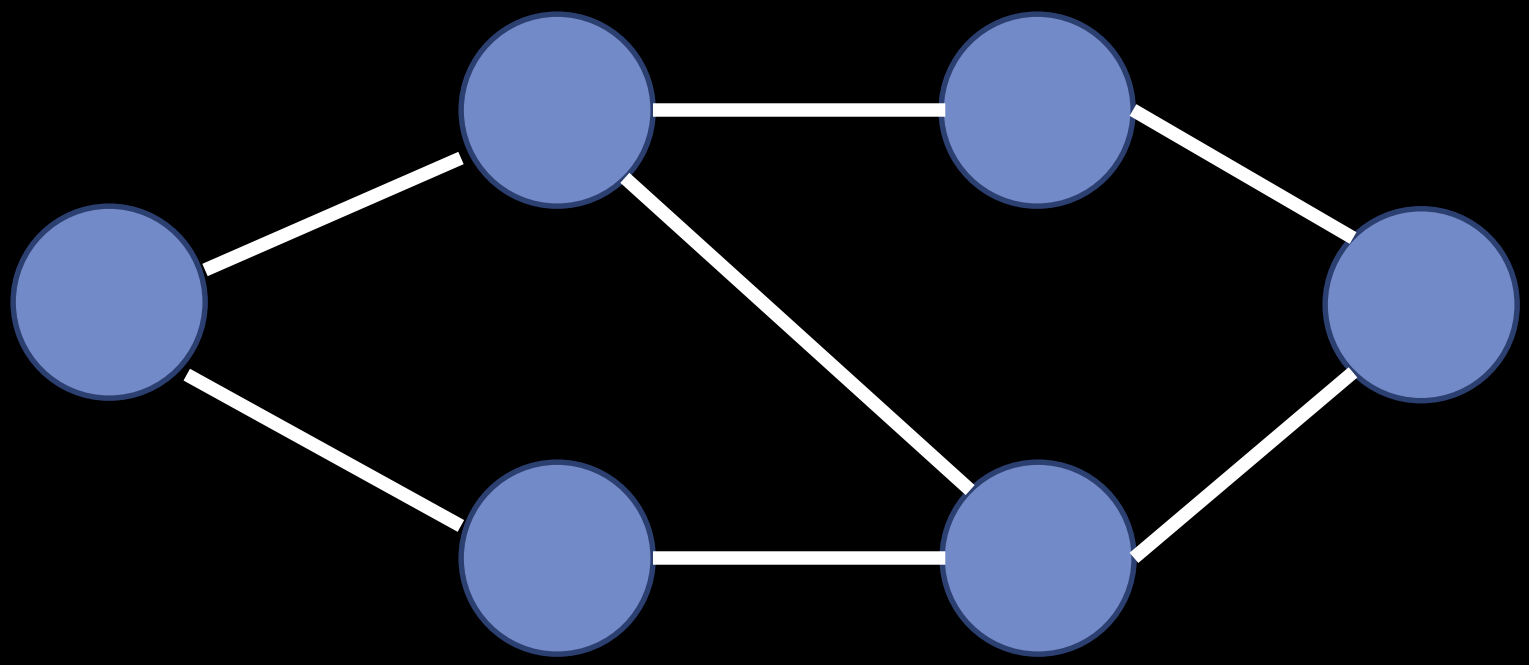
CS61BL

Lecture 5:

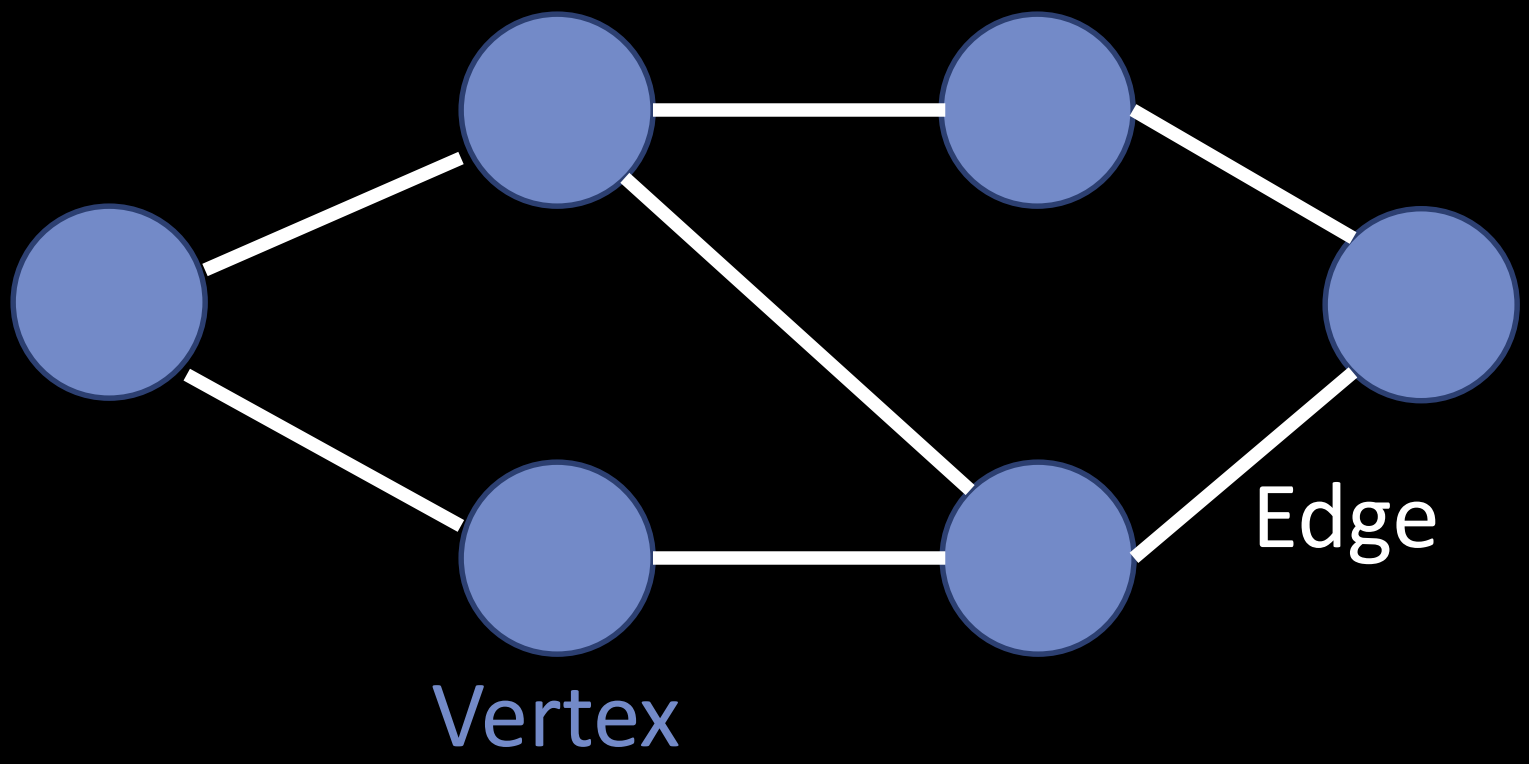
Graphs

Sorting

Graphs

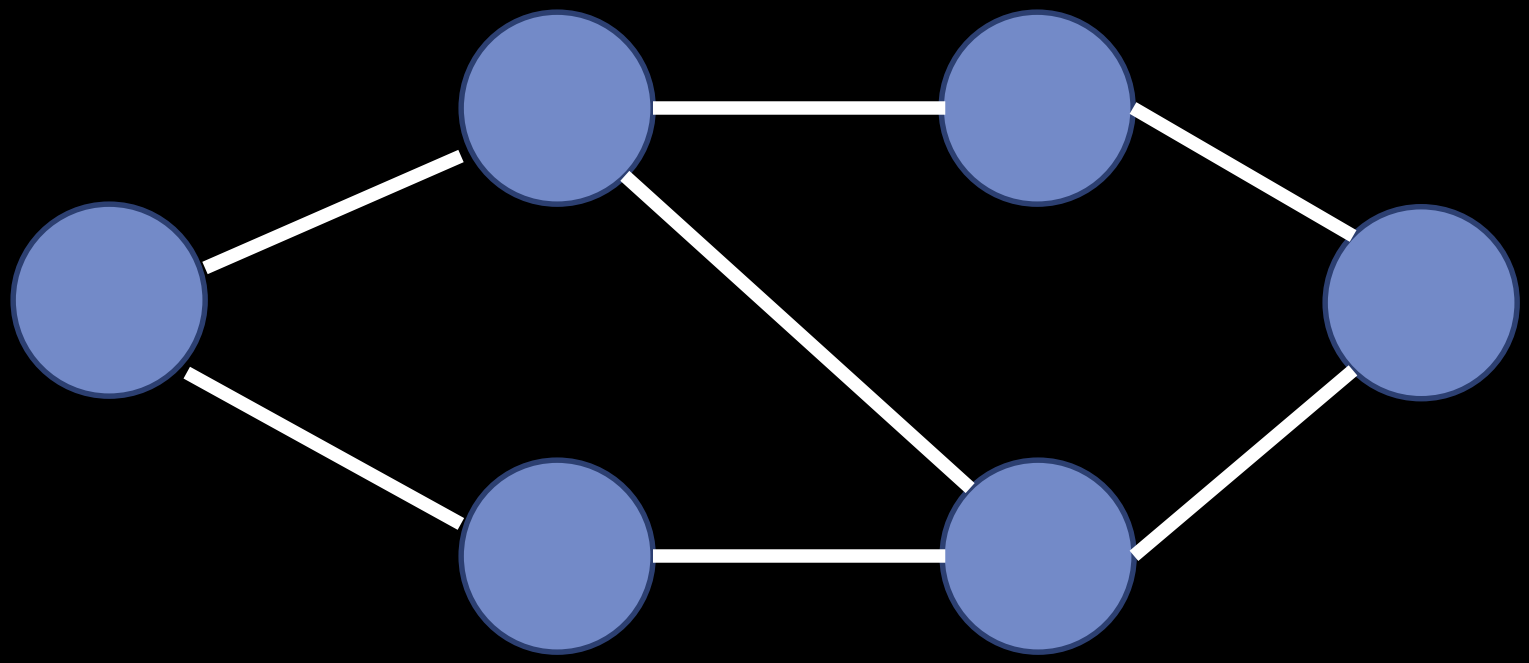


Graphs



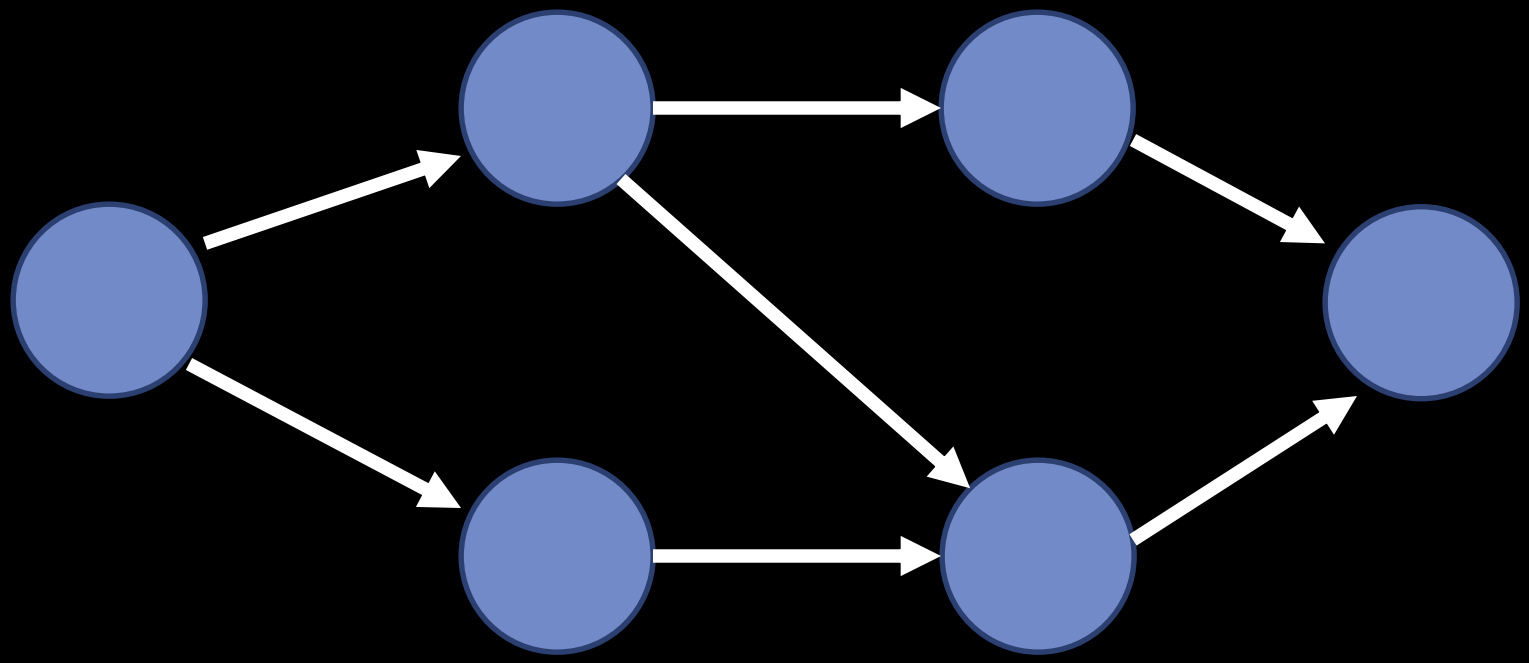
Graphs

(Undirected)



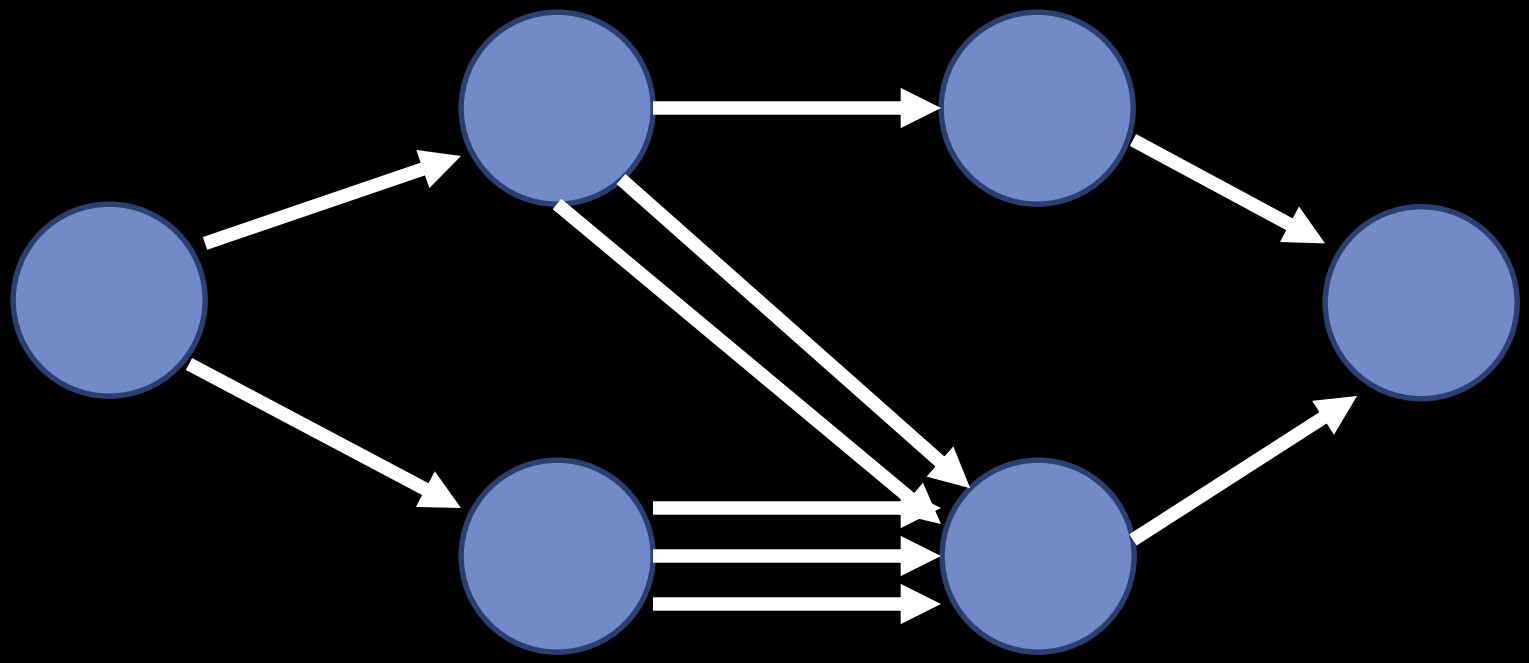
Graphs

(Directed)



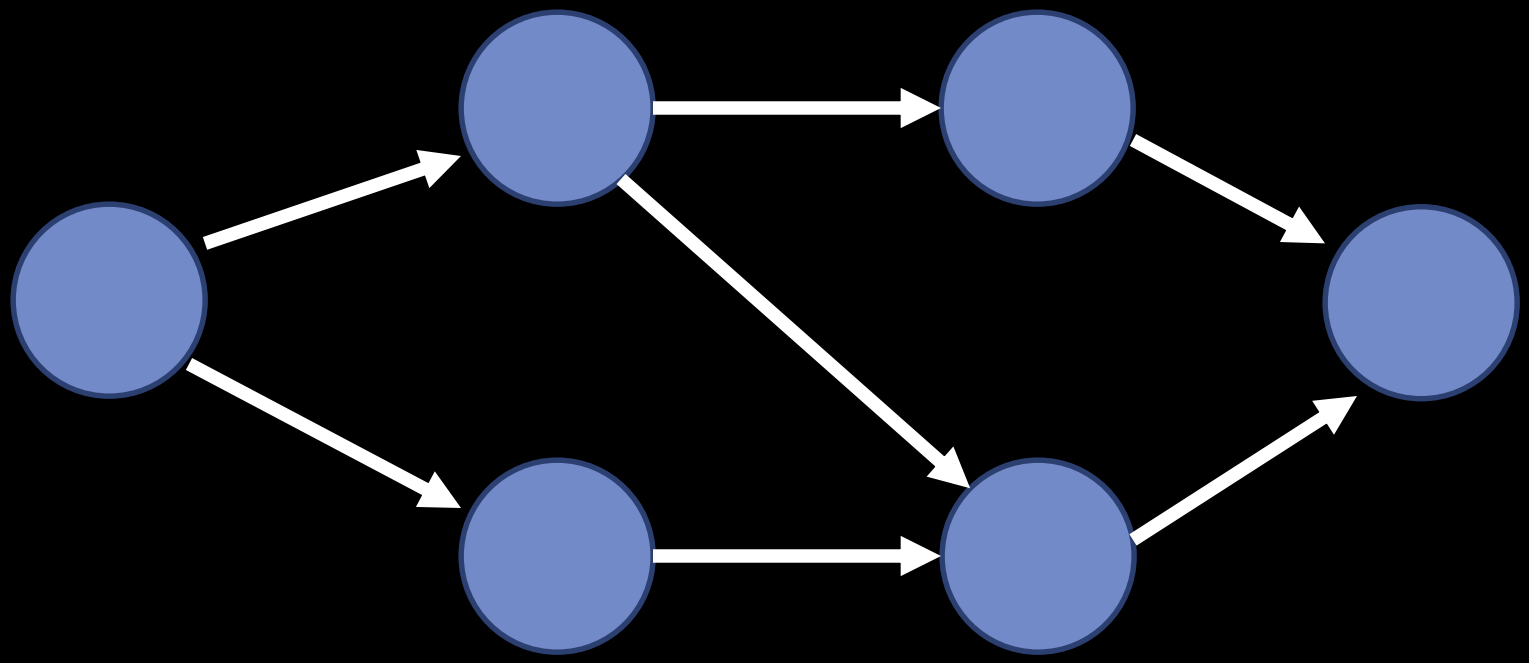
Graphs

(Multigraph)



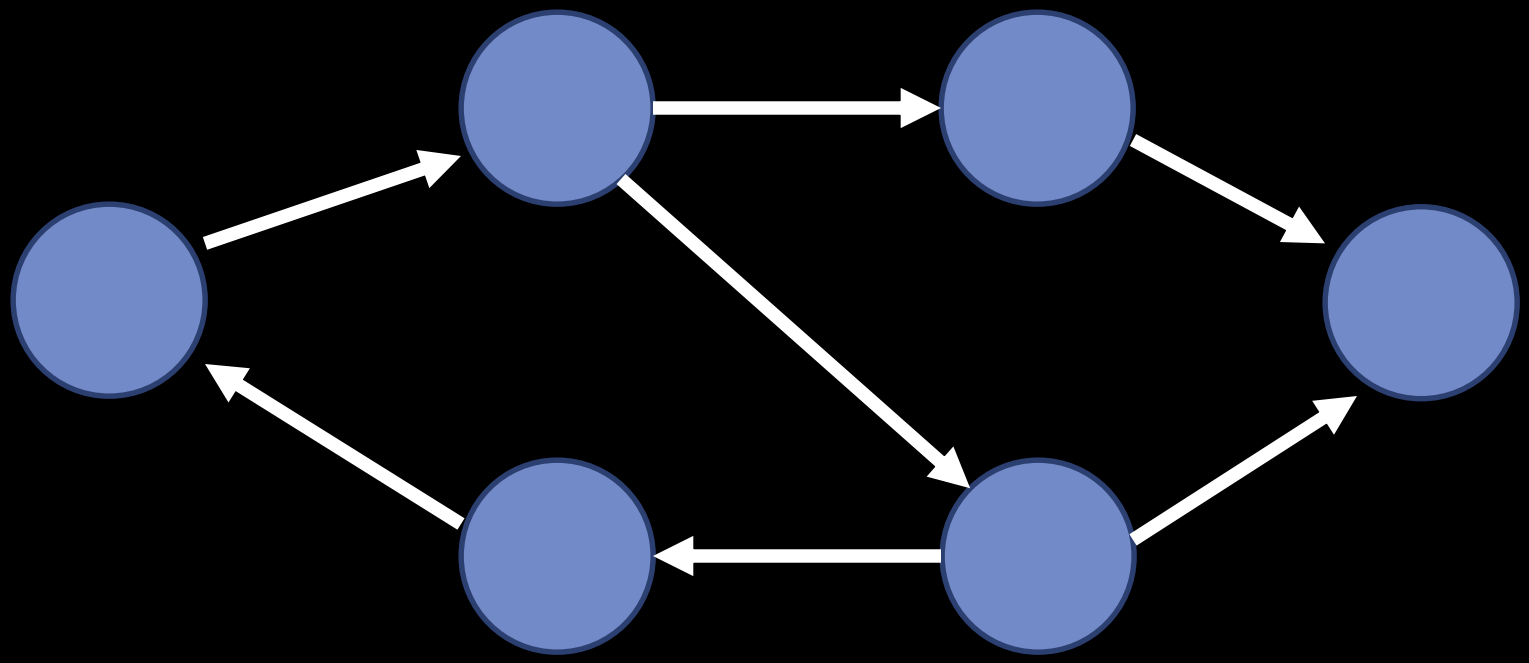
Graphs

(Acyclic)



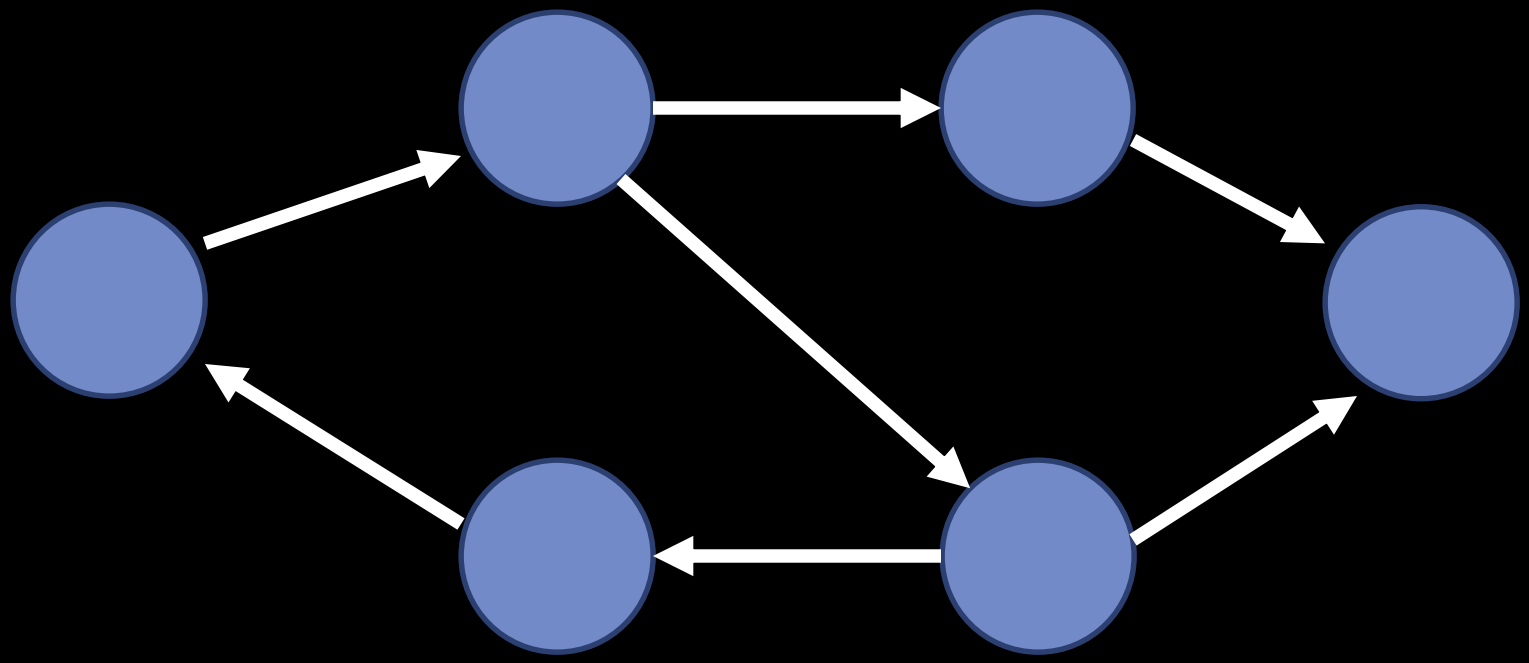
Graphs

(Cyclic)



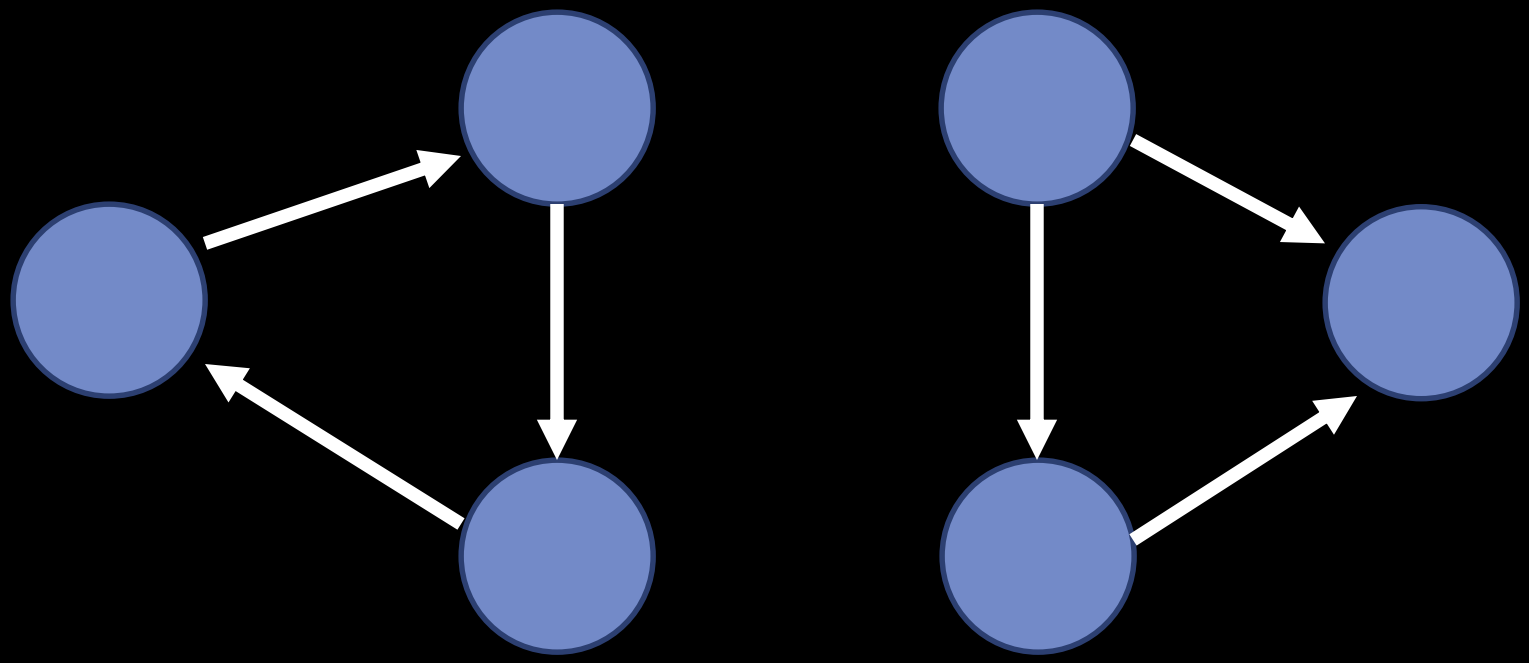
Graphs

(Connected)



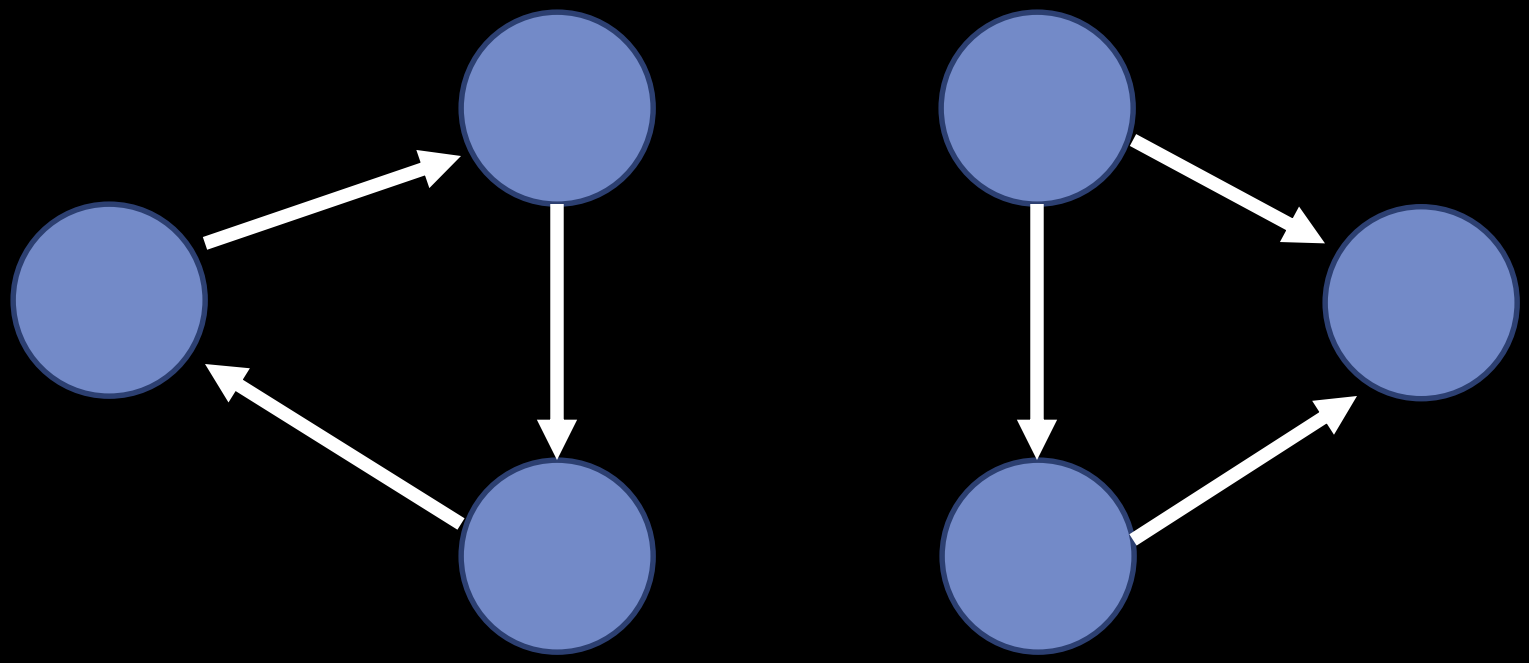
Graphs

(Disconnected)



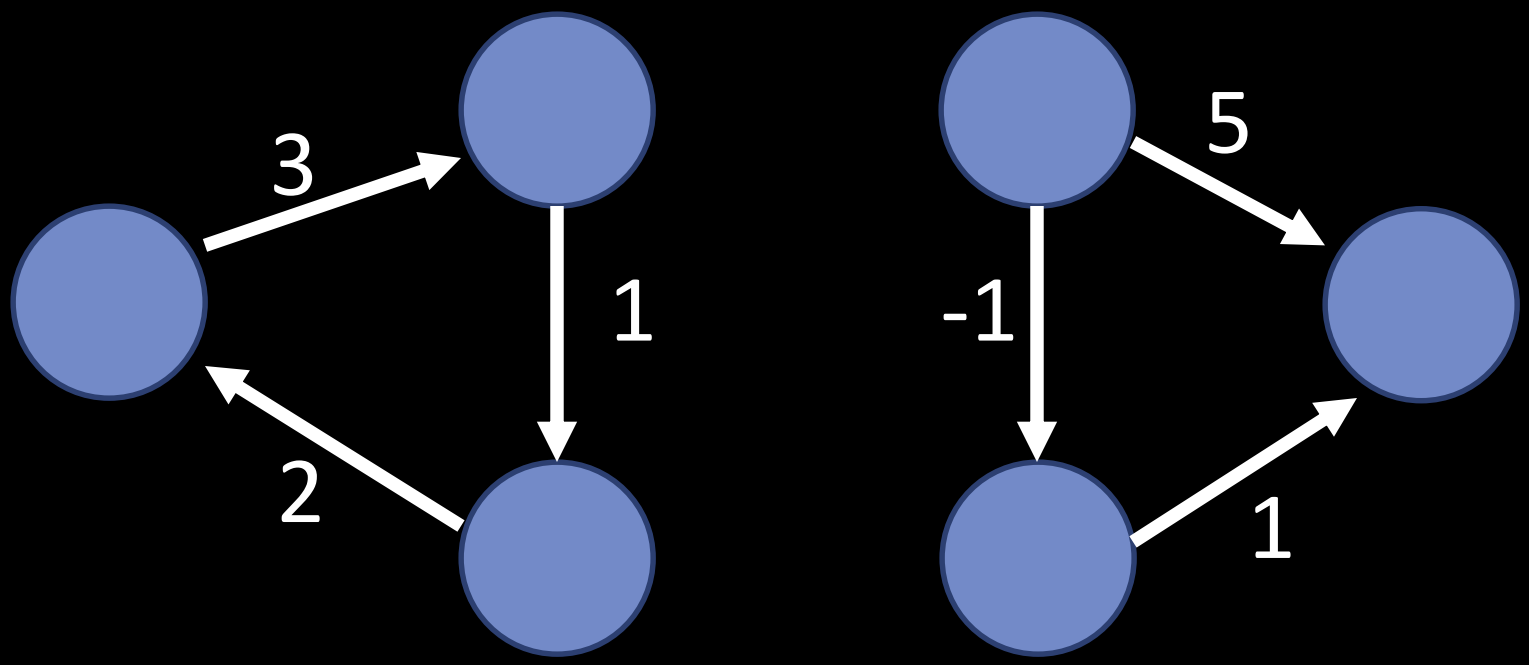
Graphs

(Unweighted)



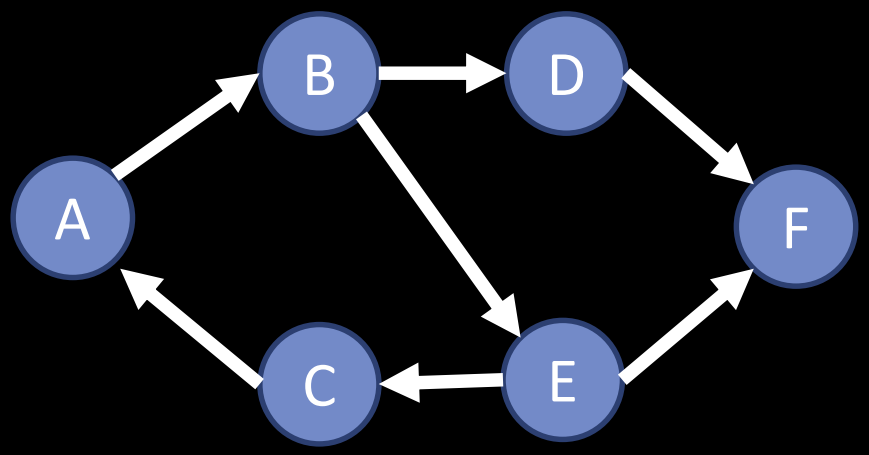
Graphs

(Weighted)



Graph Representation

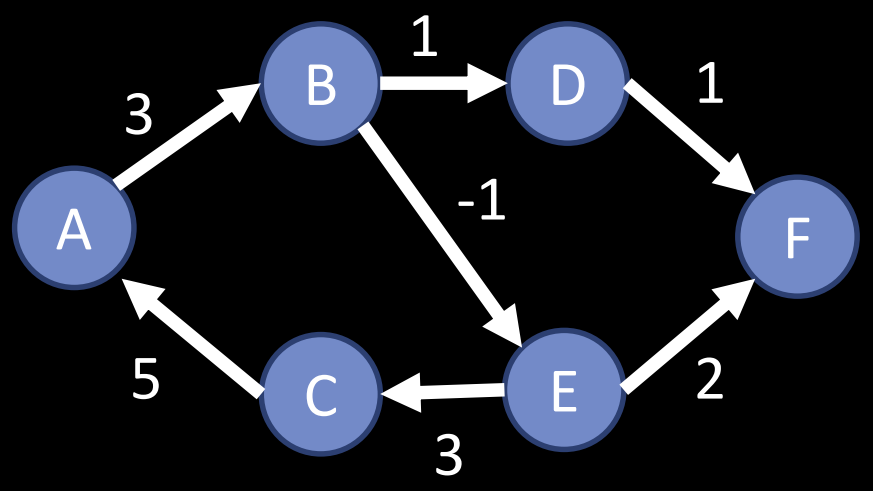
- Adjacency Matrix
  - Matrix (e.g. 2D array) specifying which edges are present in a graph



$$
\begin{array}{c c}
 & \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\begin{array}{c} A \\ B \\ C \\ D \\ E \\ F \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0
\end{pmatrix}
\end{array}
$$

Graph Representation

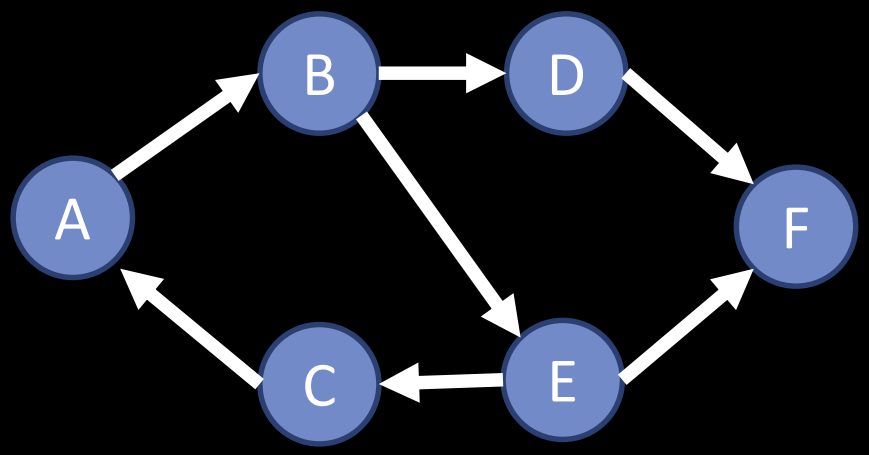
- Adjacency Matrix
  - Matrix (e.g. 2D array) specifying which edges are present in a graph



$$
\begin{array}{c}
 \\ A \\ B \\ C \\ D \\ E \\ F
\end{array}
\begin{array}{c}
\begin{array}{cccccc} A & B & C & D & E & F \end{array} \\
\begin{pmatrix}
X & 3 & X & X & X & X \\
X & X & X & 1 & -1 & X \\
5 & X & X & X & X & X \\
X & X & X & X & X & 1 \\
X & X & 3 & X & X & 2 \\
X & X & X & X & X & X
\end{pmatrix}
\end{array}
$$

Graph Representation

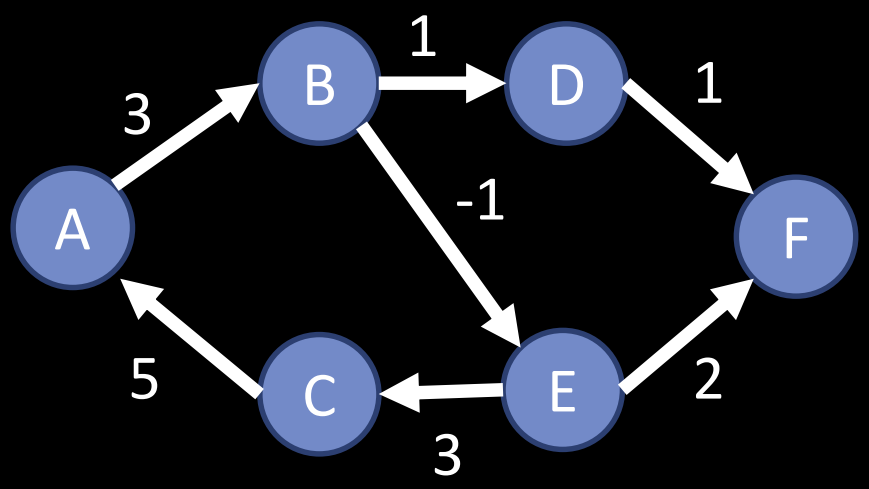
- Adjacency List
  - Collection of lists of neighbors
  - Each list specifies a single vertex's neighbors



A: [B]
B: [D, E]
C: [A]
D: [F]
E: [C, F]
F: [ ]

Graph Representation

- Adjacency List
  - Collection of lists of neighbors
  - Each list specifies a single vertex's neighbors



A: [(B, 3)]
B: [(D, 1), (E, -1)]
C: [(A, 5)]
D: [(F, 1)]
E: [(C, 3), (F, 2)]
F: [ ]

Graph Traversal

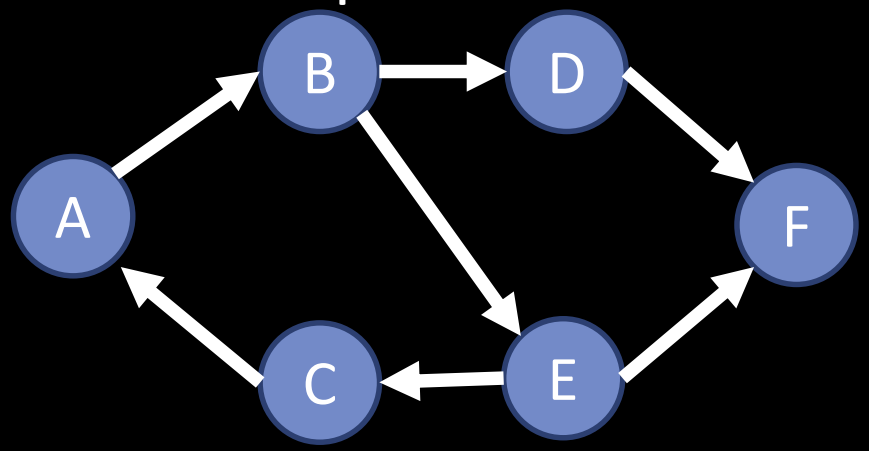
- Almost same as tree traversal
  - Keep track of which vertices have been already been visited
  - Pick any vertex to start at instead of root
  - BFS, DFS

Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue



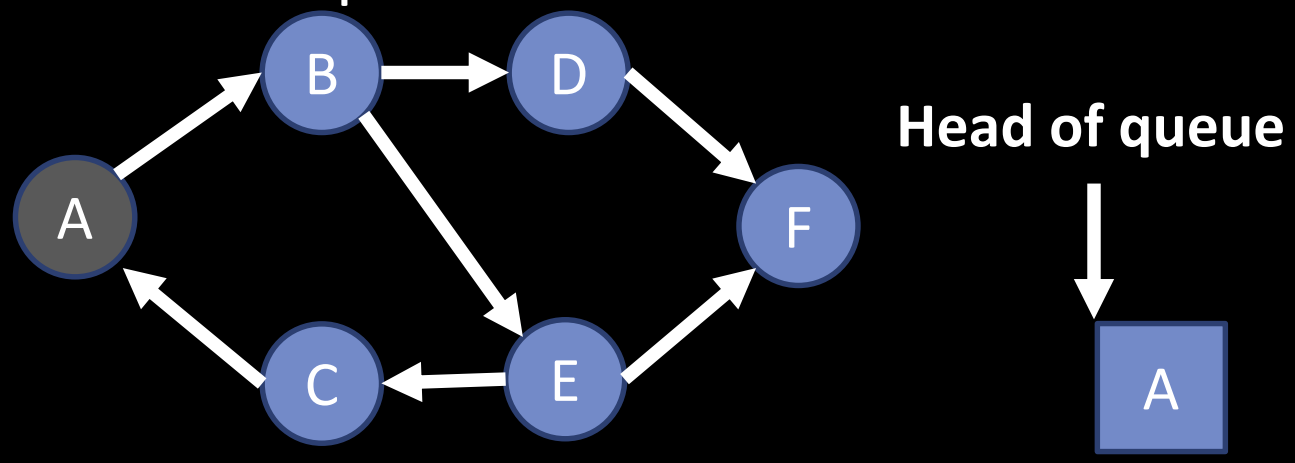
Head of queue

Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue

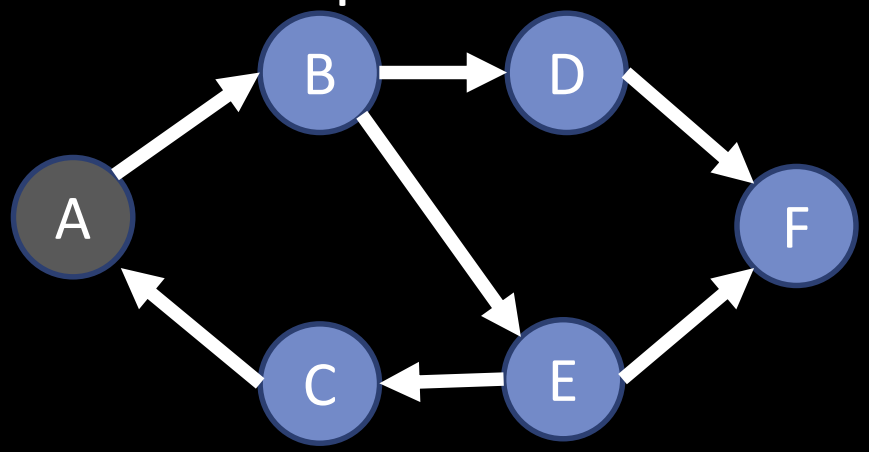


Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue



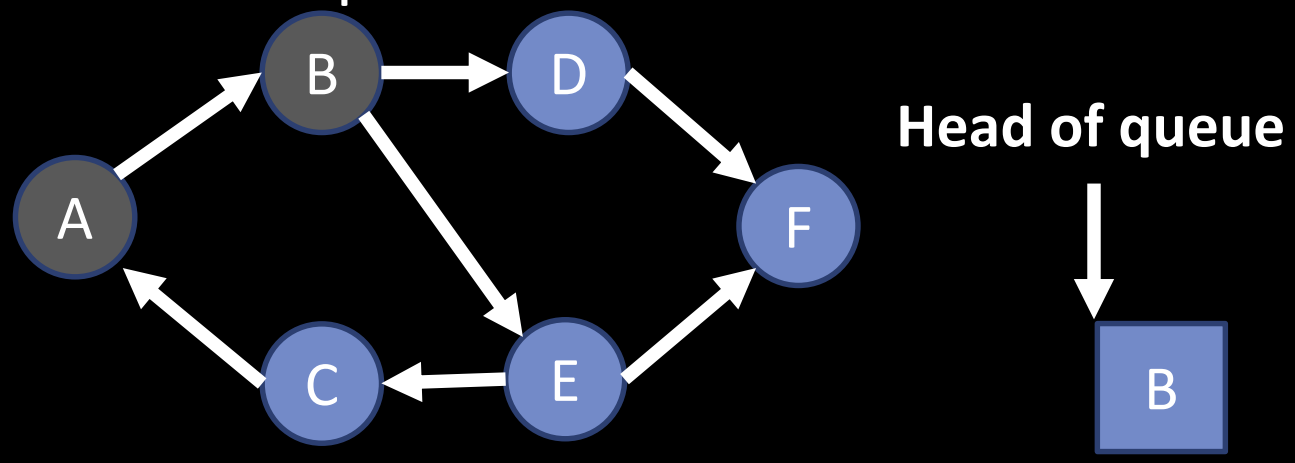
Head of queue

Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue

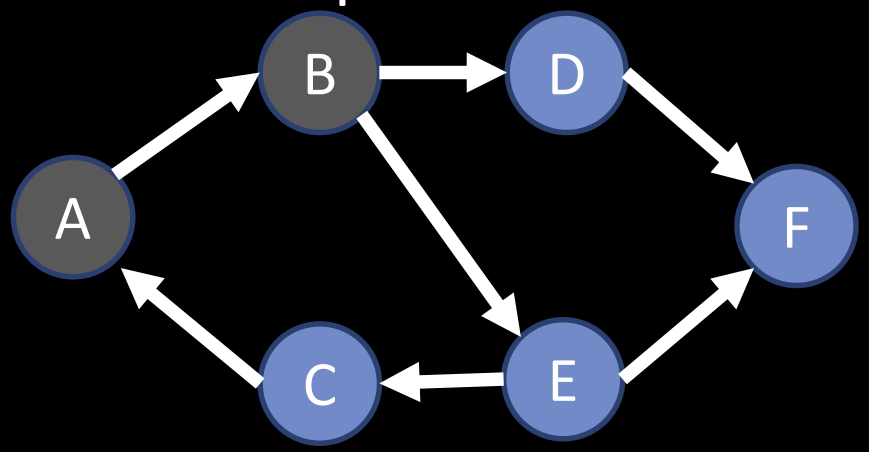


Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue



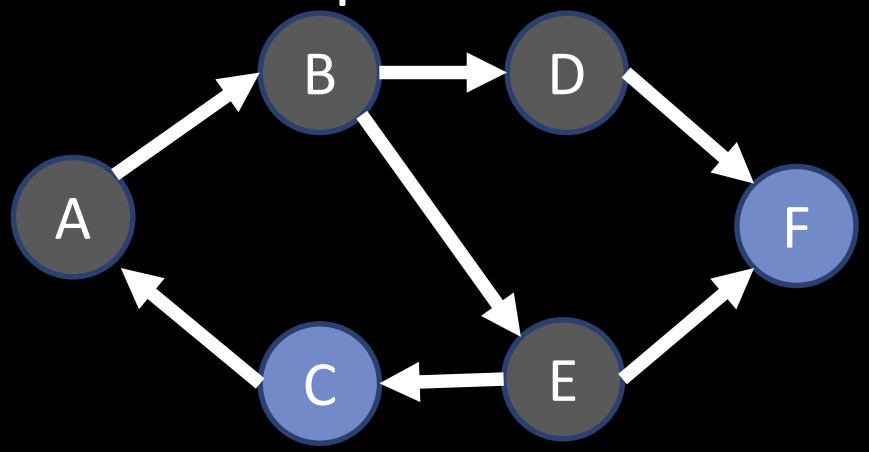
Head of queue

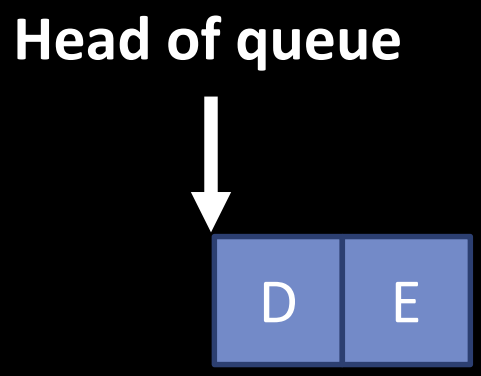
Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue



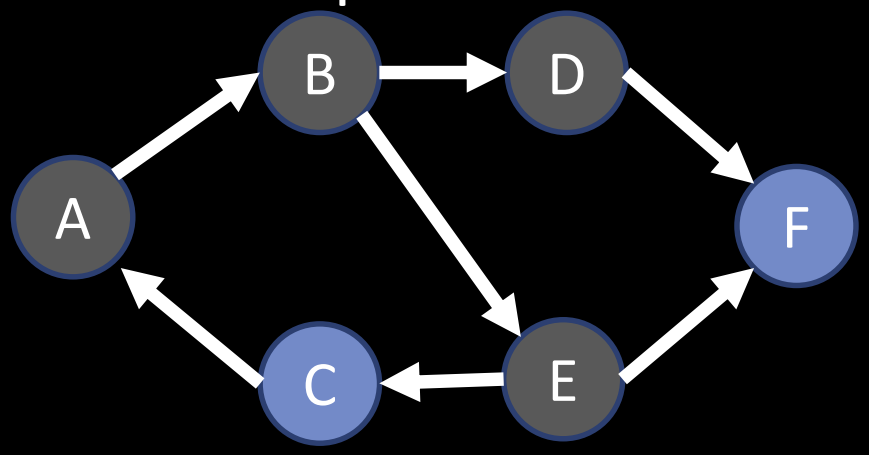


Graph Traversal: BFS Example

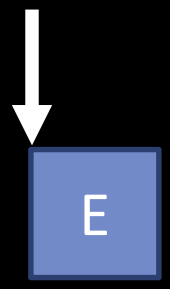
Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue



Head of queue

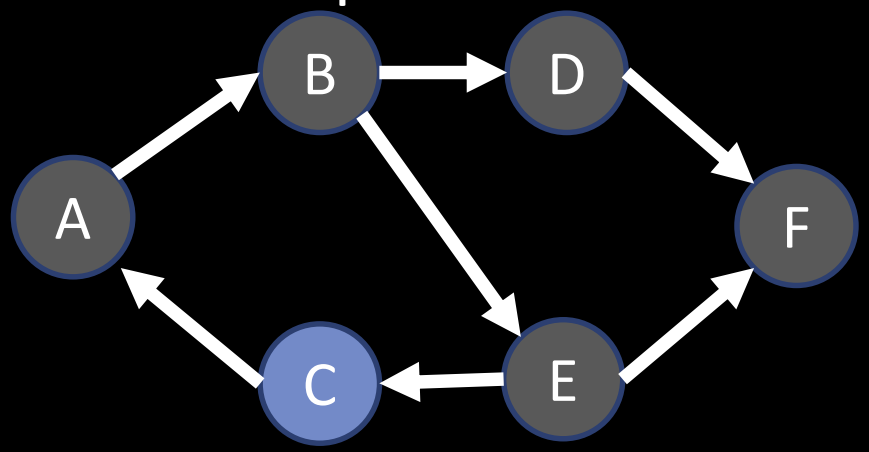


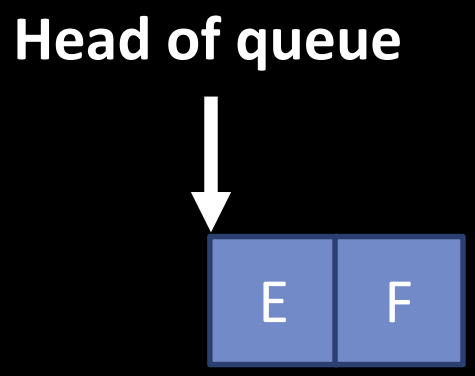
Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue



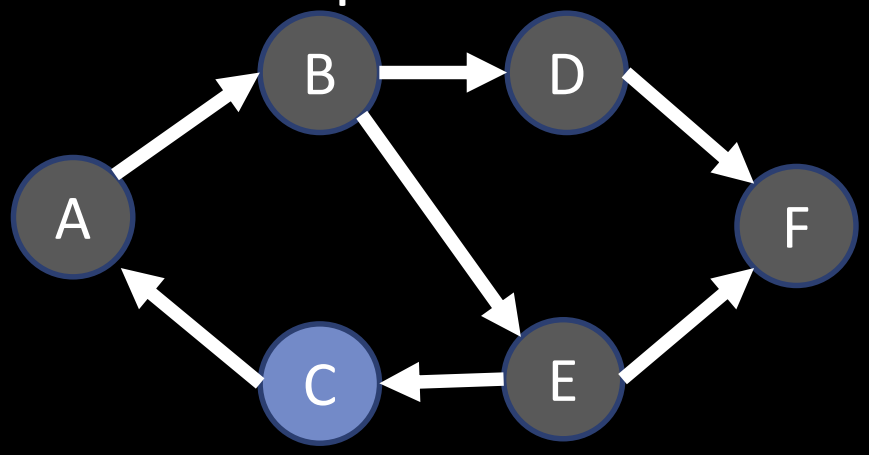


Graph Traversal: BFS Example

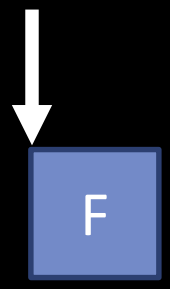
Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue



Head of queue

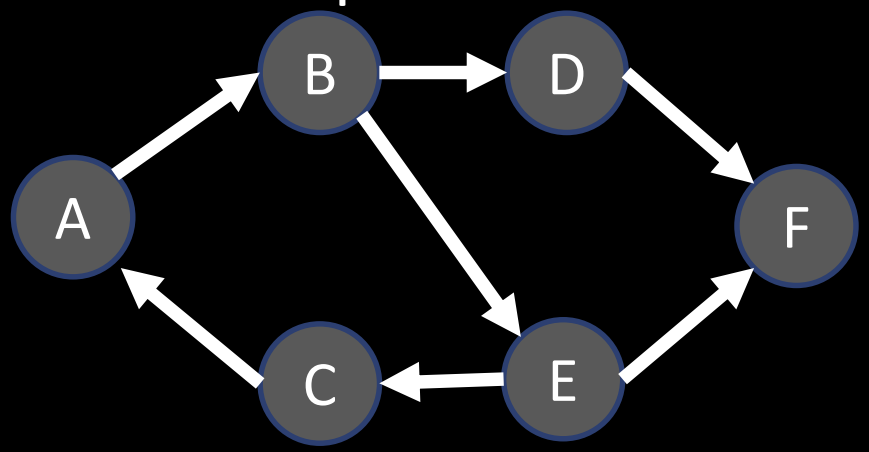


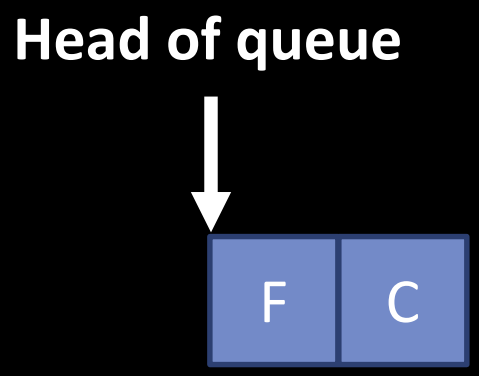
Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue



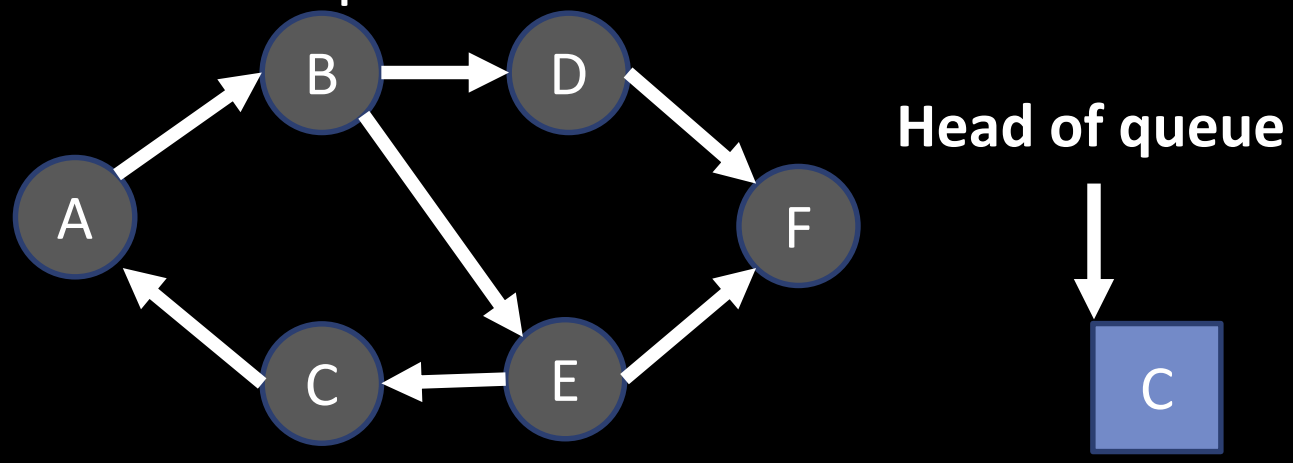


Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue

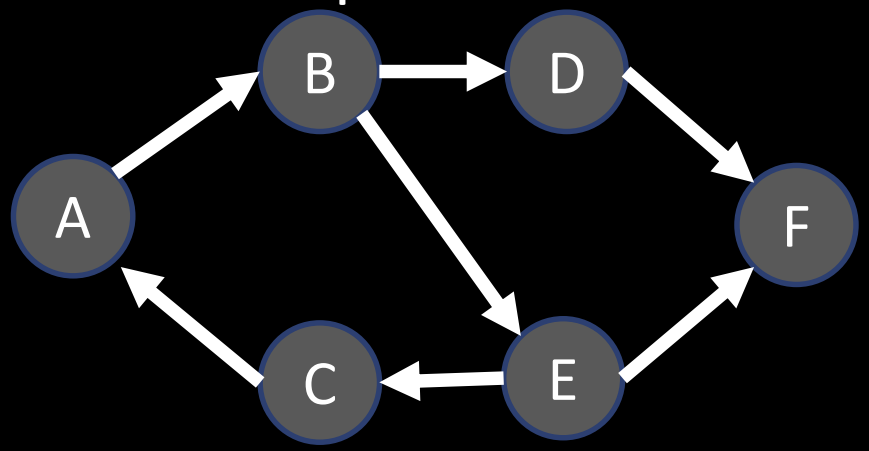


Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue



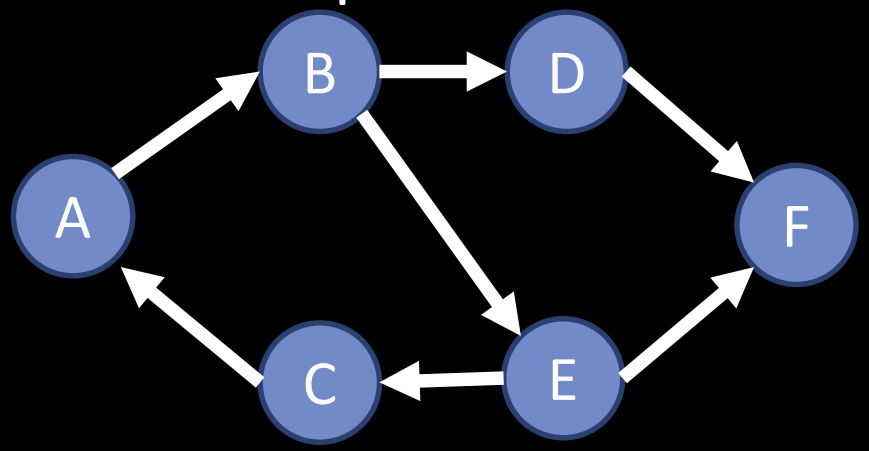
Head of queue

Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue



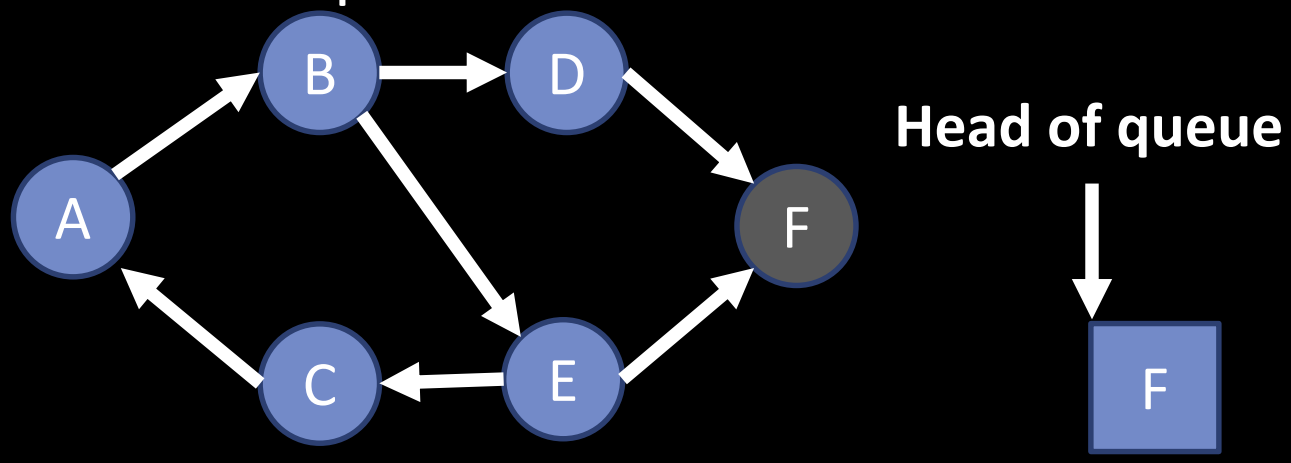
Head of queue

Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue

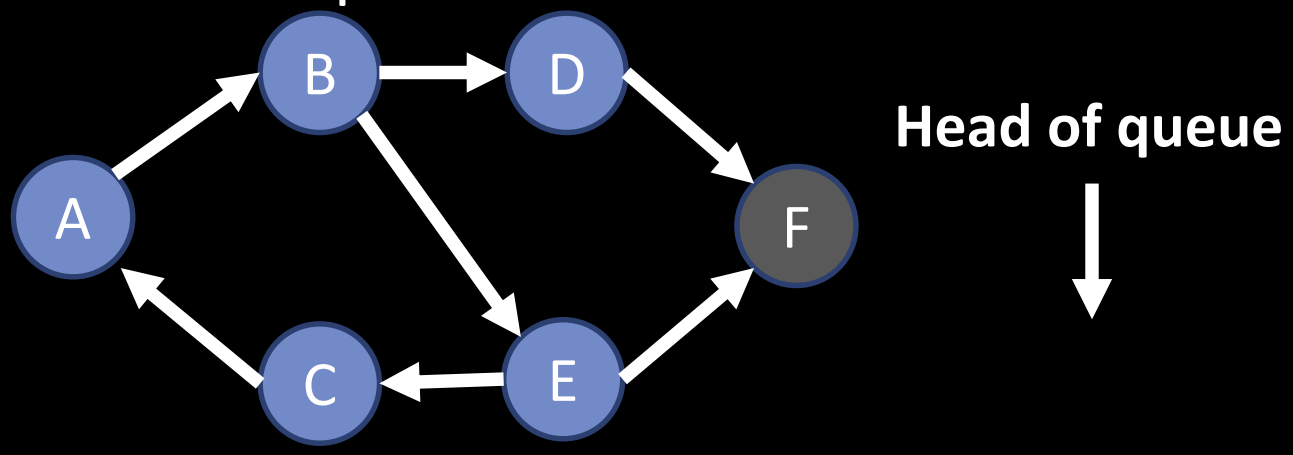


Graph Traversal: BFS Example

Mark starting vertex as “visited” and add to queue

While queue is not empty:

- Remove a vertex from queue
- Mark its unvisited neighbors as “visited” and add to queue

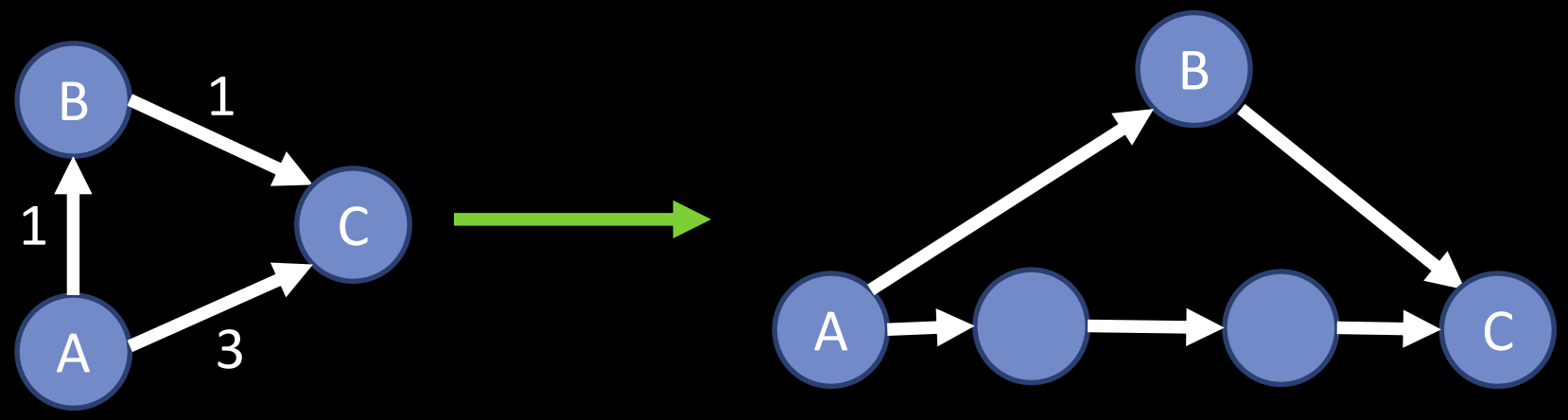


Shortest Path Problem

- Find shortest path from some start vertex *s* to some end vertex *t*
- Google Maps, puzzles, pathing AIs, etc…
- Unweighted graph: use BFS
- Weighted graph: ?

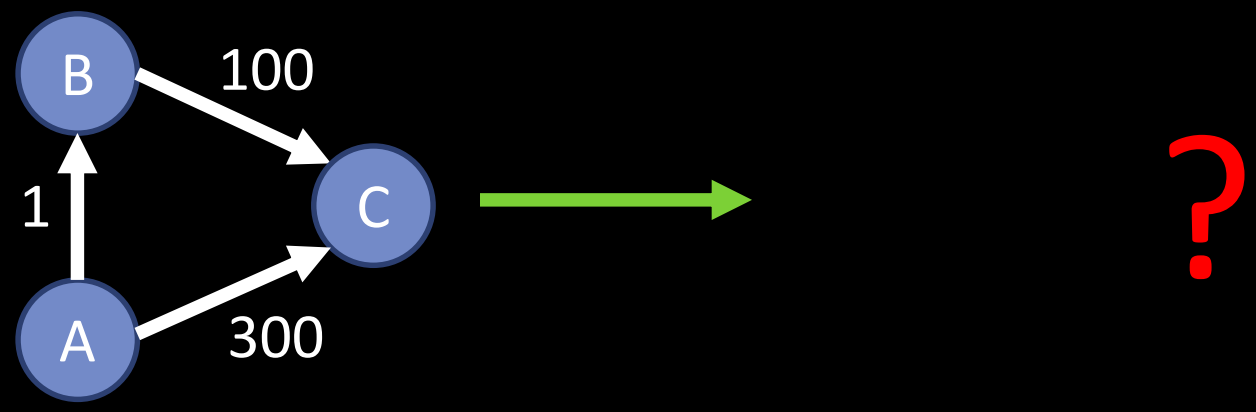
Shortest Path Problem

- Weighted graph?
- Split each weighted edge into multiple intermediary vertices and edges?
  - Then run BFS?



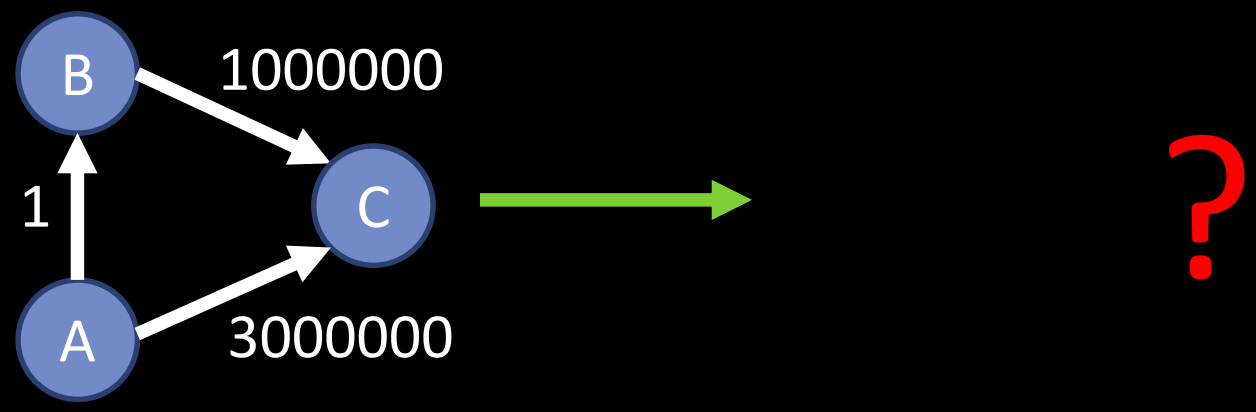
Shortest Path Problem

- Weighted graph?
- Split each weighted edge into multiple intermediary vertices and edges?
  - Then run BFS?



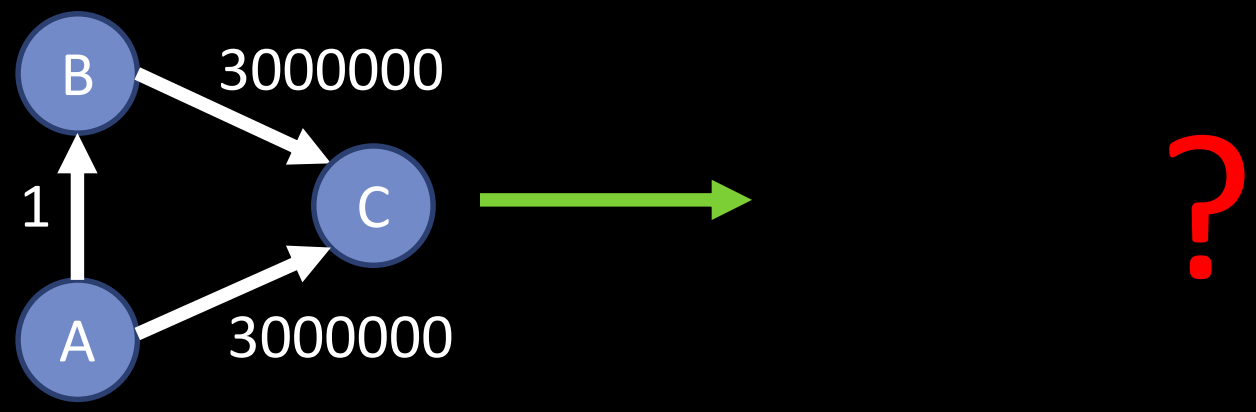
Shortest Path Problem

- Weighted graph?
- Split each weighted edge into multiple intermediary vertices and edges?
  - Then run BFS?



Shortest Path Problem

- Weighted graph?
- Split each weighted edge into multiple intermediary vertices and edges?
  - Then run BFS?



Dijkstra's Algorithm

- Finds shortest path from start vertex to every other vertex in graph
- Use priority queue as fringe
  - Priority value of vertex *u* is shortest path length we've found from starting vertex *s* so far

Dijkstra's Algorithm Example

Add starting vertex *s* to priority queue w/ priority 0

Add all other vertices with priority ∞

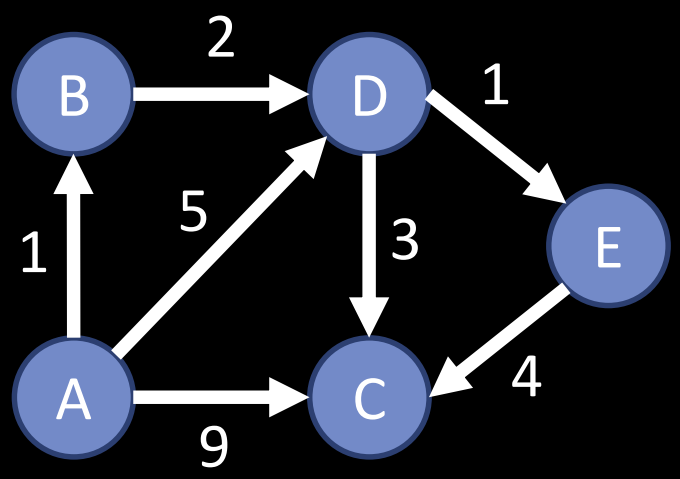
While priority queue is not empty:

- *u* = vertex dequeued from priority queue
- For each neighbor *v*:
  - If *v* is in priority queue, update *v*'s priority value with minimum of:
    - Current priority
    - Priority of *u* + weight of edge (*u*, *v*)

Dijkstra's Algorithm Example

Add *s* to priority queue with priority 0

Add all other vertices with priority ∞



Priority Queue

A	B	C	D	E
0	∞	∞	∞	∞

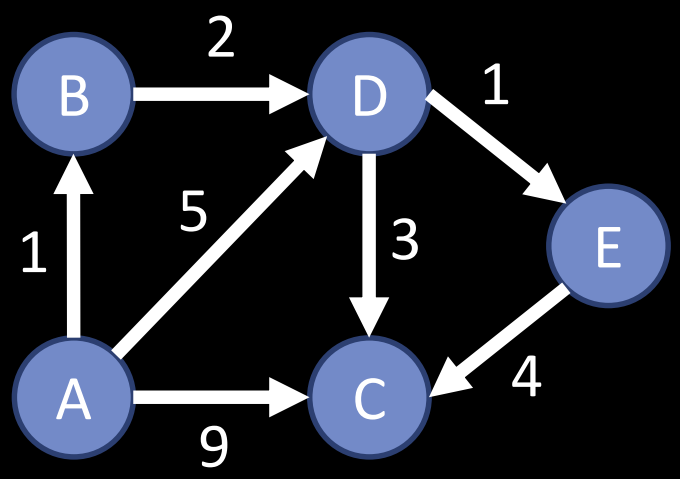
Shortest Distances

A	B	C	D	E
?	?	?	?	?

Dijkstra's Algorithm Example

While priority queue is not empty:

u = vertex dequeue'd from priority queue



Priority Queue

A	B	C	D	E
	∞	∞	∞	∞

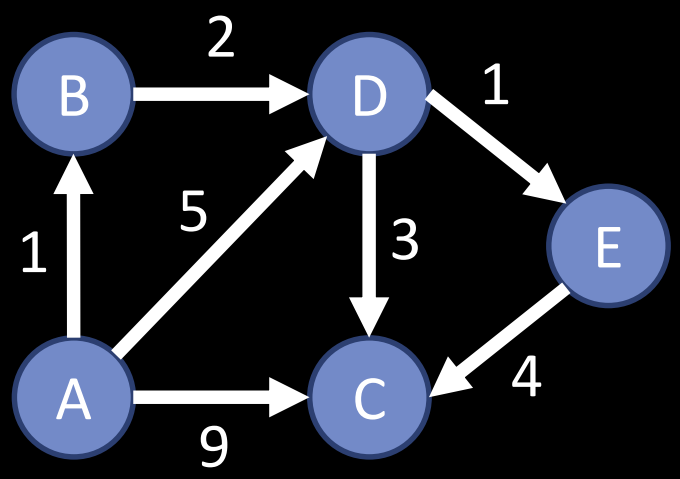
Shortest Distances

A	B	C	D	E
0	?	?	?	?

Dijkstra's Algorithm Example

For each neighbor *v*, if *v* is in priority queue, update *v*'s priority value with minimum of:

[Current priority, Priority of *u* + weight of edge (*u*, *v*)]



Priority Queue

A	B	C	D	E
	∞	∞	∞	∞

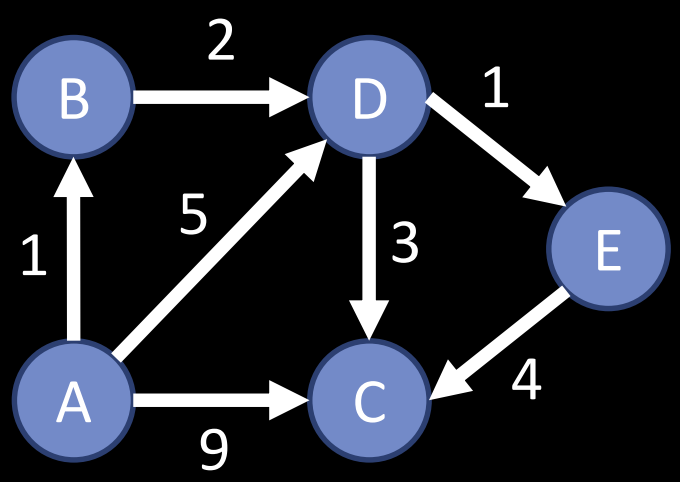
Shortest Distances

A	B	C	D	E
0	?	?	?	?

Dijkstra's Algorithm Example

For each neighbor *v*, if *v* is in priority queue, update *v*'s priority value with minimum of:

[Current priority, Priority of *u* + weight of edge (*u*, *v*)]



Priority Queue

A	B	C	D	E
	1	9	5	∞

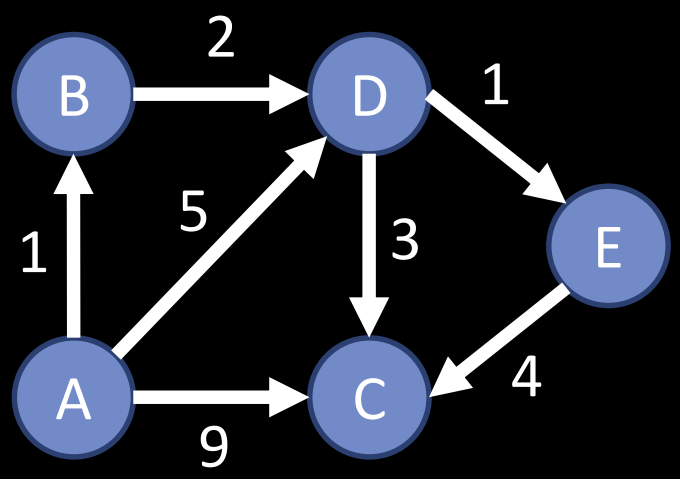
Shortest Distances

A	B	C	D	E
0	?	?	?	?

Dijkstra's Algorithm Example

While priority queue is not empty:

u = vertex dequeued from priority queue



Priority Queue

A	B	C	D	E
		9	5	∞

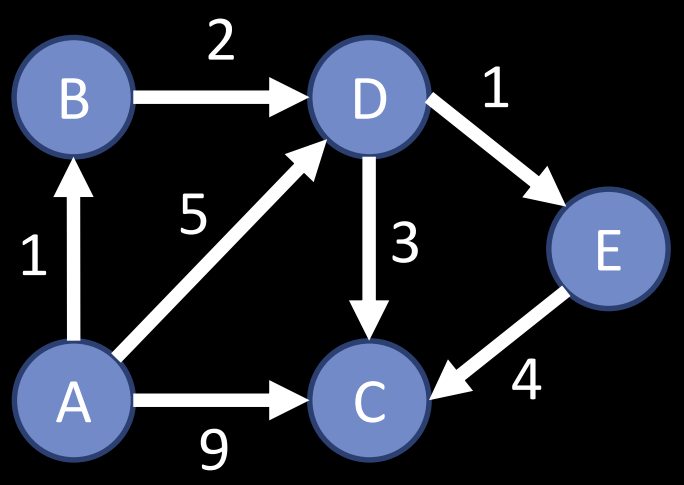
Shortest Distances

A	B	C	D	E
0	1	?	?	?

Dijkstra's Algorithm Example

For each neighbor *v*, if *v* is in priority queue, update *v*'s priority value with minimum of:

[Current priority, Priority of *u* + weight of edge (*u*, *v*)]



Priority Queue

A	B	C	D	E
		9	5	∞

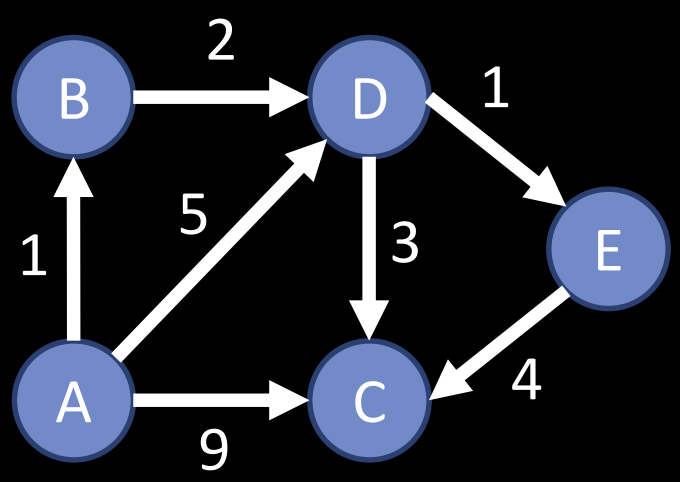
Shortest Distances

A	B	C	D	E
0	1	?	?	?

Dijkstra's Algorithm Example

For each neighbor *v*, if *v* is in priority queue, update *v*'s priority value with minimum of:

[Current priority, Priority of *u* + weight of edge (*u*, *v*)]



Priority Queue

A	B	C	D	E
		9	3	∞

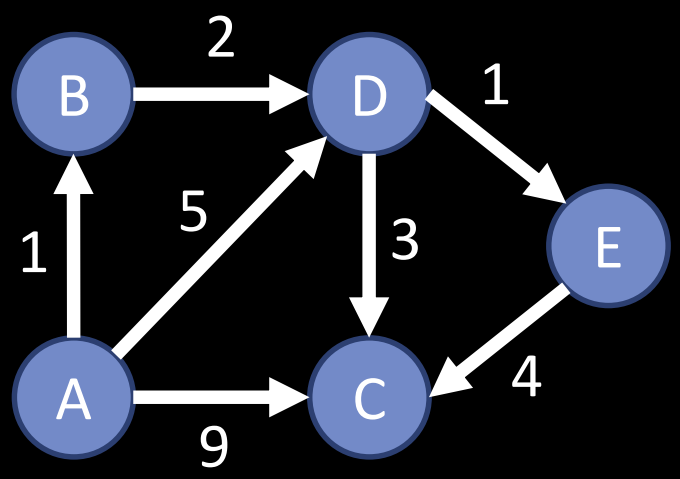
Shortest Distances

A	B	C	D	E
0	1	?	?	?

Dijkstra's Algorithm Example

While priority queue is not empty:

u = vertex dequeued from priority queue



Priority Queue

A	B	C	D	E
		9		∞

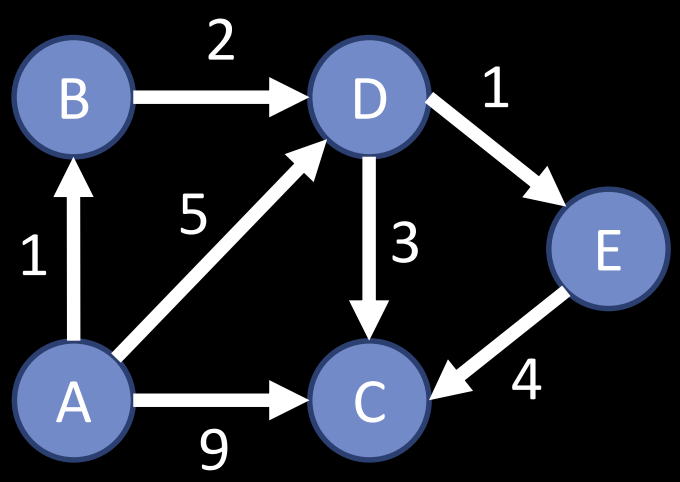
Shortest Distances

A	B	C	D	E
0	1	?	3	?

Dijkstra's Algorithm Example

For each neighbor *v*, if *v* is in priority queue, update *v*'s priority value with minimum of:

[Current priority, Priority of *u* + weight of edge (*u*, *v*)]



Priority Queue

A	B	C	D	E
		9		∞

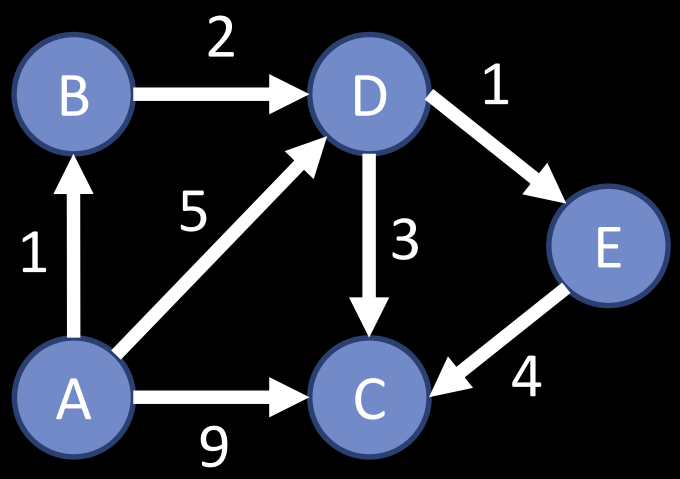
Shortest Distances

A	B	C	D	E
0	1	?	3	?

Dijkstra's Algorithm Example

For each neighbor *v*, if *v* is in priority queue, update *v*'s priority value with minimum of:

[Current priority, Priority of *u* + weight of edge (*u*, *v*)]



Priority Queue

A	B	C	D	E
		6		4

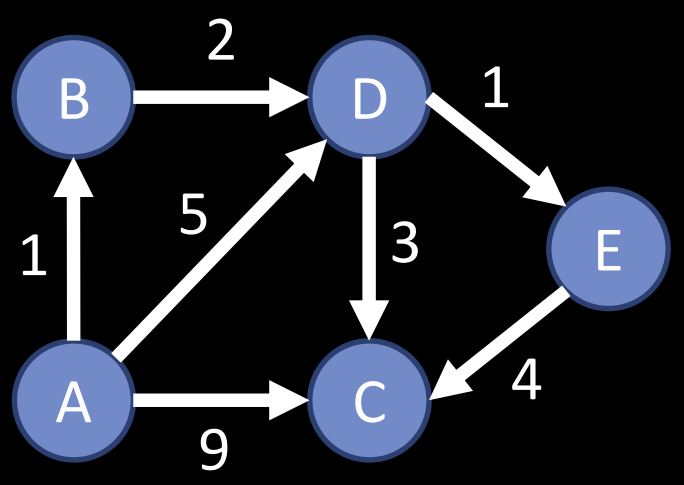
Shortest Distances

A	B	C	D	E
0	1	?	3	?

Dijkstra's Algorithm Example

While priority queue is not empty:

u = vertex dequeued from priority queue



Priority Queue

A	B	C	D	E
		6		

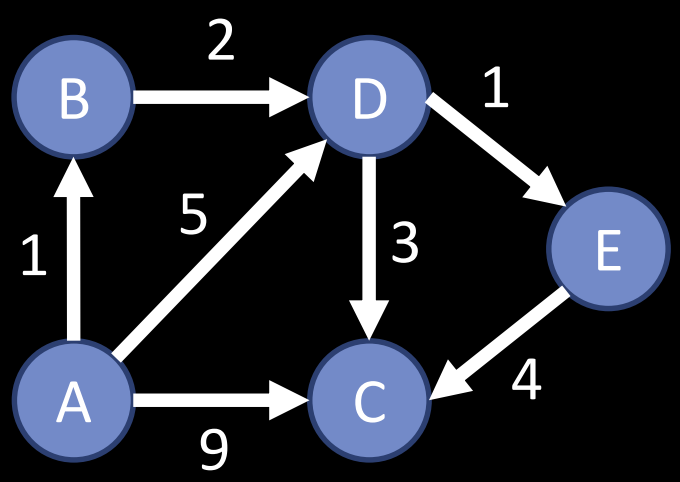
Shortest Distances

A	B	C	D	E
0	1	?	3	4

Dijkstra's Algorithm Example

For each neighbor *v*, if *v* is in priority queue, update *v*'s priority value with minimum of:

[Current priority, Priority of *u* + weight of edge (*u*, *v*)]



Priority Queue

A	B	C	D	E
		6		

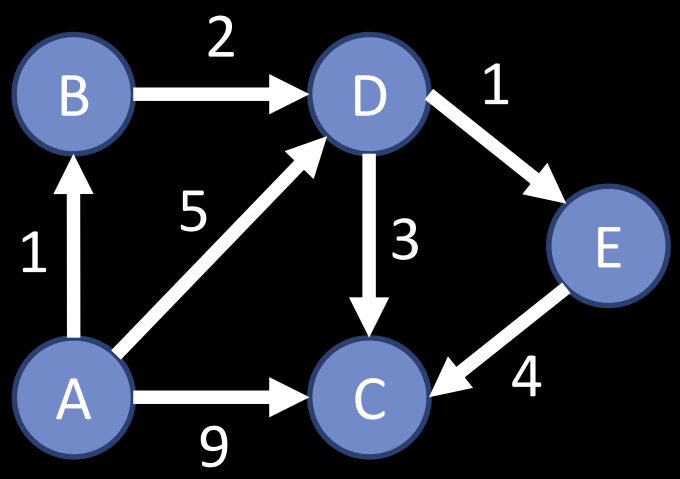
Shortest Distances

A	B	C	D	E
0	1	?	3	4

Dijkstra's Algorithm Example

While priority queue is not empty:

u = vertex dequeued from priority queue



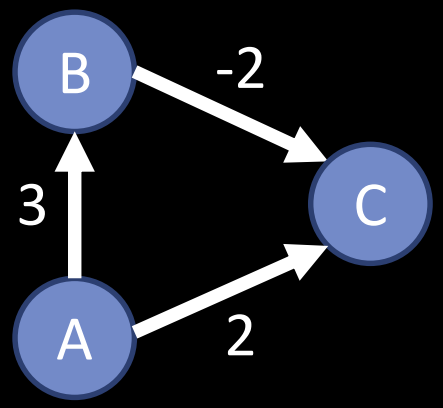
Priority Queue

A	B	C	D	E

Shortest Distances

A	B	C	D	E
0	1	6	3	4

Dijkstra Example



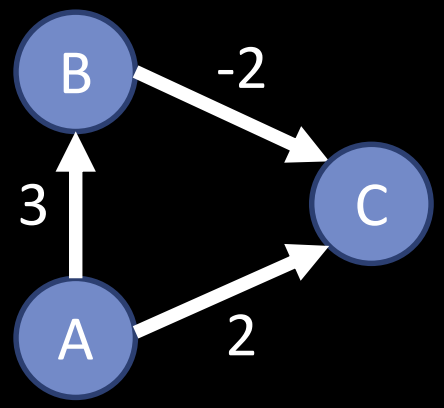
Priority Queue

A	B	C
0	∞	∞

Shortest Distances

A	B	C
?	?	?

Dijkstra Example



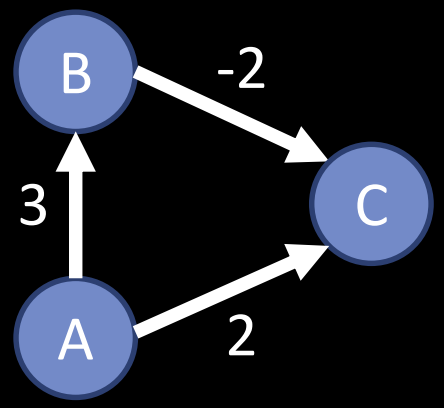
Priority Queue

A	B	C
	∞	∞

Shortest Distances

A	B	C
0	?	?

Dijkstra Example



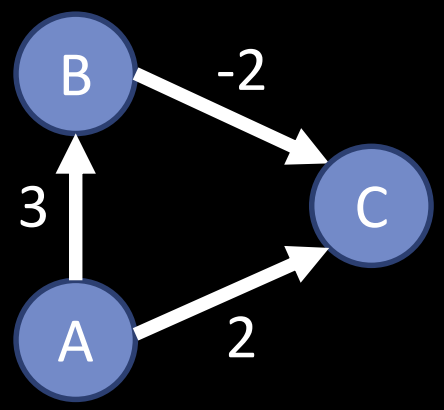
Priority Queue

A	B	C
	3	2

Shortest Distances

A	B	C
0	?	?

Dijkstra Example



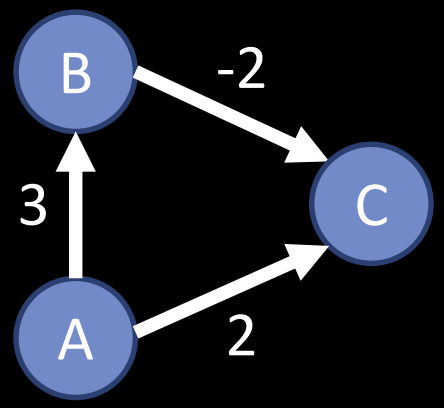
Priority Queue

A	B	C
	3	

Shortest Distances

A	B	C
0	?	2

Dijkstra Example



Priority Queue

A	B	C

Shortest Distances

A	B	C
0	3	2

Dijkstra Example

Priority Queue

A

B

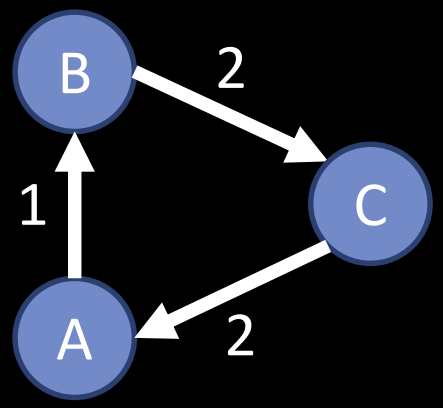
-2

Distances

A	B	C
0	3	2

Dijkstra's doesn't work with negative edge weights

Another Example



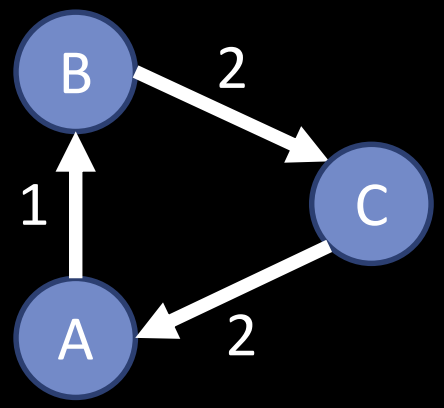
Priority Queue

A	B	C
0	∞	∞

Shortest Distances

A	B	C
?	?	?

Another Example



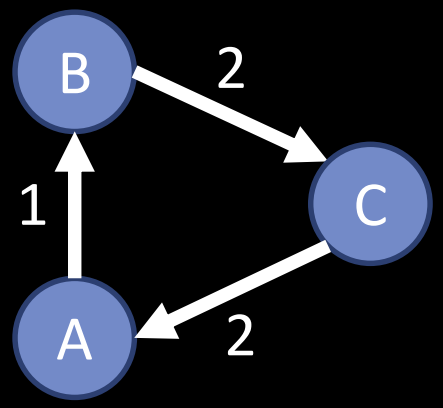
Priority Queue

A	B	C
	∞	∞

Shortest Distances

A	B	C
0	?	?

Another Example



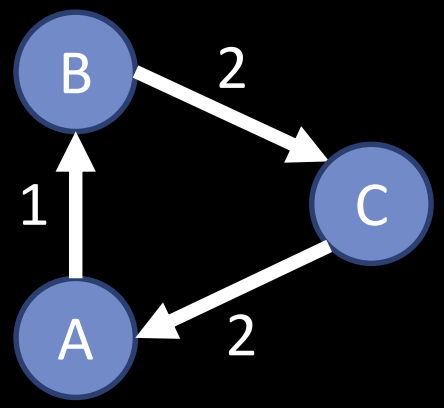
Priority Queue

A	B	C
	1	∞

Shortest Distances

A	B	C
0	?	?

Another Example



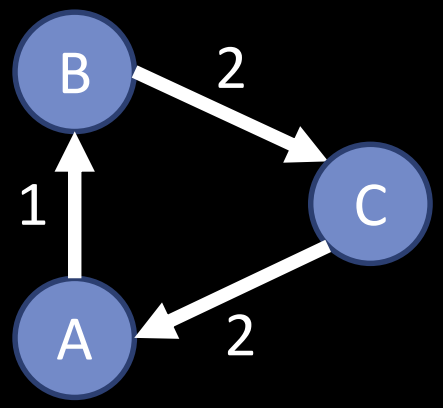
Priority Queue

A	B	C
		∞

Shortest Distances

A	B	C
0	1	?

Another Example



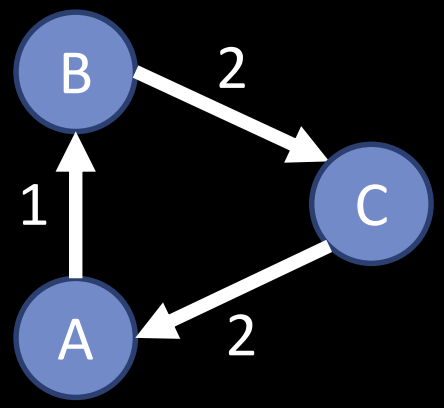
Priority Queue

A	B	C
		3

Shortest Distances

A	B	C
0	1	?

Another Example



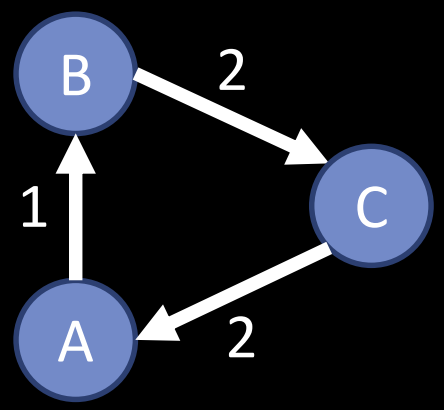
Priority Queue

A	B	C

Shortest Distances

A	B	C
0	1	3

Another Example



Priority Queue

A	B	C

Shortest Distances

A	B	C
0	1	3

Cycles are fine

Why do we care about graphs?

- Facebook, LinkedIn
- Maps (Google Maps, GPS)
- The internet
- Puzzle configurations (Rubik's cube, project 3)

Final Exam Information

- Next Wednesday (8/13), 3-6pm
- Last name A-H, Q-Z: 2050 VLSB
- Last name J-P: 100 GPB
- All lab and quiz material up to and including this Friday's (balanced search trees)
- Review Session
  - Sunday (8/10), 1-3pm in 100 GPB

Sorting

- Given a collection of elements, sort them from smallest to largest

Terminology: Stable

- Elements with the same value are ordered the same before and after sorting

Original: 1 2 3 1 2 3 3 4

Sorted (stable): 1 1 2 2 3 3 3 4

Sorted (unstable): 1 1 2 2 3 3 3 4

Terminology: In-place

- Only uses O(1) additional space
  - E.g. doesn't copy all contents of input array into a new array
- Used to describe any algorithm, not just sorting algorithms

Selection Sort

for `i = 0, 1, ... , (n-1)`:

Search positions `i` to `(n-1)` for smallest element

Swap with element at position `i`

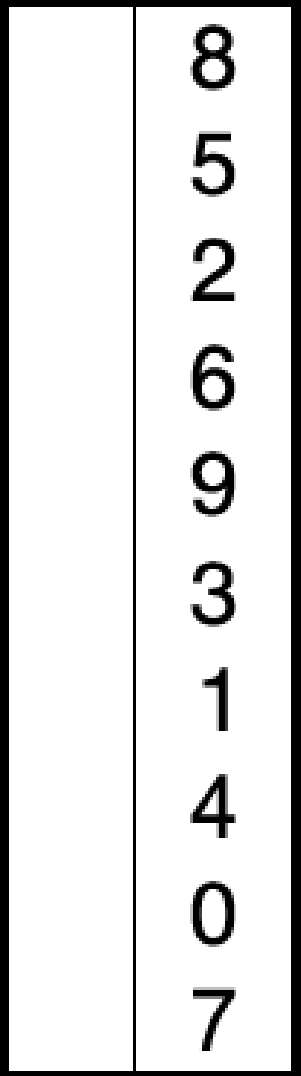
- Stable, in-place
- Best / average / worst case running time: $O(n^2)$ for a list with n elements

Selection Sort Example

Current index

Smallest found this iteration

Already sorted



http://en.wikipedia.org/wiki/Selection_sort#mediaviewer/File:Selection-Sort-Animation.gif

Bubble Sort

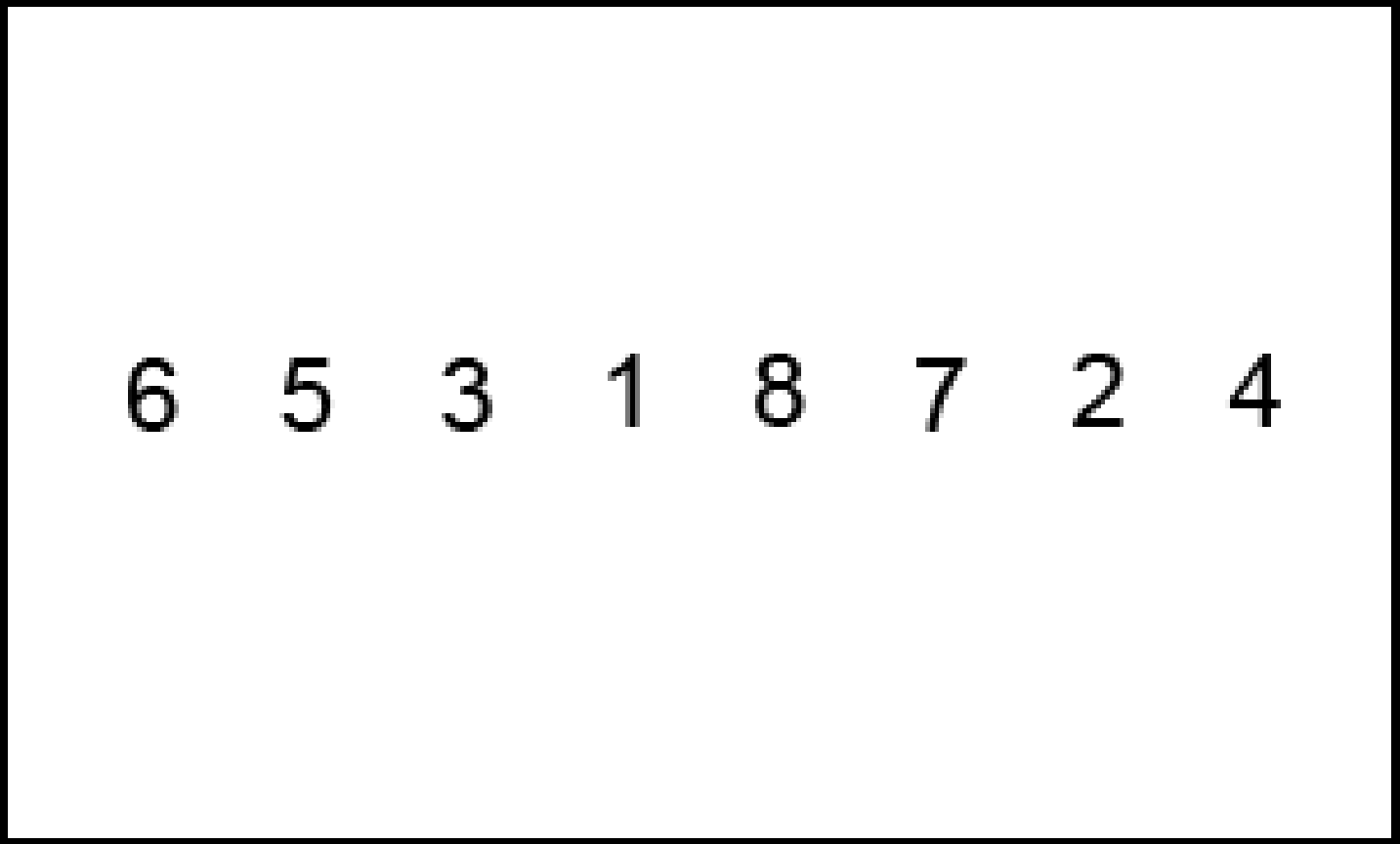
for `i = 0, 1, ... , (n-1)`:

Go through the first `n-i` elements

Swap pairs of adjacent elements that are out of order

- Stable, in-place
- Average / worst case running time: $O(n^2)$ for a list with n elements
- Best case running time: $O(n)$ with optimization

Bubble Sort Example



http://en.wikipedia.org/wiki/Bubble_sort#mediaviewer/File:Bubble-sort-example-300px.gif

Insertion Sort

for `i = 0, 1, ... , (n-1)`:

Take element at index `i`

Insert it into the correct location in the sorted sublist consisting of the first `i+1` elements

- Stable, in-place
- Average / worst case running time: $O(n^2)$ for a list with n elements
- Best case running time: $O(n)$

Insertion Sort Example

6 5 3 1 8 7 2 4

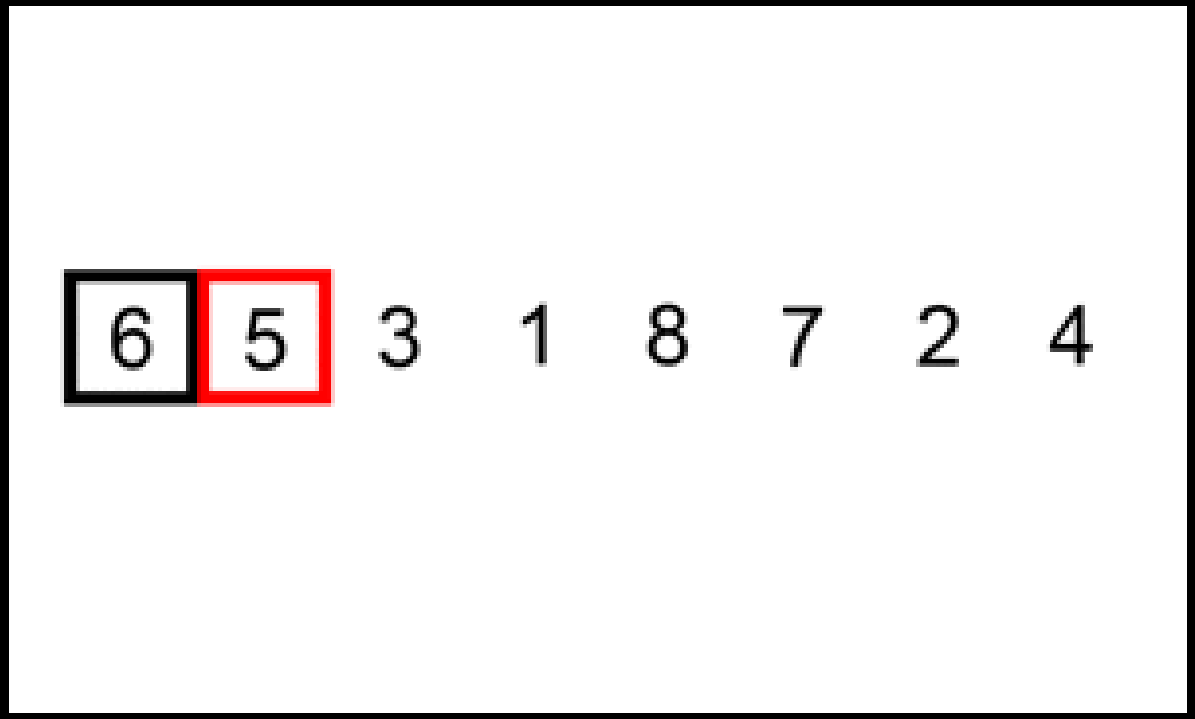
Insertion Sort Example

6 5 3 1 8 7 2 4

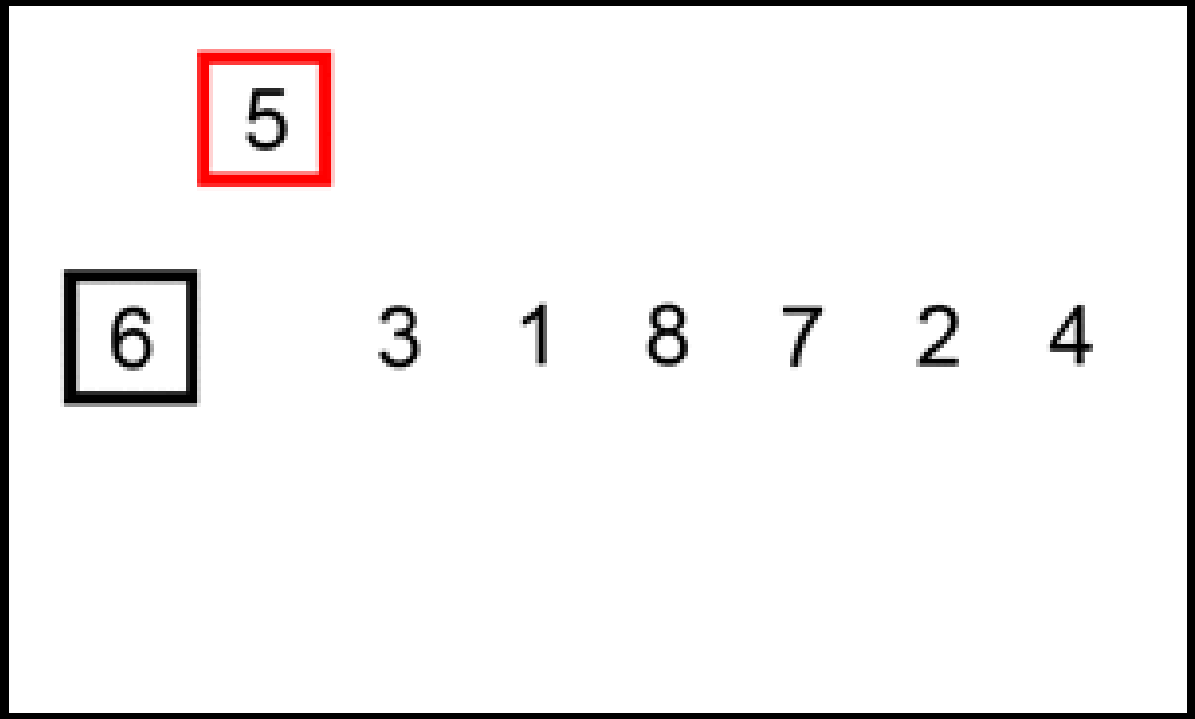
Insertion Sort Example

6	5	3	1	8	7	2	4

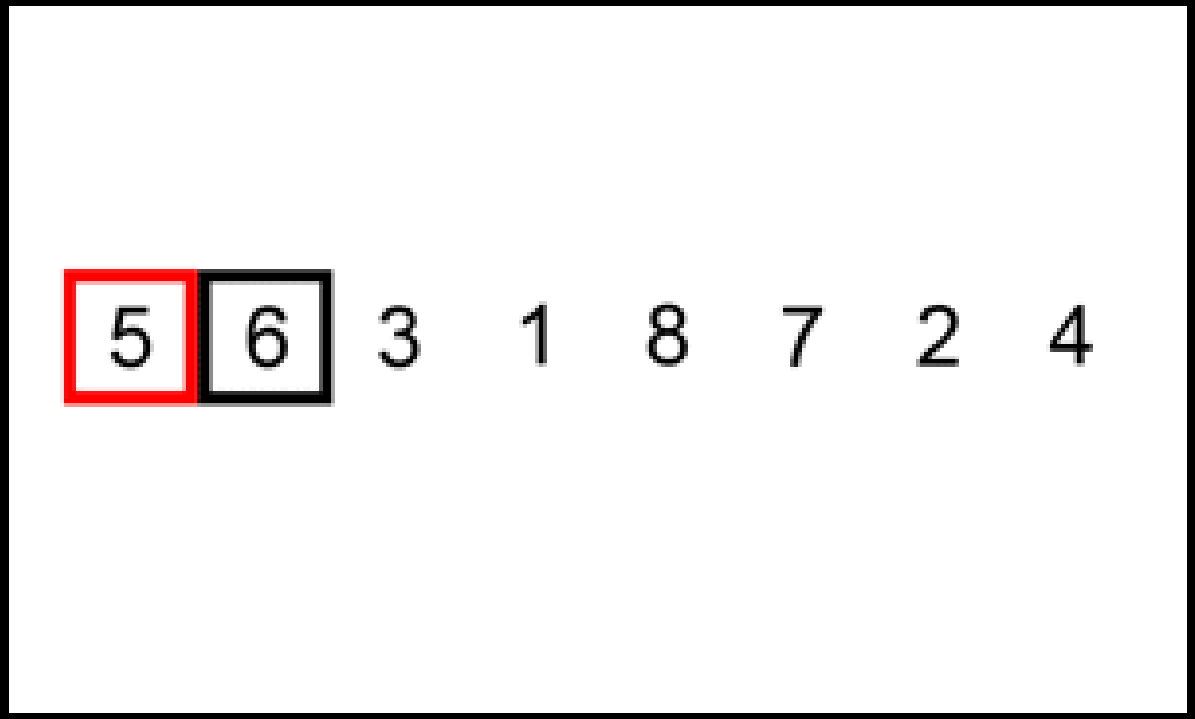
Insertion Sort Example



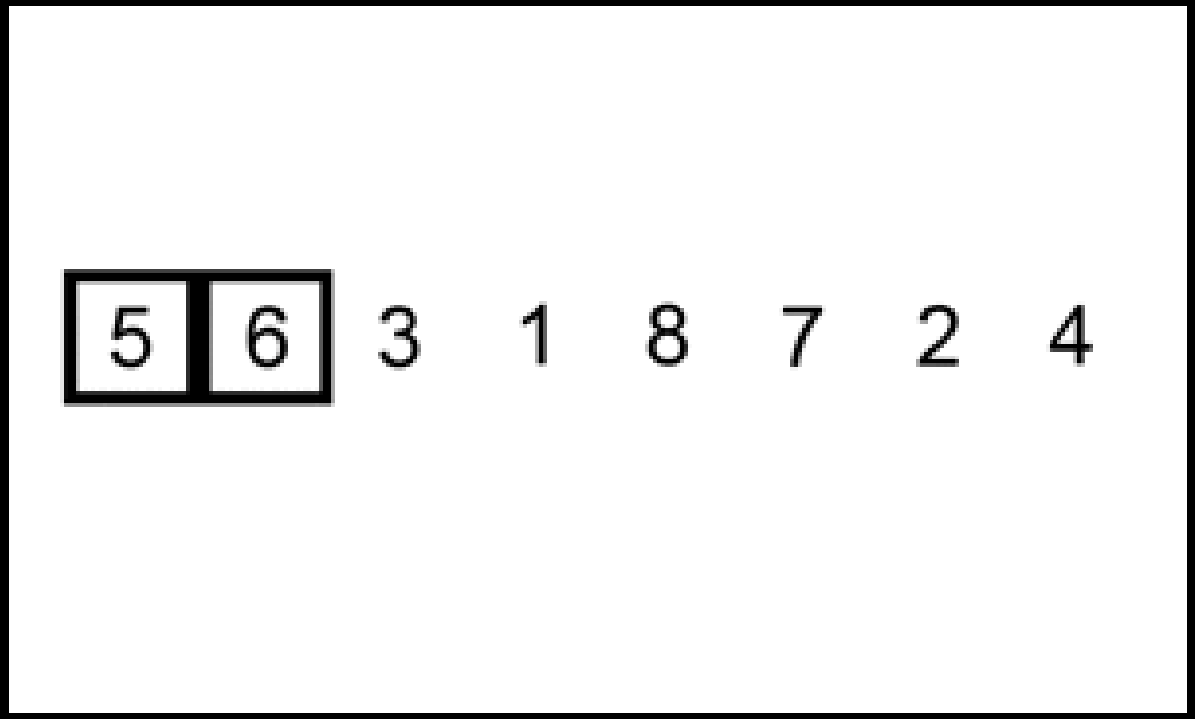
Insertion Sort Example



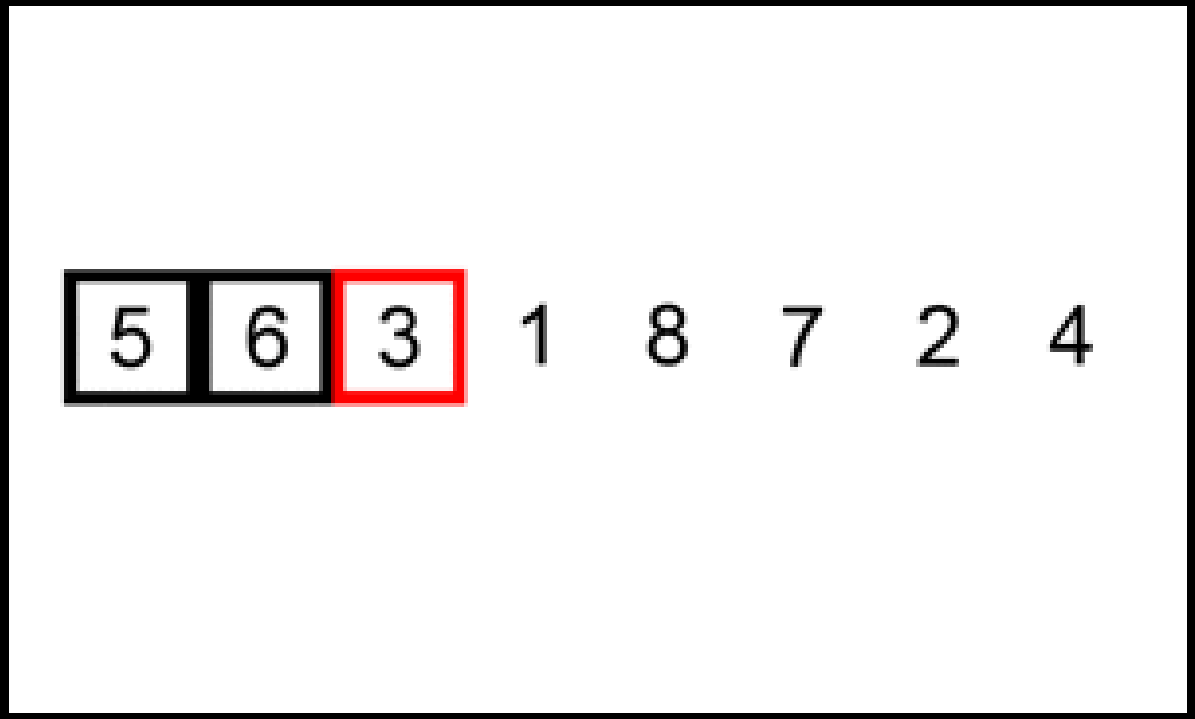
Insertion Sort Example



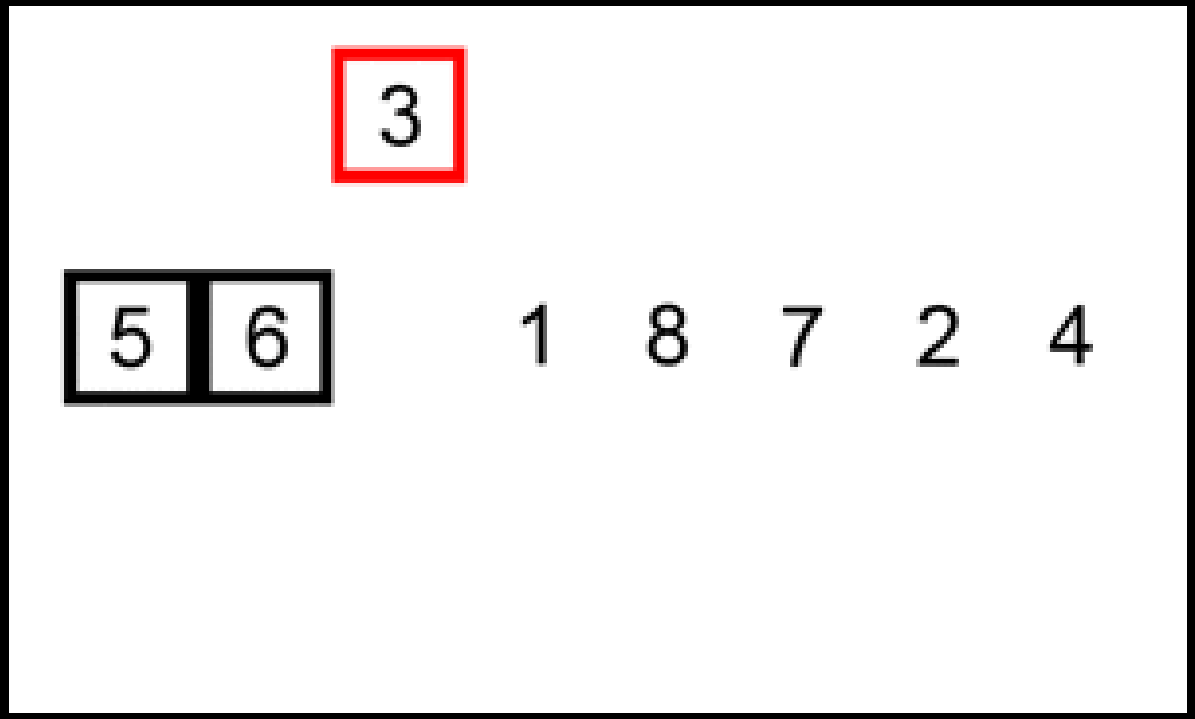
Insertion Sort Example



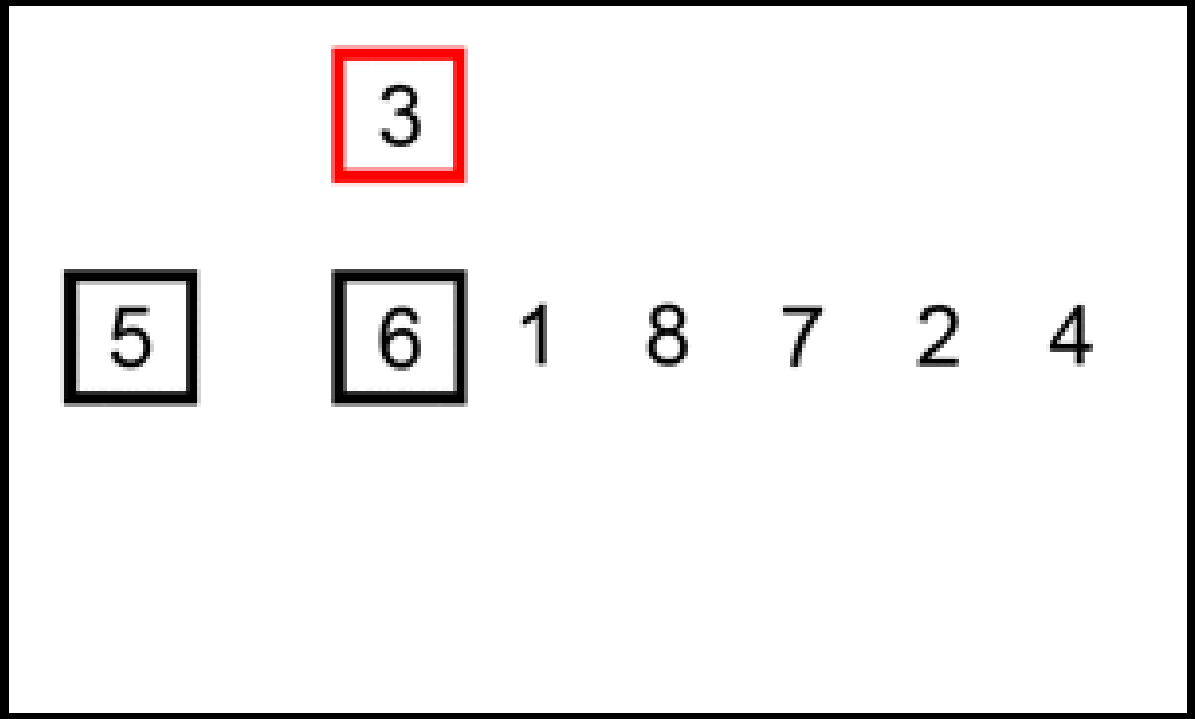
Insertion Sort Example



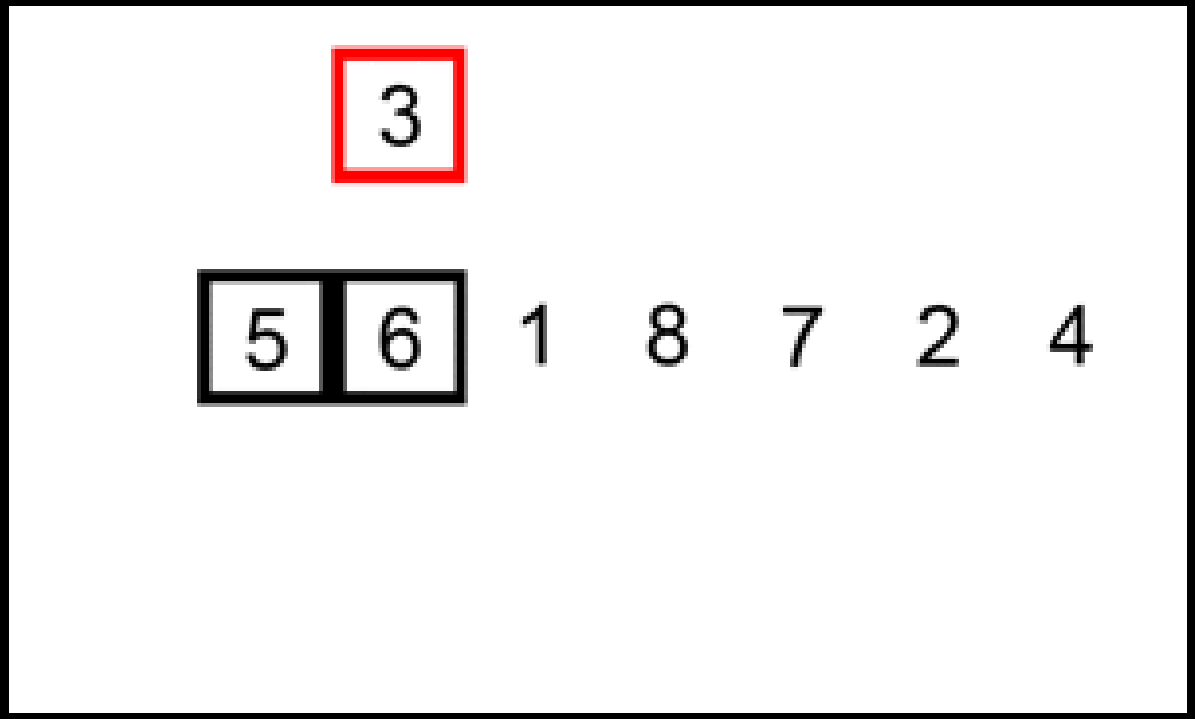
Insertion Sort Example



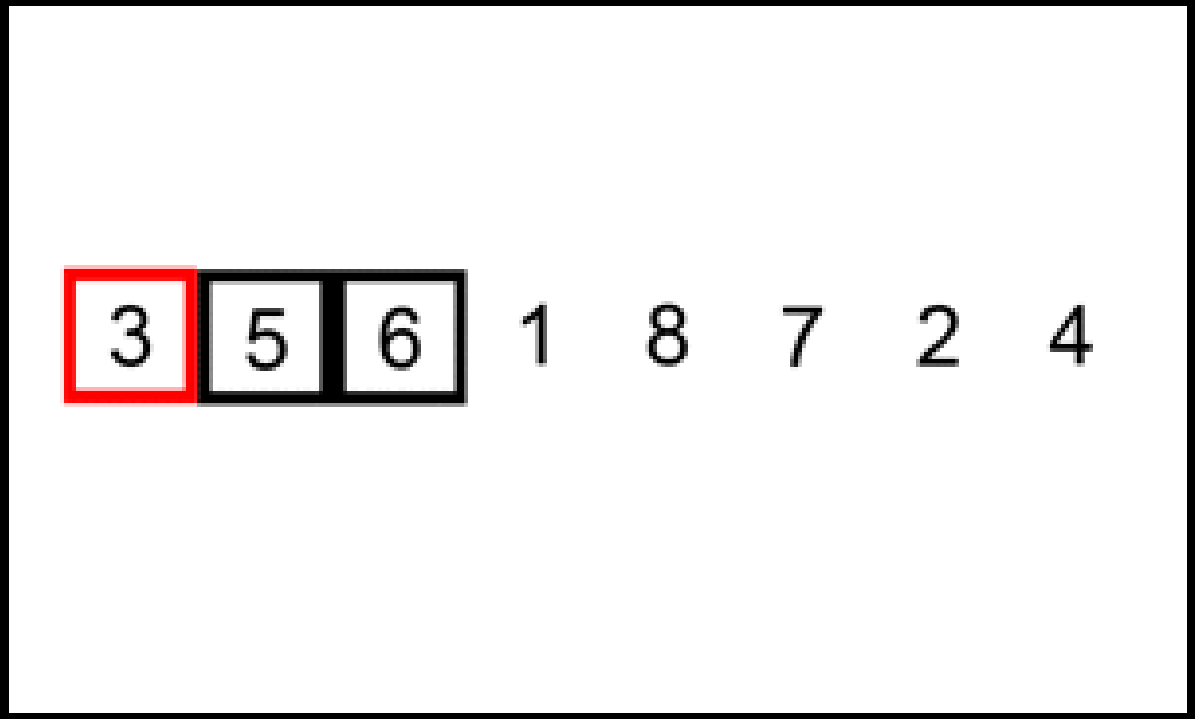
Insertion Sort Example



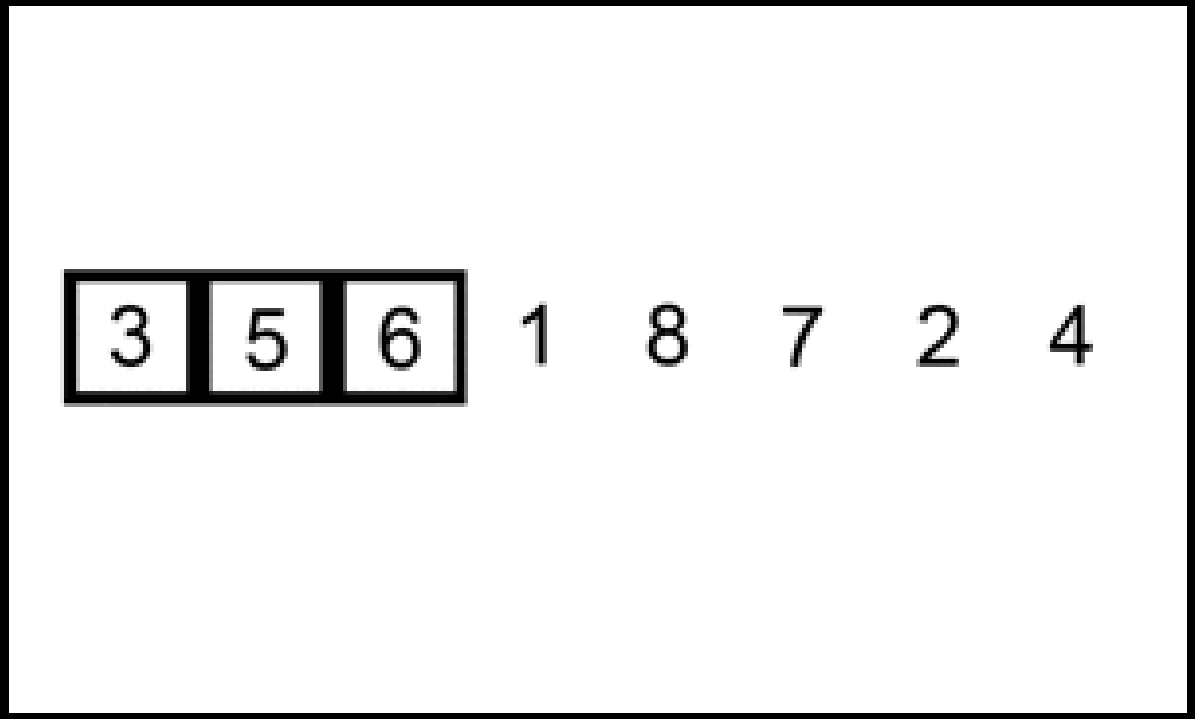
Insertion Sort Example



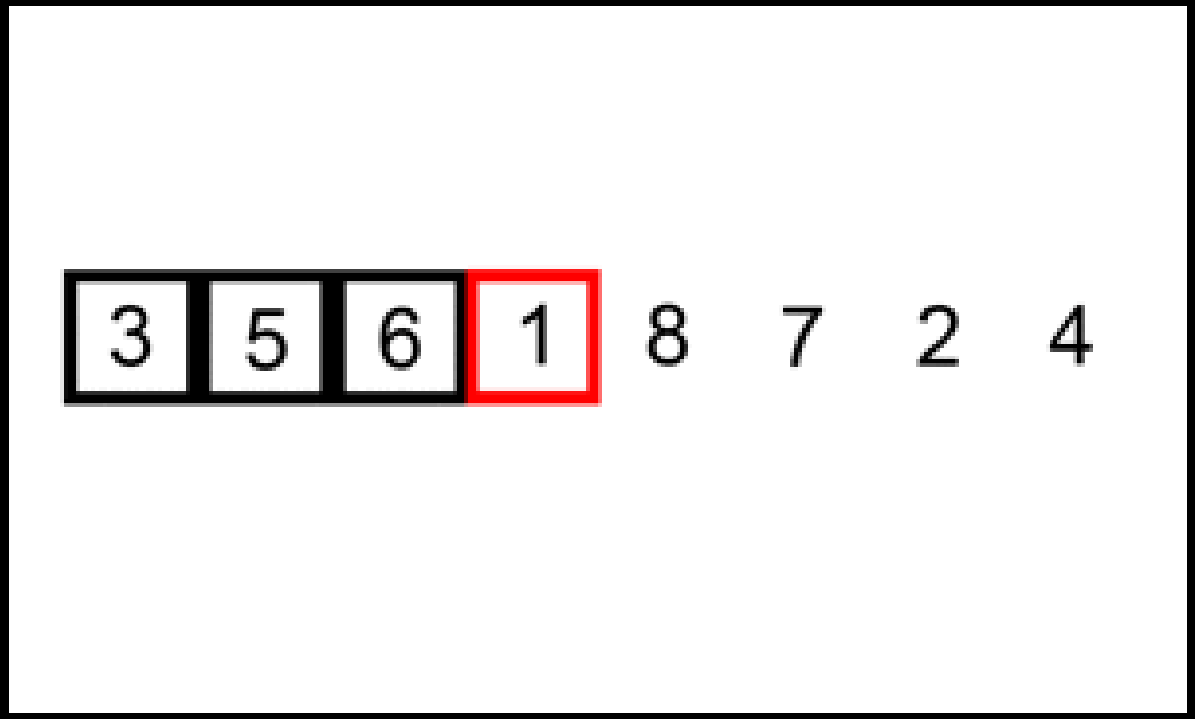
Insertion Sort Example



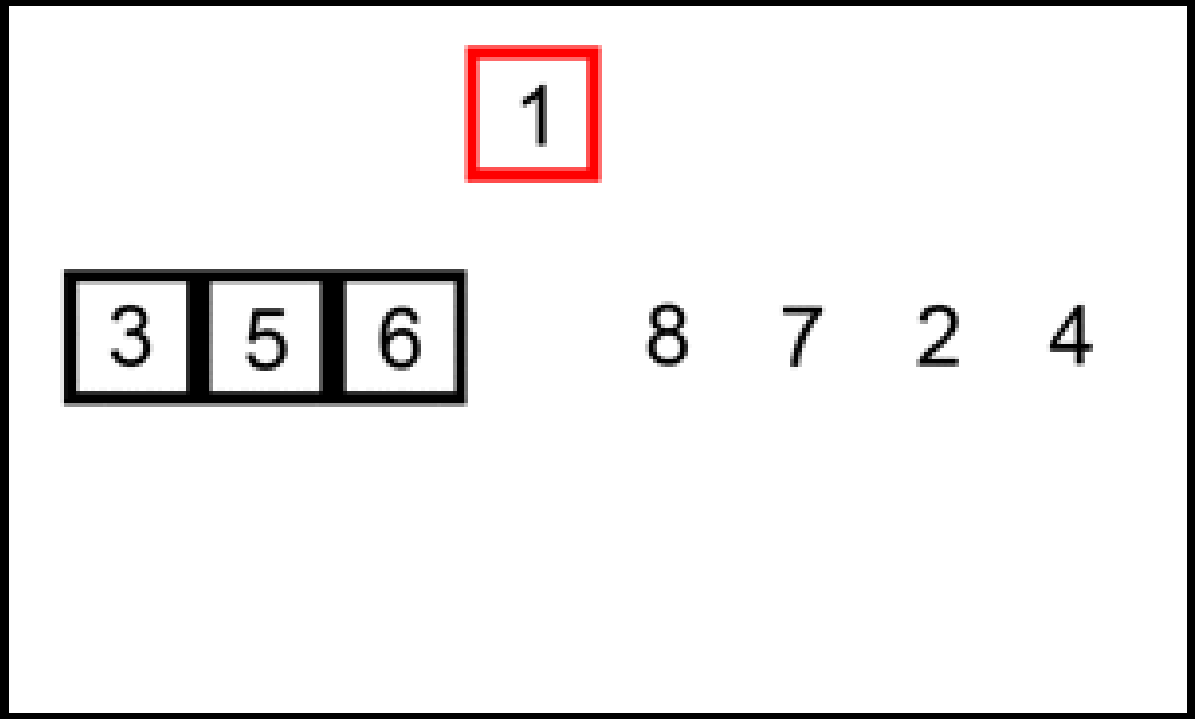
Insertion Sort Example



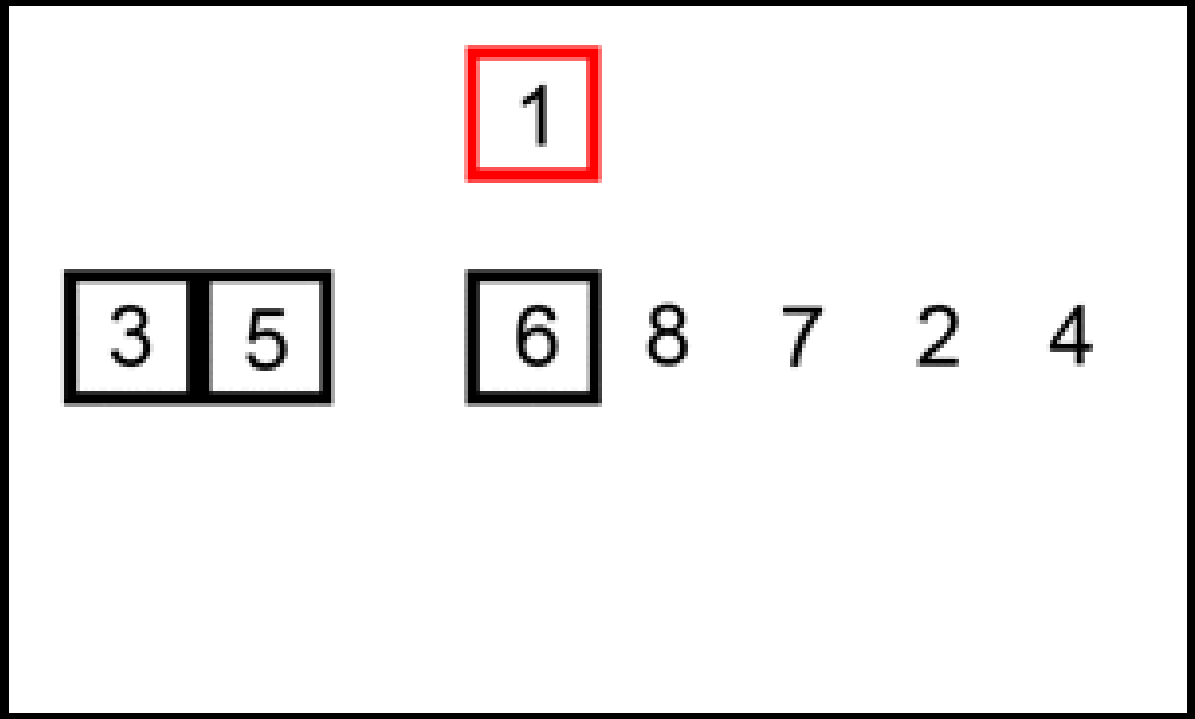
Insertion Sort Example



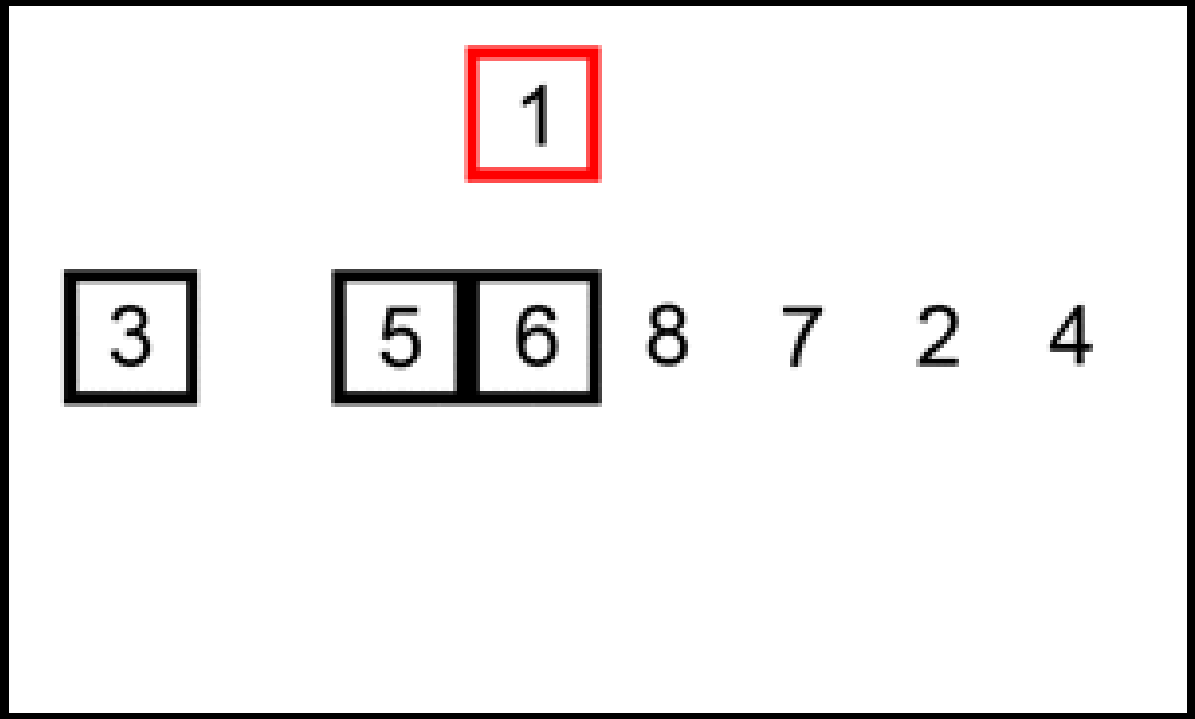
Insertion Sort Example



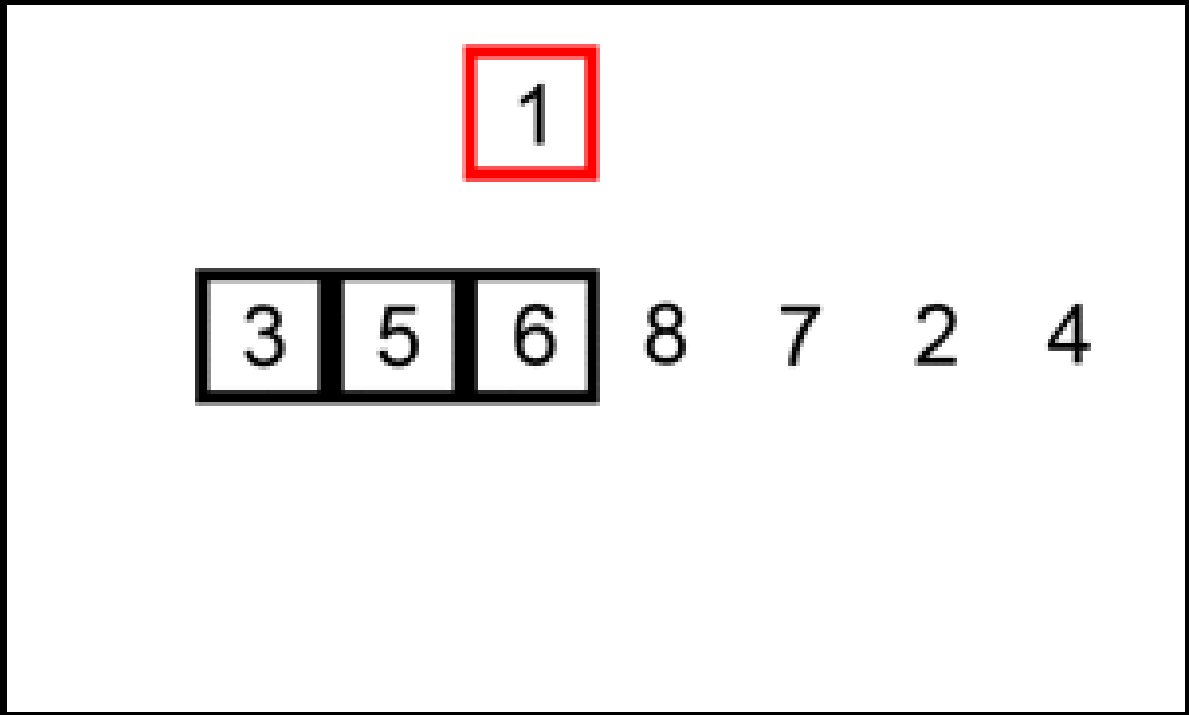
Insertion Sort Example



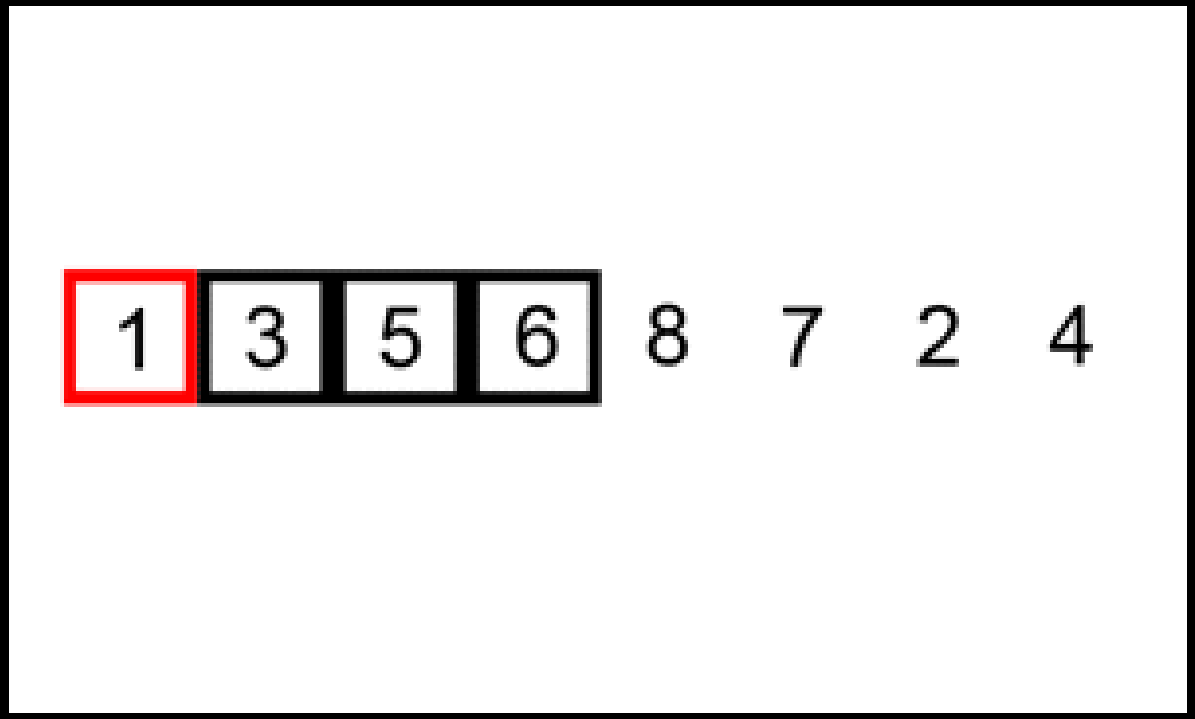
Insertion Sort Example



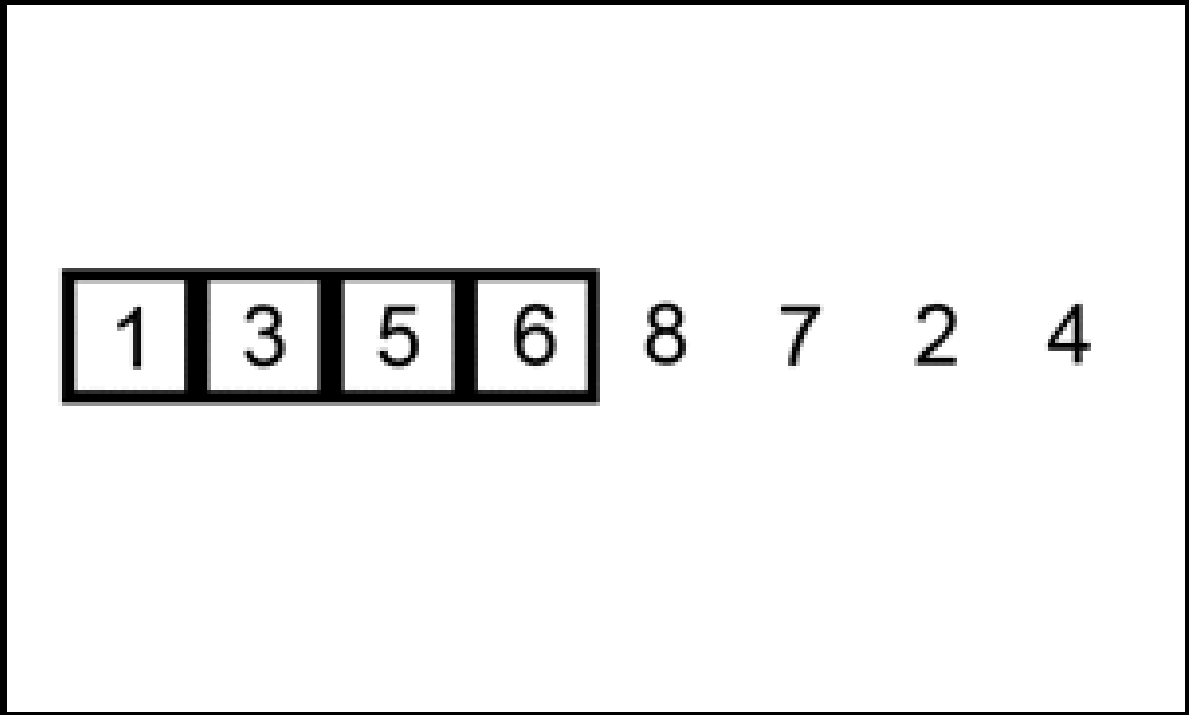
Insertion Sort Example



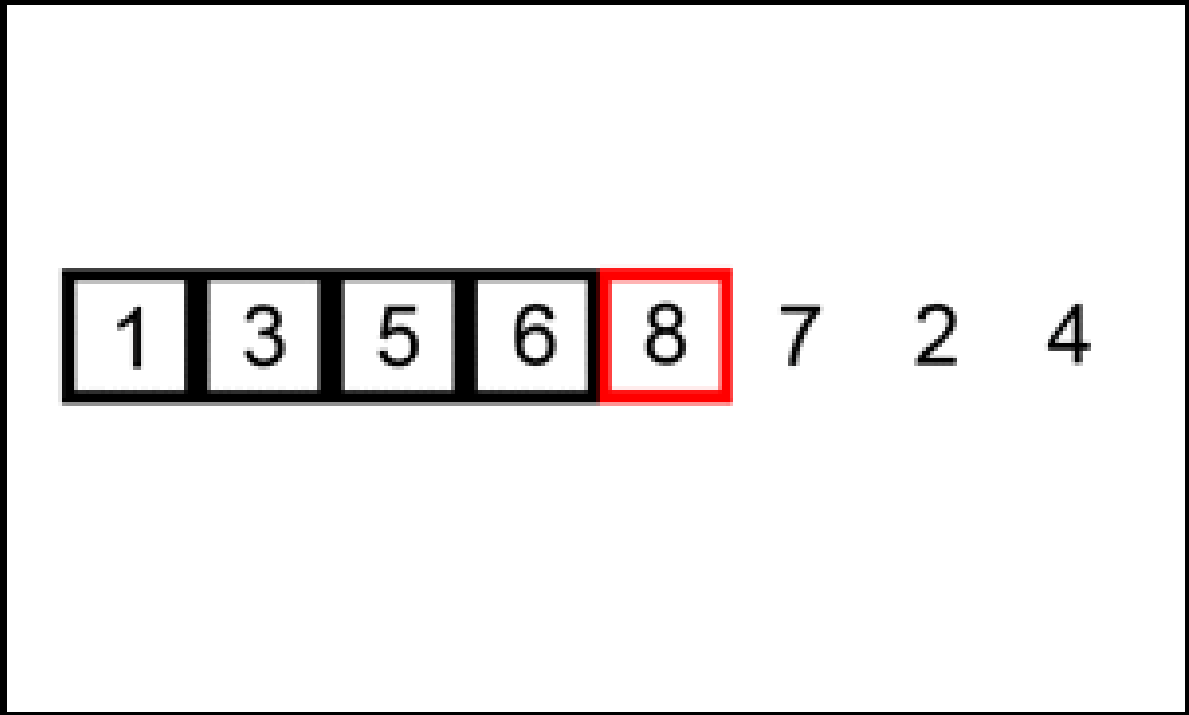
Insertion Sort Example



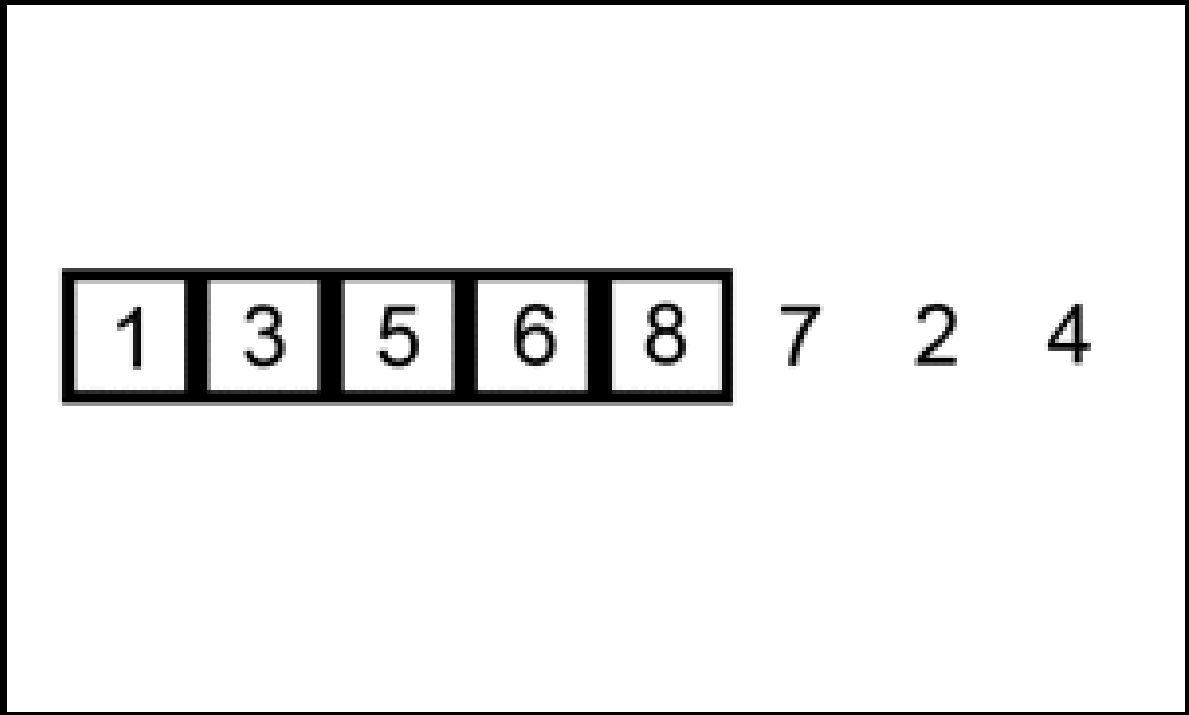
Insertion Sort Example



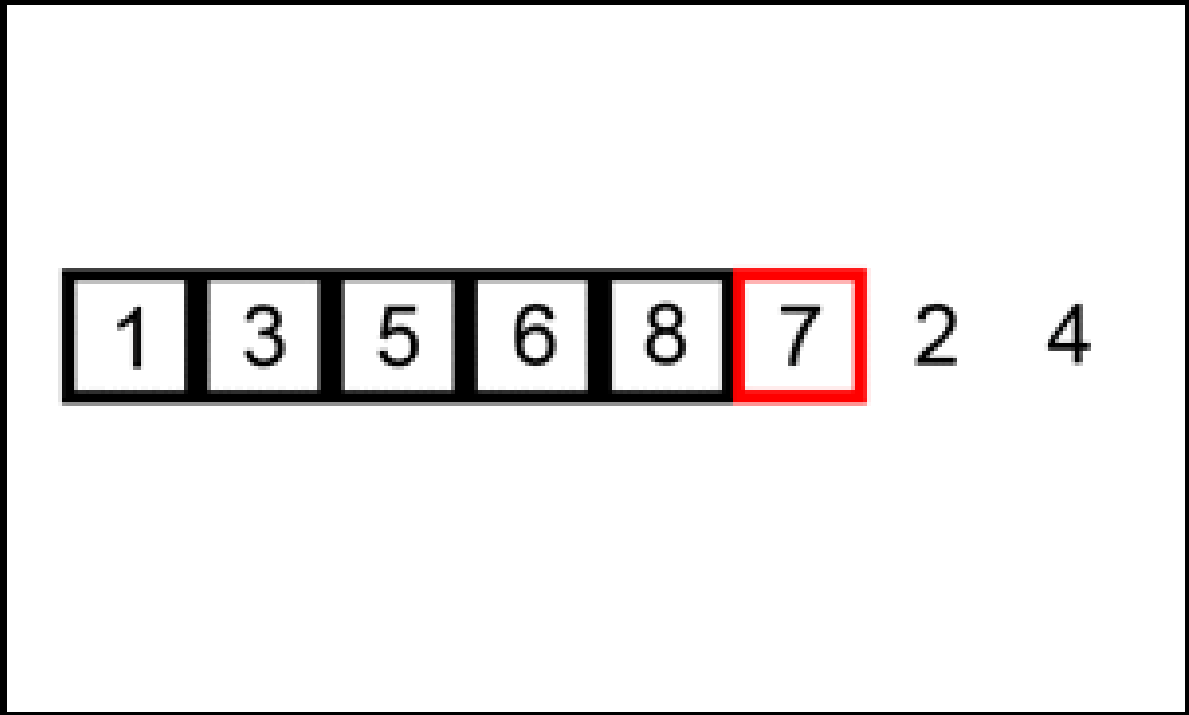
Insertion Sort Example



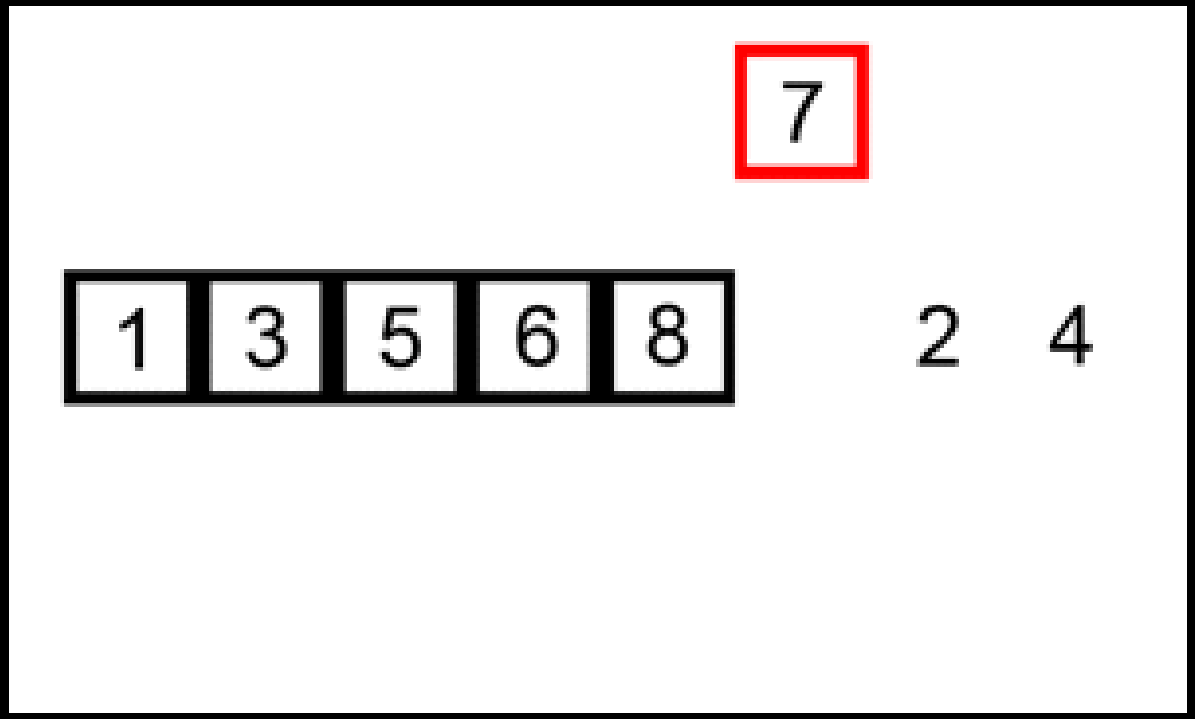
Insertion Sort Example



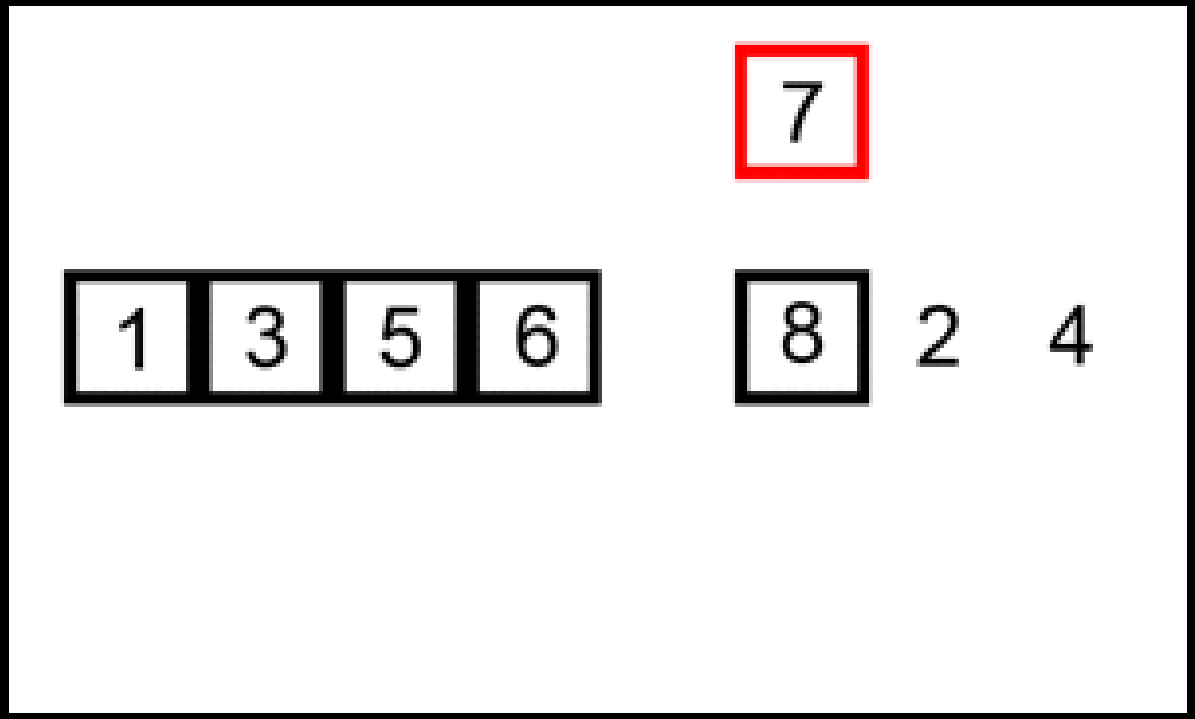
Insertion Sort Example



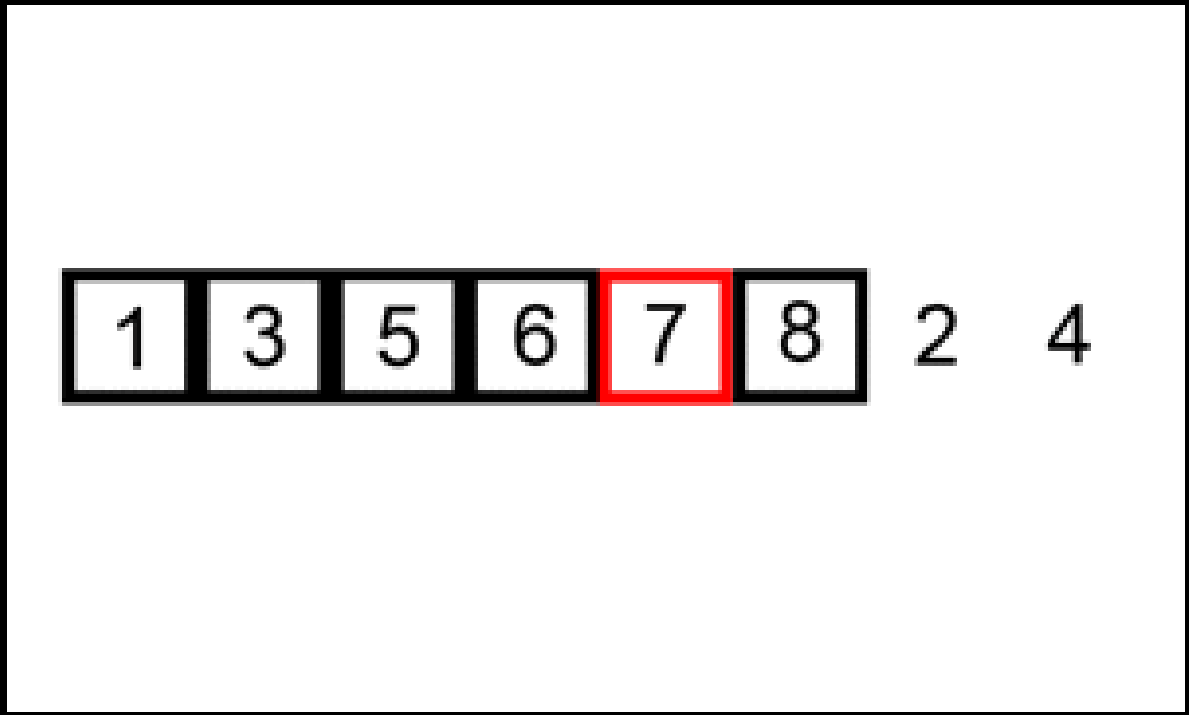
Insertion Sort Example



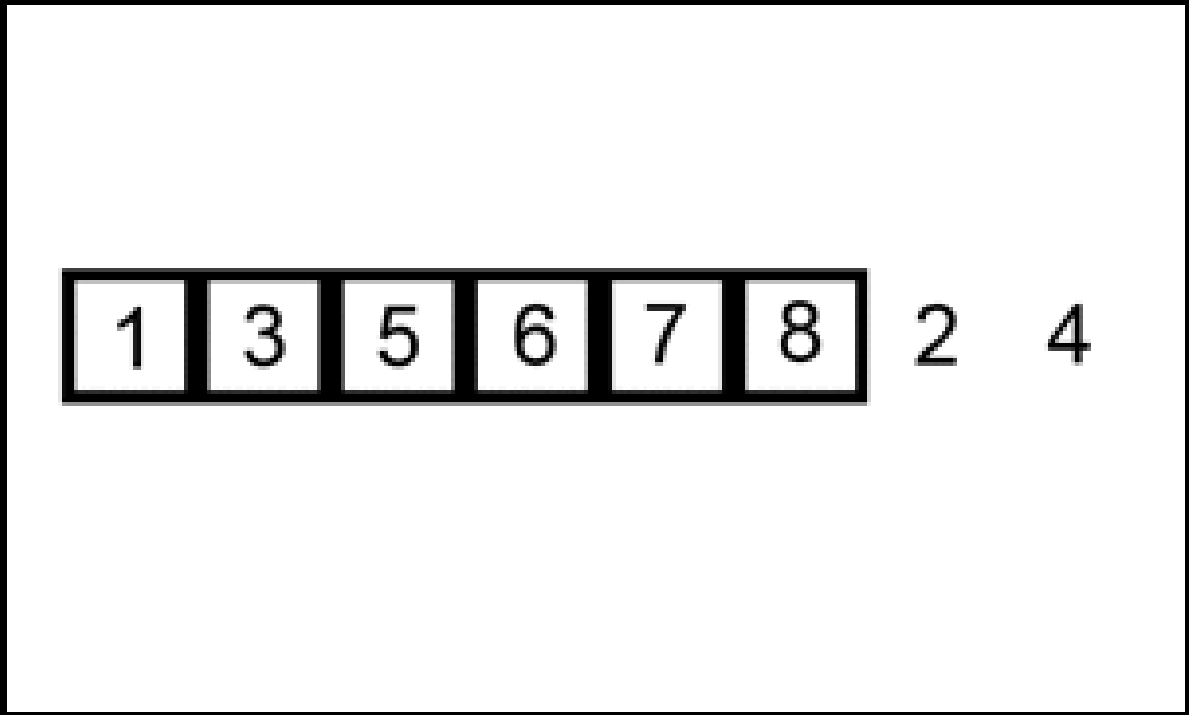
Insertion Sort Example



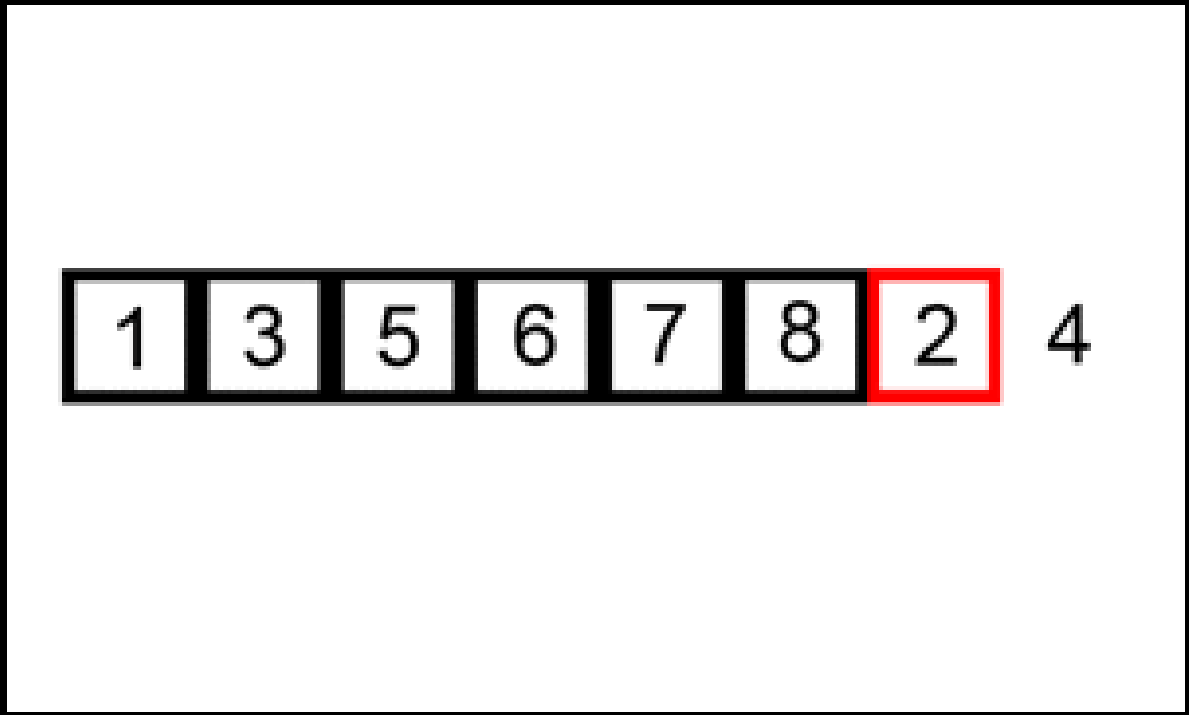
Insertion Sort Example



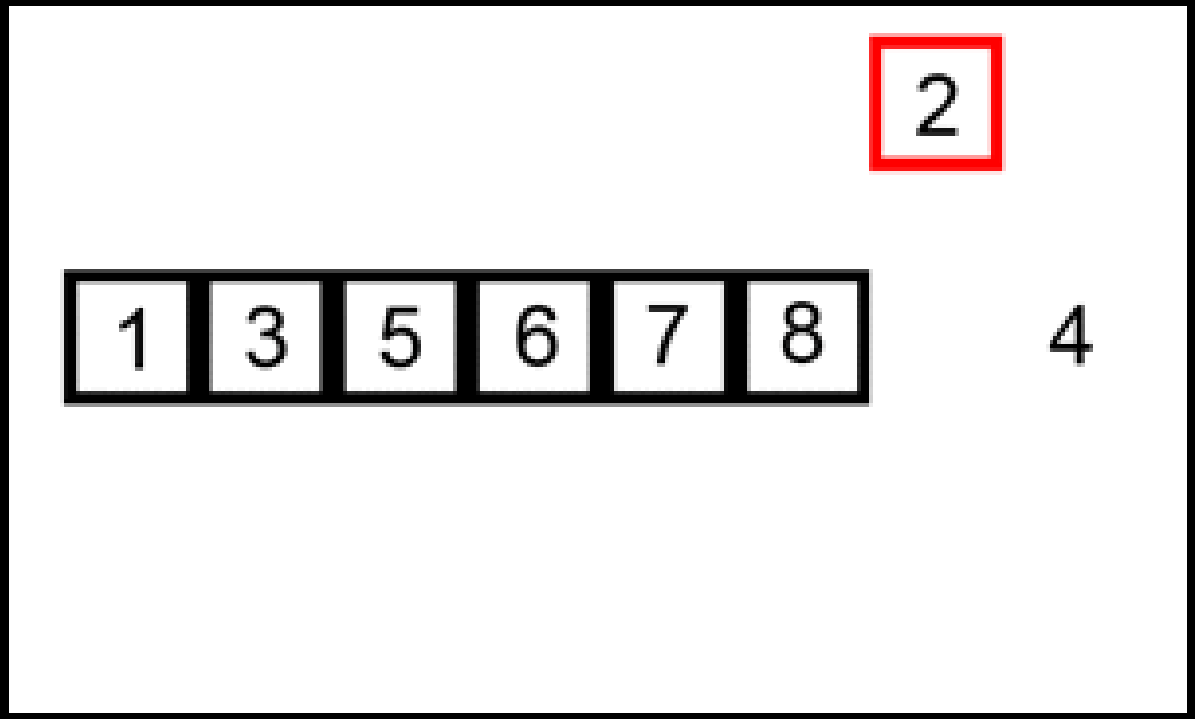
Insertion Sort Example



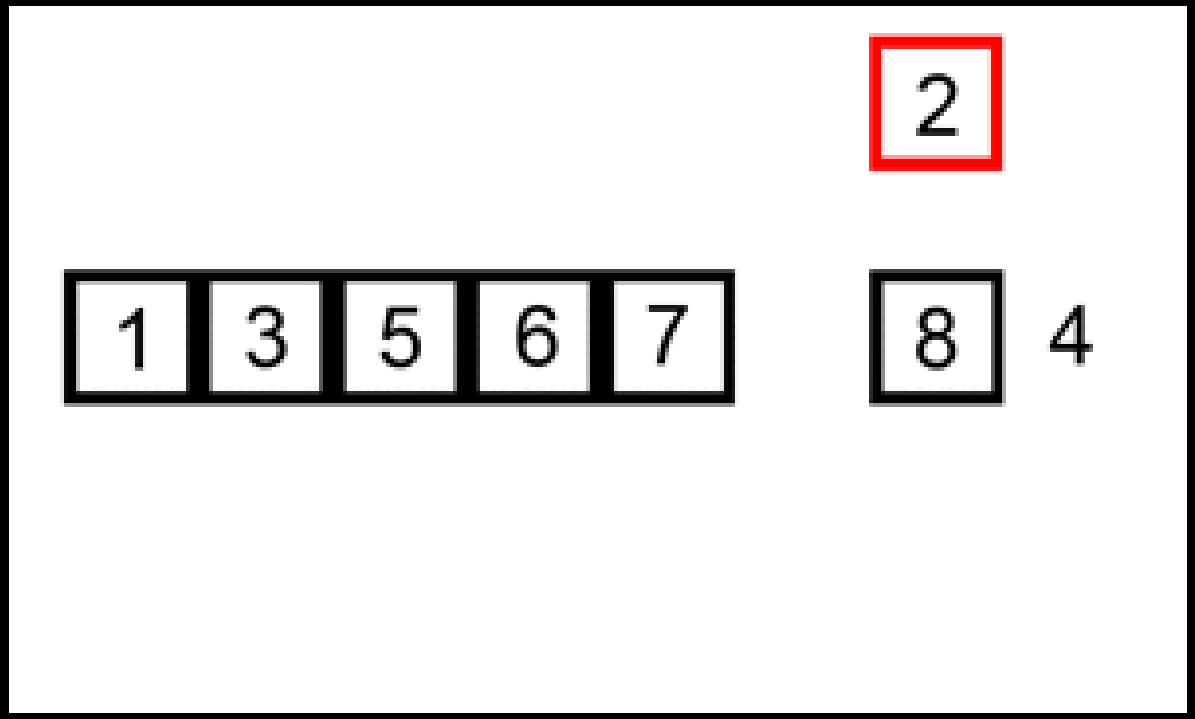
Insertion Sort Example



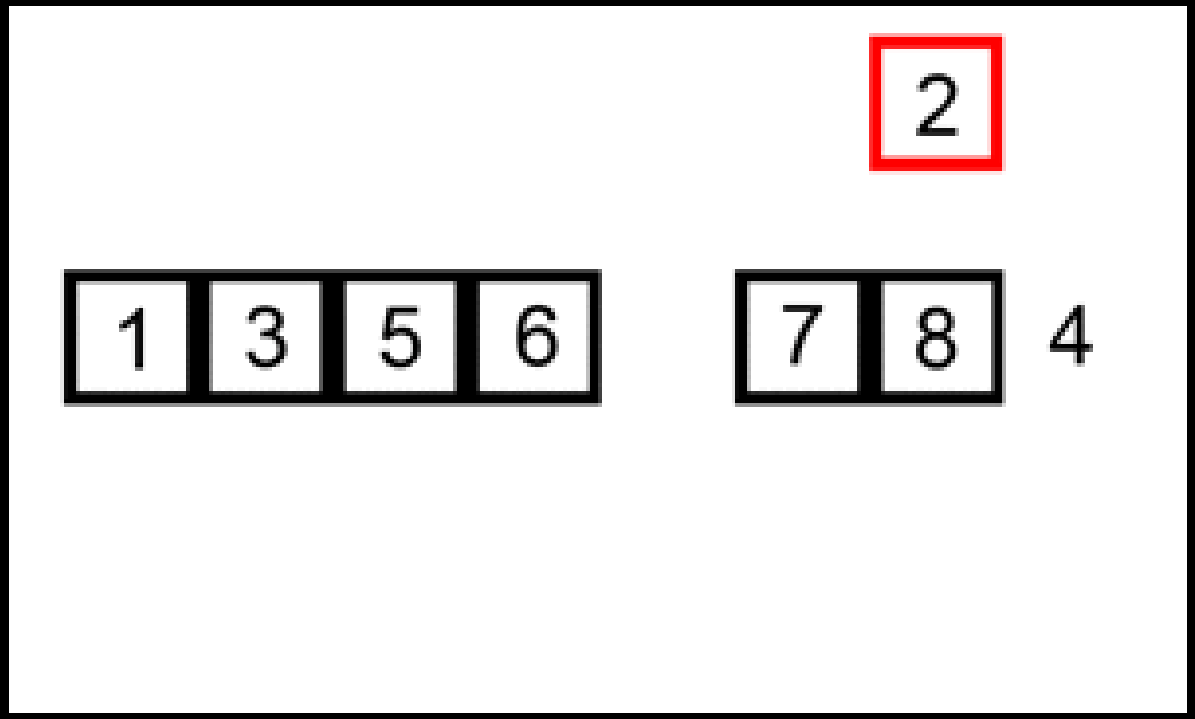
Insertion Sort Example



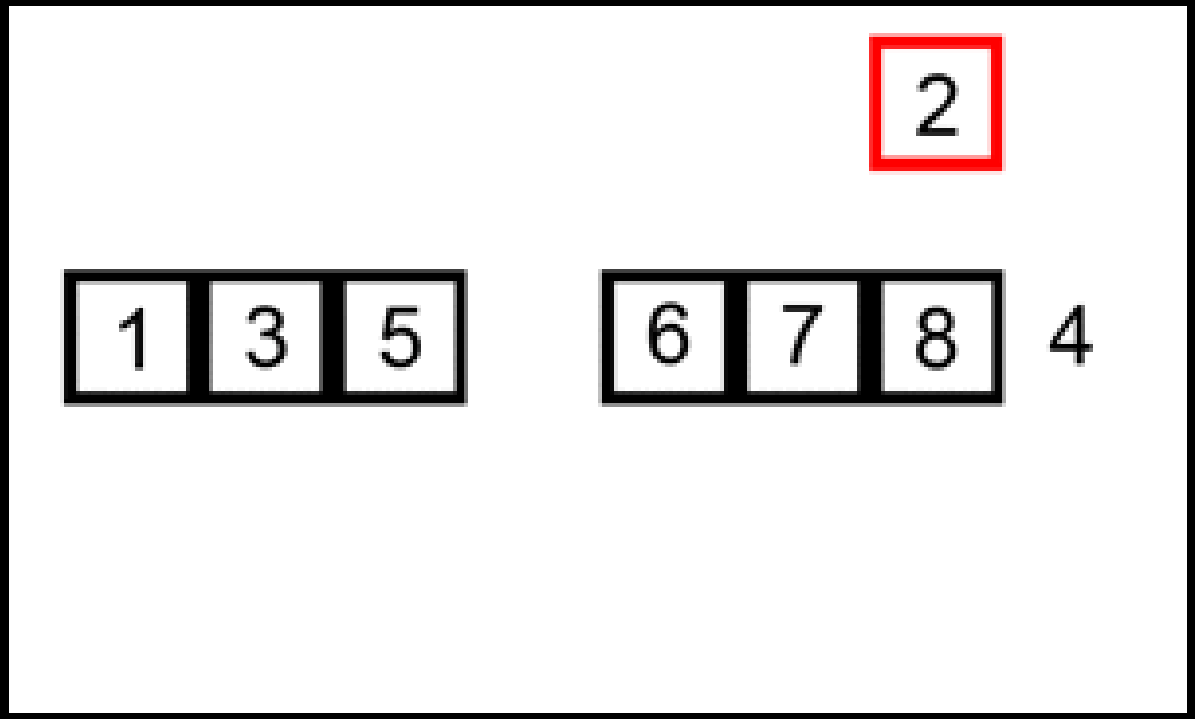
Insertion Sort Example



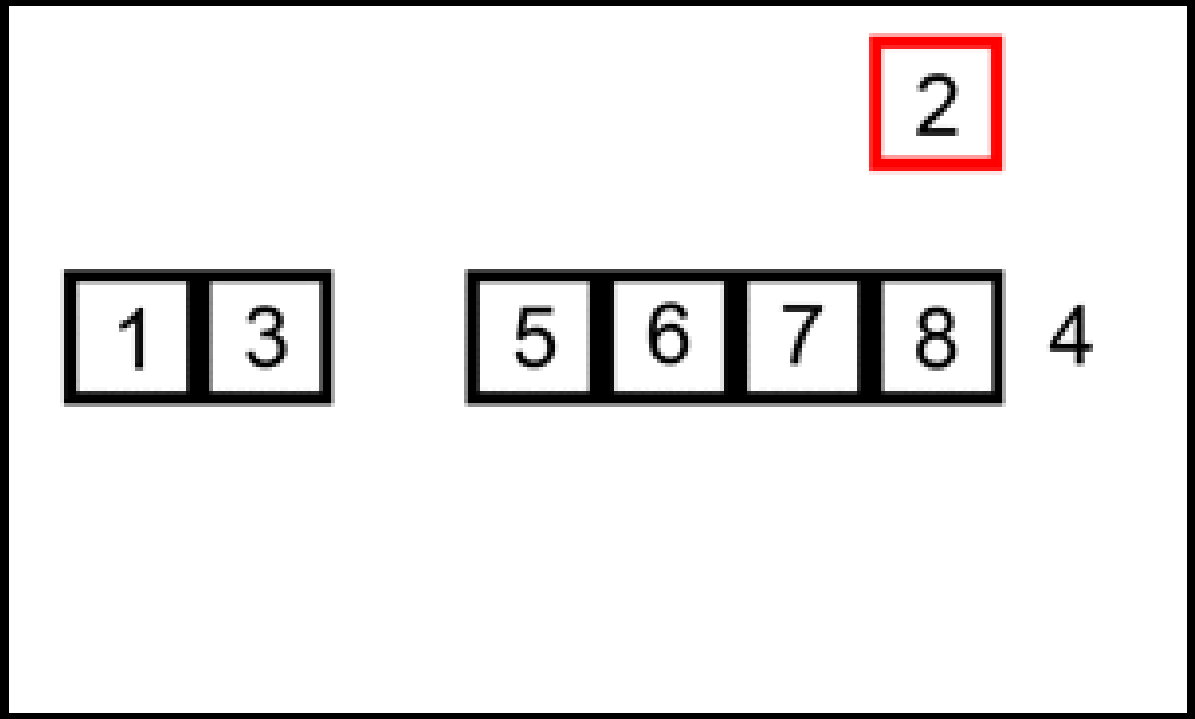
Insertion Sort Example



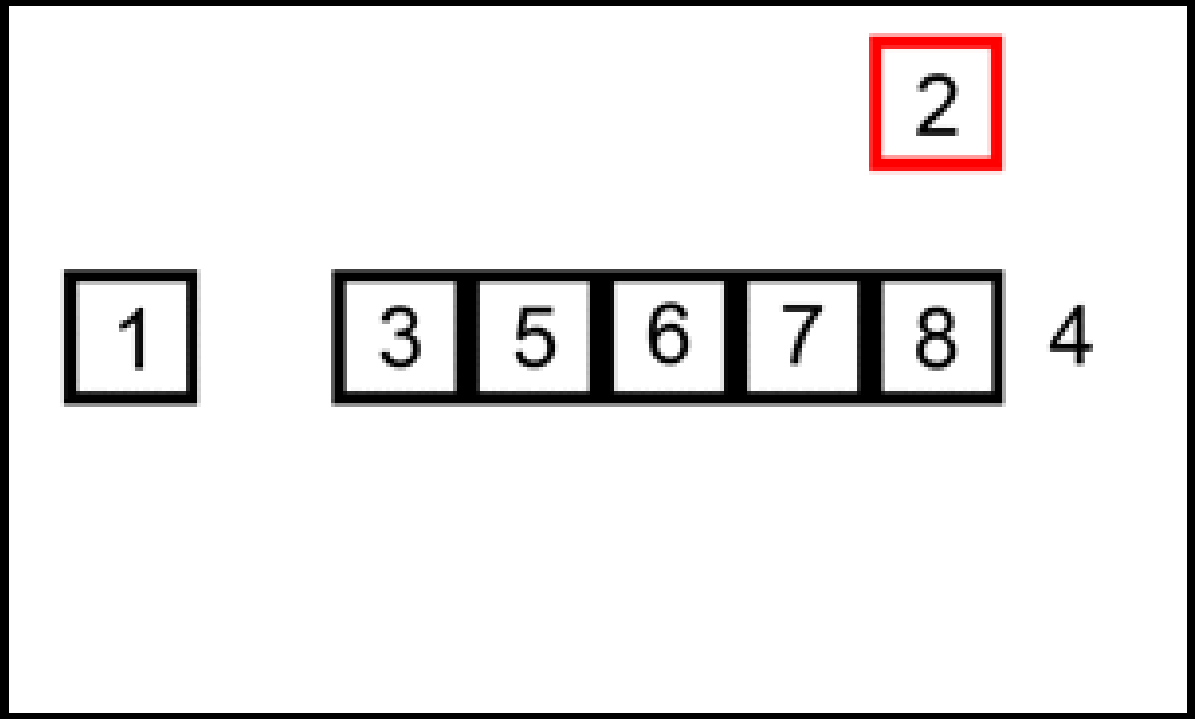
Insertion Sort Example



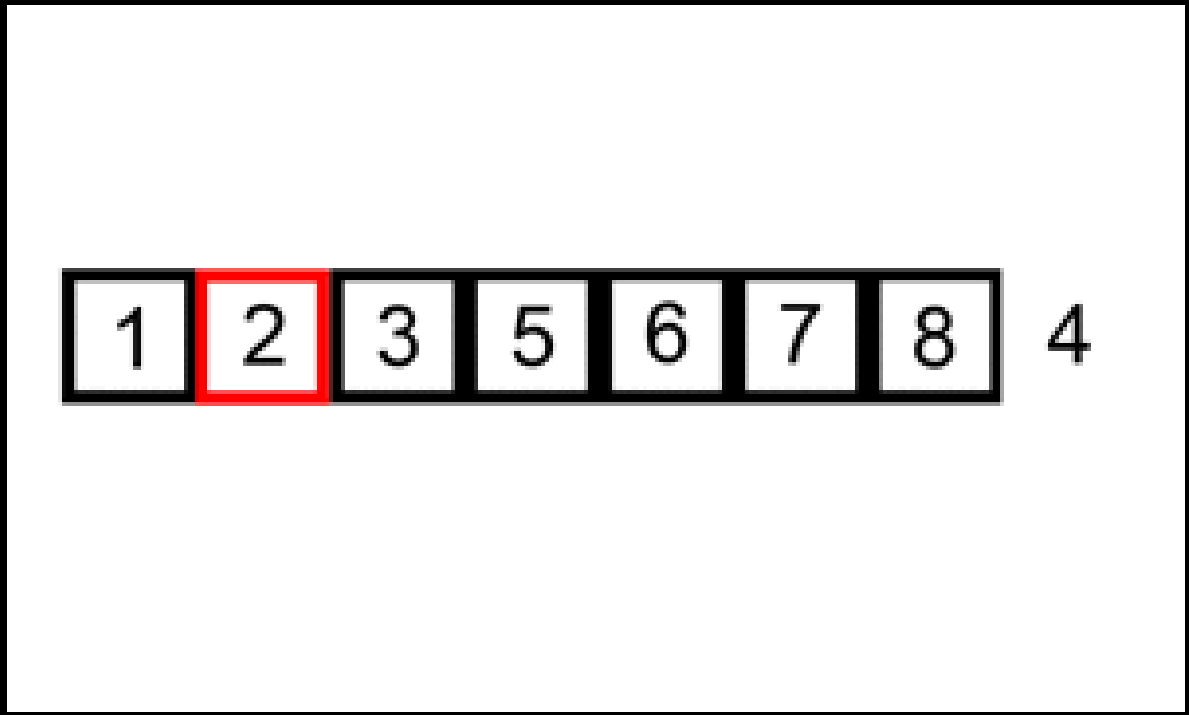
Insertion Sort Example



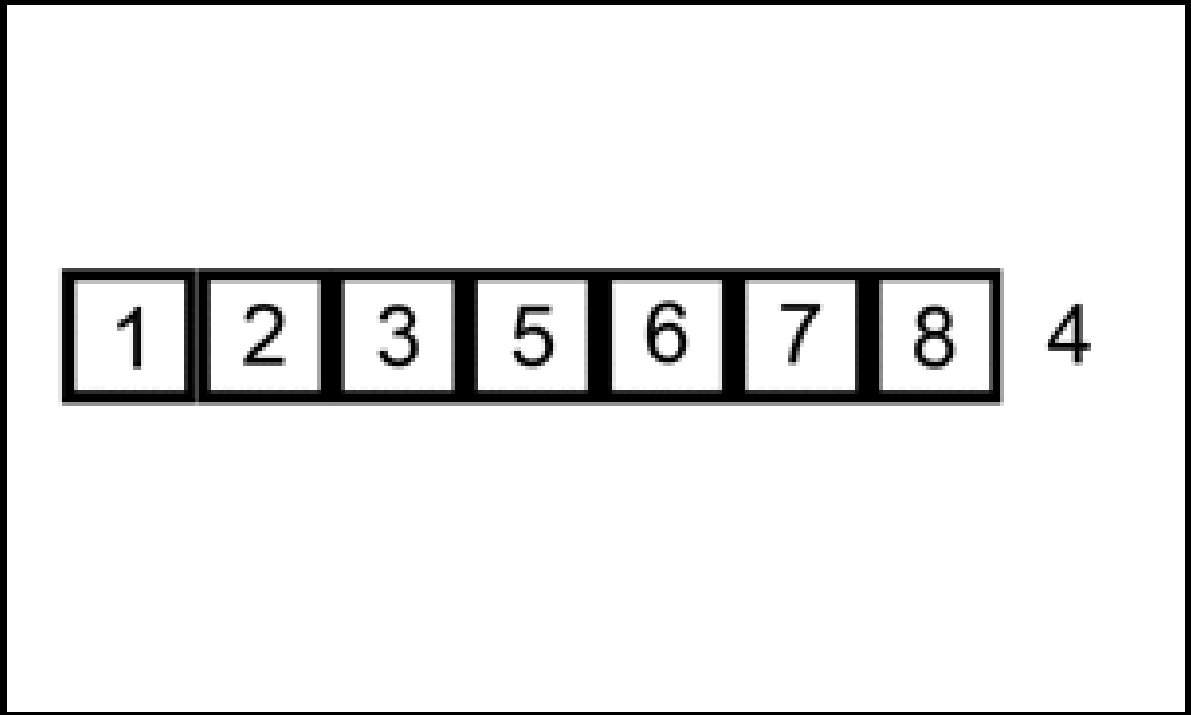
Insertion Sort Example



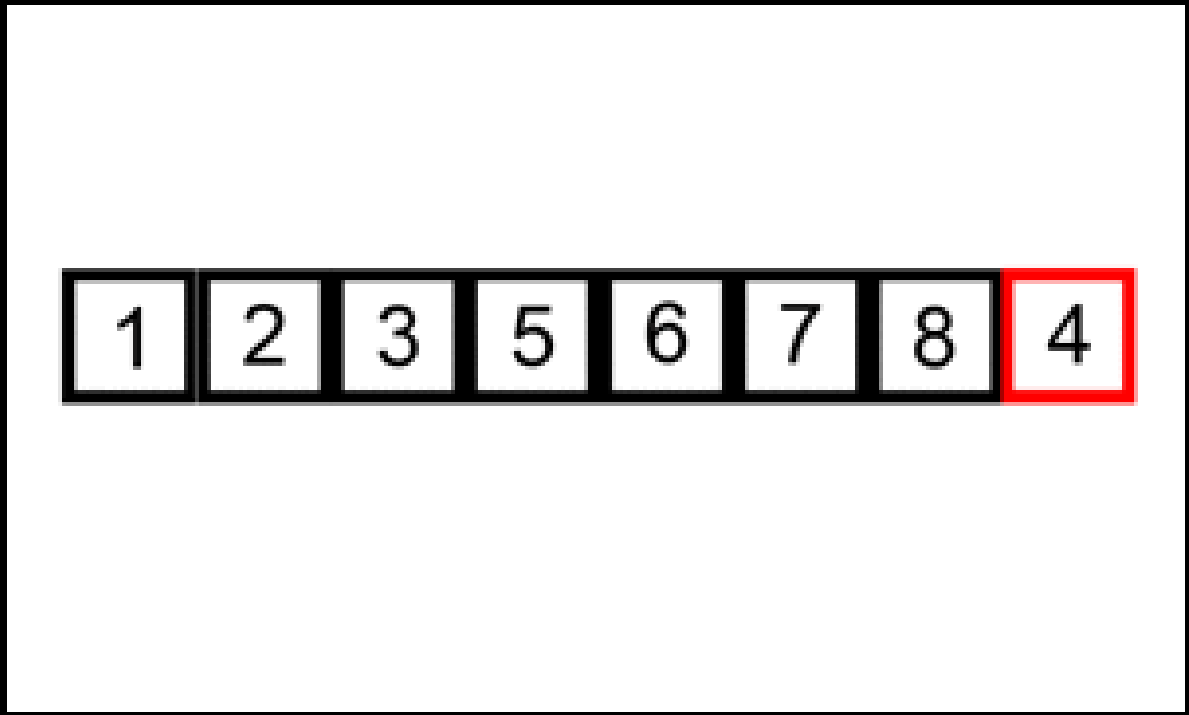
Insertion Sort Example



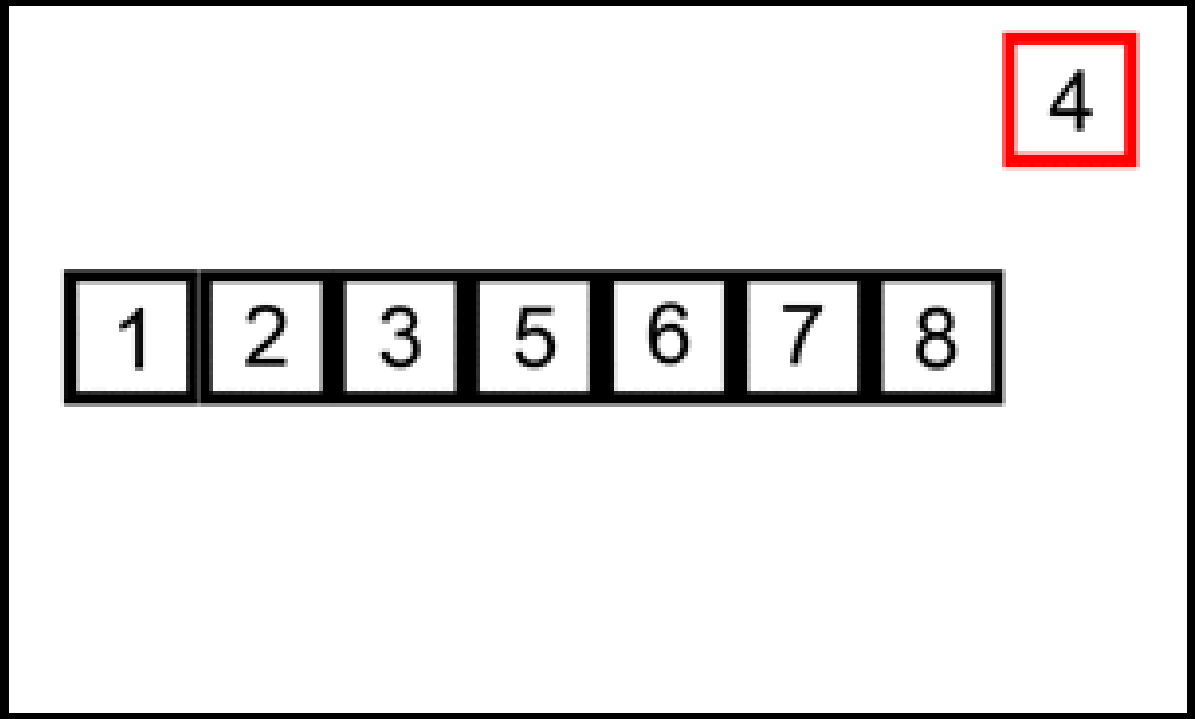
Insertion Sort Example



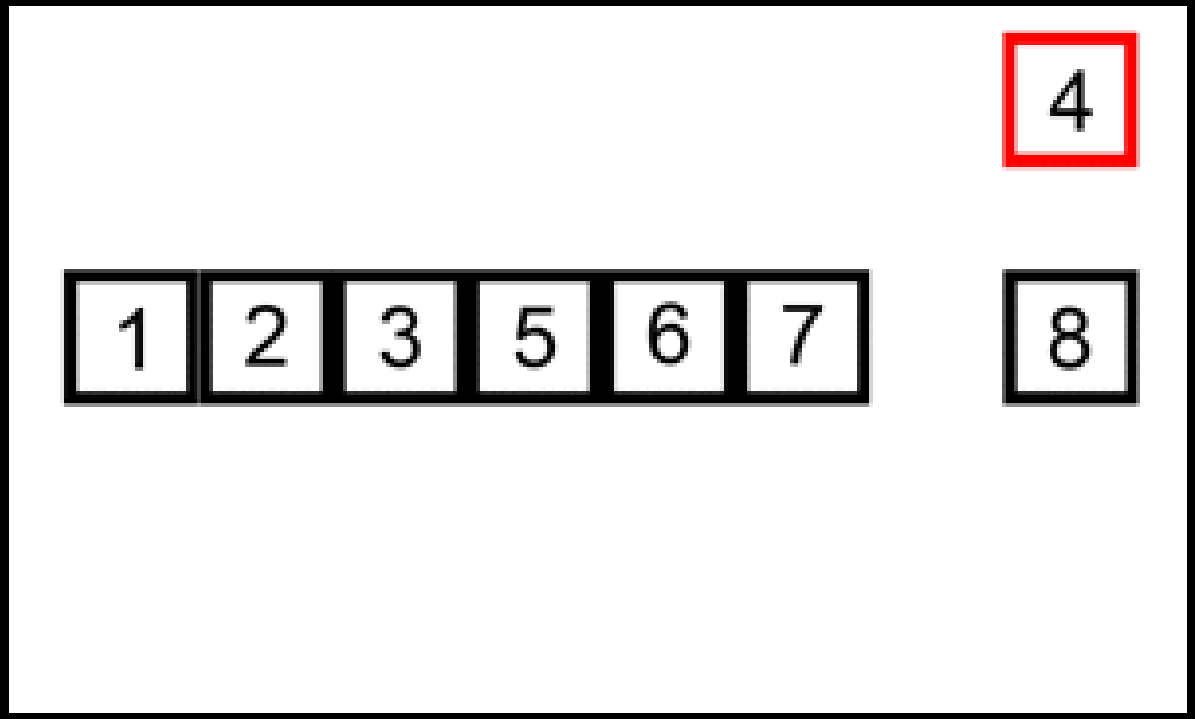
Insertion Sort Example



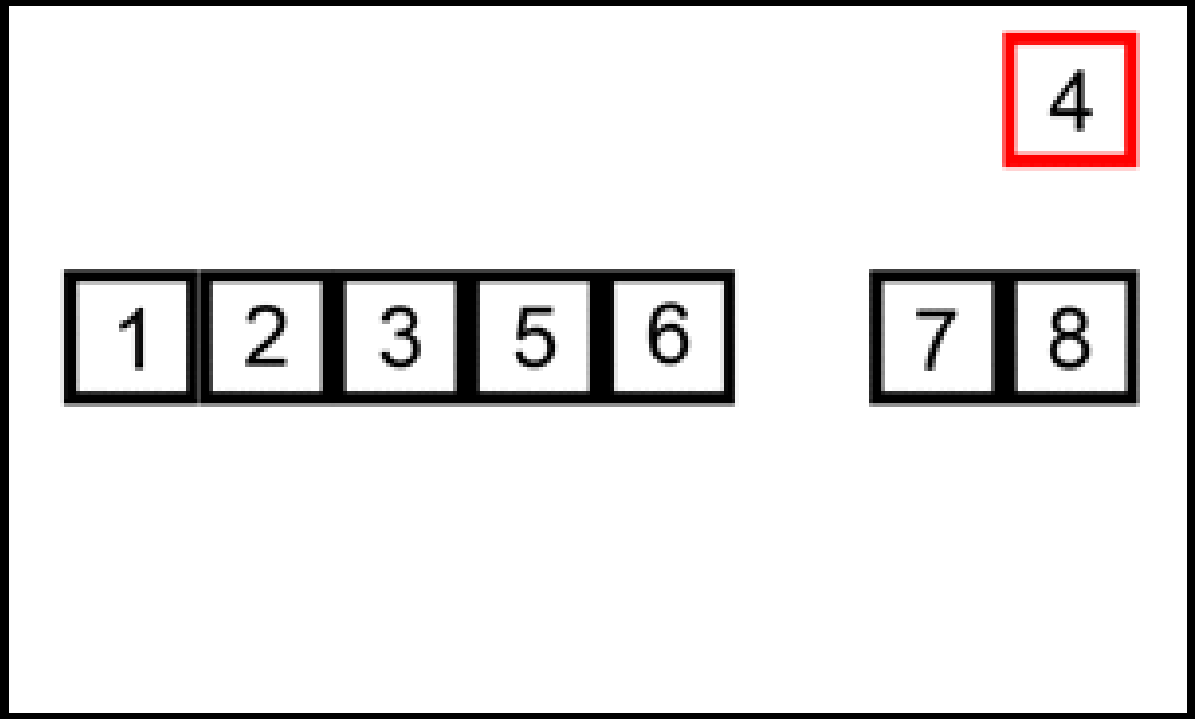
Insertion Sort Example



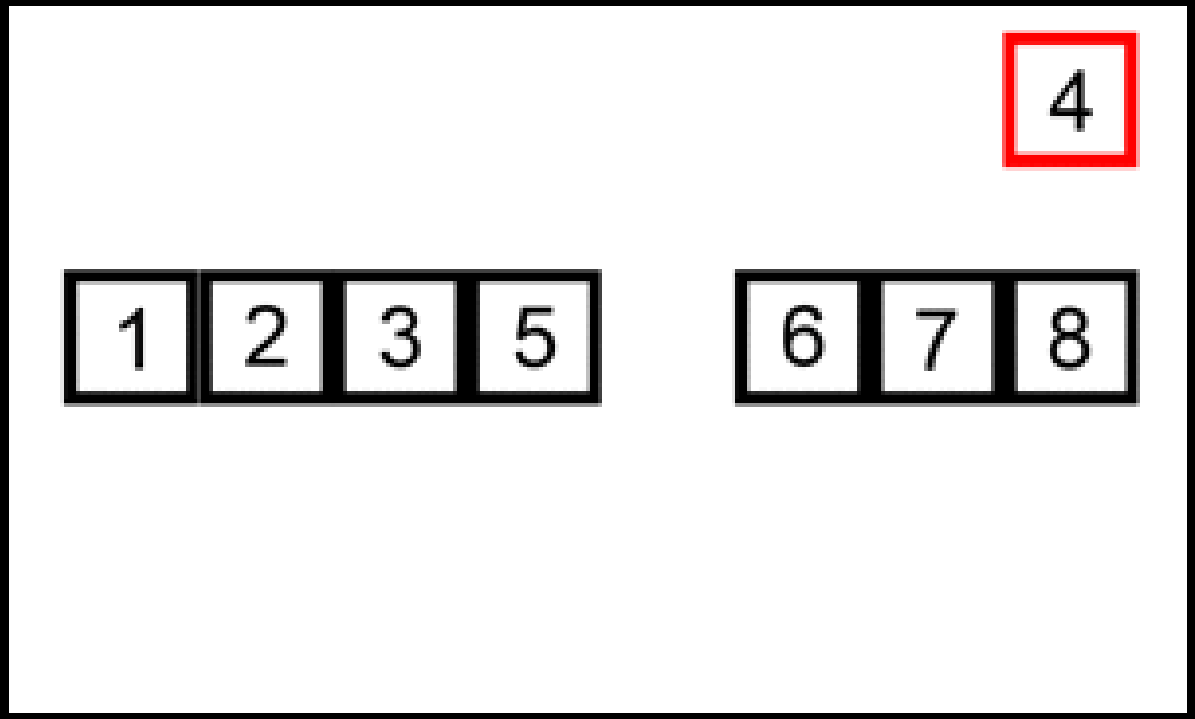
Insertion Sort Example



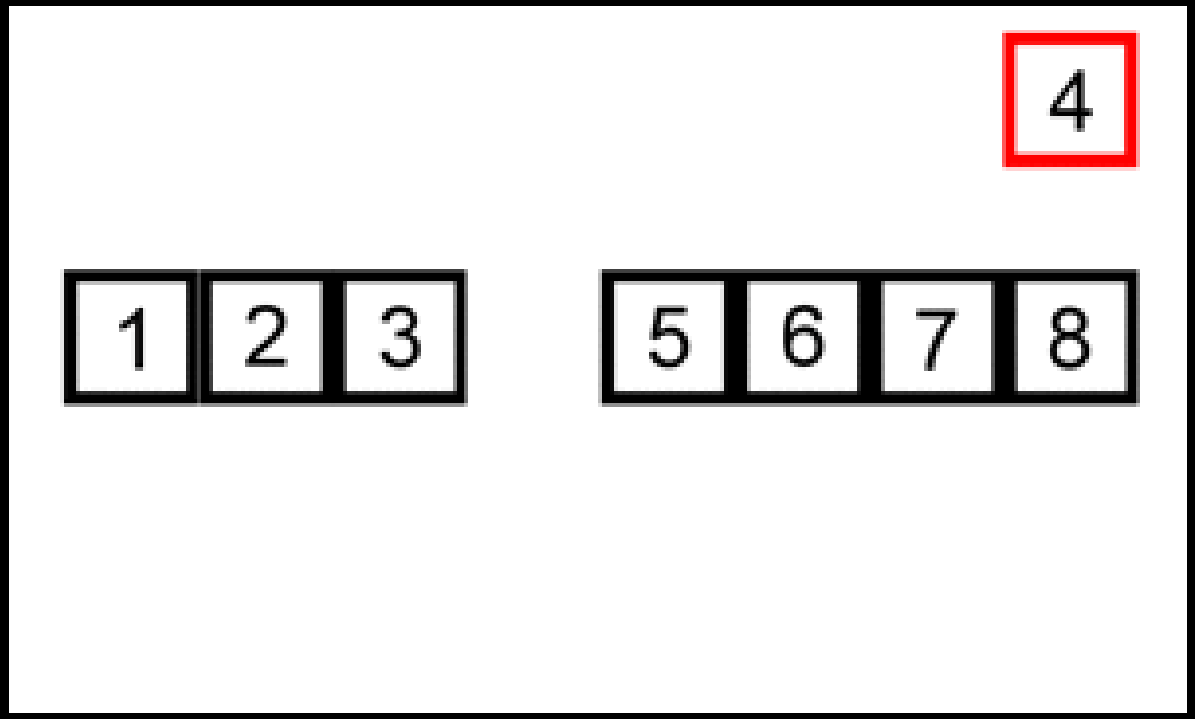
Insertion Sort Example



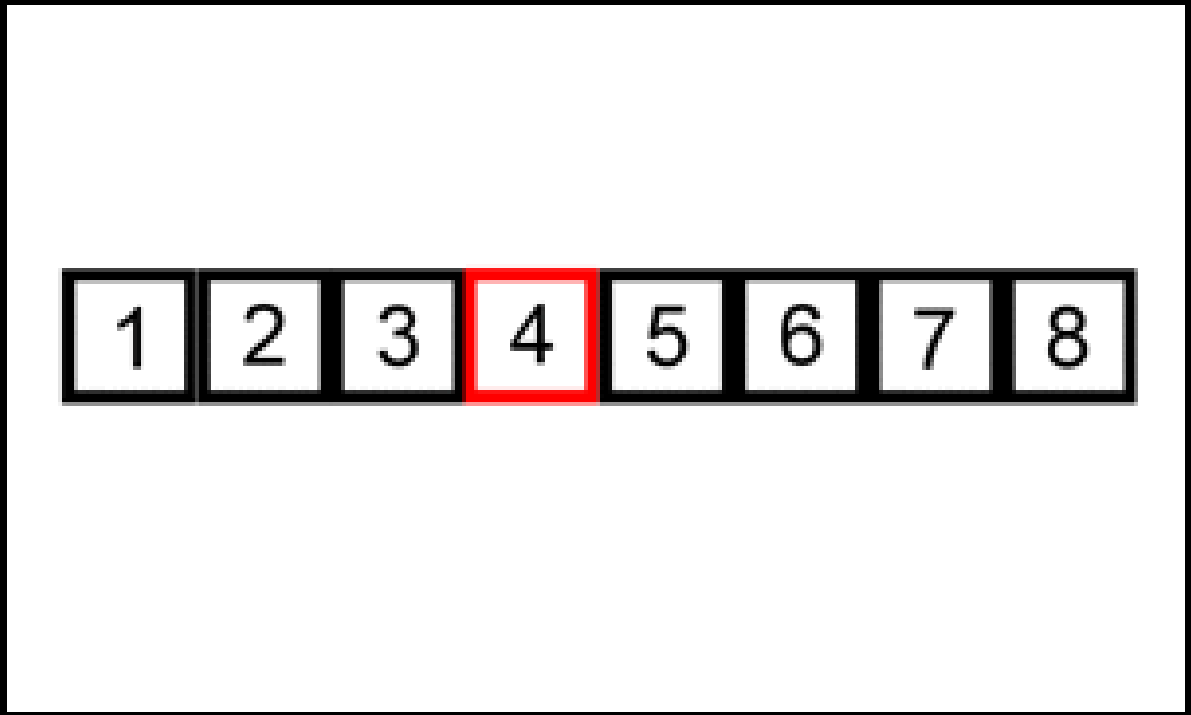
Insertion Sort Example



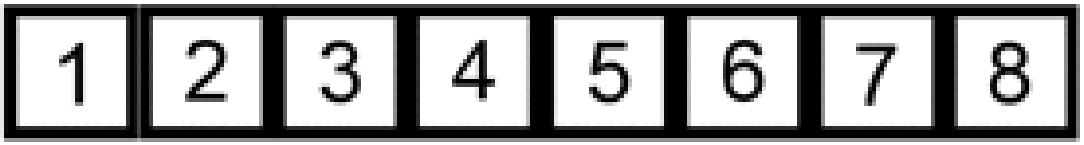
Insertion Sort Example



Insertion Sort Example



Insertion Sort Example



Faster Sorting

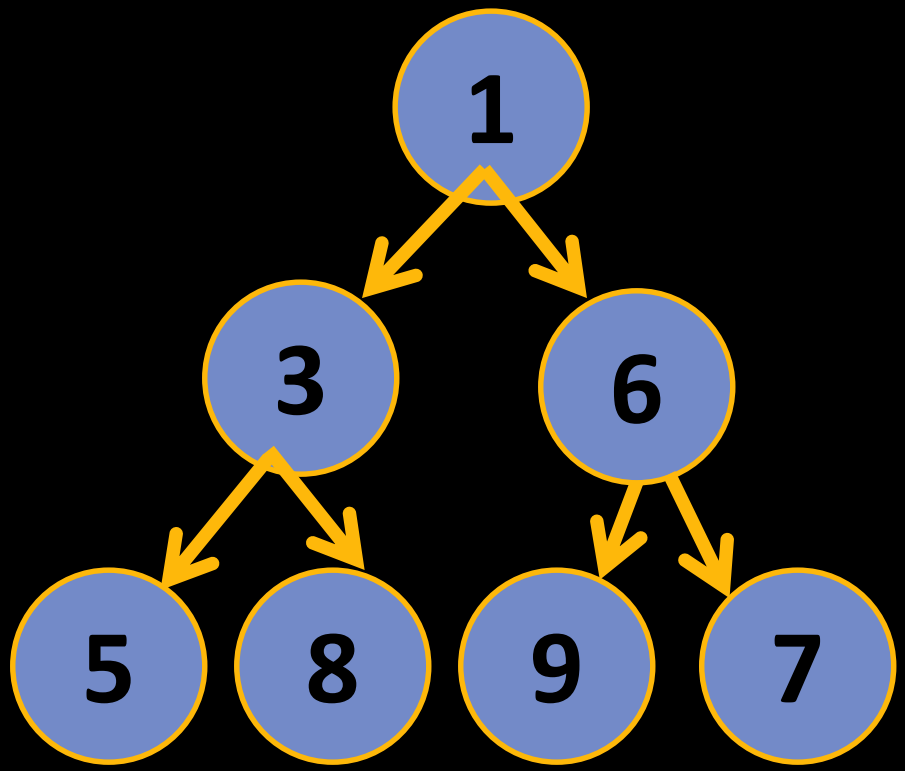
- Divide and Conquer:
  - Solve large problem by splitting it into smaller instances of same problem

Heap Sort

- (From last lecture)
- Create heap
- Remove smallest element from heap, one at a time

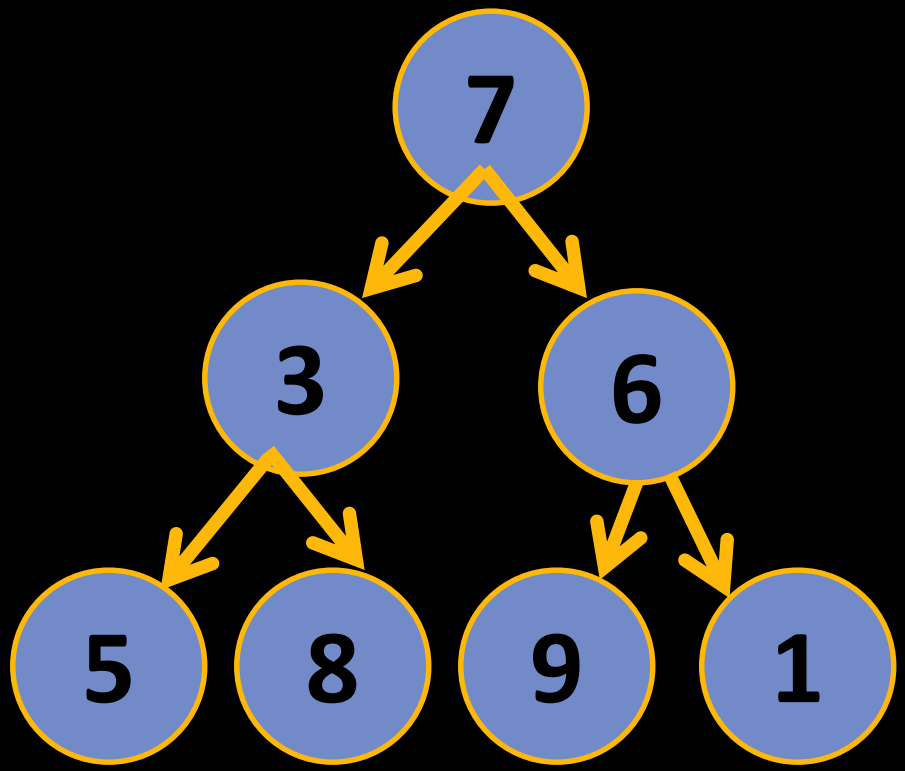
- NOT Stable
- In-place*
- Best / average / worst case running time: $O(n \log n)$ for a list with n elements

In-place Heap Sort



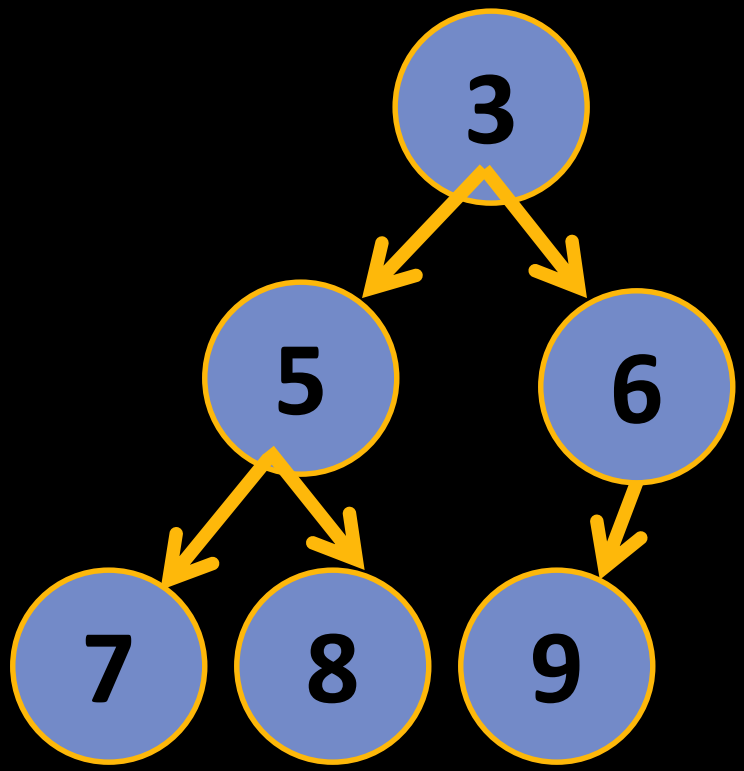
0	1	2	3	4	5	6	7
X	1	3	6	5	8	9	7

In-place Heap Sort



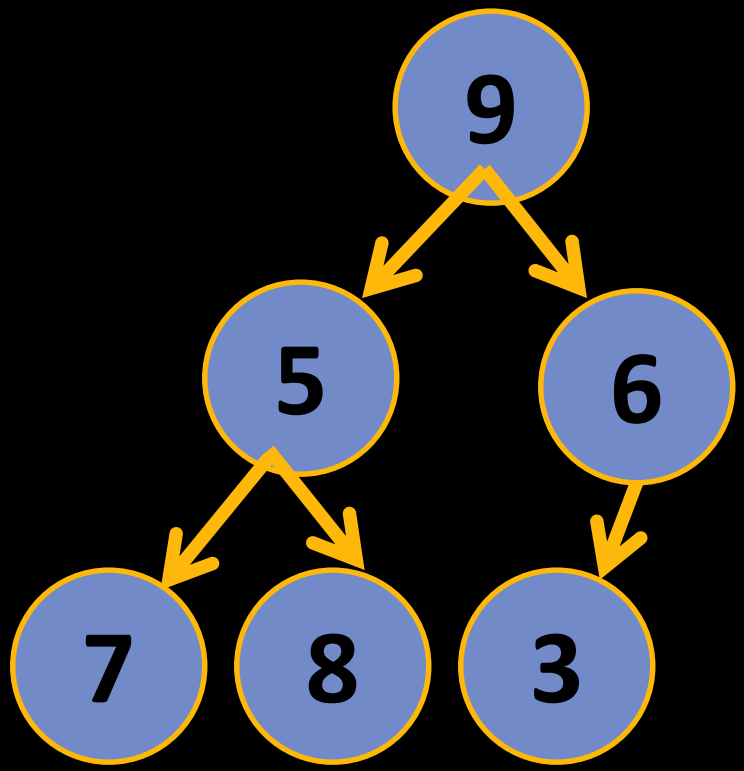
0	1	2	3	4	5	6	7
X	7	3	6	5	8	9	1

In-place Heap Sort



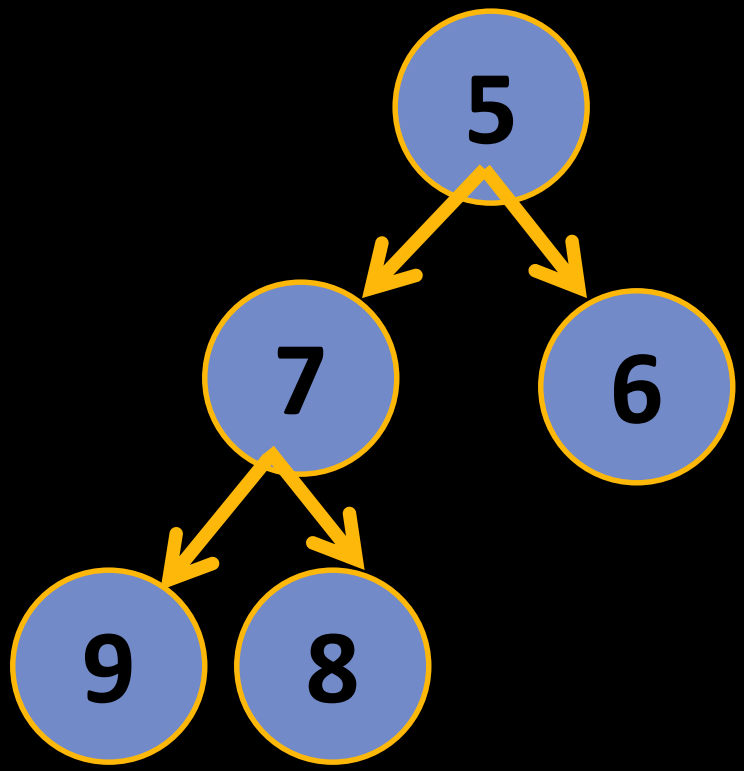
0	1	2	3	4	5	6	0
X	3	5	6	7	8	9	1

In-place Heap Sort



0	1	2	3	4	5	6	0
X	9	5	6	7	8	3	1

In-place Heap Sort



0	1	2	3	4	5	0	1
X	5	7	6	9	8	3	1

In-place Heap Sort

Etc…

In-place Heap Sort

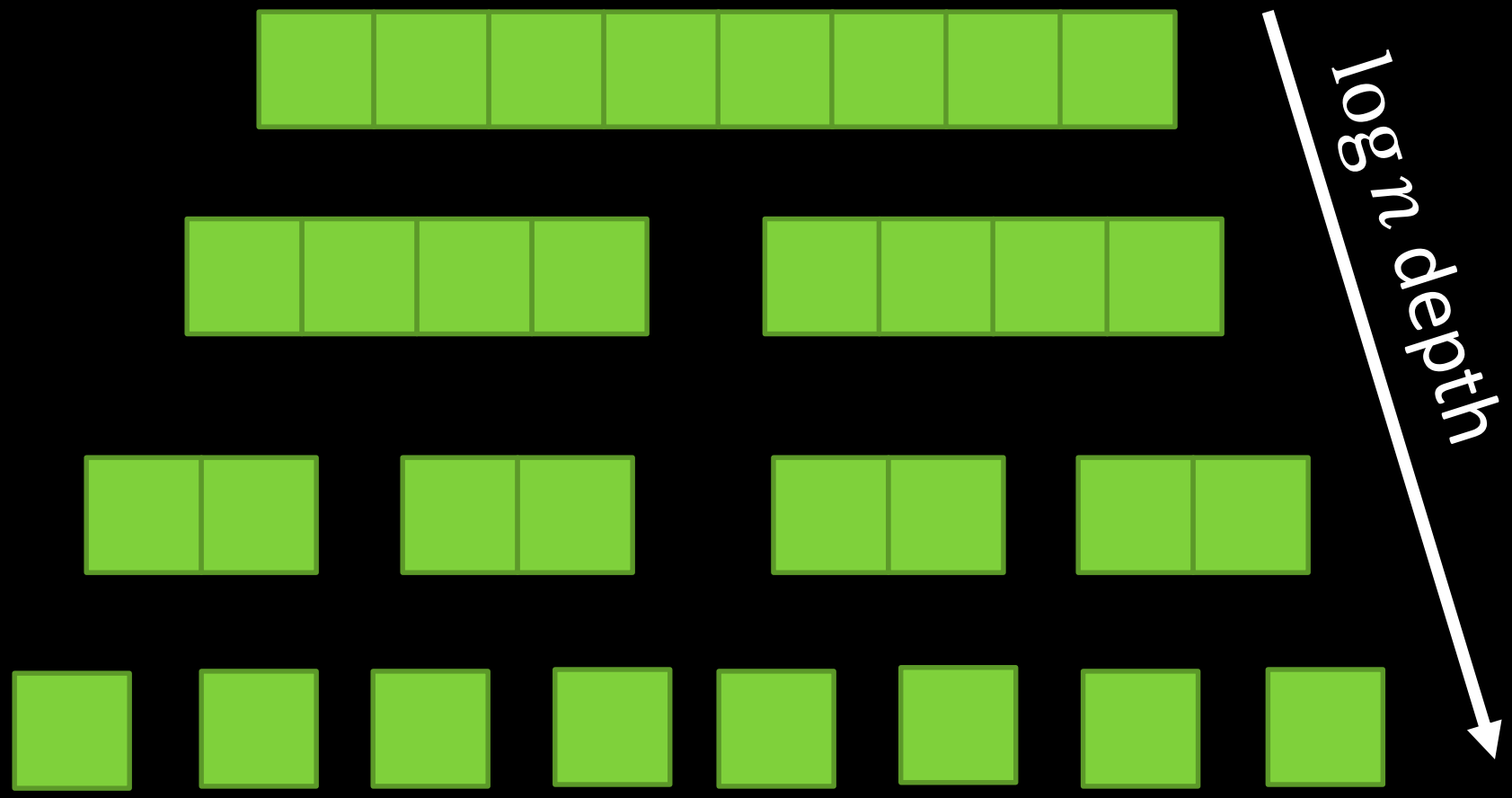
	0	1	2	3	4	5	6
X	9	8	7	6	5	3	1

Merge Sort

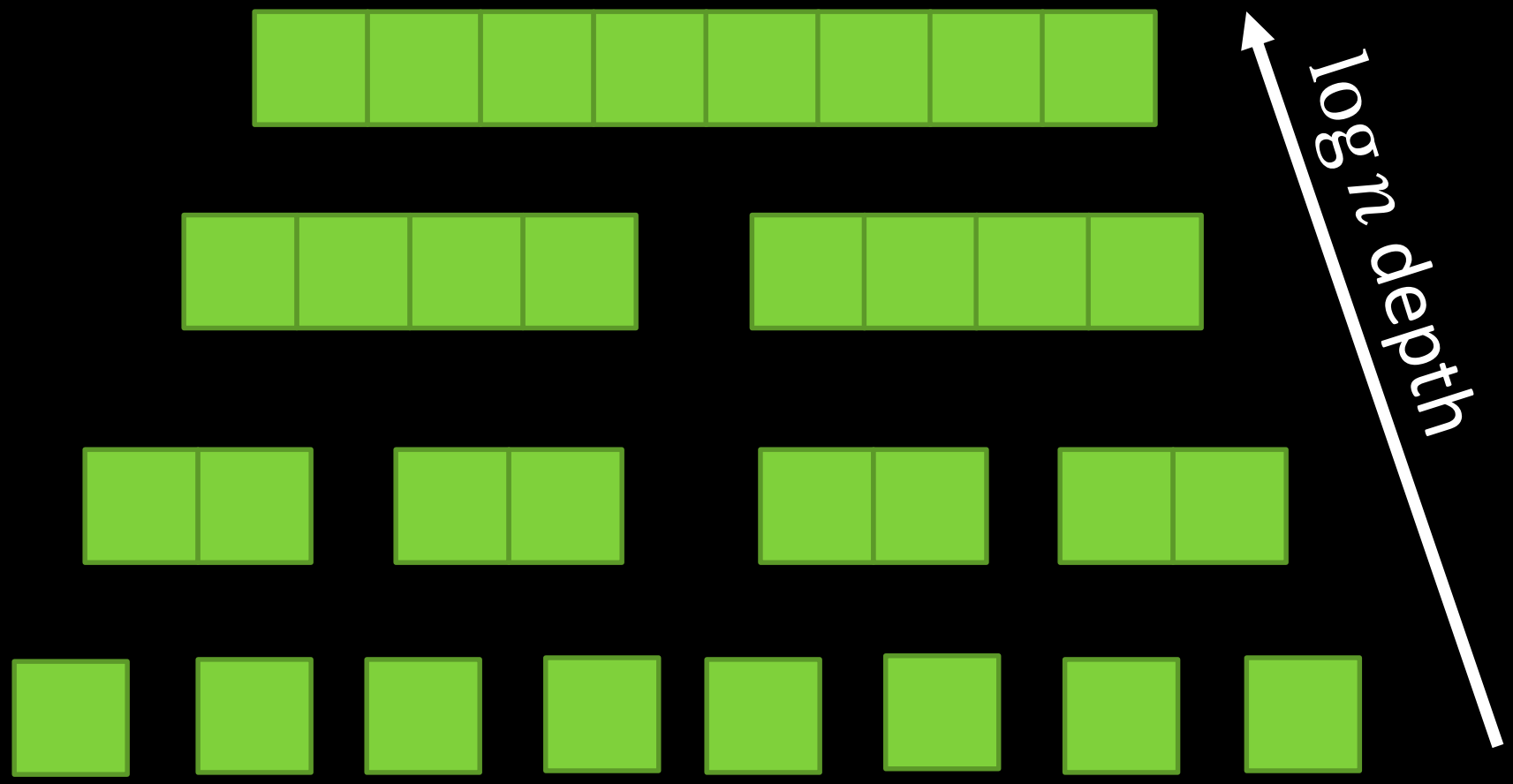
- Base case: list of 1 element is already sorted
- Recursively sort left and right halves
- Merge

- Stable*
- Can be done in-place (not efficient)
- Best / average / worst case running time: $O(n \log n)$ for a list with n elements

Merge Sort Running Time



Merge Sort Running Time



Merge Sort Running Time

1 merge: lists of $\frac{n}{2}$ elements each

2 merges: lists of $\frac{n}{4}$ elements each

$\vdots$

$\frac{n}{2}$ merges: lists of 1 element each

Merge Sort Running Time

1 merge: lists of $\frac{n}{2}$ elements each $\quad O(n)$

2 merges: lists of $\frac{n}{4}$ elements each $\quad O(n)$

$\vdots \qquad \vdots$

$\frac{n}{2}$ merges: lists of 1 element each $\quad O(n)$

$\log n$ depth

Merge Sort Running Time

1 merge: lists of $\frac{n}{2}$ elements each	$O(n)$	$\log n$ depth
2 merges: lists of $\frac{n}{4}$ elements each	$O(n)$	
$\vdots$	$\vdots$	
$\frac{n}{2}$ merges: lists of 1 element each	$O(n)$	

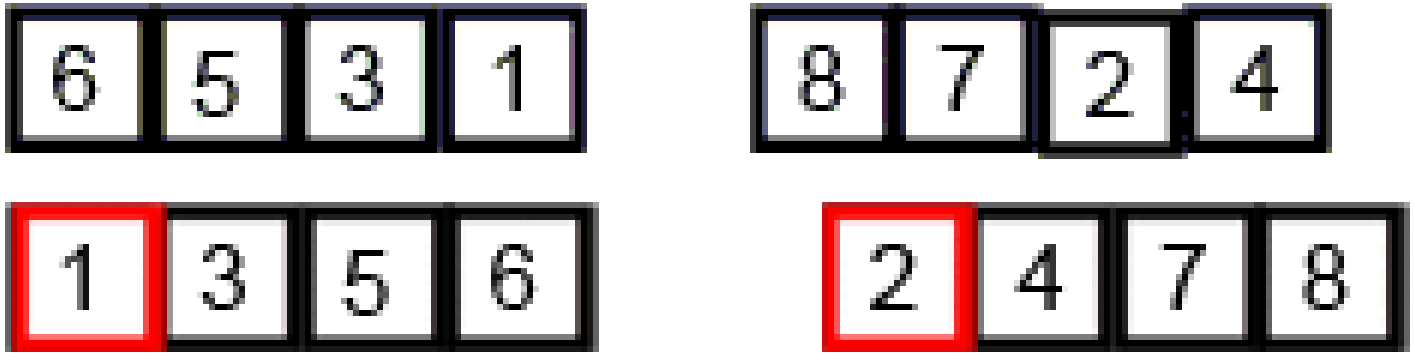
$$O(n \log n)$$

6 5 3 1 8 7 2 4

6	5	3	1	8	7	2	4

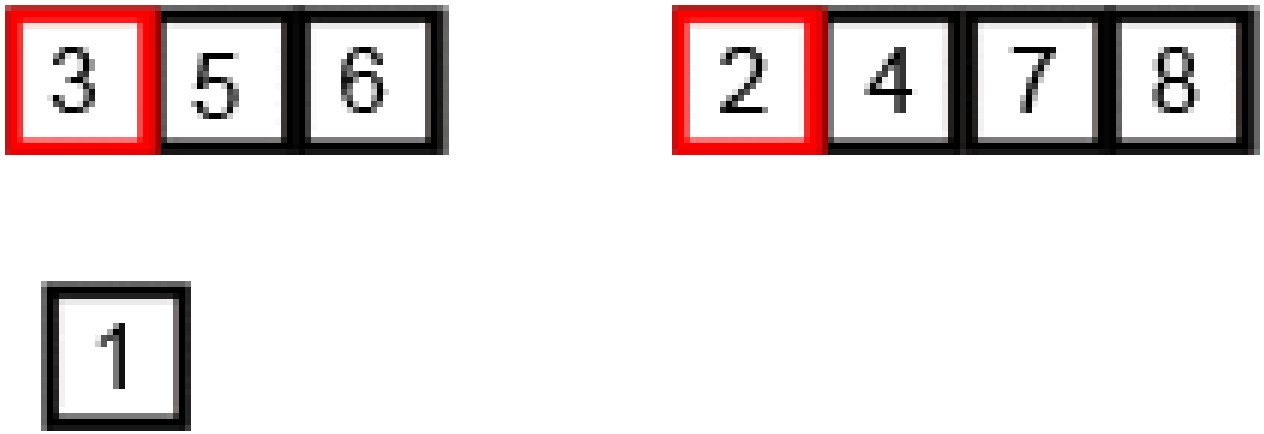
6	5	3	1

8	7	2	4

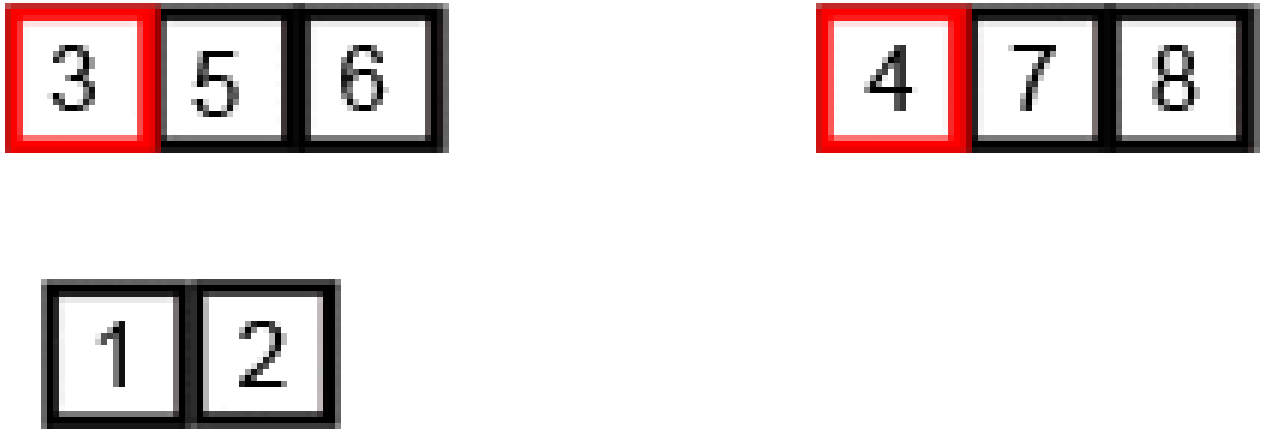

6 5 3 1
8 7 2 4
1 3 5 6
2 4 7 8

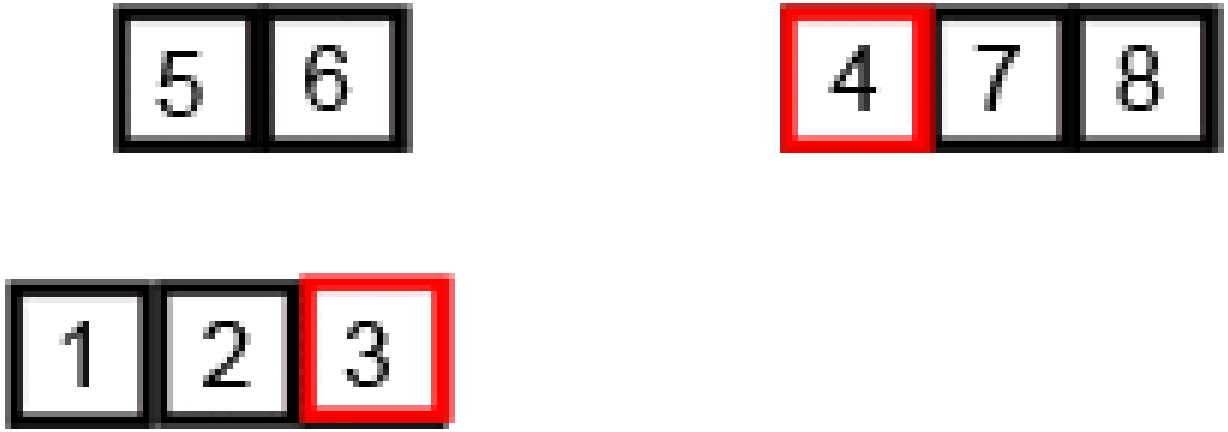

1 3 5 6
2 4 7 8

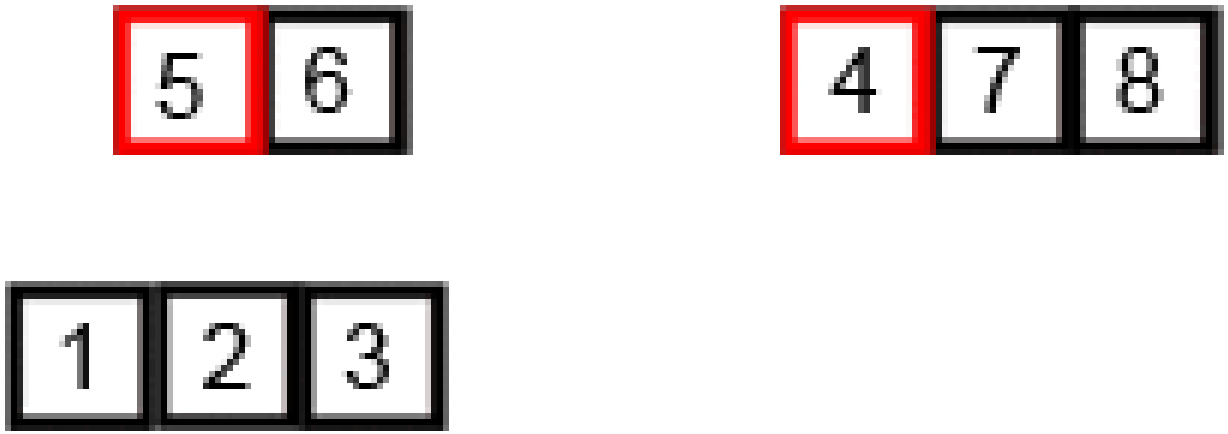

3
5
6
2
4
7
8
1

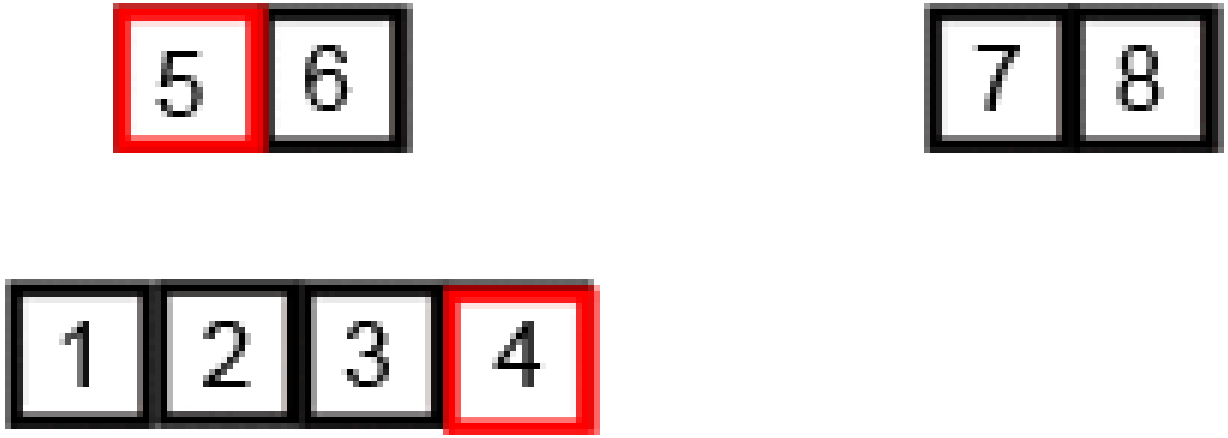

3
5
6
2
4
7
8
1

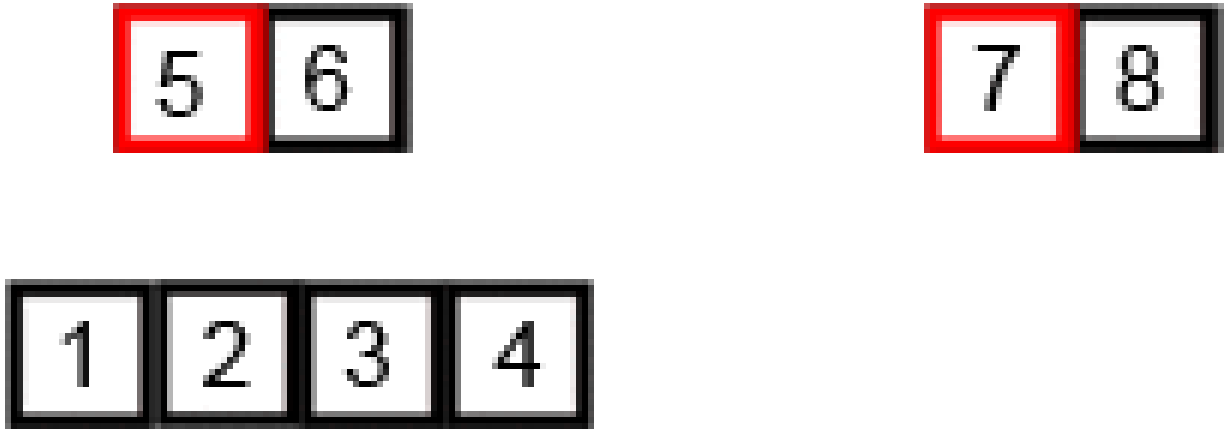

3
5
6
4
7
8
1
2

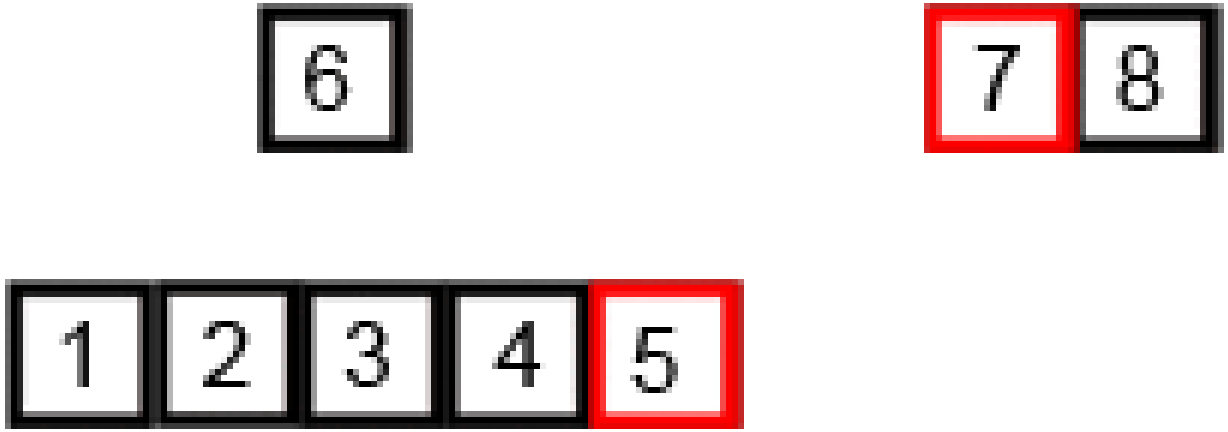

3
5
6
4
7
8
1
2

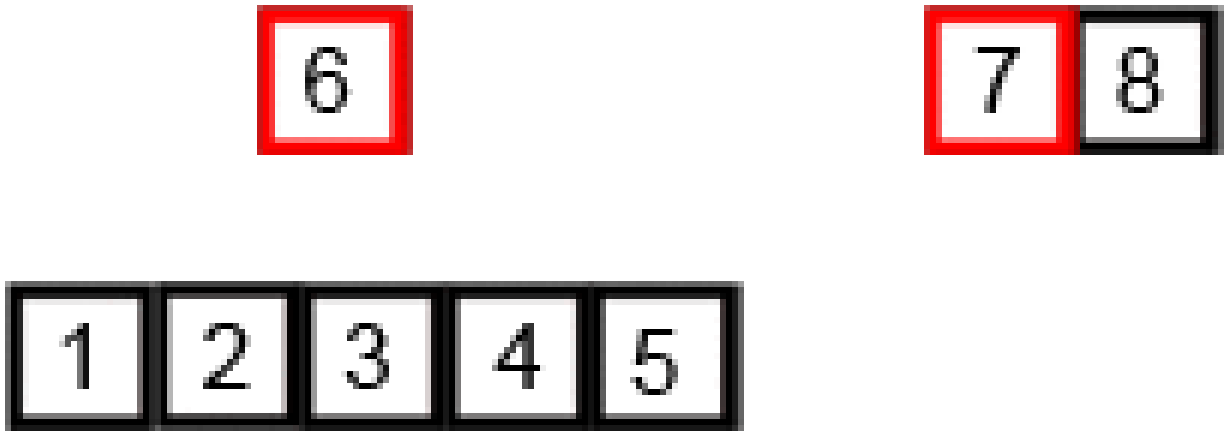

5
6
4
7
8
1
2
3

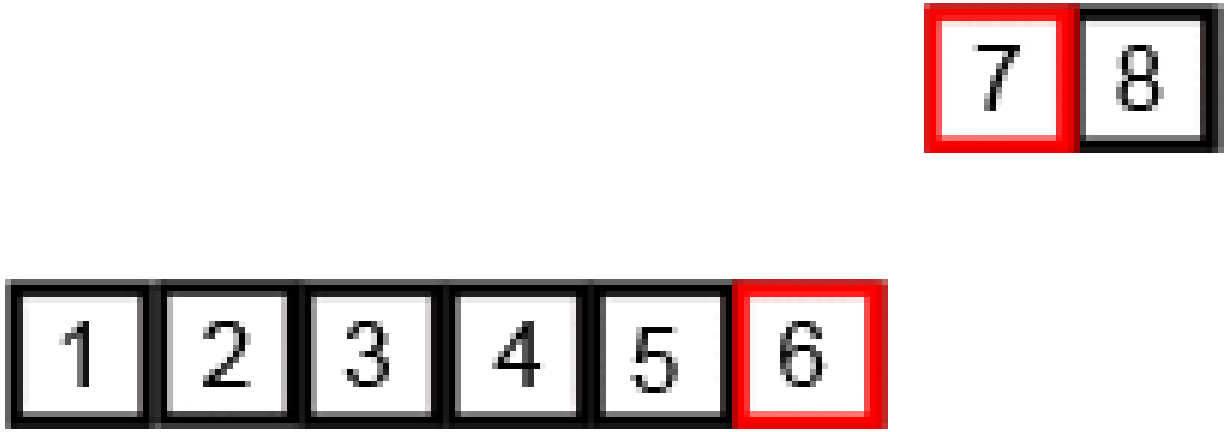

5
6
4
7
8
1
2
3

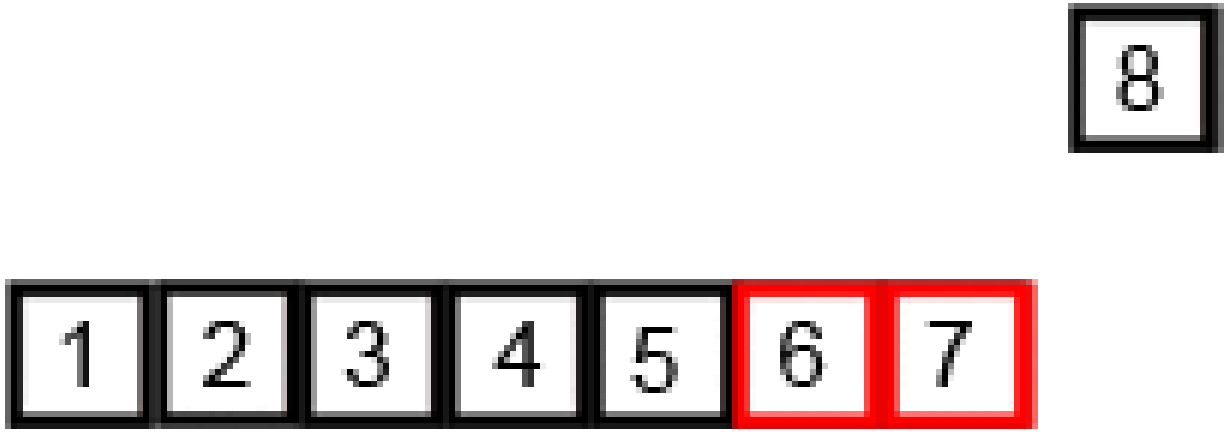

5
6
7
8
1
2
3
4

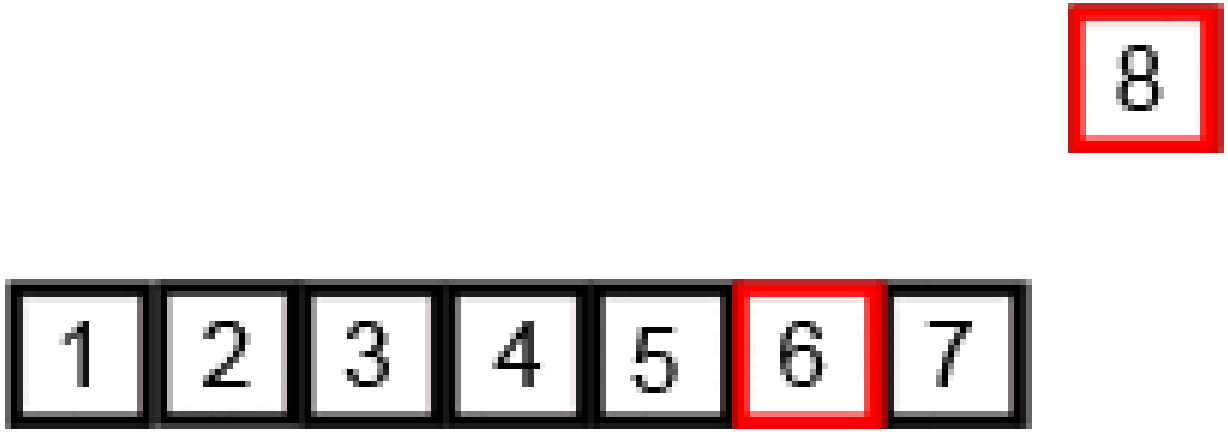

5
6
7
8
1
2
3
4


6
7
8
1
2
3
4
5


6
7
8
1
2
3
4
5


7
8
1
2
3
4
5
6


8
1
2
3
4
5
6
7


8
1
2
3
4
5
6
7

1	2	3	4	5	6	7	8

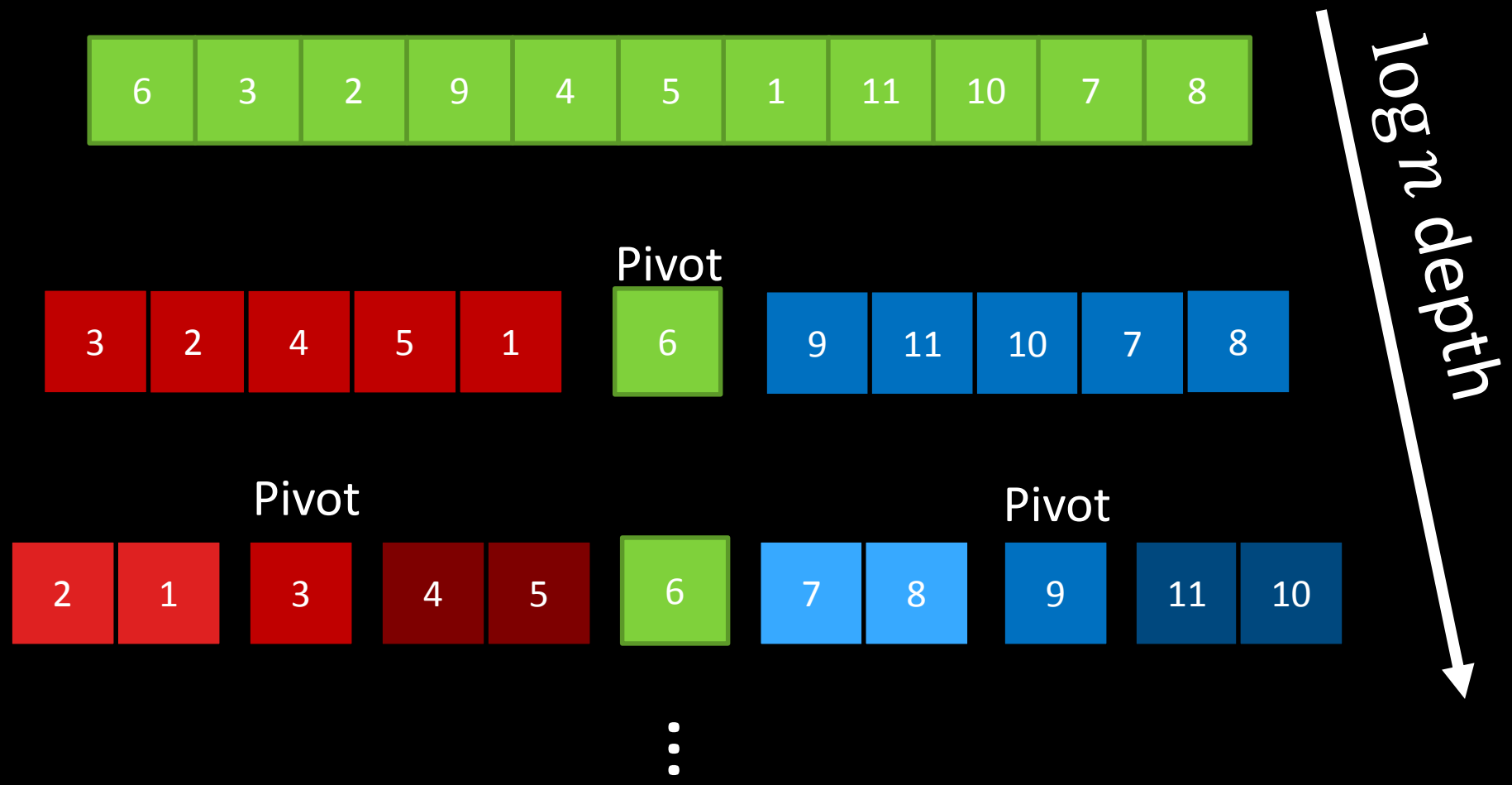
1	2	3	4	5	6	7	8

Quick Sort

- Base case: list of 1 element is already sorted
- Choose one of the list elements as the <u>pivot</u>
- Partition list into elements that are less than pivot and elements that are greater than pivot
- Recursively do this on both partitions

- Stable, in-place
- Best / average case running time: $O(n \log n)$ where n is number of elements in input list
- Worst case running time: $O(n^2)$

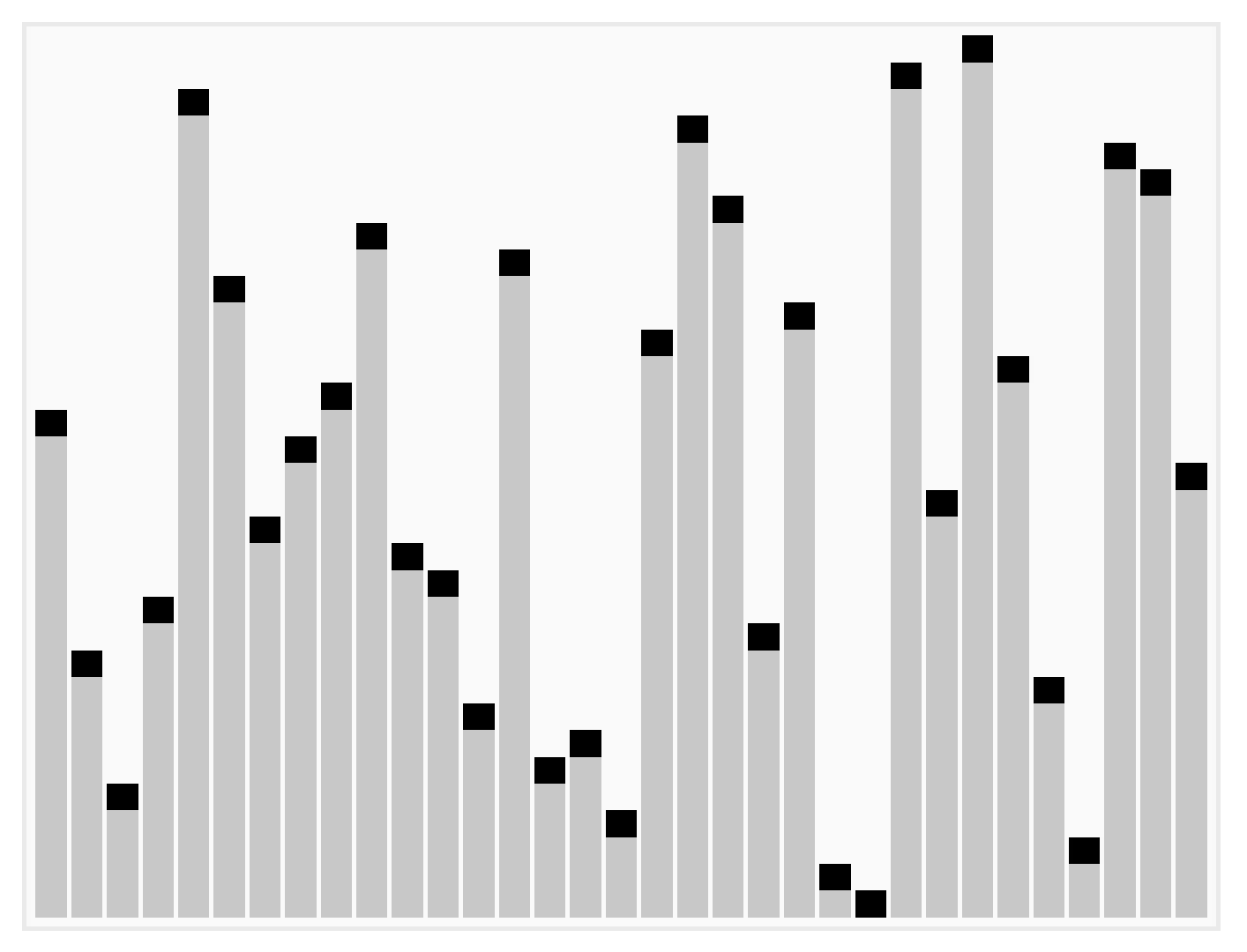
Quick Sort Running Time

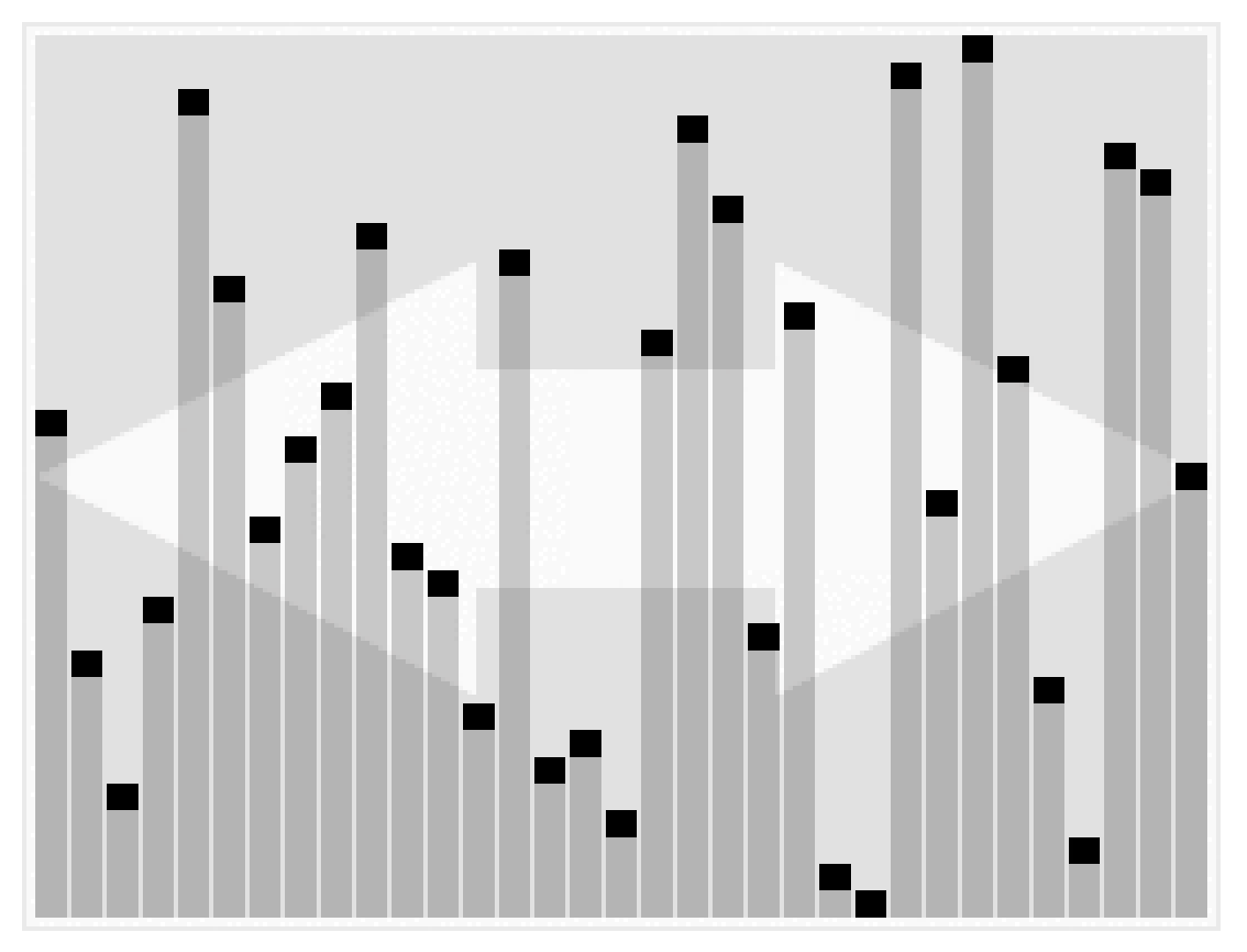


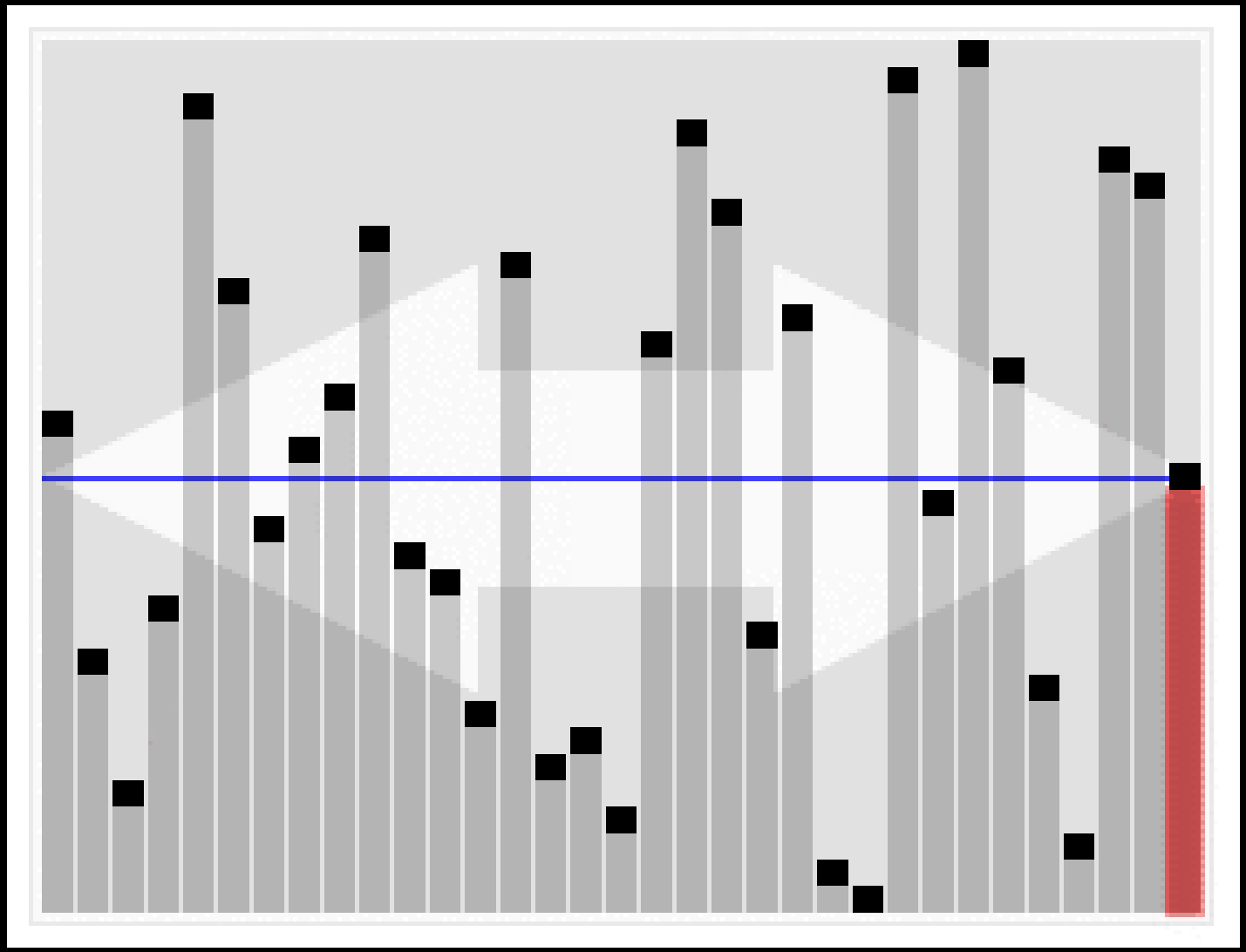
Quick Sort Running Time

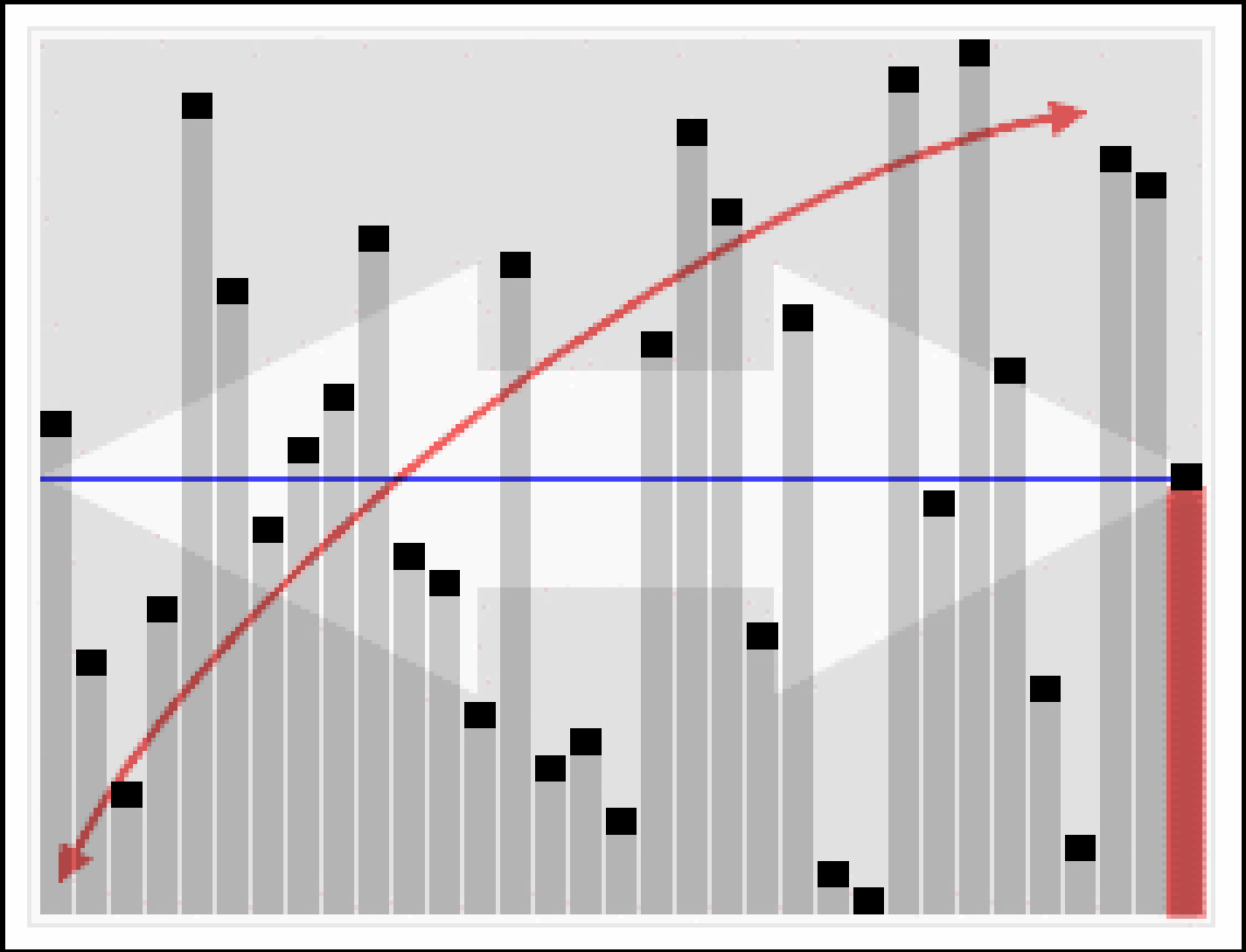
Quick Sort Running Time

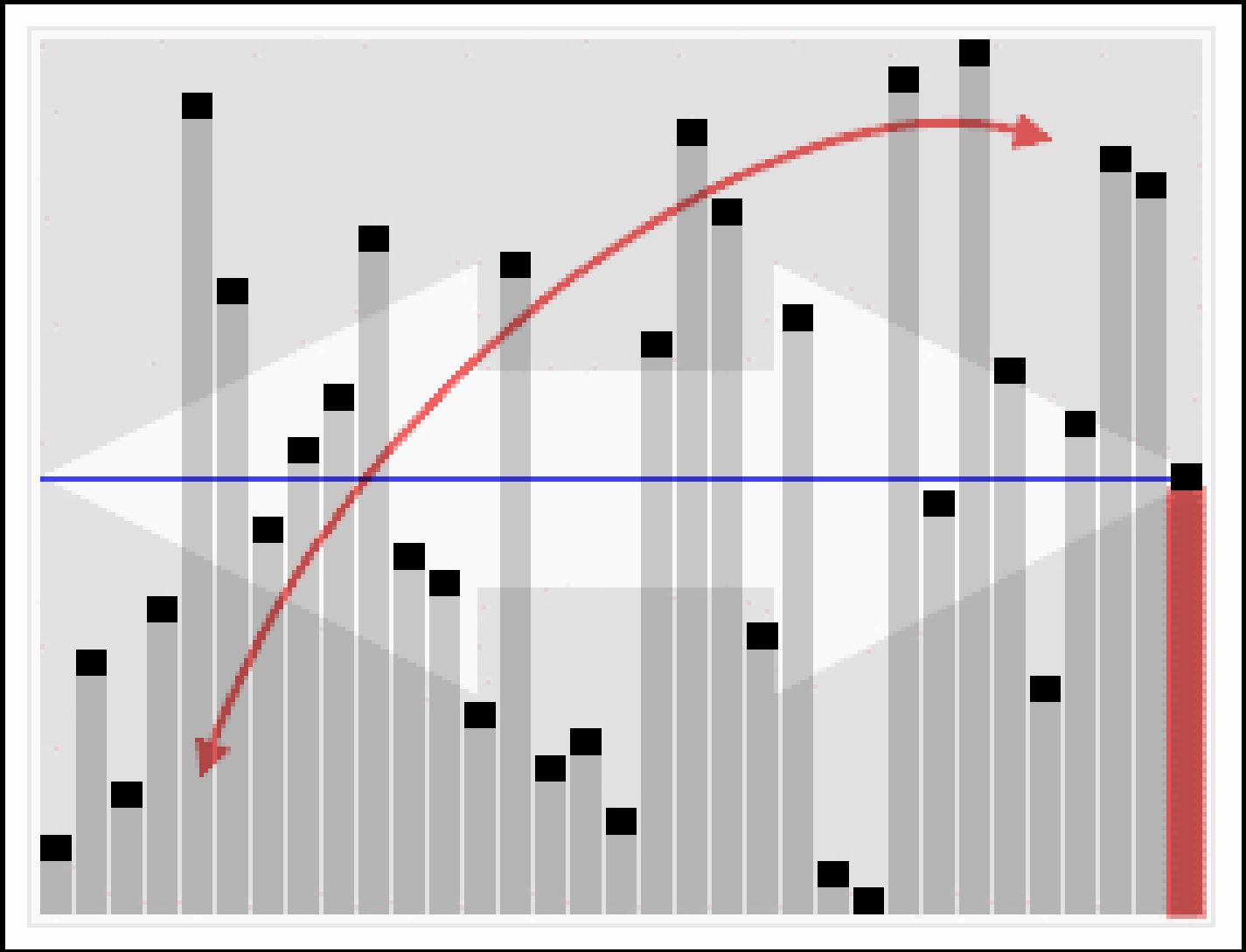
- All partitioning at each depth level: $O(n)$ time
- Number of depth levels: $O(\log n)$ to $O(n)$
- Total running time: $O(n \log n)$ to $O(n^2)$

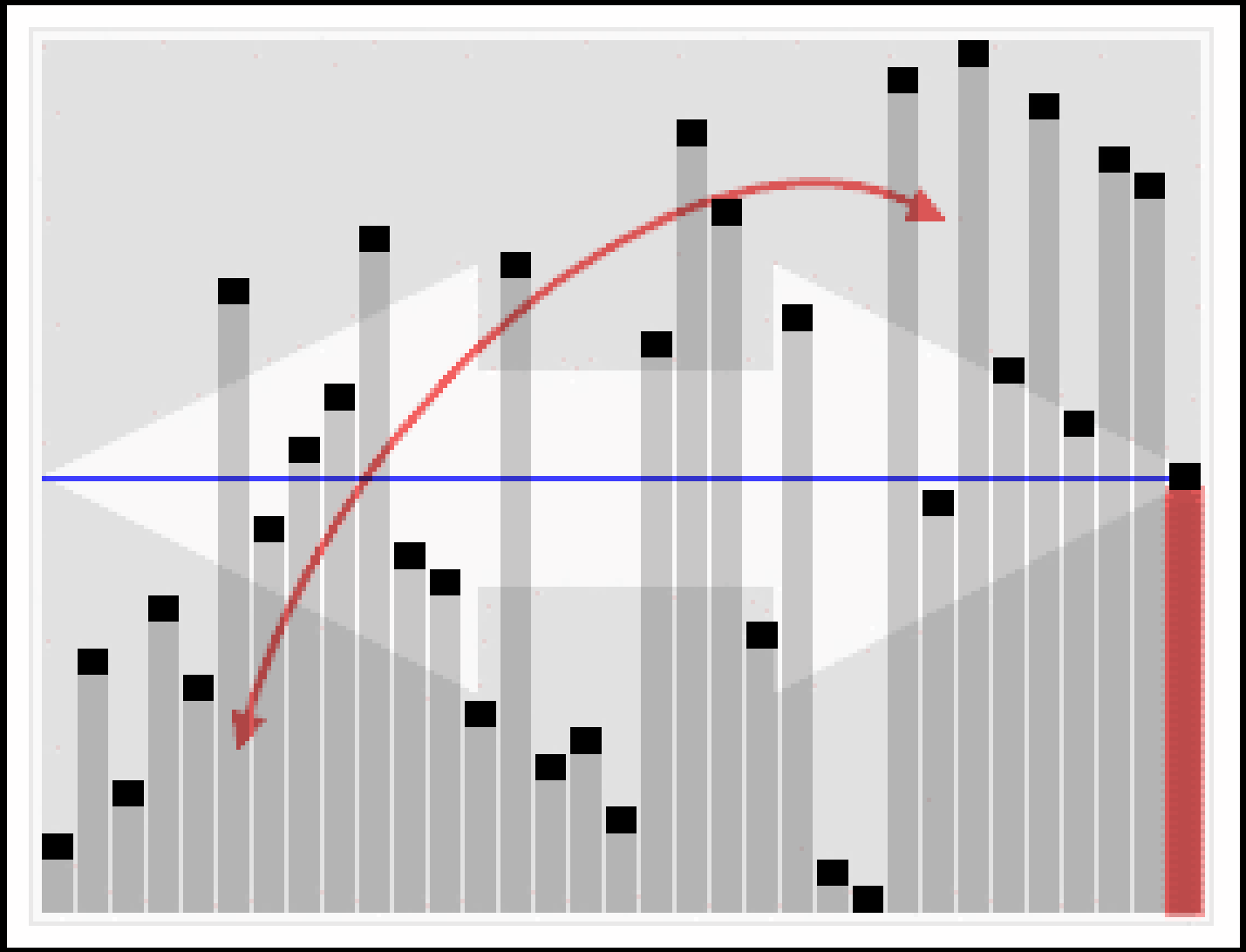


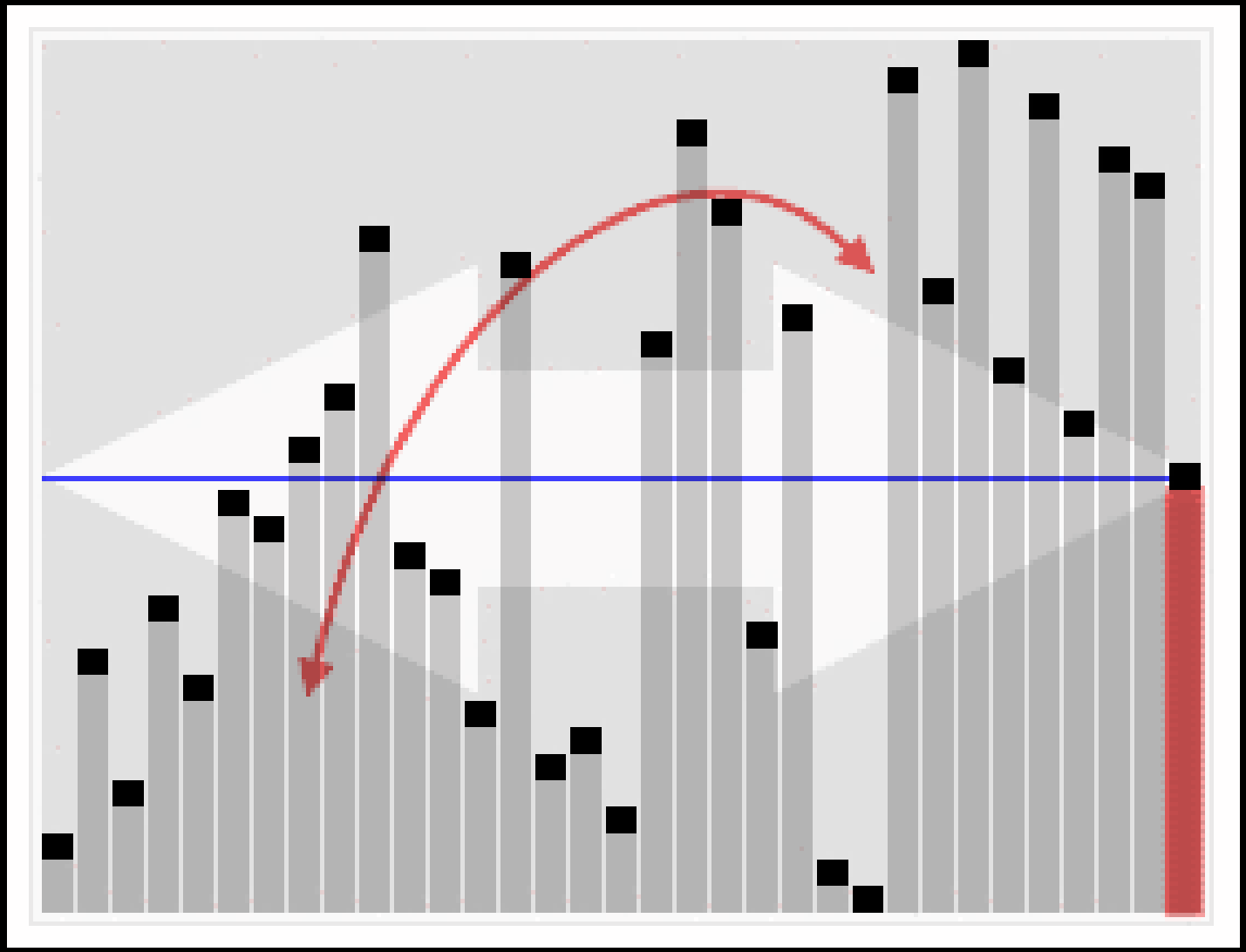


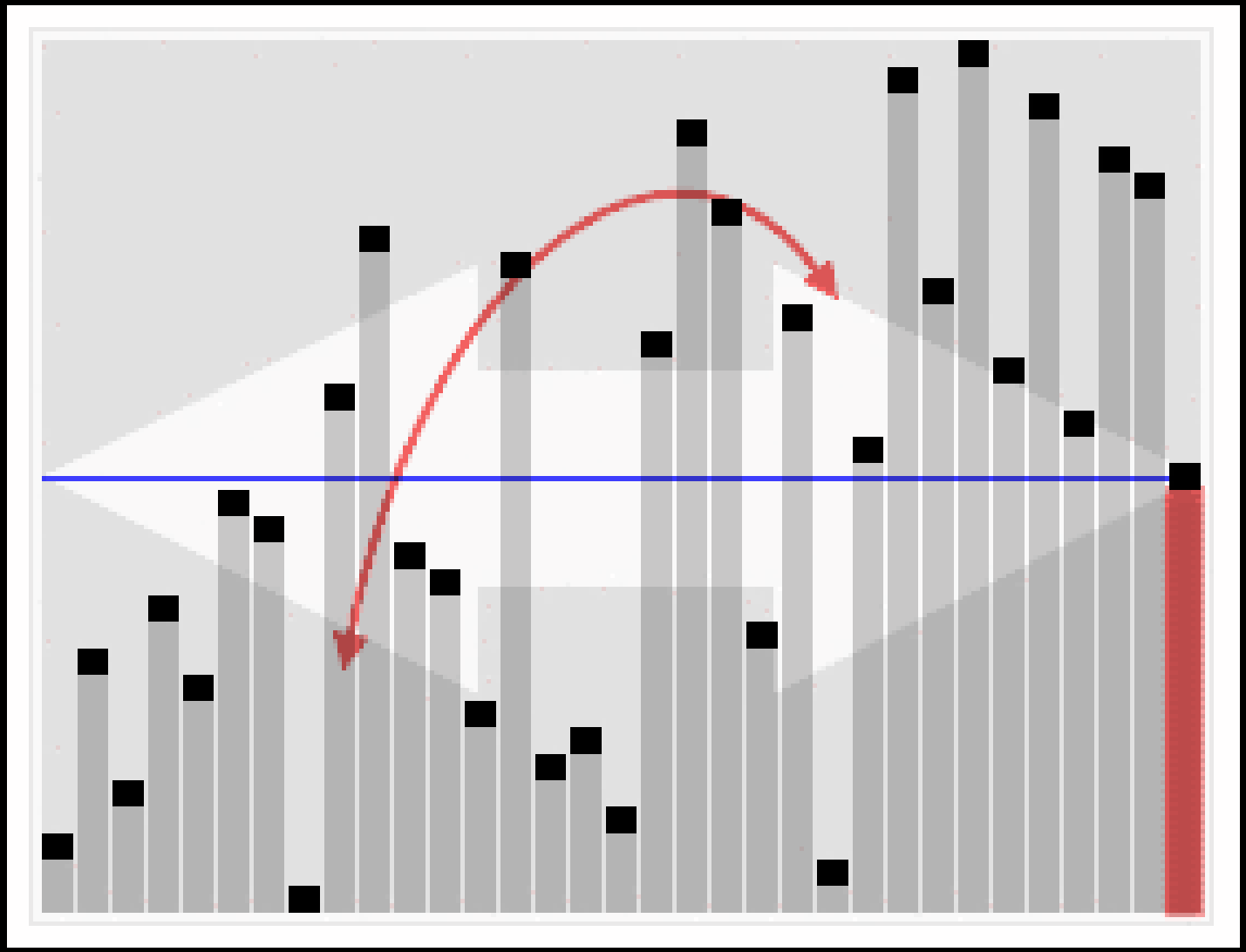


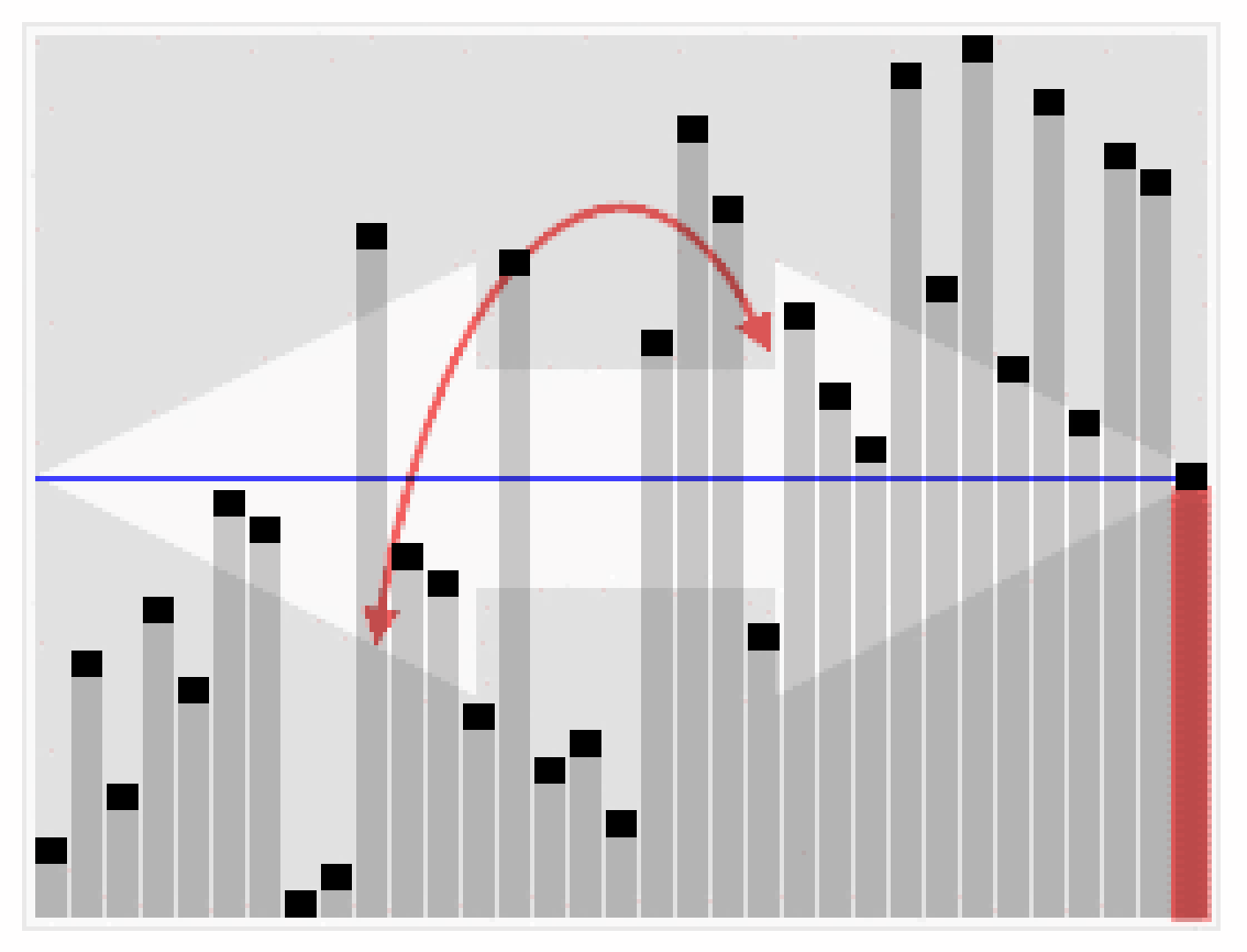


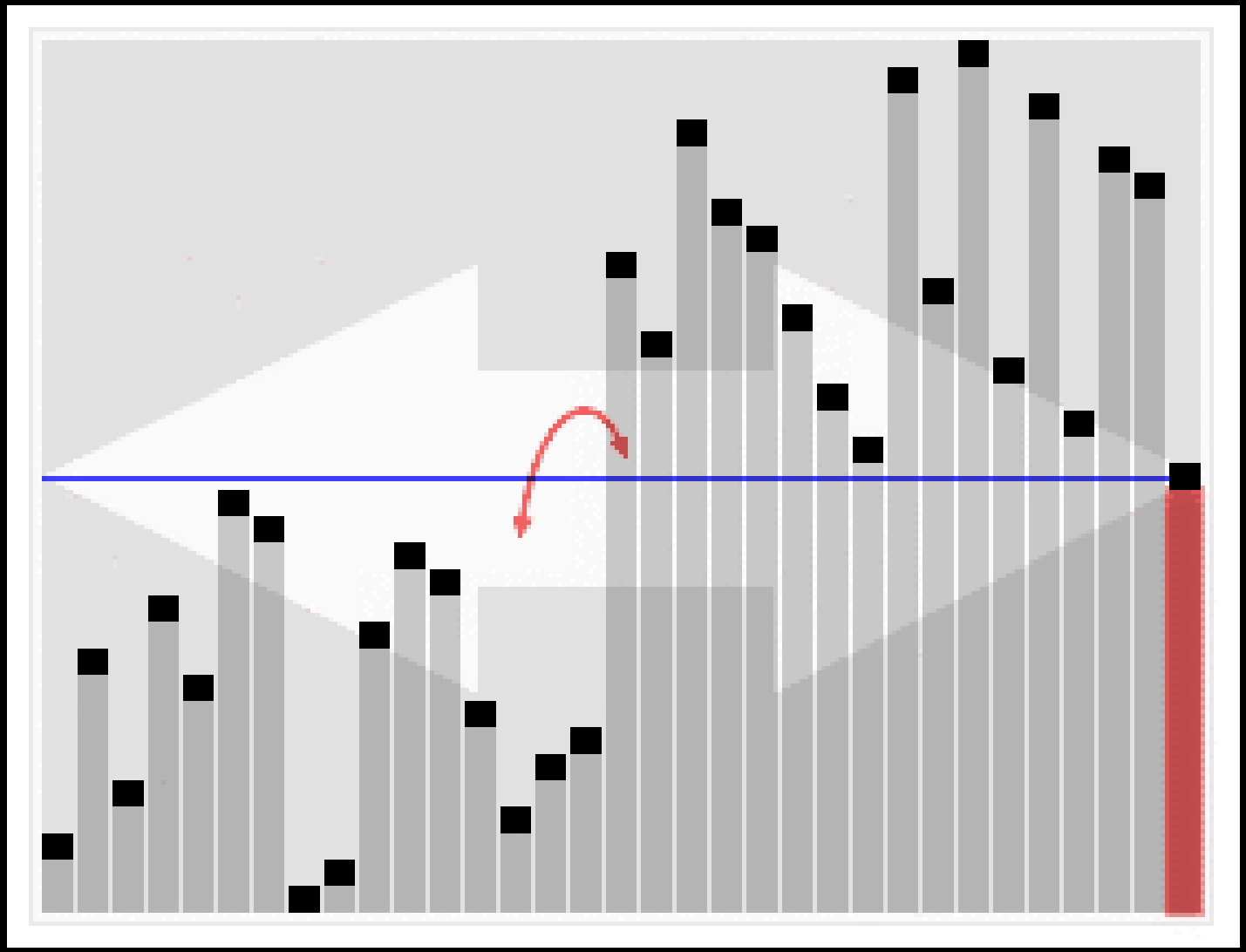


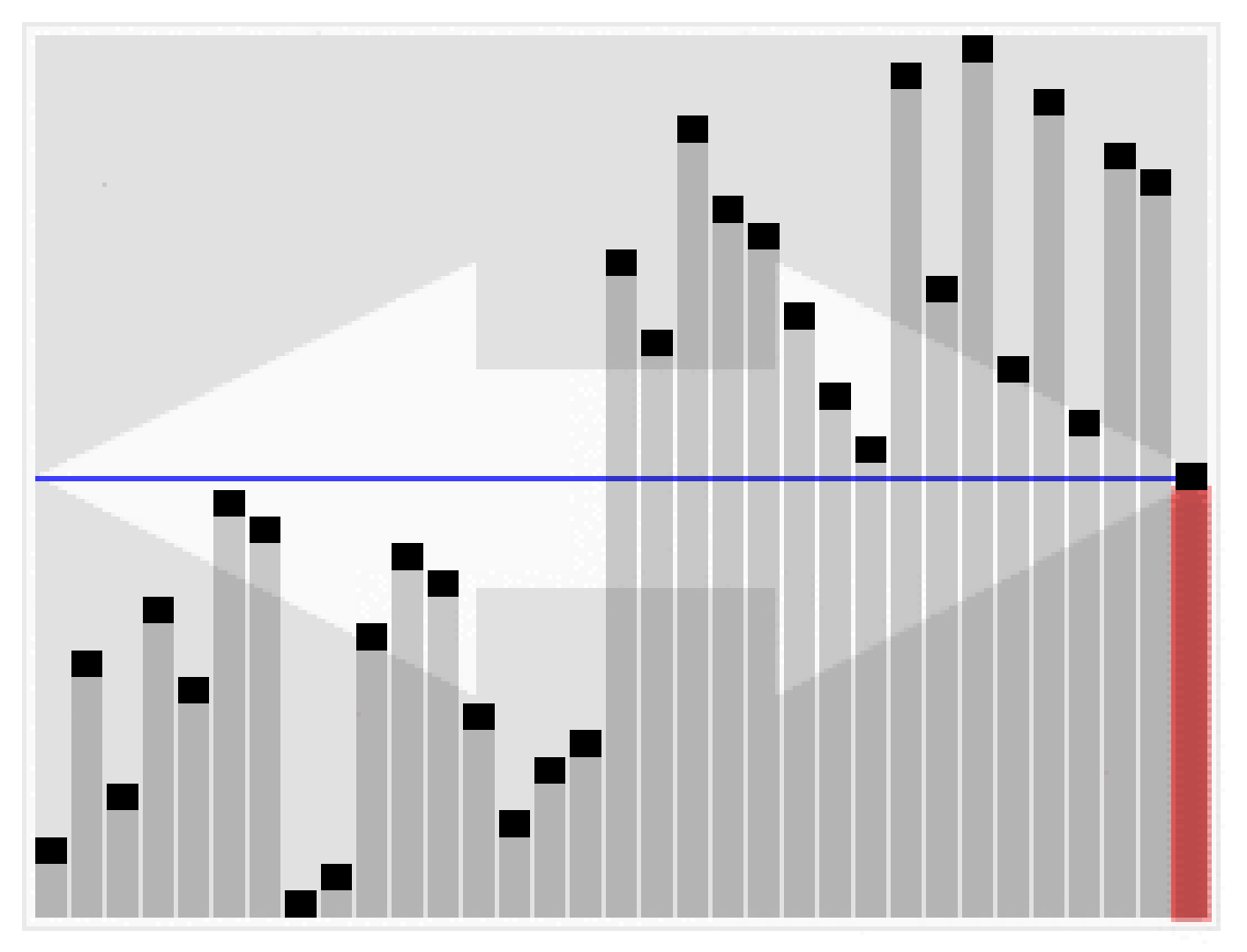


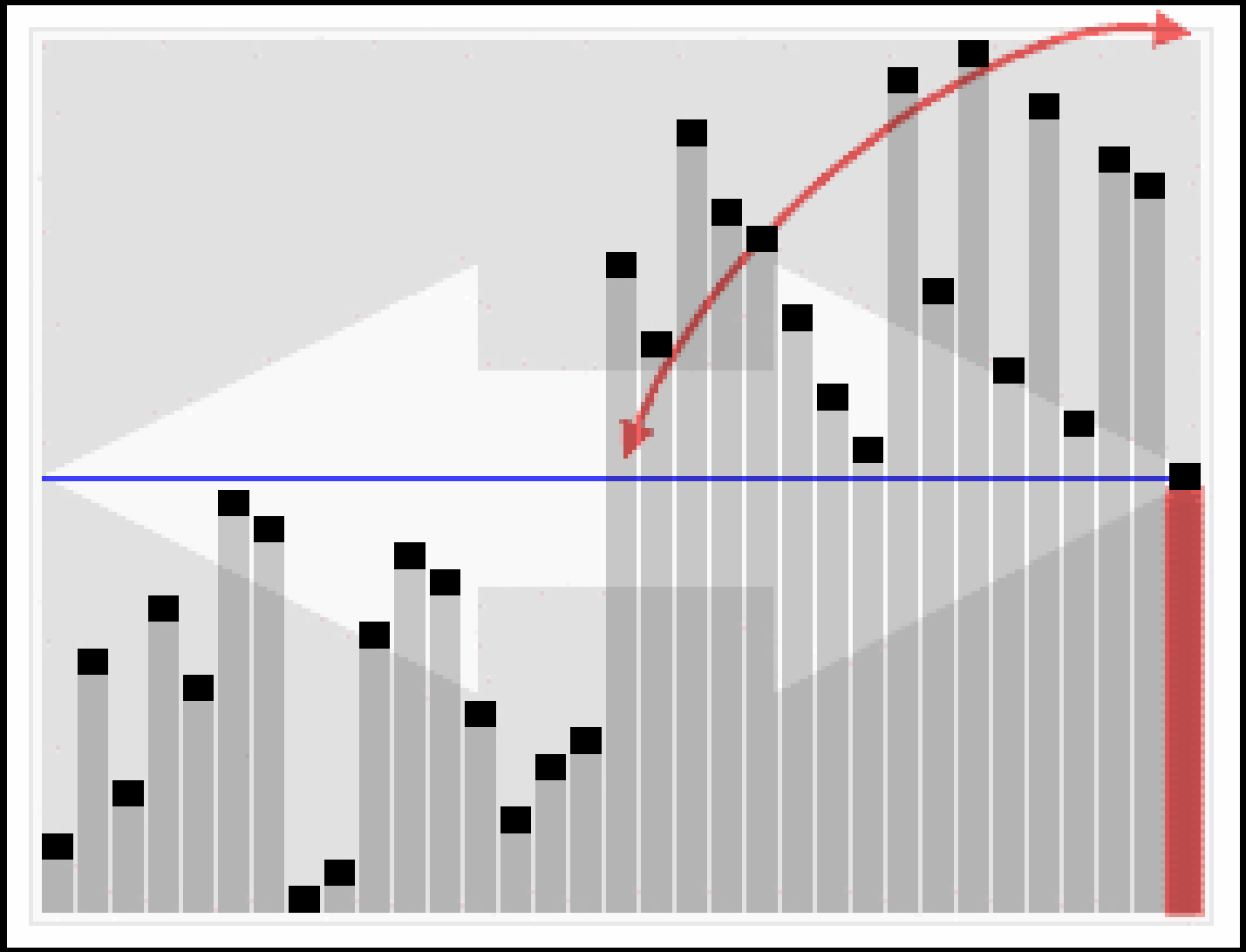


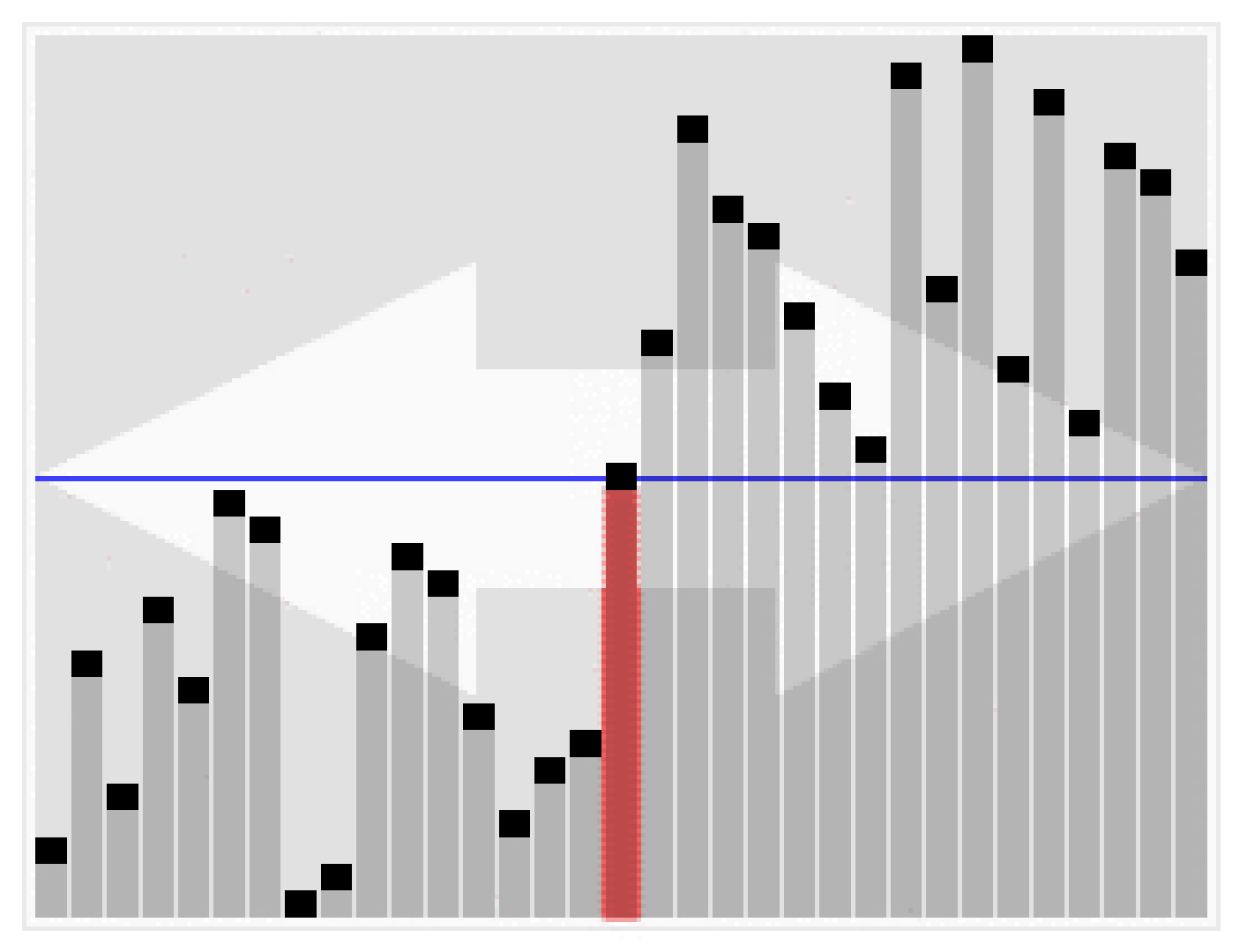


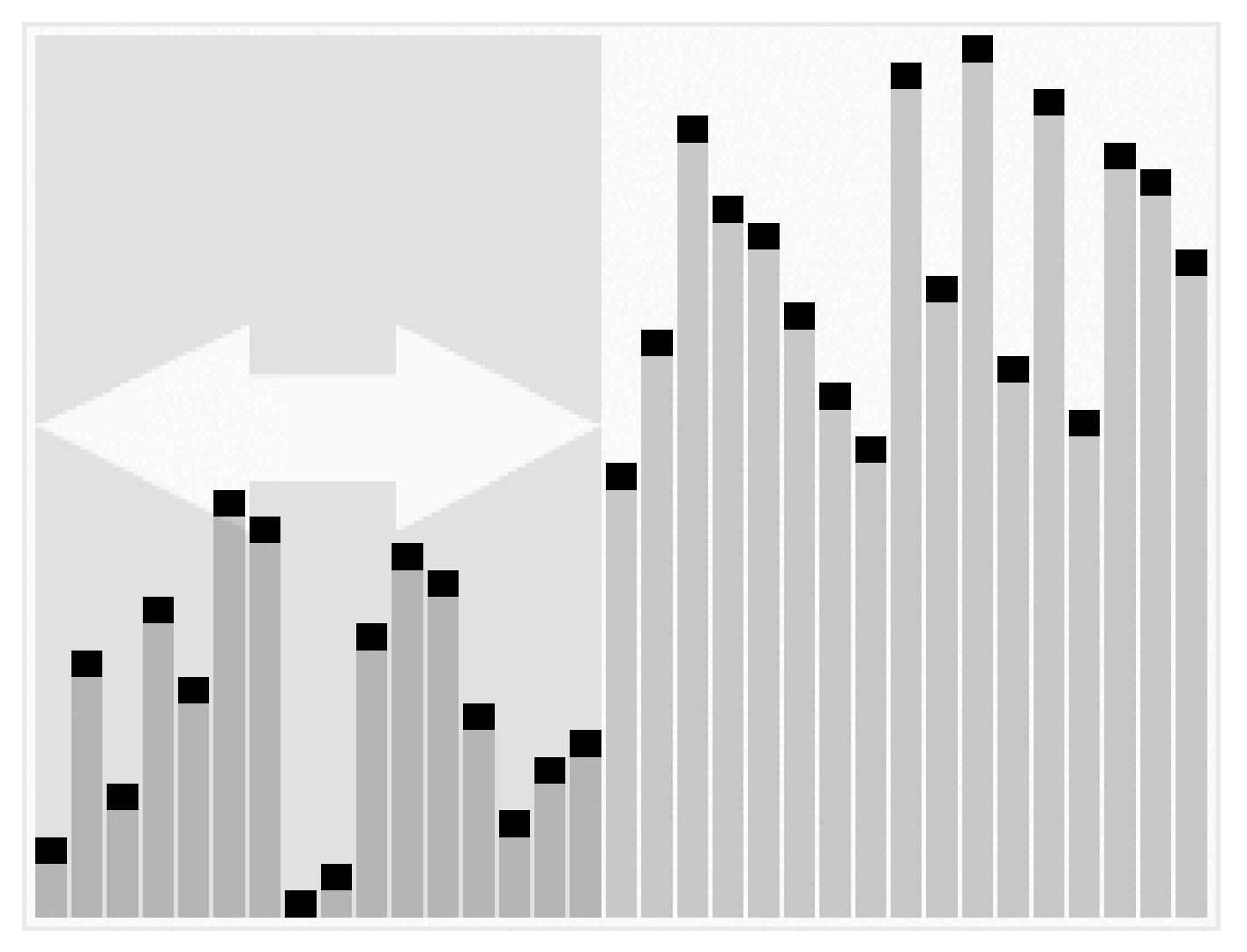


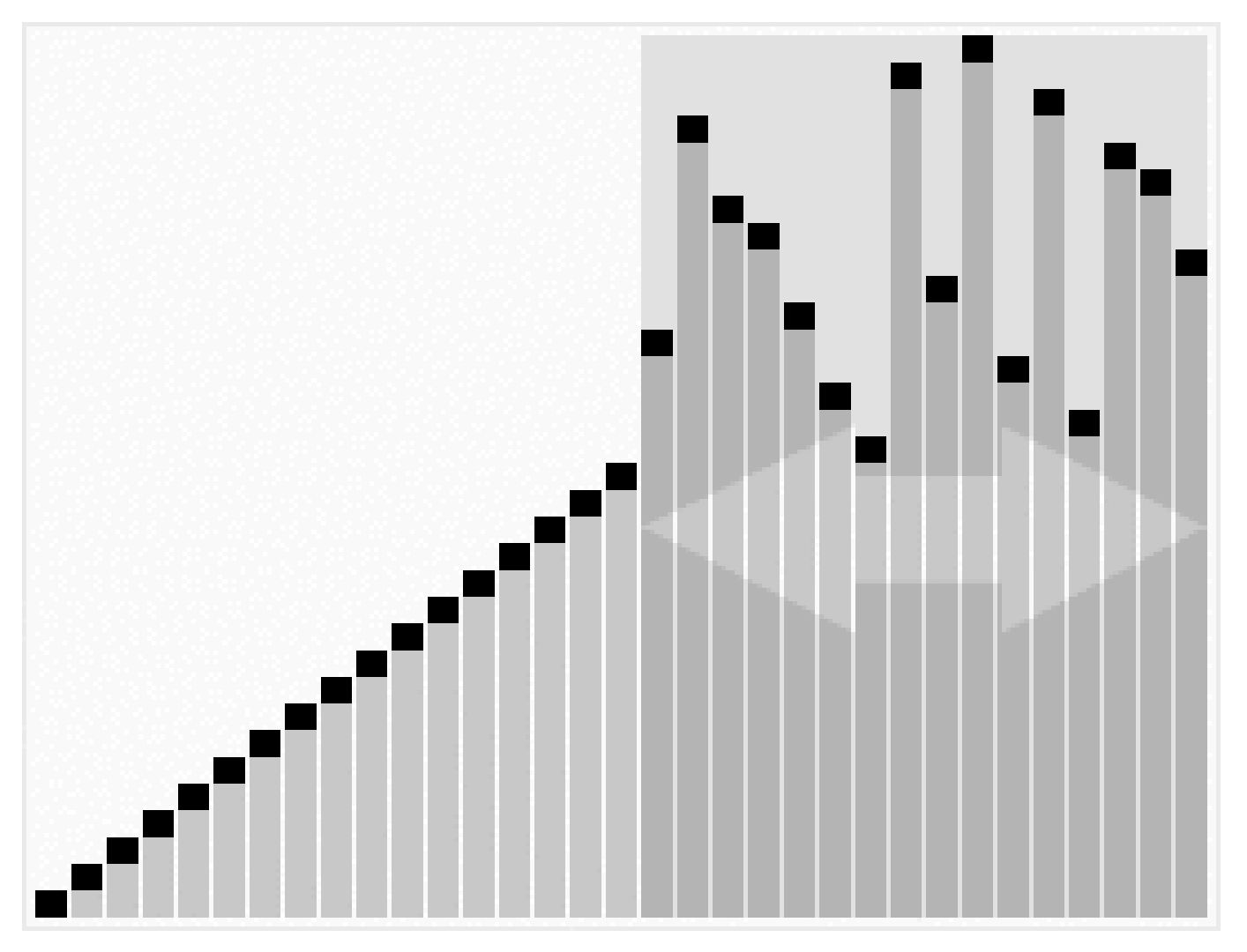


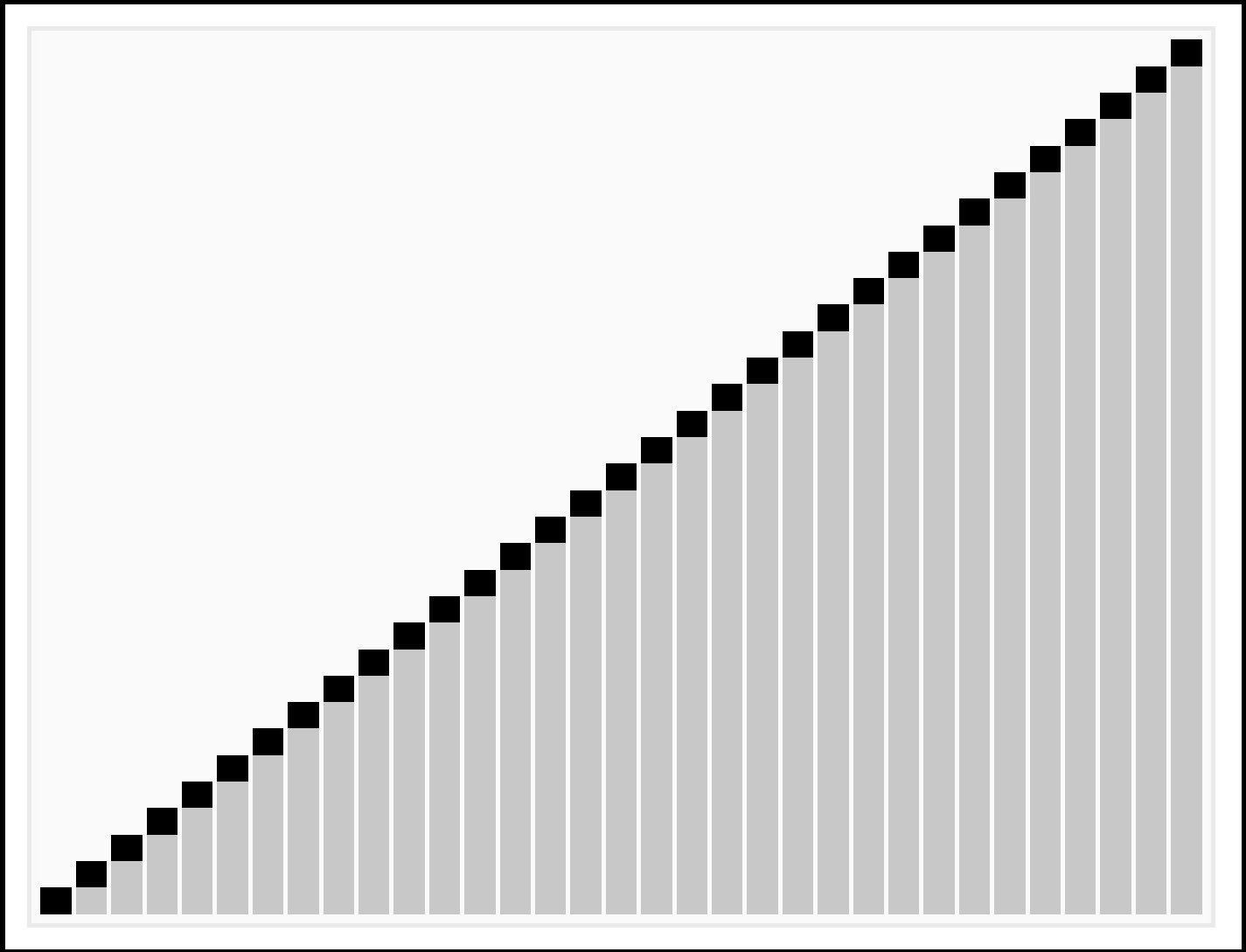












Sorting in practice…

- Base case: use insertion sort on smaller lists
- Quick sort is fastest on randomized lists
- Choosing a pivot (options):
  - Choose first element of list
  - Calculate median
  - Pick 3 elements at random and take median

Java's `Arrays.sort`

- Dual-pivot quicksort!
  - Vladimir Yaroslavskiy, 2009
  - Used in Java's `Arrays.sort` starting with Java 7 (2011)

Comparison Sorts

- Compares elements
- Given items a and b:
  - a > b?
  - a < b?
  - (a == b?)
- All sorting algorithms we've learned so far
  - Selection, bubble, insertion, merge, heap, quick…

Comparison Sorts: Number of Comparisons (Lower Bound)

- Each comparison distinguishes between 2 possibilities
  - $f(n)$ comparisons $\rightarrow 2^{f(n)}$ list permutations
- n elements: $n!$ possible list permutations
  - $2^{f(n)} \geq n!$
  - $f(n) \geq \log_2(n!)$

Comparison Sorts

- Lower bound on number of comparisons: $\log_2(n!)$
- Want to show: $\log_2(n!) \in \Omega(n \log_2 n)$

Want to show: $\log_2(n!) \in \Omega(n \log_2 n)$

- $\log_2(n!)$
  $= \log_2 n + \log_2(n-1) + \cdots + \log_2(2)$
  $> \log_2 n + \log_2(n-1) + \cdots + \log_2 \frac{n}{2}$
  $> \log_2 \frac{n}{2} + \log_2 \frac{n}{2} + \cdots + \log_2 \frac{n}{2} = \frac{n}{2} \log_2 \frac{n}{2}$

Want to show: $\log_2(n!) \in \Omega(n \log_2 n)$

- $\log_2(n!) > \frac{n}{2} \log_2 \frac{n}{2}$
  $\in \Theta(n \log_2 \frac{n}{2})$
  $\in \Theta(n \log_2 n - n \log_2 2)$
  $\in \Theta(n \log_2 n)$
- $\log_2(n!) \in \Omega(n \log_2 n)$
- Lower bound on number of comparisons: $\Omega(n \log_2 n)$

Comparison Sorts

Lower bound on number of comparisons: $\Omega(n \log_2 n)$

Non-comparison Sorts

We can do better

Counting Sort

- Input:
  - Large unsorted list of n integers in the range $[0, k]$ (hopefully $k \ll n$)
- Solution:
  - Create an int array of size k
  - Traverse input list
  - Increment int at array position i whenever you come across i

Counting Sort Example

Input list

4	1	2	1

Counter array

0	0	0	0	0
0	1	2	3	4

Counting Sort Example

Input list

4	1	2	1

Counter array

0	0	0	0	1
0	1	2	3	4

Counting Sort Example

Input list

4	1	2	1

Counter array

0	0	0	0	1
0	1	2	3	4

Counting Sort Example

Input list

4	1	2	1

Counter array

0	1	0	0	1
0	1	2	3	4

Counting Sort Example

Input list

4	1	2	1

Counter array

0	1	0	0	1
0	1	2	3	4

Counting Sort Example

Input list

4	1	2	1

Counter array

0	1	1	0	1
0	1	2	3	4

Counting Sort Example

Input list

4	1	2	1

Counter array

0	1	1	0	1
0	1	2	3	4

Counting Sort Example

Input list

4	1	2	1

Counter array

0	2	1	0	1
0	1	2	3	4

Counting Sort Example

Output:

Counter array

0	2	1	0	1
0	1	2	3	4

Counting Sort Example

Output:

Counter array

0	2	1	0	1
0	1	2	3	4

Counting Sort Example

Output: 1 1

Counter array

0	2	1	0	1
0	1	2	3	4

Counting Sort Example

Output: 1 1

Counter array

0	2	1	0	1
0	1	2	3	4

Counting Sort Example

Output: 1 1 2

Counter array

0	2	1	0	1
0	1	2	3	4

Counting Sort Example

Output: 1 1 2

Counter array

0	2	1	0	1
0	1	2	3	4

Counting Sort Example

Output: 1 1 2

Counter array

0	2	1	0	1
0	1	2	3	4

Counting Sort Example

Output: 1 1 2 4

Counter array

0	2	1	0	1
0	1	2	3	4

Counting Sort Example

Output: 1 1 2 4

Counter array

0	2	1	0	1
0	1	2	3	4

Counting Sort

- Best / average / worst case running time: $O(n + k)$
- Stable? N/A
- In-place? No

Radix Sort (version 1)

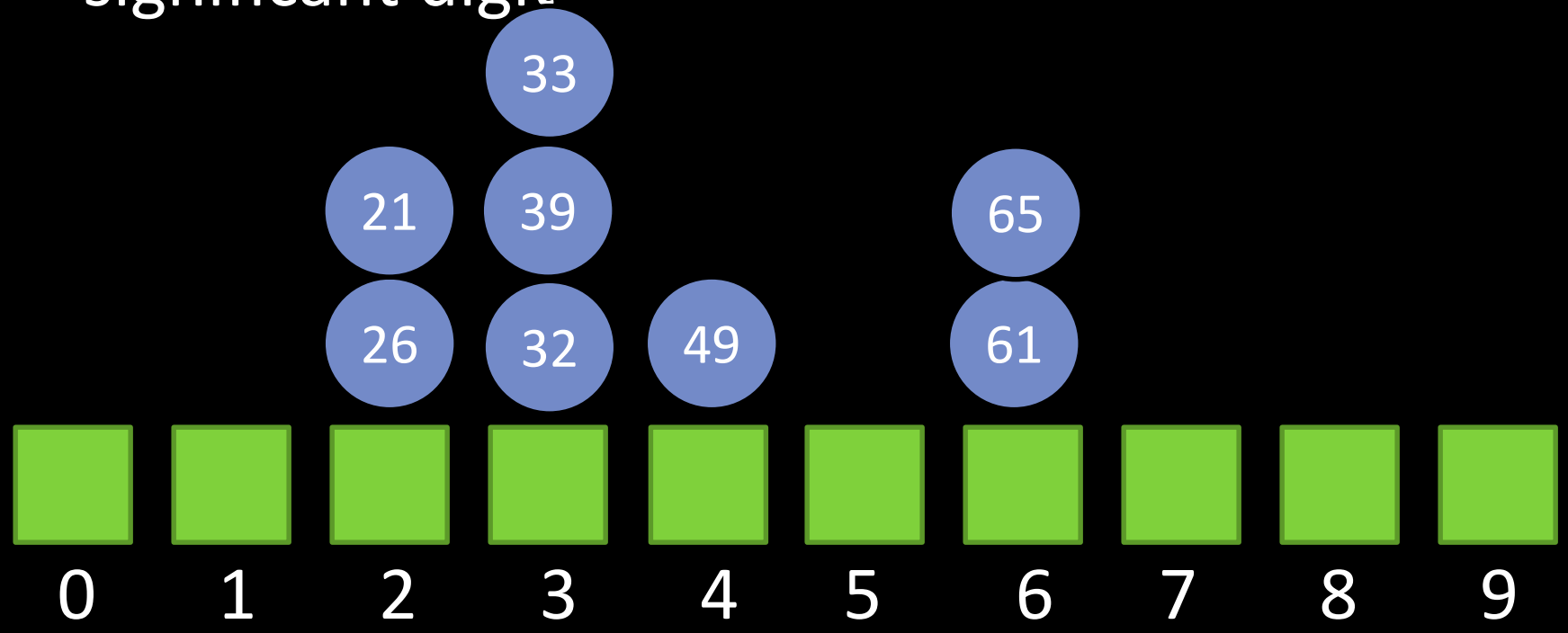
- Base case: list of only 1 element is already sorted
- Put elements into buckets according to <u>most</u> significant digit
- Recursively sort each bucket on next most significant digit (and so forth)

Radix Sort v1 Example

- Input list: 49, 26, 32, 39, 65, 61, 33, 31

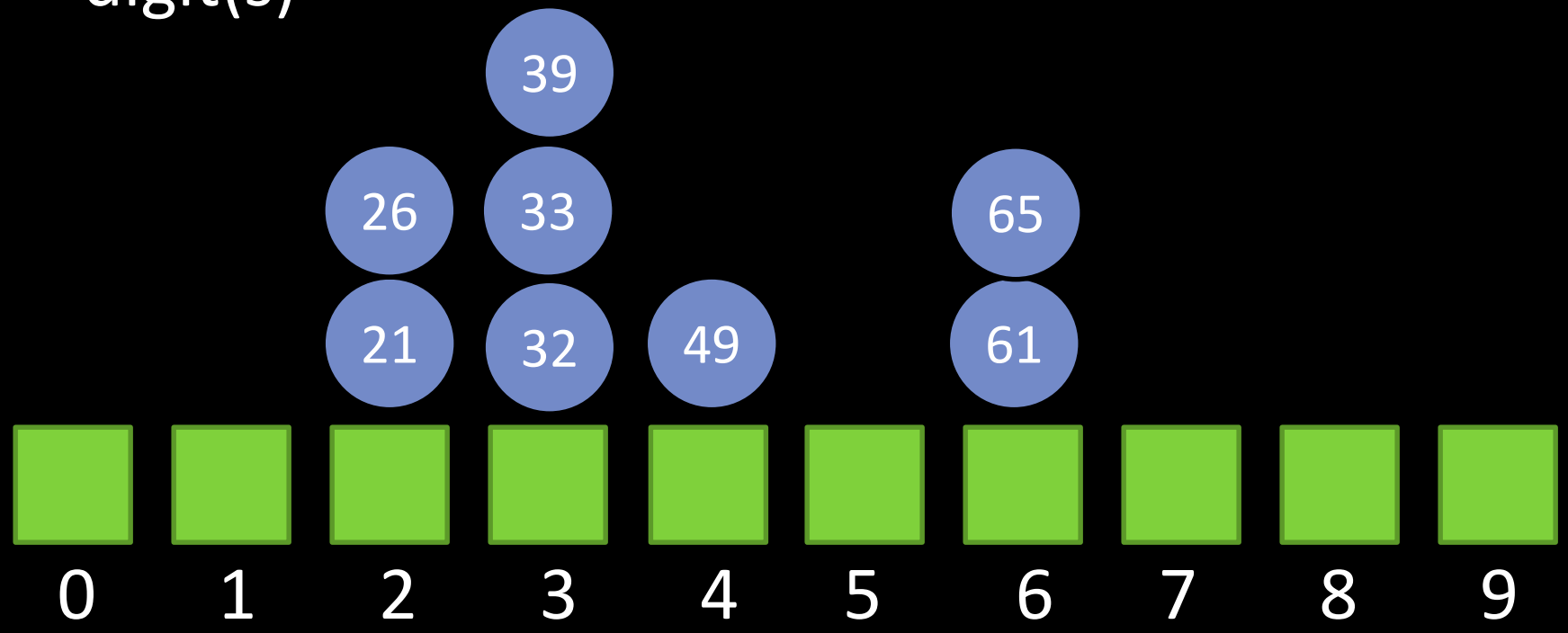
Radix Sort v1 Example

- Input list: 49, 26, 32, 39, 61, 65, 33, 21
- Put elements into buckets according to <u>most</u> significant digit



Radix Sort v1 Example

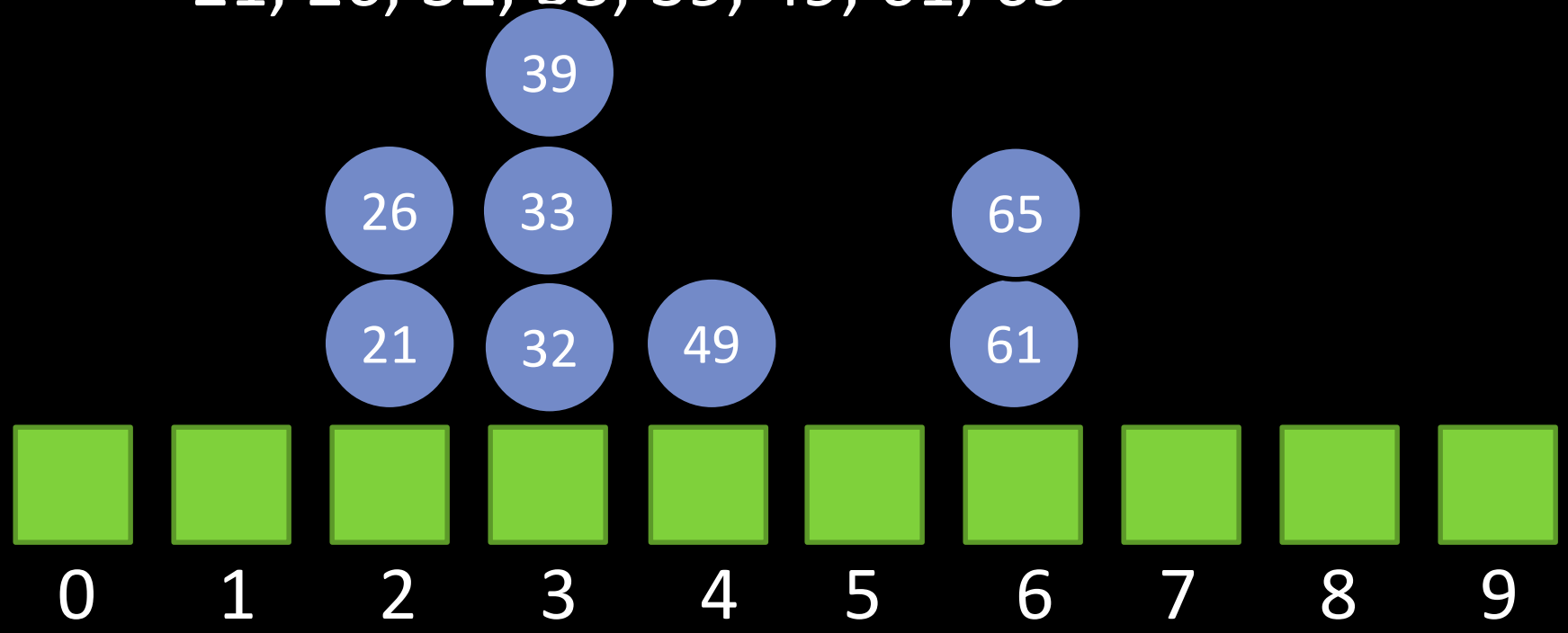
- Input list: 49, 26, 32, 39, 61, 65, 33, 21
- Recursively sort on next most significant digit(s)



Radix Sort v1 Example

- Input list: 49, 26, 32, 39, 61, 65, 33, 21
- Output sorted list:

  21, 26, 32, 33, 39, 49, 61, 65



Radix Sort (version 2)

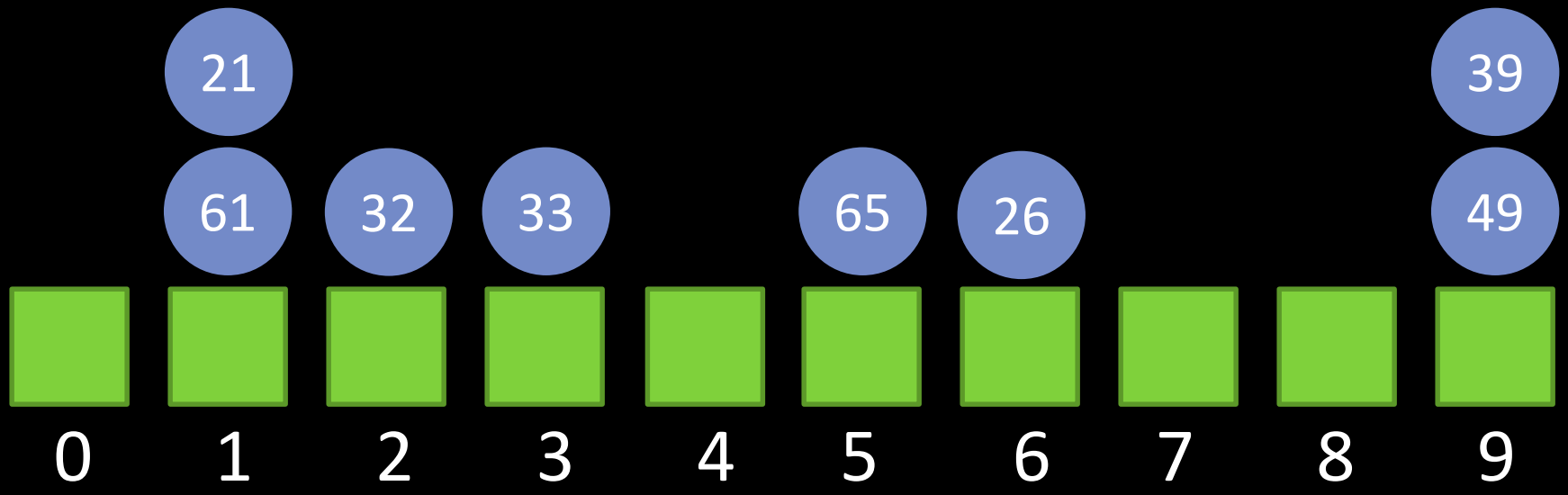
- Base case: list of size 1 is already sorted
- Put elements into buckets according to <u>least</u> significant digit
- Output all of the elements to a list:
  - In bucket order
  - In the order they were added to each bucket
- Repeat with next digit until you reach the most significant digit

Radix Sort v2 Example

- Input list: 49, 26, 32, 39, 61, 65, 33, 21

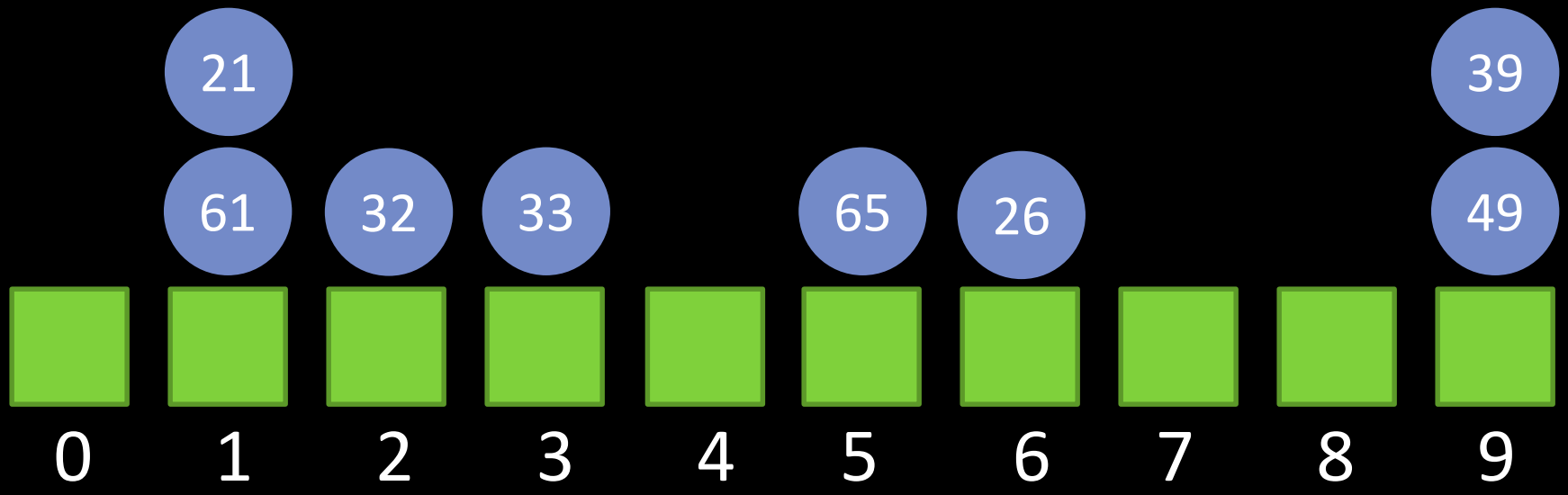
Radix Sort v2 Example

- Input list: 49, 26, 32, 39, 61, 65, 33, 21
- Put elements into buckets according to <u>least</u> significant digit



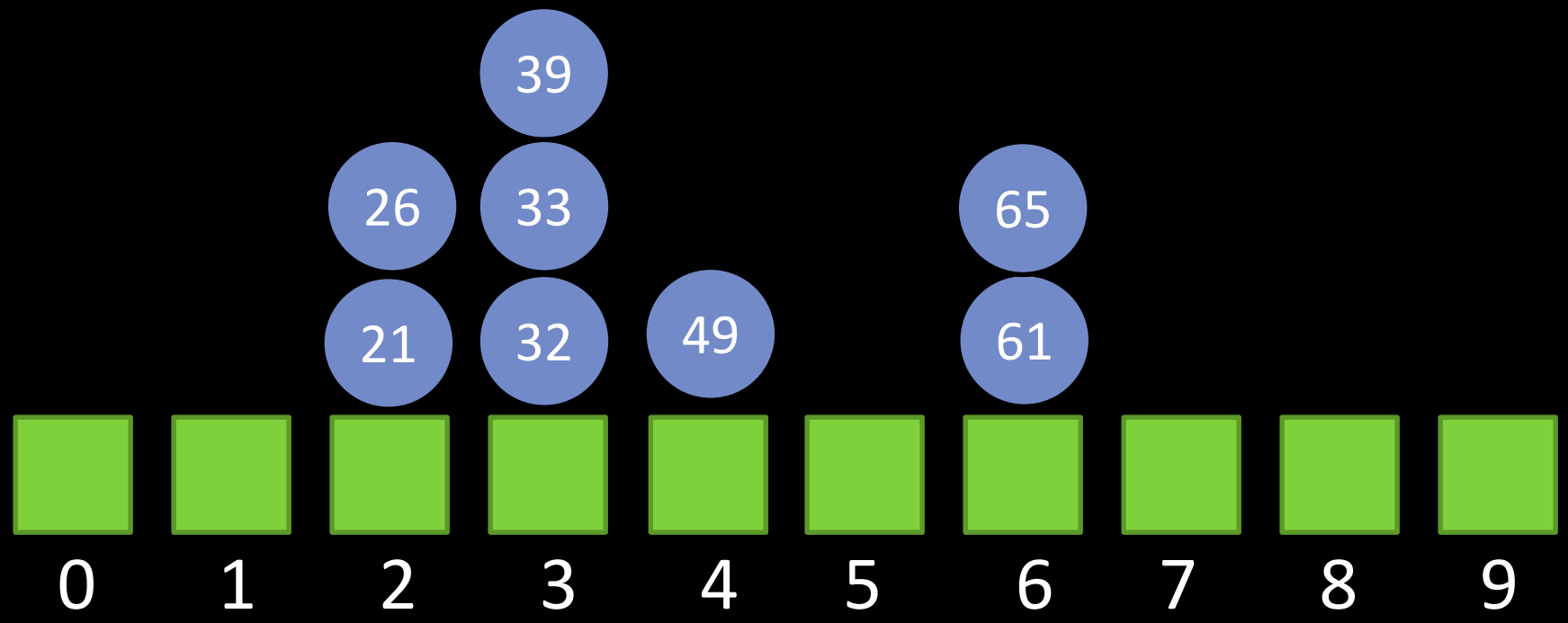
Radix Sort v2 Example

- Output all of the elements to a list:
  - In bucket order
  - In the order they were added to each bucket
- Output: 61, 21, 32, 33, 65, 26, 49, 39



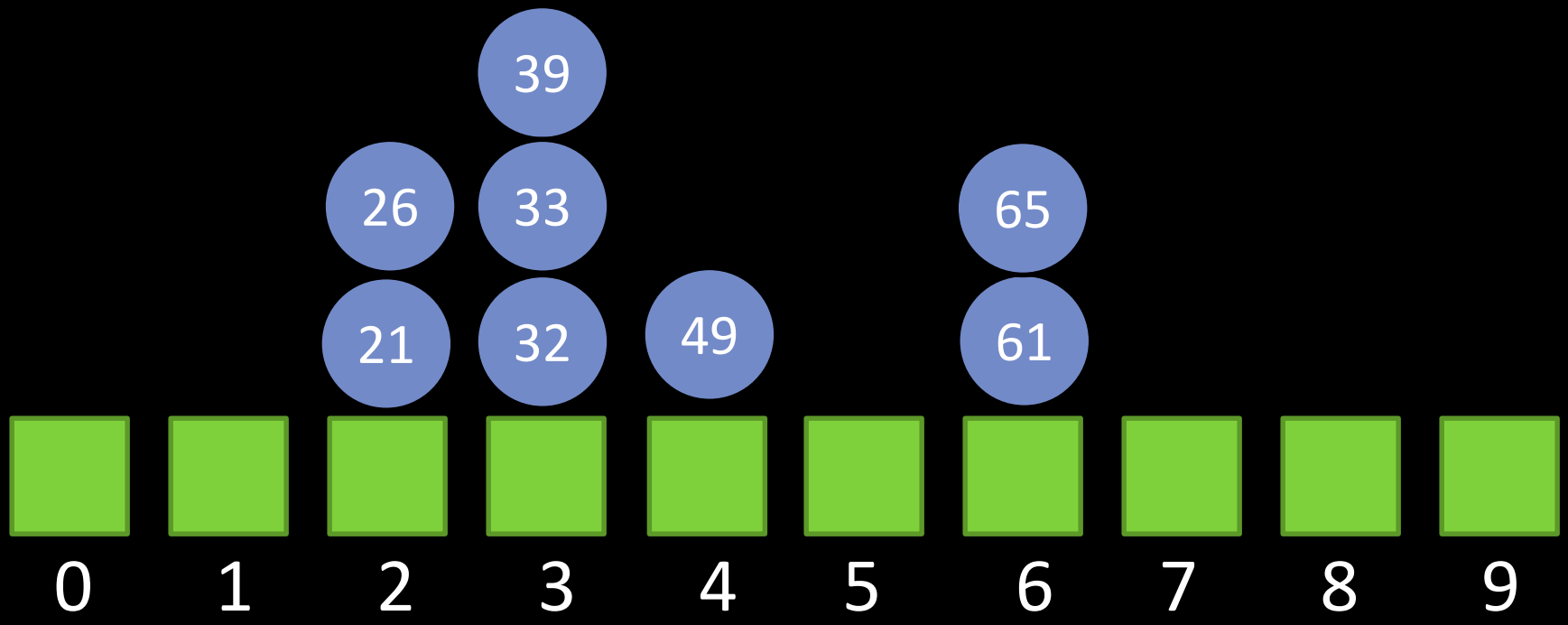
Radix Sort v2 Example

- Input: 61, 21, 32, 33, 65, 26, 49, 39
- Repeat with next digit



Radix Sort v2 Example

- Input: 61, 21, 32, 33, 65, 26, 49, 39
- Output: 21, 26, 32, 33, 39, 49, 61, 65



Radix Sort Running Time

- Best / average / worst case: $O((n + b)k)$
  - n: Number of elements in input list
  - b: Number of buckets
  - k: Number of digits of largest number
- Usually, $b \ll n$, so running time can be reduced to: $O(nk)$
- If k is also small (basically a constant), running time can be reduced to: $O(n)$
  - Sorting Java `int`s: b=2, k = 32

Good luck next week.