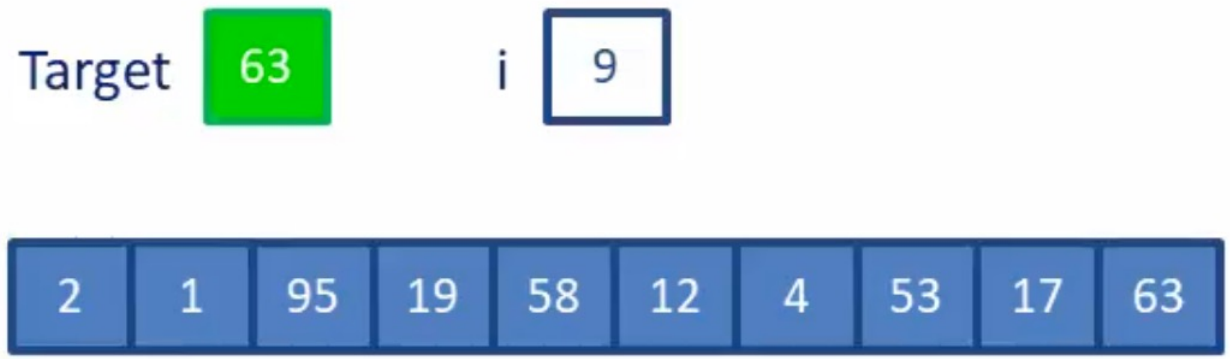
Space Complexity of a program i.e., How the memory requirments of program differ with amount of the data?

Linear Search




```
FOR i = 0 TO n - 1
    IF Array(i) = Target THEN
        bFound = True
        EXIT FOR
    END IF
NEXT i
```

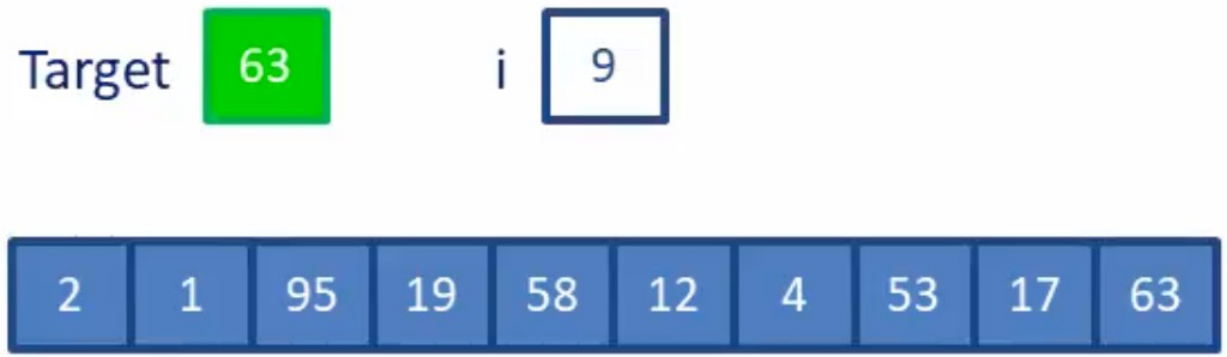
Linear Search




```
FOR i = 0 TO n - 1
    IF Array(i) = Target THEN
        bFound = True
        EXIT FOR
    END IF
NEXT i
```

Linear time complexity **O(n)**

Linear Search




```
FOR i = 0 TO n - 1
    IF Array(i) = Target THEN
        bFound = True
        EXIT FOR
    END IF
NEXT i
```

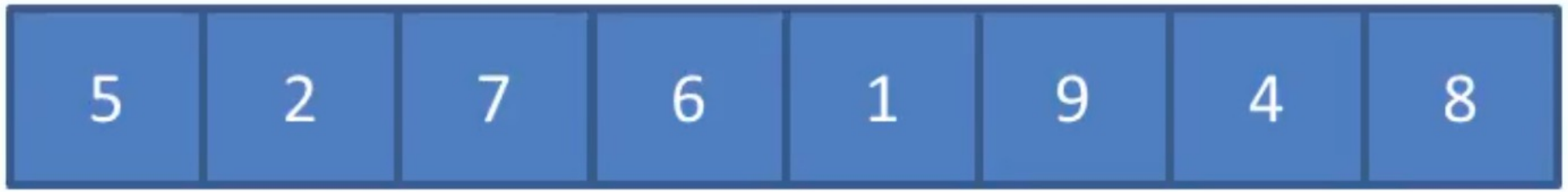
Other Examples: Inseration Sort, Bubble Sort, Section Sort

Linear time complexity **O(n)**

Constant space complexity **O(1)**

Quicksort

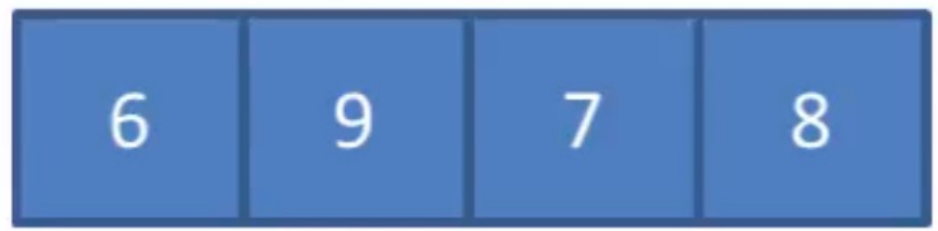
Imp: Partionoing Process




4
2
1
5
6
9
7
8


4
2
1


5


6
9
7
8

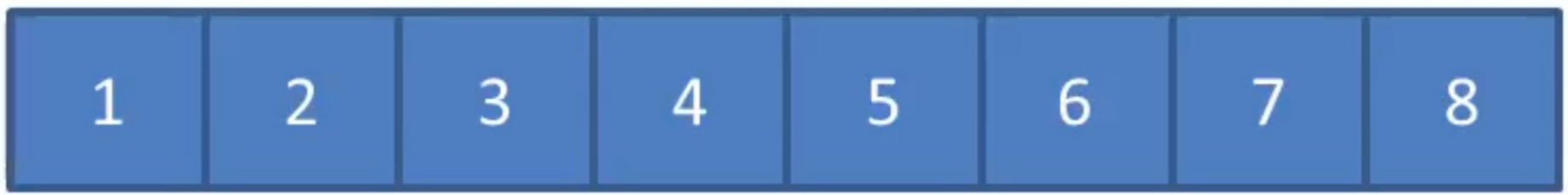
Quicksort

Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

Quicksort

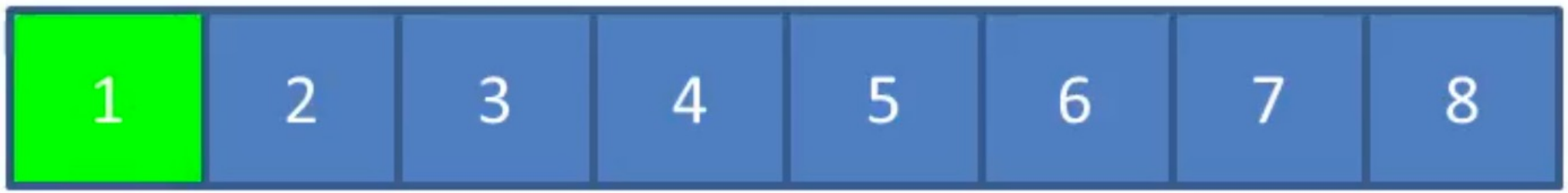


Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

Quicksort

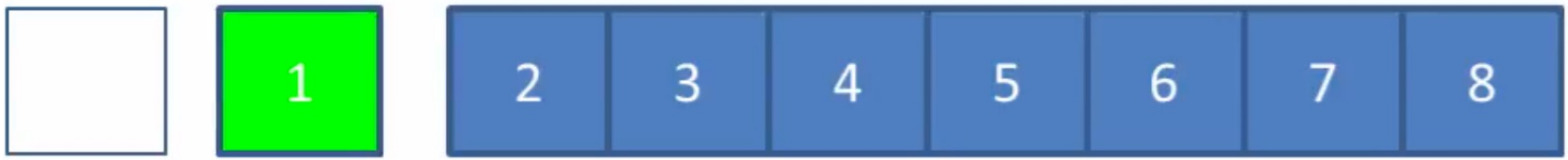


Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

Quicksort

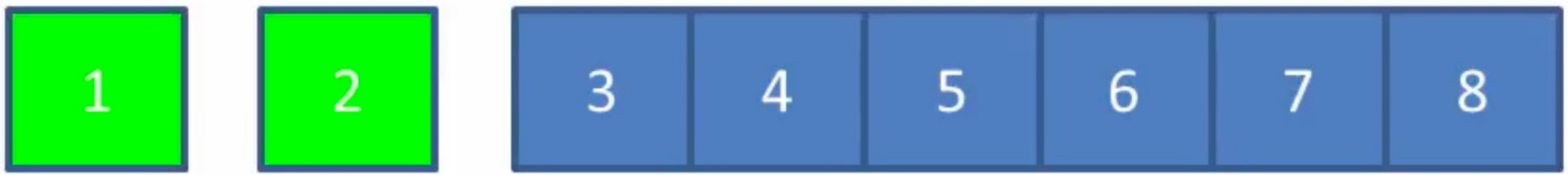


Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

Quicksort



Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

Quicksort

Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

Quicksort

Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

Quicksort

Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

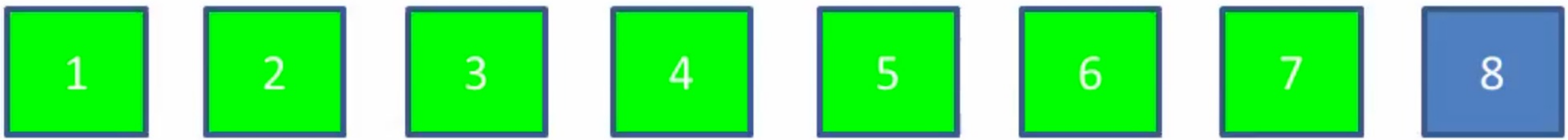
Quicksort

Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

Quicksort

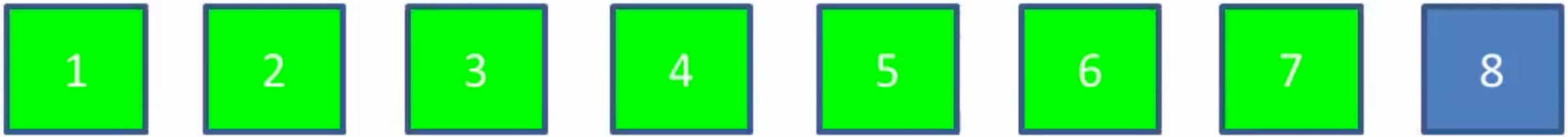


Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

Quicksort



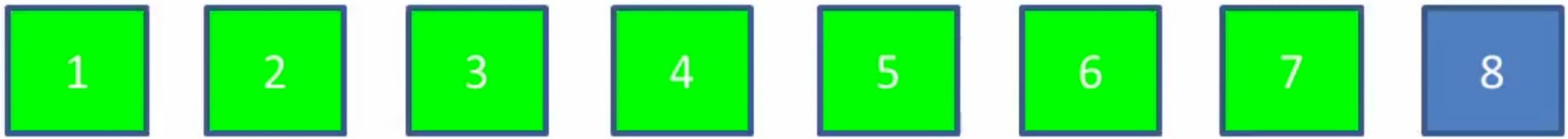
Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

7+6+5+4+3+2+1 = 28

Quicksort



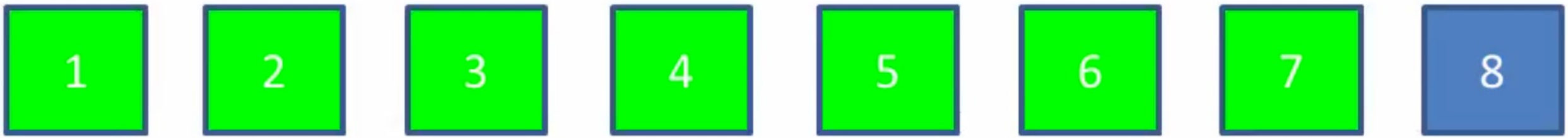
Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

$(n - 1) + (n - 2) + (n - 3) \ldots + 2 + 1$

Quicksort



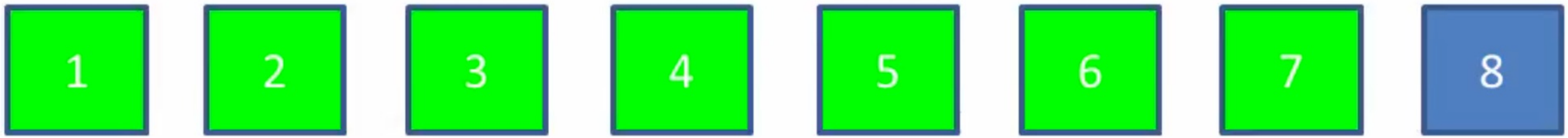
Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

$(n (n - 1)) / 2$

Quicksort



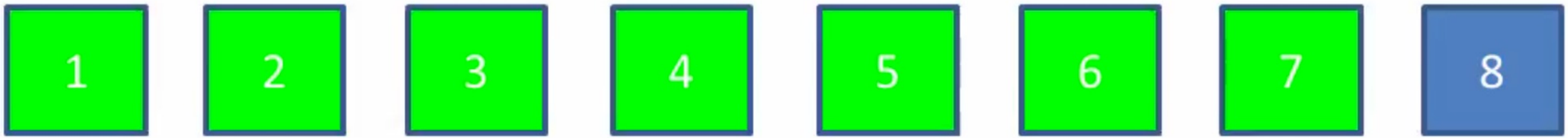
Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

$(n^2 - n) / 2$

Quicksort



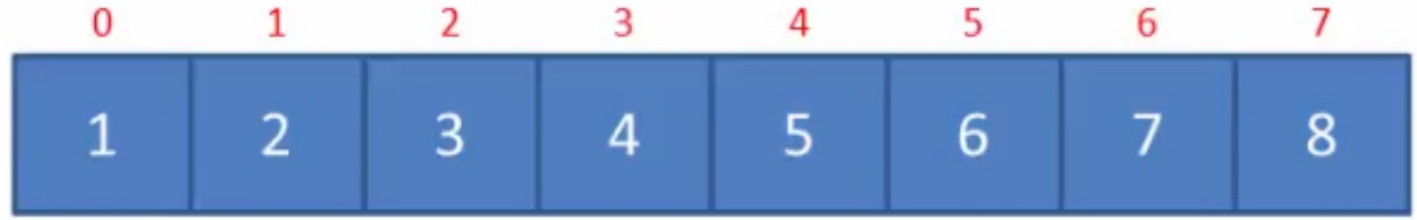
Repeat for each partition with more than one item

Partition the list

Until all partitions contain only one item

Quadratic time complexity **$O(n^2)$**

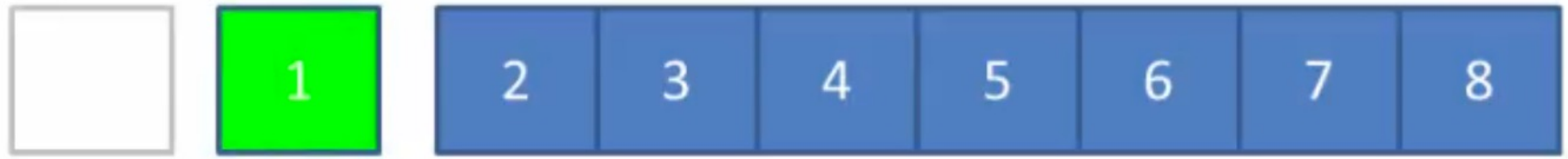
Quicksort




```
Sub QuickSort(Data(), Left, Right)

    If Left < Right Then
        PivotPosition = Partition(Data, Left, Right)
        QuickSort(Data, Left, PivotPosition - 1)
        QuickSort(Data, PivotPosition + 1, Right)
    End If

End Sub
```

Quadratic time complexity **$O(n^2)$**

Quicksort




```
Sub QuickSort(Data(), Left, Right)

    If Left < Right Then
        PivotPosition = Partition(Data, Left, Right)
        QuickSort(Data, Left, PivotPosition - 1)
        QuickSort(Data, PivotPosition + 1, Right)
    End If

End Sub
```

Left = 0 Right = 7 PivotPosition = 0

Quadratic time complexity **$O(n^2)$**

Quicksort


```
Sub QuickSort(Data(), Left, Right)

    If Left < Right Then
        PivotPosition = Partition(Data, Left, Right)
        QuickSort(Data, Left, PivotPosition - 1)
        QuickSort(Data, PivotPosition + 1, Right)
    End If

End Sub
```

Left = 6 Right = 5
Left = 6 Right = 7 PivotPosition = 6
Left = 5 Right = 7 PivotPosition = 5
Left = 4 Right = 7 PivotPosition = 4
Left = 3 Right = 7 PivotPosition = 3
Left = 2 Right = 7 PivotPosition = 2
Left = 1 Right = 7 PivotPosition = 1
Left = 0 Right = 7 PivotPosition = 0

Quadratic time complexity **$O(n^2)$**

Quicksort


```
Sub QuickSort(Data(), Left, Right)

    If Left < Right Then
        PivotPosition = Partition(Data, Left, Right)
        QuickSort(Data, Left, PivotPosition - 1)
        QuickSort(Data, PivotPosition + 1, Right)
    End If

End Sub
```

Left = 7 Right = 7
Left = 6 Right = 7 PivotPosition = 6
Left = 5 Right = 7 PivotPosition = 5
Left = 4 Right = 7 PivotPosition = 4
Left = 3 Right = 7 PivotPosition = 3
Left = 2 Right = 7 PivotPosition = 2
Left = 1 Right = 7 PivotPosition = 1
Left = 0 Right = 7 PivotPosition = 0

Quadratic time complexity **$O(n^2)$**

Quicksort


```
Sub QuickSort(Data(), Left, Right)

    If Left < Right Then
        PivotPosition = Partition(Data, Left, Right)
        QuickSort(Data, Left, PivotPosition - 1)
        QuickSort(Data, PivotPosition + 1, Right)
    End If

End Sub
```

Left = 3	Right = 7	PivotPosition = 3
Left = 2	Right = 7	PivotPosition = 2
Left = 1	Right = 7	PivotPosition = 1
Left = 0	Right = 7	PivotPosition = 0

Quadratic time complexity **$O(n^2)$**

Quicksort


```
Sub QuickSort(Data(), Left, Right)

    If Left < Right Then
        PivotPosition = Partition(Data, Left, Right)
        QuickSort(Data, Left, PivotPosition - 1)
        QuickSort(Data, PivotPosition + 1, Right)
    End If

End Sub
```

Left = 0 Right = 7 PivotPosition = 0

Quadratic time complexity **$O(n^2)$**

Linear space complexity **$O(n)$**