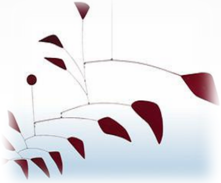


# EECS 3101 - Design and Analysis of Algorithms

**Shahin Kamali**

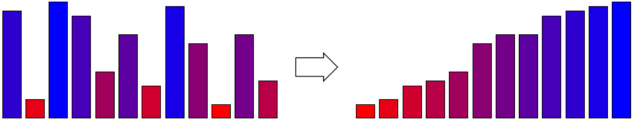
Topic 3 - Sorting

# Overview

- The sorting problem and its significance in practice
- Insertion sort, merge sort and their drawbacks
- Priority queues, heaps, and heapsort
- Comparison based sorting lower bound
- Linear-time sorting

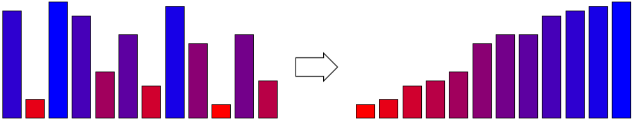
# Sorting

- **Input:**
  - a sequence of $n$ objects: $A[0], \ldots, A[n-1]$ (typically an array or a linked list)
  - a comparison predicate, $\leq$, that defines a total order on $A$
- **Output:**
  - an ordered representation of the objects in A



# Sorting

- **Input:**
  - a sequence of $n$ objects: $A[0], \ldots, A[n-1]$ (typically an array or a linked list)
  - a comparison predicate, $\leq$, that defines a total order on $A$
- **Output:**
  - an ordered representation of the objects in A



- Many sorting algorithms exist: bubble sort, insertion sort, merge sort, heapsort, radix sort, bucket sort, quicksort, etc.

## Significance of Sorting

- Data structures and database: we often **preprocess data** to answer queries faster; many times, preprocessing involves sorting data for faster query operations.

## Significance of Sorting

- Data structures and database: we often **preprocess data** to answer queries faster; many times, preprocessing involves sorting data for faster query operations.
- Sorting has applications in **graphics** and **computational geometry** algorithms (e.g., finding closest pair element uniqueness, etc.)

# Significance of Sorting

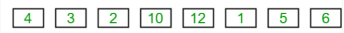
- Data structures and database: we often **preprocess data** to answer queries faster; many times, preprocessing involves sorting data for faster query operations.
- Sorting has applications in **graphics** and **computational geometry** algorithms (e.g., finding closest pair element uniqueness, etc.)
- From a pedagogical point of view, many algorithms exist for sorting; studying them provides great material for learning algorithm design.

# Significance of Sorting

- Data structures and database: we often **preprocess data** to answer queries faster; many times, preprocessing involves sorting data for faster query operations.
- Sorting has applications in **graphics** and **computational geometry** algorithms (e.g., finding closest pair element uniqueness, etc.)
- From a pedagogical point of view, many algorithms exist for sorting; studying them provides great material for learning algorithm design.
- A rough estimate suggest that over 25 percent of all computing time is spent on sorting!

# Insertion Sort

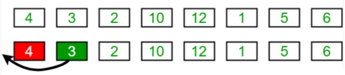
- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
    - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
    - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |

# Insertion Sort

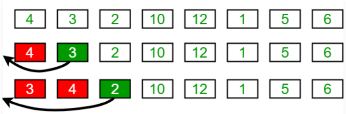
- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |
| 🟥 4 | 🟩 3 | 2 | 10 | 12 | 1 | 5 | 6 |

# Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
    - We know the sub-array $A[0] \dots A[i-1]$ is sorted
    - **Insert** $x$ in its correct position in the sub-array $A[i] \dots A[i]$.



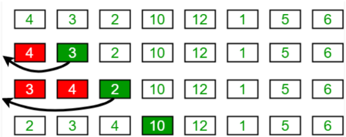
4 3 2 10 12 1 5 6

4 3 2 10 12 1 5 6

3 4 2 10 12 1 5 6

# Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \dots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \dots A[i]$.



4 3 2 10 12 1 5 6

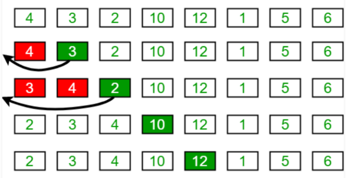
4 3 2 10 12 1 5 6

3 4 2 10 12 1 5 6

2 3 4 10 12 1 5 6

# Insertion Sort

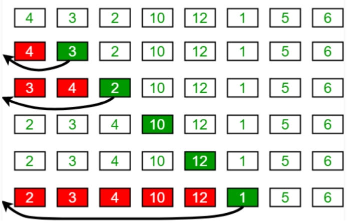
- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \dots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \dots A[i]$.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |
| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |
| 3 | 4 | 2 | 10 | 12 | 1 | 5 | 6 |
| 2 | 3 | 4 | 10 | 12 | 1 | 5 | 6 |
| 2 | 3 | 4 | 10 | 12 | 1 | 5 | 6 |

# Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$.



4 3 2 10 12 1 5 6

4 3 2 10 12 1 5 6

3 4 2 10 12 1 5 6

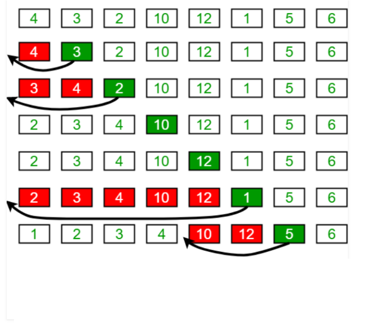
2 3 4 10 12 1 5 6

2 3 4 10 12 1 5 6

2 3 4 10 12 1 5 6

# Insertion Sort

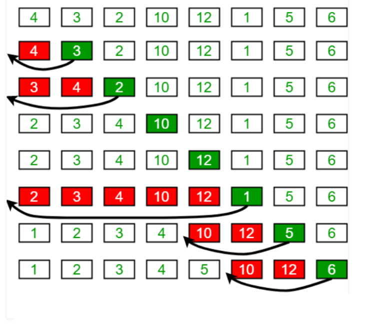
- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \ldots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \ldots A[i]$.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |
| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |
| 3 | 4 | 2 | 10 | 12 | 1 | 5 | 6 |
| 2 | 3 | 4 | 10 | 12 | 1 | 5 | 6 |
| 2 | 3 | 4 | 10 | 12 | 1 | 5 | 6 |
| 2 | 3 | 4 | 10 | 12 | 1 | 5 | 6 |
| 1 | 2 | 3 | 4 | 10 | 12 | 5 | 6 |

# Insertion Sort

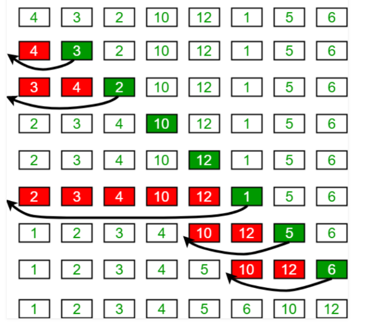
- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
  - We know the sub-array $A[0] \dots A[i-1]$ is sorted
  - **Insert** $x$ in its correct position in the sub-array $A[i] \dots A[i]$.



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |
| 4 | 3 | 2 | 10 | 12 | 1 | 5 | 6 |
| 3 | 4 | 2 | 10 | 12 | 1 | 5 | 6 |
| 2 | 3 | 4 | 10 | 12 | 1 | 5 | 6 |
| 2 | 3 | 4 | 10 | 12 | 1 | 5 | 6 |
| 2 | 3 | 4 | 10 | 12 | 1 | 5 | 6 |
| 1 | 2 | 3 | 4 | 10 | 12 | 5 | 6 |
| 1 | 2 | 3 | 4 | 5 | 10 | 12 | 6 |

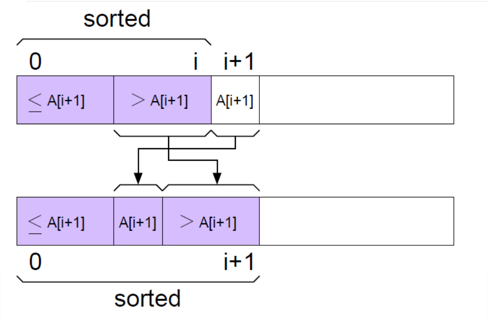
# Insertion Sort

- Go through the items in the array (list) one by one
- For each item $x$ at index $i$:
    - We know the sub-array $A[0] \dots A[i-1]$ is sorted
    - **Insert** $x$ in its correct position in the sub-array $A[i] \dots A[i]$.



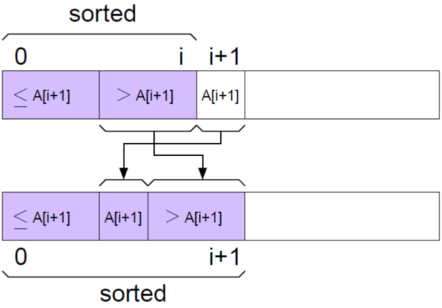
# Insertion Sort Summary

- One Iteration of the Insertion Sort Algorithm:
    - After the $i$th iteration, $A[0..i]$ is sorted.
    - Insert item $A[i+1]$ in its proper place in $A[0..i]$.



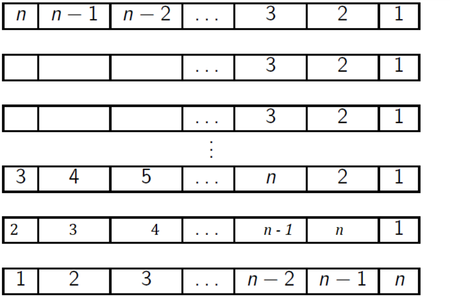
# Insertion Sort Summary

- One Iteration of the Insertion Sort Algorithm:
  - After the $i$th iteration, $A[0..i]$ is sorted.
  - Insert item $A[i+1]$ in its proper place in $A[0..i]$.
- In the **worst case**, $i$ items are moved in the $(i+1)$'th iteration!



# Insertion Sort Analysis

- In the worst case the array is sorted backwards.



|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| $n$ | $n-1$ | $n-2$ | ... | 3 | 2 | 1 |
|  |  |  | ... | 3 | 2 | 1 |
|  |  |  | ... | 3 | 2 | 1 |
|  |  |  | ⋮ |  |  |  |
| 3 | 4 | 5 | ... | $n$ | 2 | 1 |
| 2 | 3 | 4 | ... | $n$ - 1 | $n$ | 1 |
| 1 | 2 | 3 | ... | $n-2$ | $n-1$ | $n$ |

# Insertion Sort Analysis

- In the worst case the array is sorted backwards.

| | | | | | | |
|---|---|---|---|---|---|---|
| $n$ | $n-1$ | $n-2$ | $\dots$ | 3 | 2 | 1 |
| | | | $\dots$ | 3 | 2 | 1 |
| | | | $\dots$ | 3 | 2 | 1 |
| | | | $\vdots$ | | | |
| 3 | 4 | 5 | $\dots$ | $n$ | 2 | 1 |
| 2 | 3 | 4 | $\dots$ | $n$ - 1 | $n$ | 1 |
| 1 | 2 | 3 | $\dots$ | $n-2$ | $n-1$ | $n$ |

- The total number of moved items:

$$1 + 2 + \dots + n - 1 = n(n-1)/2 \in \Theta(n^2)$$

.

# Insertion Sort Time Complexity

- The **worst-case** running time of insertion sort is $\Theta(n^2)$.

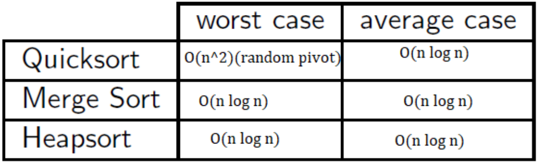
# Insertion Sort Time Complexity

- The **worst-case** running time of insertion sort is $\Theta(n^2)$.
- As it turns out, the **average-case** running time is also $\Theta(n^2)$.
- Faster sorting algorithms exist. These include:

| | worst case | average case |
|---|---|---|
| Quicksort | | |
| Merge Sort | | |
| Heapsort | | |

# Insertion Sort Time Complexity

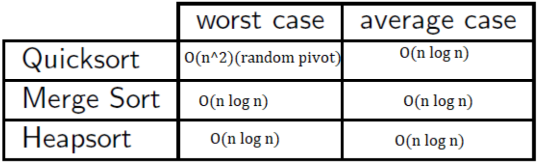
- The **worst-case** running time of insertion sort is $\Theta(n^2)$.
- As it turns out, the **average-case** running time is also $\Theta(n^2)$.
- Faster sorting algorithms exist. These include:



| | worst case | average case |
|---|---|---|
| Quicksort | O(n^2)(random pivot) | O(n log n) |
| Merge Sort | O(n log n) | O(n log n) |
| Heapsort | O(n log n) | O(n log n) |

# Insertion Sort Time Complexity

- The **worst-case** running time of insertion sort is $\Theta(n^2)$.
- As it turns out, the **average-case** running time is also $\Theta(n^2)$.
- Faster sorting algorithms exist. These include:



|  | worst case | average case |
|---|---|---|
| Quicksort | O(n^2)(random pivot) | O(n log n) |
| Merge Sort | O(n log n) | O(n log n) |
| Heapsort | O(n log n) | O(n log n) |

- The lower bound on the worst-case time complexity of any comparison-based sorting algorithm is also $\Omega(n \log n)$.

# Merge Sort

- Merge sort is an example of a **divide-and-conquer** algorithm.
  - **Divide** the input into two or more disjoint subsets.
  - Recursively solve each sub-problem.
  - Combine solutions to the sub-problems to give the solution to the original problem.

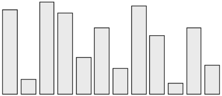
# Merge Sort Algorithm

- **Input:** an array $A[0..n-1]$ of comparable elements.
  - **Divide** $A$ into two subarrays $A[0..\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1, n-1]$
  - **Recursively** sort each sub-array
  - **Combine** the two subarray via merging them

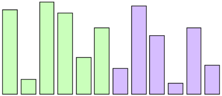
# Merge Sort Algorithm

- **Input:** an array $A[0..n-1]$ of comparable elements.
  - **Divide** $A$ into two subarrays $A[0..\lfloor n/2 \rfloor]$ and $A[\lfloor n/2 \rfloor + 1, n-1]$
  - **Recursively** sort each sub-array
  - **Combine** the two subarray via merging them
- The base of recursion is an array of size 1 which is sorted
  - In practice, when the length of sub-array is less than 100, selection sort is applied.

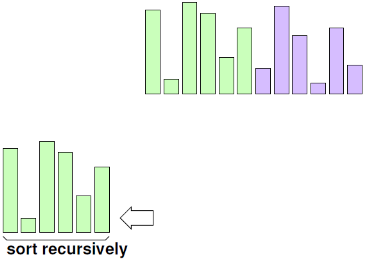
# Merge Sort Scheme



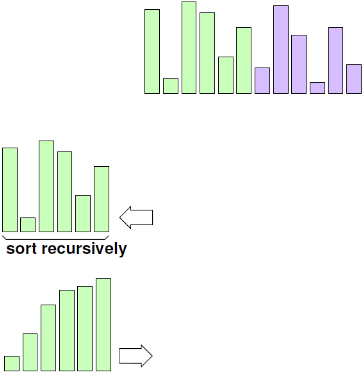
# Merge Sort Scheme



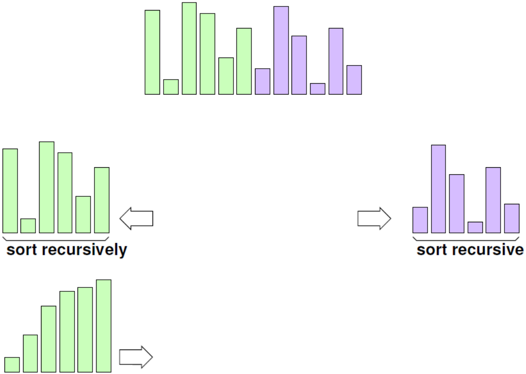
# Merge Sort Scheme



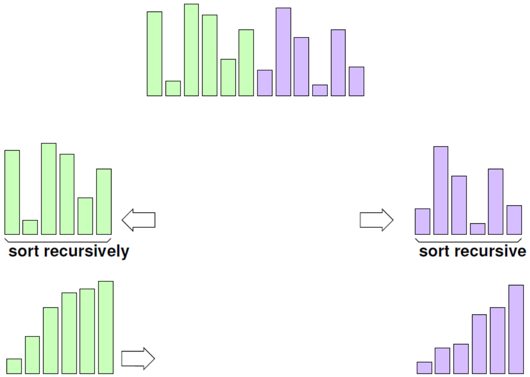
# Merge Sort Scheme



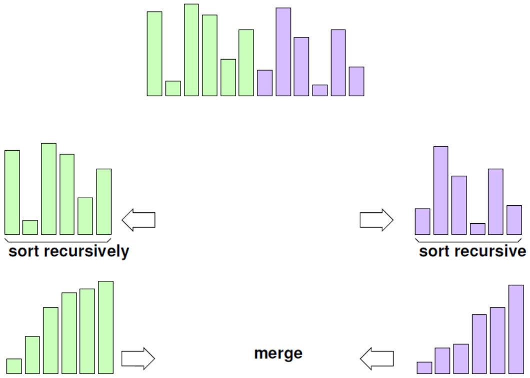
# Merge Sort Scheme



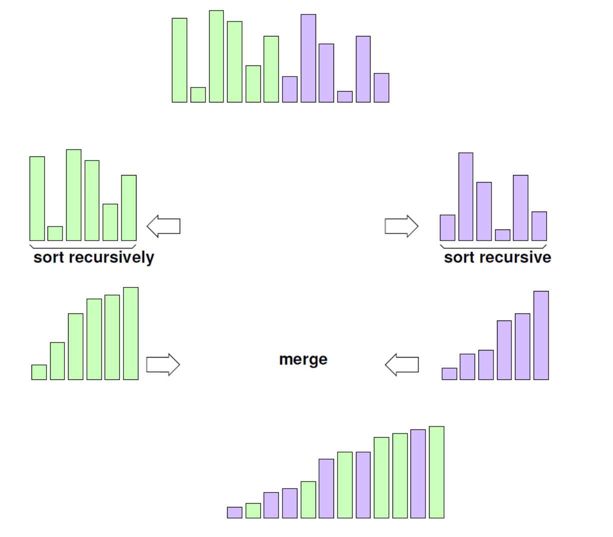
# Merge Sort Scheme



# Merge Sort Scheme



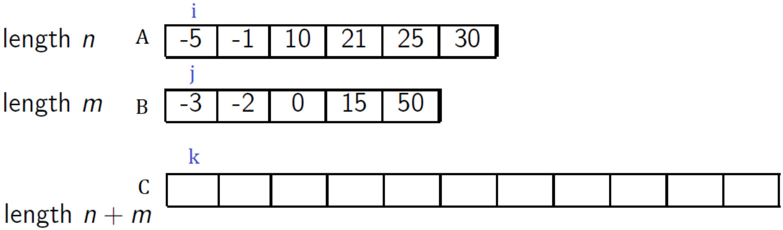
# Merge Sort Scheme



# Merging Sorted Sub-arrays

- Given two sorted arrays $A$ and $B$ of size $n$ and $m$, merge them into array $C$ of size $m + n$.
    - $i$, $j$, and $k$ are three indices moving on $A, B$, and $C$.
    - They are initially 0.
- At each step, copy the smaller of $A[i]$ and $B[j]$ to $C[k]$.
    - Increment $k$
    - If $A[i]$ is copied, increment $i$; otherwise increment $j$.
- If one array ends ($i = n$ or $j = m$), copy the remaining items of the other array to $C$.

# Merge Example



i

length $n$ A

| -5 | -1 | 10 | 21 | 25 | 30 |
|---|---|---|---|---|---|

j

length $m$ B

| -3 | -2 | 0 | 15 | 50 |
|---|---|---|---|---|

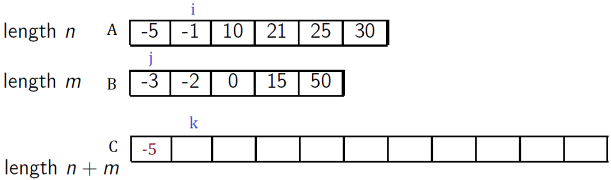
k

C

|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|

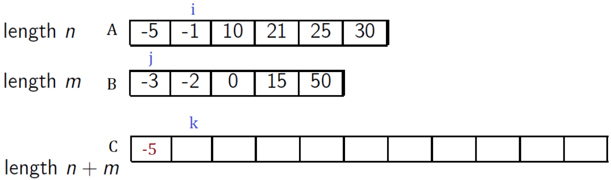
length $n + m$

# Merge Example



| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | i | | | | | | | | | |
| length $n$ | A | -5 | -1 | 10 | 21 | 25 | 30 | | | | | |
| | | j | | | | | | | | | | |
| length $m$ | B | -3 | -2 | 0 | 15 | 50 | | | | | | |
| | | | k | | | | | | | | | |
| length $n+m$ | C | -5 | | | | | | | | | | |

# Merge Example

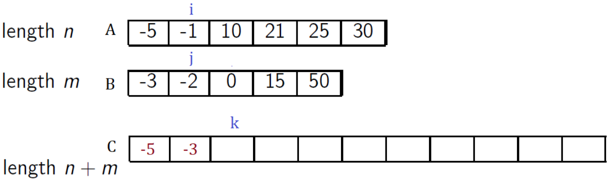


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | i | | | | |
| length $n$ | A | -5 | -1 | 10 | 21 | 25 | 30 |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | j | | | | |
| length $m$ | B | -3 | -2 | 0 | 15 | 50 |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | k | | | | | | | | | |
| length $n+m$ | C | -5 | | | | | | | | | | |

# Merge Example



length $n$ A

| | i | | | | |
|---|---|---|---|---|---|
| -5 | -1 | 10 | 21 | 25 | 30 |

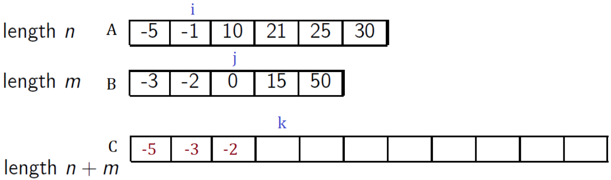
length $m$ B

| | j | | | |
|---|---|---|---|---|
| -3 | -2 | 0 | 15 | 50 |

C length $n + m$

| | | k | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -5 | -3 | | | | | | | | | |

# Merge Example



length $n$ A

|  | i |  |  |  |  |
|---|---|---|---|---|---|
| -5 | -1 | 10 | 21 | 25 | 30 |

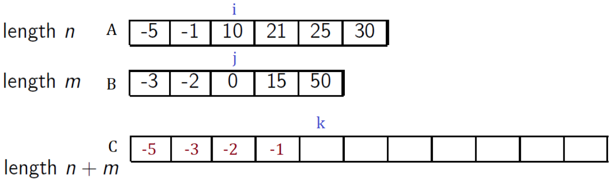
length $m$ B

|  |  | j |  |  |
|---|---|---|---|---|
| -3 | -2 | 0 | 15 | 50 |

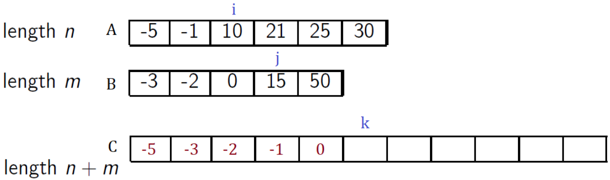
C length $n + m$

|  |  |  | k |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| -5 | -3 | -2 |  |  |  |  |  |  |  |  |

# Merge Example



# Merge Example



length $n$ A

|  |  | i |  |  |  |
|---|---|---|---|---|---|
| -5 | -1 | 10 | 21 | 25 | 30 |

length $m$ B

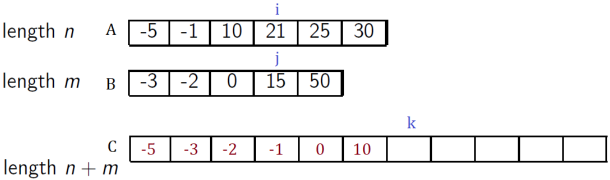
|  |  |  | j |  |
|---|---|---|---|---|
| -3 | -2 | 0 | 15 | 50 |

C

length $n + m$

|  |  |  |  |  | k |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| -5 | -3 | -2 | -1 | 0 |  |  |  |  |  |  |

# Merge Example



length $n$ A

|  |  |  |  |  |  |
|---|---|---|---|---|---|
| -5 | -1 | 10 | 21 | 25 | 30 |

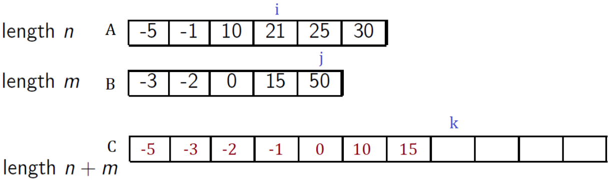
length $m$ B

|  |  |  |  |  |
|---|---|---|---|---|
| -3 | -2 | 0 | 15 | 50 |

C
length $n + m$

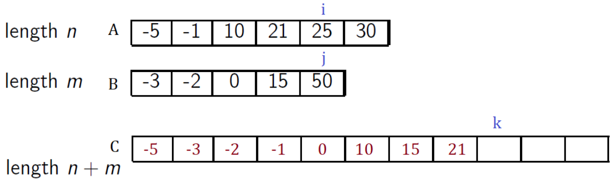
|  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|
| -5 | -3 | -2 | -1 | 0 | 10 |  |  |  |  |  |

# Merge Example



|  |  |  |  |  |  |  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  | i |  |  |  |  |  |  |  |  |
| length $n$ | A | -5 | -1 | 10 | 21 | 25 | 30 |  |  |  |  |  |
|  |  |  |  |  | j |  |  |  |  |  |  |  |
| length $m$ | B | -3 | -2 | 0 | 15 | 50 |  |  |  |  |  |  |
|  |  |  |  |  |  |  |  | k |  |  |  |  |
| length $n + m$ | C | -5 | -3 | -2 | -1 | 0 | 10 | 15 |  |  |  |  |

# Merge Example



i

length $n$ A: -5, -1, 10, 21, 25, 30

j

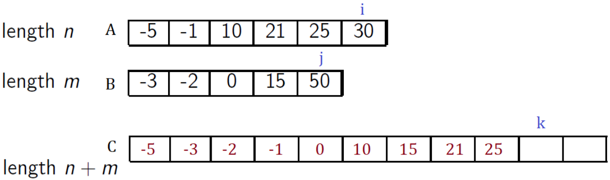
length $m$ B: -3, -2, 0, 15, 50

k

C: -5, -3, -2, -1, 0, 10, 15, 21, , ,

length $n + m$

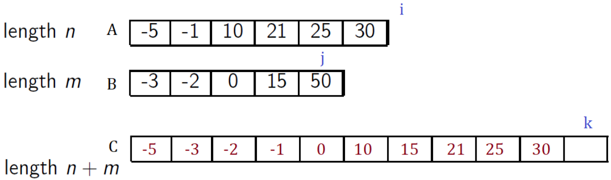
# Merge Example



| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | | | | | | i |
| length $n$ | A | -5 | -1 | 10 | 21 | 25 | 30 |
| | | | | | | j | |
| length $m$ | B | -3 | -2 | 0 | 15 | 50 | |

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | | | k | |
| length $n + m$ | C | -5 | -3 | -2 | -1 | 0 | 10 | 15 | 21 | 25 | | |

# Merge Example



i

length $n$ A

| -5 | -1 | 10 | 21 | 25 | 30 |
|---|---|---|---|---|---|

j

length $m$ B

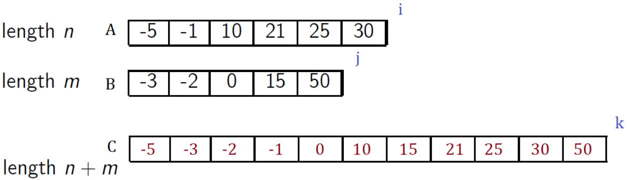
| -3 | -2 | 0 | 15 | 50 |
|---|---|---|---|---|

k

C length $n + m$

| -5 | -3 | -2 | -1 | 0 | 10 | 15 | 21 | 25 | 30 | |
|---|---|---|---|---|---|---|---|---|---|---|

# Merge Example



i

length $n$ A | -5 | -1 | 10 | 21 | 25 | 30 |

j

length $m$ B | -3 | -2 | 0 | 15 | 50 |

k

C | -5 | -3 | -2 | -1 | 0 | 10 | 15 | 21 | 25 | 30 | 50 |

length $n + m$

# Merge Sort Summary

- Recursively sort the left half of the input array $A$
- Recursively sort the left half of the input array $B$
- Merge the two sub-arrays into a new one
  - note that merging requires a new array, that is, it cannot be done **in place**.

# Time complexity of merge sort

- What is the time complexity of merge-sort?

$$T(n) = \begin{cases} 1, & n \leq 1 \\ 2T(n/2) + O(n) & n \geq 2 \end{cases}$$

- We solve this using replacement method to get $T(n) \in O(n \log n)$.

# Analysis of Merge Sort

$$T(n) = \begin{cases} 1, & n \le 1 \\ 2T(n/2) + O(n) & n \ge 2 \end{cases}$$

As we saw earlier, we have $T(n) = \Theta(n \log n)$ (case 2 of Master theorem).

# Should we use merge sort?

- Unlike insertion sort, merge sort does not work in place.
- Merging two arrays requires temporary storage.

# Should we use merge sort?

- Unlike insertion sort, merge sort does not work in place.
- Merging two arrays requires temporary storage.

In practice, we prefer Heap Sort and Quick Sort over Merge Sort.

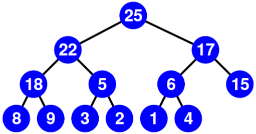
# HeapSort Preliminaries: Arrays vs. Dynamic Allocation

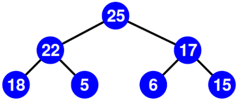
- Typically, a tree is implemented using dynamic allocation of nodes. This is both:
  - more efficient with respect to memory usage, and
  - allows for faster operations.
- It is, however, also possible to implement a tree using an array. Practical only in restricted cases:
  - binary tree is complete
  - maximum size is known
  - only insert/delete rightmost leaf on bottom level

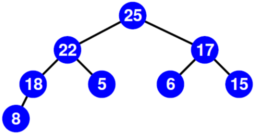
# Complete Binary Trees

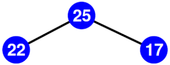
**Definition**

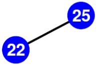
A complete binary tree is a binary tree in which every level is full, except possibly the bottom level. In the bottom level the nodes are in the leftmost positions.





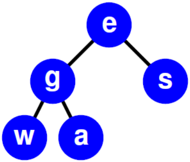




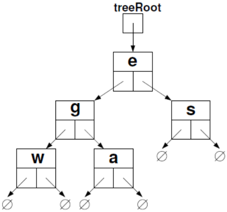


# Array Implementation of Binary Trees

- Simply write nodes level-by-level from left to right into an array.



T



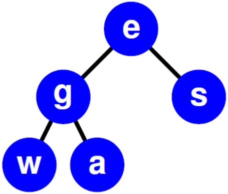
Dynamic Allocation

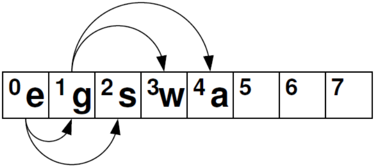
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| e | g | s | w | a |  |  |  |

Array Implementation

# Array Implementation of Binary Trees

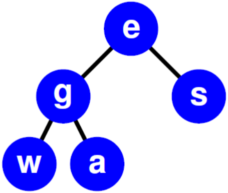
- When using an array to implement a complete tree:
    - The root is stored at $A[0]$.
    - The left child of the root is stored at $A[1]$.
    - The right child of the root is stored at $A[2]$.
    - The left child of $A[1]$ is stored at $A[3]$.
    - The right child of $A[1]$ is stored at $A[4]$.

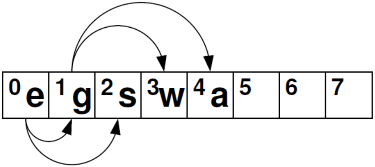




# Array Implementation of Binary Trees

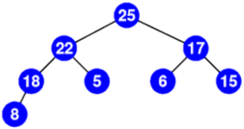
- In general:
  - The **left child** of the node at index i is stored at A[2i + 1].
  - The **right child** of the node at index i is stored at A[2i + 2].
  - The **parent** of the node at index i is stored at $A[\lceil i/2 \rceil - 1]$.



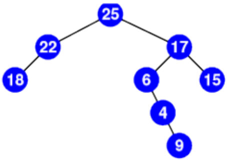


# Array Implementation of Binary Trees

- An array implementation can work for storing any tree



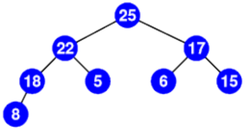
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 25 | 22 | 17 | 18 | 5 | 6 | 15 | 8 |



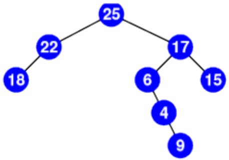
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 25 | 22 | 17 | 18 | | 6 | 15 | | | | | | 4 | | ••• | | 9 |

# Array Implementation of Binary Trees

- An array implementation can work for storing any tree
  - But if the tree is not complete or nearly complete, the memory will be wasted and the insert/delete take time as bad as $O(n)$.



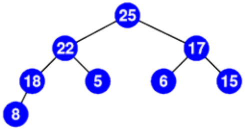
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 25 | 22 | 17 | 18 | 5 | 6 | 15 | 8 |



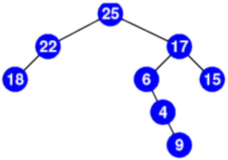
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 25 | 22 | 17 | 18 | | 6 | 15 | | | | | | 4 | | ••• | | 9 |

# Array Implementation of Binary Trees

- An array implementation can work for storing any tree
  - But if the tree is not complete or nearly complete, the memory will be wasted and the insert/delete take time as bad as $O(n)$.
- We often assume the array is complete and insert/delete take place at the last index → takes $O(1)$.



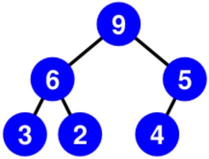
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 25 | 22 | 17 | 18 | 5 | 6 | 15 | 8 |



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | | 25 | 26 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 25 | 22 | 17 | 18 | | 6 | 15 | | | | | | 4 | | ••• | | 9 |

# Binary Heaps

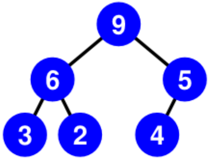
- A **maximum binary heap** is a complete binary tree such that the key of each node is less than or equal to the key of its parent, i.e., for any node $a$ in the tree, $key(a) \leq key(parent(a))$.
  - The heap property is recursive: each node's left and right subtrees are also maximum binary heaps.



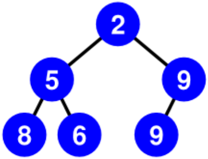
a maximum binary heap

# Binary Heaps

- A **maximum binary heap** is a complete binary tree such that the key of each node is less than or equal to the key of its parent, i.e., for any node $a$ in the tree, $key(a) \leq key(parent(a))$.
  - The heap property is recursive: each node's left and right subtrees are also maximum binary heaps.
  - If the opposite property holds then the data structure is a minimum binary heap.



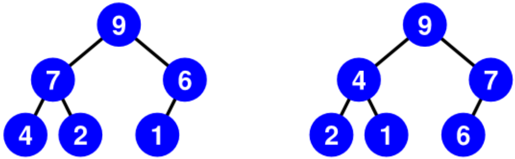
a maximum binary heap



a minimum binary heap

# Non-uniqueness of heaps

- Consider the following two heaps:
  - Both trees are complete binary trees.
  - Both trees store the same keys.
  - Both trees satisfy the maximum binary heap property.
  - The positions of some keys differ in the two trees.
  - This scenario is not possible for binary search trees.



# Heap's Application

- Heaps are used to implement **priority queues**.
- We want to maintain a collection of elements, each of which has a key corresponding to its **priority**:

# Heap's Application

- Heaps are used to implement **priority queues**.
- We want to maintain a collection of elements, each of which has a key corresponding to its **priority**:
    - `Insert(key)`: insert a new item to the queue
    - `extractMax()`: remove and return the item with max key (priority)
    - `getMax()`: return the item with max key (priority)

# Heap's Application

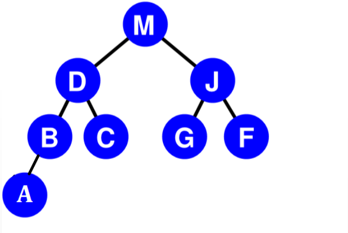
- Heaps are used to implement **priority queues**.
- We want to maintain a collection of elements, each of which has a key corresponding to its **priority**:
  - `Insert(key):` insert a new item to the queue
  - `extractMax()`: remove and return the item with max key (priority)
  - `getMax():` return the item with max key (priority)
  - other operations include isEmpty() and size();

# Heap's Application

- Heaps are used to implement **priority queues**.
- We want to maintain a collection of elements, each of which has a key corresponding to its **priority**:
  - `Insert(key)`: insert a new item to the queue
  - `extractMax()`: remove and return the item with max key (priority)
  - `getMax()`: return the item with max key (priority)
  - other operations include isEmpty() and size();
- We will show how to use a heap to implement these operations.

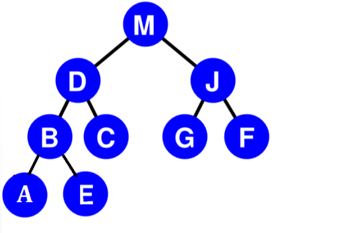
## Insertion in Heaps

- A new element is inserted as the next empty leaf in the complete binary tree.
- A **reheapUp** operation is performed on the new node:
  - compare the new item with its parent
  - if its key is larger than its parent's key, swap the two nodes
  - continue recursively upwards
- E.g.: insert(E)



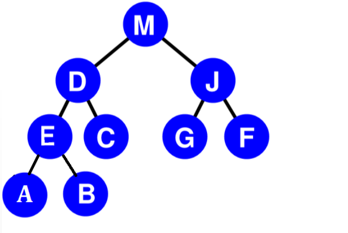
# Insertion in Heaps

- A new element is inserted as the next empty leaf in the complete binary tree.
- A **reheapUp** operation is performed on the new node:
  - compare the new item with its parent
  - if its key is larger than its parent's key, swap the two nodes
  - continue recursively upwards
- E.g.: insert(E)



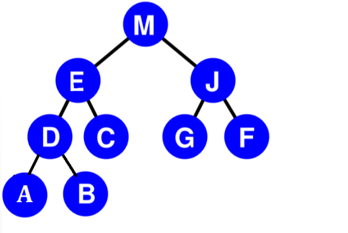
# Insertion in Heaps

- A new element is inserted as the next empty leaf in the complete binary tree.
- A **reheapUp** operation is performed on the new node:
  - compare the new item with its parent
  - if its key is larger than its parent's key, swap the two nodes
  - continue recursively upwards
- E.g.: insert(E)



# Insertion in Heaps

- A new element is inserted as the next empty leaf in the complete binary tree.
- A **reheapUp** operation is performed on the new node:
  - compare the new item with its parent
  - if its key is larger than its parent's key, swap the two nodes
  - continue recursively upwards
- E.g.: insert(E)



# Insertion into a Heap

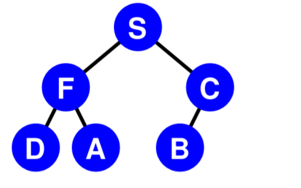
- `insert` is a public method that calls the private `reheapUp`

```
insert ($key$)
    $size \leftarrow$ no. elements in the heap
    $heap(size) \leftarrow key$
    reheapUp($size$)
```

```
reheapUp ($index$)
    $parent \leftarrow (index + 1)/2 - 1$
    if $index > 0$ and $heap[parent] < heap[key]$
        swap($parent, index$)
        reheapUp($parent$)
```

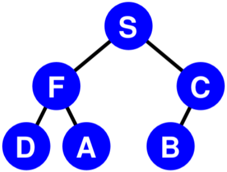
# ExtractMax

- We want to remove the item of highest priority and return it.
- An item of highest priority is always located at the root of the tree.
  - Copy the item at the root to return it later.
  - Take the rightmost element on the bottom level of the tree (the last item currently in the array) and move it to the root. This preserves the structure of the complete binary tree but the heap ordering property is lost.
  - Perform a **reheapDown** operation on the root.
    - If necessary, swap the current node (initially the root) with its largest child, and repeat!



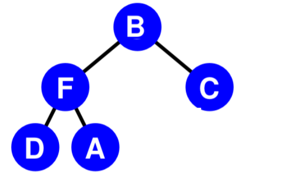
# ExtractMax

- We want to remove the item of highest priority and return it.
- An item of highest priority is always located at the root of the tree.
    - Copy the item at the root to return it later.
    - Take the rightmost element on the bottom level of the tree (the last item currently in the array) and move it to the root. This preserves the structure of the complete binary tree but the heap ordering property is lost.
    - Perform a **reheapDown** operation on the root.
        - If necessary, swap the current node (initially the root) with its largest child, and repeat!



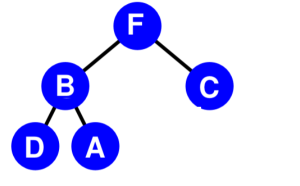
# ExtractMax

- We want to remove the item of highest priority and return it.
- An item of highest priority is always located at the root of the tree.
  - Copy the item at the root to return it later.
  - Take the rightmost element on the bottom level of the tree (the last item currently in the array) and move it to the root. This preserves the structure of the complete binary tree but the heap ordering property is lost.
  - Perform a **reheapDown** operation on the root.
    - If necessary, swap the current node (initially the root) with its largest child, and repeat!



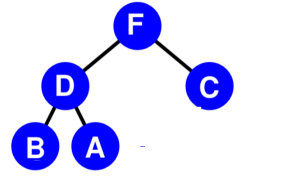
# ExtractMax

- We want to remove the item of highest priority and return it.
- An item of highest priority is always located at the root of the tree.
  - Copy the item at the root to return it later.
  - Take the rightmost element on the bottom level of the tree (the last item currently in the array) and move it to the root. This preserves the structure of the complete binary tree but the heap ordering property is lost.
  - Perform a **reheapDown** operation on the root.
    - If necessary, swap the current node (initially the root) with its largest child, and repeat!



# ExtractMax

- We want to remove the item of highest priority and return it.
- An item of highest priority is always located at the root of the tree.
    - Copy the item at the root to return it later.
    - Take the rightmost element on the bottom level of the tree (the last item currently in the array) and move it to the root. This preserves the structure of the complete binary tree but the heap ordering property is lost.
    - Perform a **reheapDown** operation on the root.
        - If necessary, swap the current node (initially the root) with its largest child, and repeat!



# Extracting the max item from a Heap

```
extractMax ()
    $result \leftarrow$ null
    if no. elements in the heap $> 0$
        $result \leftarrow heap[0]$
        swap ($heap[0], heap[size]$)
        $size \leftarrow size - 1$
        reheapDown(0);
    return $result$
```

# Extracting the max item from a Heap

```
extractMax ()
    $result \leftarrow$ null
    if no. elements in the heap $> 0$
        $result \leftarrow heap[0]$
        swap ($heap[0], heap[size]$)
        $size \leftarrow size - 1$
        reheapDown(0);
    return $result$
```

```
reheapDown (top)
    $left \leftarrow 2 \times top + 1$
    $right \leftarrow 2 \times top + 2$
    $size \leftarrow$ no. elements in the heap
    if ($left < size$)
        if ($right \geq size$ or $heap[left] > heap[right]$)
            $maxChild \leftarrow left$
        else $maxChild \leftarrow right$
    if ($heap[top] < heap[maxChild]$)
        swap($top, maxChild$)
        reheapDown($maxChild$)
```

# Time Complexity

- What is the time Complexity of ReheapUp and ReheapDow?
- When performing either a reheapUp or reheapDown operation, the number of steps depends on the **height** of the tree.
    - In the worst case a single path from root to leaf is traversed.
- A complete binary tree of height $h$ has between $2^h$ and $2^{h+1} - 1$ nodes.
    - Therefore, a complete binary tree with n nodes has height between $\log(n+1) - 1$ and $\log n$
    - Hence, **the time complexity of reheapUp and reheapDown is $\Theta(\log n)$ in the worst case**.

# Time Complexity

- What is time Complexity of Insert and ExtractMax?

# Time Complexity

- What is time Complexity of Insert and ExtractMax?
- Inserting an item into the priority queue requires one call to reheapUp which takes $\Theta(\log n)$ time in the worst case.
- Removing the maximum item from the priority queue requires one call to reheapDown which takes $\Theta(\log n)$ time in the worst case.

# Time Complexity

- What is time Complexity of Insert and ExtractMax?
- Inserting an item into the priority queue requires one call to reheapUp which takes $\Theta(\log n)$ time in the worst case.
- Removing the maximum item from the priority queue requires one call to reheapDown which takes $\Theta(\log n)$ time in the worst case.
- **Insert and Extract-Max operations in a heap take $O(\log n)$ time.**

# Builiding a Heap

- Given an arbitrary array, how can we make it into a heap? Similarly, given an arbitrary complete binary tree, how can we make it into a heap?

# Builiding a Heap

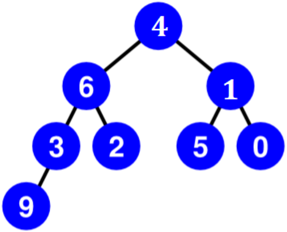
- Given an arbitrary array, how can we make it into a heap? Similarly, given an arbitrary complete binary tree, how can we make it into a heap?
- **Solution:** Start at the bottom of the tree to restore the heap property within each subtree and work upwards towards the root.

```
buildHeap ()
    for $i = size - 1$ down to 0
        reheapDown($i$);
```

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(7)

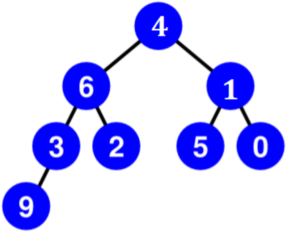


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 3 | 2 | 5 | 0 | 9 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(6)

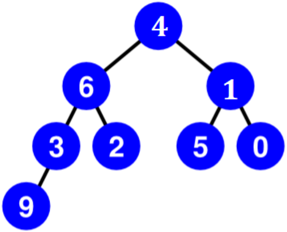


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 3 | 2 | 5 | 0 | 9 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(5)

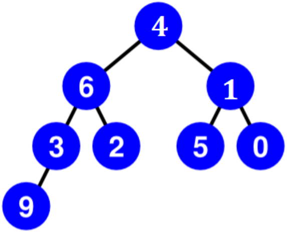


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 3 | 2 | 5 | 0 | 9 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(4)

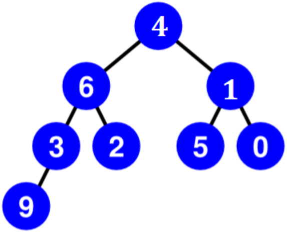


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 3 | 2 | 5 | 0 | 9 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(3)

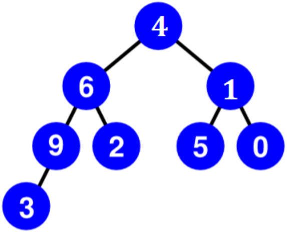


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 3 | 2 | 5 | 0 | 9 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(3)

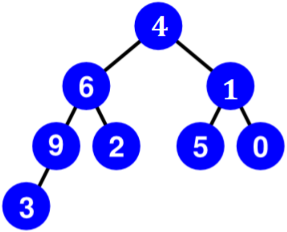


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 9 | 2 | 5 | 0 | 3 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(2)

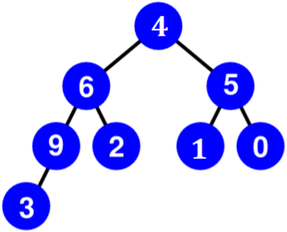


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 9 | 2 | 5 | 0 | 3 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(2)

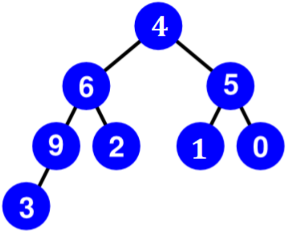


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 6 | 5 | 9 | 2 | 1 | 0 | 3 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(1)

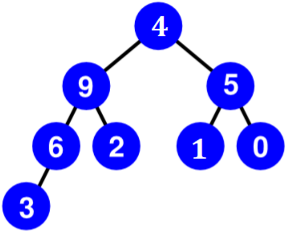


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 6 | 5 | 9 | 2 | 1 | 0 | 3 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(1)

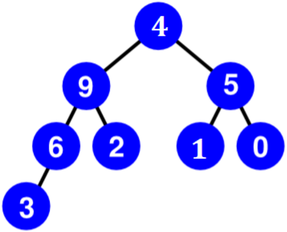


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 9 | 5 | 6 | 2 | 1 | 0 | 3 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(0)

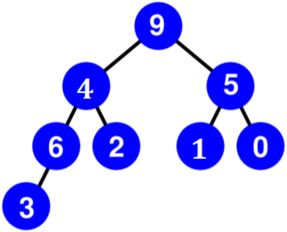


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 9 | 5 | 6 | 2 | 1 | 0 | 3 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(0)

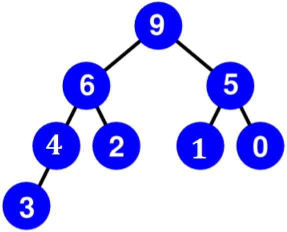


| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 9 | 4 | 5 | 6 | 2 | 1 | 0 | 3 |

# BuildHeap (Heapify) Example

- Convert the following array (complete binary tree) into a heap:

reheapDown(0)



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 9 | 6 | 5 | 4 | 2 | 1 | 0 | 3 |

# Reheap Time Complexity

- A call to `reheapDown` takes time $O(\log n)$ in the worst case.
- buildHeap makes $n$ calls to `reheapDown`.
- Therefore, the runtime of buildHeap is $O(n \log n)$ .

# Reheap Time Complexity

- A call to `reheapDown` takes time $O(\log n)$ in the worst case.
- buildHeap makes $n$ calls to `reheapDown`.
- Therefore, the runtime of buildHeap is $O(n \log n)$ .
- This is true, but we can give a better bound. In fact, the runtime of buildHeap is $O(n)$ .

# Reheap Time Complexity

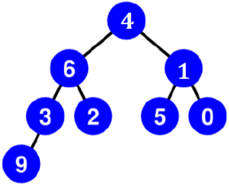
- A call to `reheapDown` takes time $O(\log n)$ in the worst case.
- buildHeap makes $n$ calls to `reheapDown`.
- Therefore, the runtime of buildHeap is $O(n \log n)$ .
- This is true, but we can give a better bound. In fact, the runtime of buildHeap is $O(n)$ .
- **Given an array of numbers, we can form a heap from them in time $O(n)$.**

# Heap Summary

- Heap is a **simple** data structure that implements priority queues.
- **extract-max** can be implemented using **reheapDown** in $O(\log n)$.
- **insert** can be implemented using **reheapUp** in $O(\log n)$.
- Given an array of keys, it is possible to build a heap from them (heapify) in $O(n)$.
  - Note that heapify and extractMax take place in place (no extra memory is used)

# Heap Sort

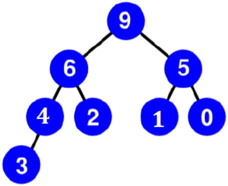
- Given an array $A$ of size $n$, apply Heapify procedure to form a heap on $A$ (this takes $O(n)$).



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 3 | 2 | 5 | 0 | 9 |

# Heap Sort

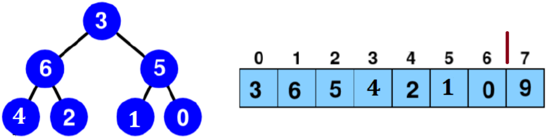
- Given an array $A$ of size $n$, apply Heapify procedure to form a heap on $A$ (this takes $O(n)$).



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 9 | 6 | 5 | 4 | 2 | 1 | 0 | 3 |

# Heap Sort

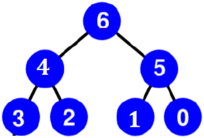
- Apply ExtractMax $n$ times; it takes a total of $n \times O(\log n) = O(n \log n)$ time.



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 3 | 6 | 5 | 4 | 2 | 1 | 0 | 9 |

# Heap Sort

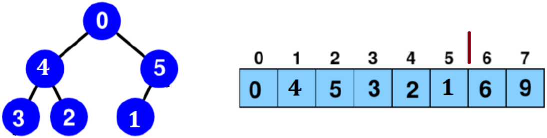
- Apply ExtractMax $n$ times; it takes a total of $n \times O(\log n) = O(n \log n)$ time.



| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 6 | 4 | 5 | 3 | 2 | 1 | 0 | 9 |

# Heap Sort

- Apply ExtractMax $n$ times; it takes a total of $n \times O(\log n) = O(n \log n)$ time.



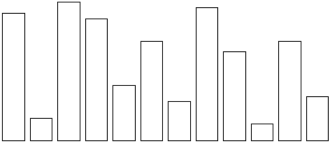
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 0 | 4 | 5 | 3 | 2 | 1 | 6 | 9 |

# Quick Sort

- Like insertion sort and heap sort, quicksort works **in place**.
- Like merge sort, quicksort employs a divide and conquer strategy.
- Quicksort is usually implemented as a randomized algorithm.

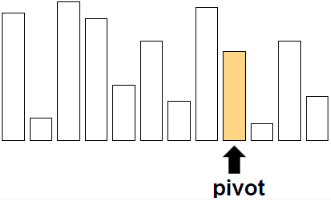
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



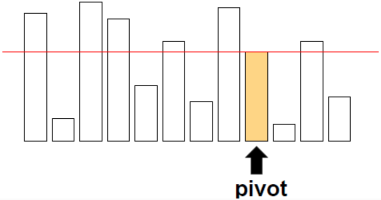
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



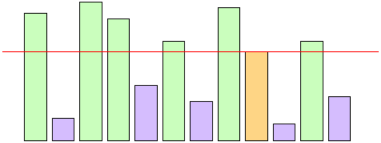
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



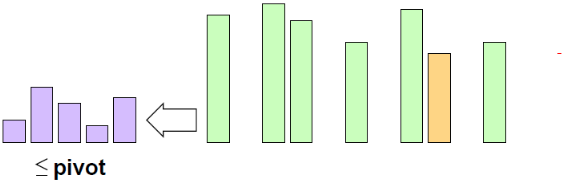
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



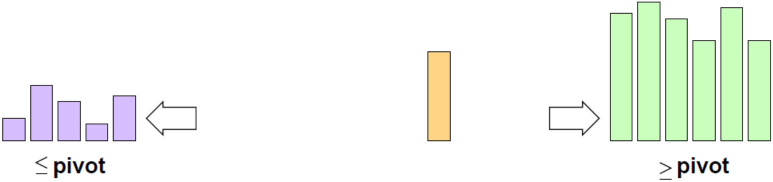
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



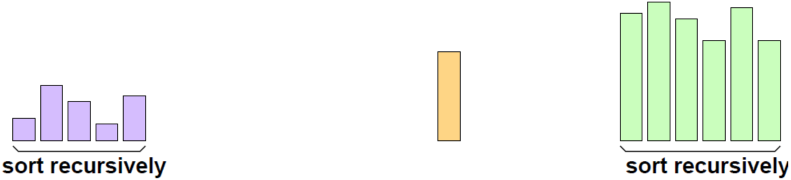
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



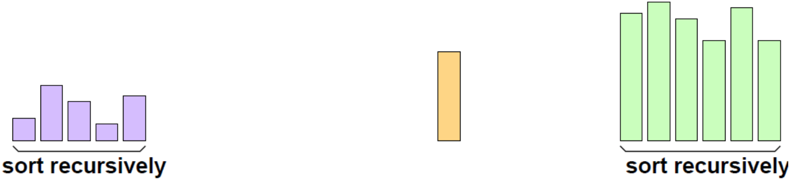
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



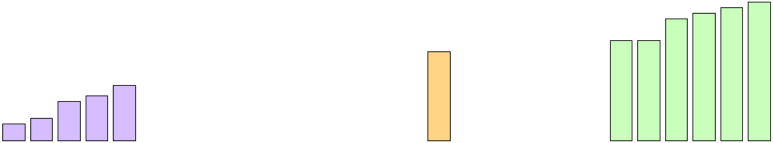
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



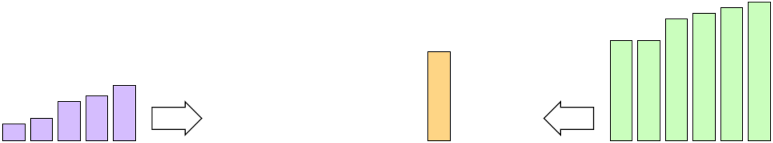
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



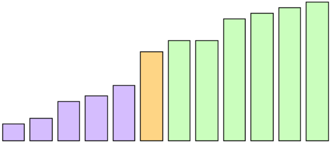
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



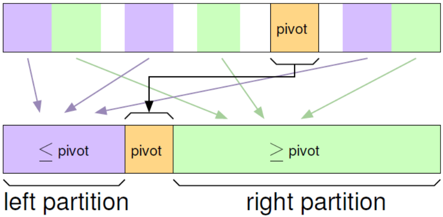
# Quick Sort

- Select an arbitrary element in the array as a **pivot**.
- Partition the array such that elements less than or equal to the pivot appear to its left and elements greater than or equal to the pivot appear to its right.
- Recursively sort each partition.
- In the base case we have an array of size 1.



# Quick Sort

- Any element in the array can be selected as the pivot.
- Typically, the pivot is selected randomly.
- Elements are partitioned into those less than or equal to the pivot and those greater than the pivot.
- In general, elements within a partition are not initially sorted.
- **Partitioning can be performed in place.**



# Partition Algorithm

- First, swap pivot with the first element.
- Store elements $\leq$ the pivot at the front of the array and elements $\geq$ the pivot at the back of the array.
  1. Scan the array starting from the front until an element is found that is $>$ the pivot.
  2. Scan the array starting from the back until an element is found that is $<$ the pivot.
  3. Swap these two items.
  4. Continue until the entire array has been partitioned.

# Quicksort Example

- Sort the following array using quicksort.

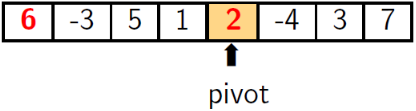
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 6 | -3 | 5 | 1 | 2 | -4 | 3 | 7 |

# Quicksort Example

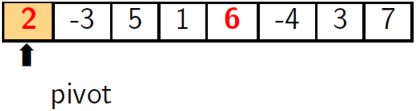
- Sort the following array using quicksort.

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

- **Step 1.** Select a pivot element and swap it with the first element.



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

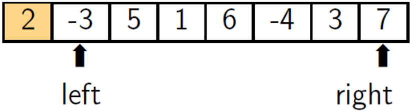


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| **2** | -3 | 5 | 1 | **6** | -4 | 3 | 7 |

- Note that you could select any element as the pivot.

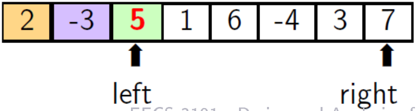
# Quicksort Example

- We now partition the array such that elements to the left of the pivot are $\leq$ than the pivot and element to the right of the pivot are $\geq$ the pivot.
  - Initialize **left** to the leftmost element after the pivot and **right** to the rightmost element.



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

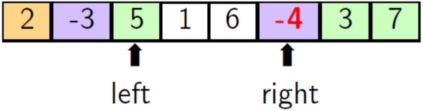
  - **step 2:** Increment **left** until we find an element that is greater than the pivot.



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

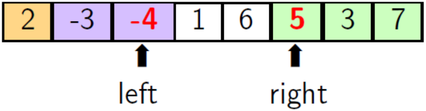
# Quicksort Example

- **Step 3:** Decrement **right** until we find an element that is less than the pivot.



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

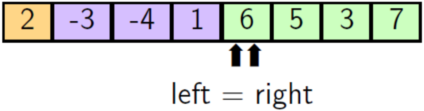
- **Step 4:** Swap the elements at positions **left** and **right**.



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

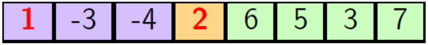
# Quicksort Example

- **Step 5:** Repeat until **left** $\geq$ **right**.



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

- **Step 6:** If the element at position **right** is less than the pivot, then swap it with the pivot. Otherwise, swap the element at position **right** $-1$ with the pivot.



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

# Quicksort Example

- The pivot element is now in the correct position in the array.
- Elements to the left of the pivot are ≤than it and elements to the right of the pivot are ≥ than it.

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

# Quicksort Example

- The pivot element is now in the correct position in the array.
- Elements to the left of the pivot are $\leq$than it and elements to the right of the pivot are $\geq$ than it.

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

- Quicksort is called recursively on the left and right partitions.

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

# Quicksort Example

- The pivot element is now in the correct position in the array.
- Elements to the left of the pivot are $\leq$than it and elements to the right of the pivot are $\geq$ than it.

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

- Quicksort is called recursively on the left and right partitions.

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

- Once the left subarray is recursively sorted and the right subarray is recursively sorted, the entire array is sorted. The base case is reached when the array has size $\leq 1$.

# QuickSort

```
$quickSort1(A)$
$A$: array of size $n$
    if $n \leq 1$ then return
    $p \leftarrow ChoosePivot1(A)$
    $i \leftarrow Partition(A, p)$
    $quickSort1(A[0, 1, \ldots, i-1])$
    $quickSort1(A[i+1, \ldots, size(A)-1])$
```

Here pivot is chosen arbitrarily (e.g., it is the first item in the array)

# Analysis of Quick-sort

**Worst case**: $T^{(worst)}(n) = T^{(worst)}(n-1) + \Theta(n)$
The algorithm has a running time of $\Theta(n^2)$ in the worst case.

# Analysis of Quick-sort

**Worst case**: $T^{(worst)}(n) = T^{(worst)}(n-1) + \Theta(n)$
The algorithm has a running time of $\Theta(n^2)$ in the worst case.

**Best case**: $T^{(best)}(n) = T^{(best)}(\lfloor \frac{n-1}{2} \rfloor) + T^{(best)}(\lceil \frac{n-1}{2} \rceil) + \Theta(n)$
Similar to Merge-sort; $T^{(best)}(n) \in \Theta(n \log n)$

# Average-case analysis of quick-sort

- In a **comparison-based sorting** the running time is proportional to the total number of comparisons performed during partitioning.

# Average-case analysis of quick-sort

- In a **comparison-based sorting** the running time is proportional to the total number of comparisons performed during partitioning.
- Let $X_n$ be a random variable denoting the number of comparisons made by Quicksort on an array of size $n$.
  $E[X_n] = expected\ no.\ of\ comparisons =$
  $\sum_{i,j \in 0,\ldots,n-1} prob(\text{the } i\text{'th and } j\text{'th smallest elements are compared})$

# Average-case analysis of quick-sort

- In a **comparison-based sorting** the running time is proportional to the total number of comparisons performed during partitioning.
- Let $X_n$ be a random variable denoting the number of comparisons made by Quicksort on an array of size $n$.
  $E[X_n] = \textit{expected no. of comparisons} = \sum_{i,j \in 0,\ldots,n-1} \textit{prob}(\text{the } i\text{'th and } j\text{'th smallest elements are compared})$
- The average-case running time of QuickSort is indeed $\Theta(E[X_n])$.

# Average-case analysis of quick-sort

- To find $E[X_n]$, we study the chance that the $i$'th and the $j$'th smallest items are compared.

# Average-case analysis of quick-sort

- To find $E[X_n]$, we study the chance that the $i$'th and the $j$'th smallest items are compared.
    - In a random (unsorted) permutation of $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$, what is the chance that 3 and 7 are compared?

# Average-case analysis of quick-sort

- To find $E[X_n]$, we study the chance that the $i$'th and the $j$'th smallest items are compared.
    - In a random (unsorted) permutation of $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$, what is the chance that 3 and 7 are compared? $\rightarrow$ the chance that $4, 5, 6$ are Not selected as pivot before $3, 7 \rightarrow 2/5$
    - If 4 (or 5 or 6) are pivot at a recursive call that includes both 3 and 7, then 3 and 7 are placed in different sides of 4 and will not be compared.

# Average-case analysis of quick-sort

- To find $E[X_n]$, we study the chance that the $i$'th and the $j$'th smallest items are compared.
    - In a random (unsorted) permutation of $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$, what is the chance that 3 and 7 are compared? → the chance that $4, 5, 6$ are Not selected as pivot before $3, 7$ → $2/5$
    - If 4 (or 5 or 6) are pivot at a recursive call that includes both 3 and 7, then 3 and 7 are placed in different sides of 4 and will not be compared.

- Elements $i$ and $j$ are compared iff one of them is selected as a pivot at some point before any other element in $\{i+1, i+2, \ldots, j-1\}$. This occurs with probability $\frac{2}{j-i+1}$.

# Average-case analysis of quick-sort

- To find $E[X_n]$, we study the chance that the $i$'th and the $j$'th smallest items are compared.
    - In a random (unsorted) permutation of $1, 2, 3, 4, 5, 6, 7, 8, 9, 10$, what is the chance that 3 and 7 are compared? $\rightarrow$ the chance that $4, 5, 6$ are Not selected as pivot before $3, 7 \rightarrow 2/5$
    - If 4 (or 5 or 6) are pivot at a recursive call that includes both 3 and 7, then 3 and 7 are placed in different sides of 4 and will not be compared.
- Elements $i$ and $j$ are compared iff one of them is selected as a pivot at some point before any other element in $\{i+1, i+2, \ldots, j-1\}$. This occurs with probability $\frac{2}{j-i+1}$.
- The expected time complexity will be $\sum\limits_{i,j \in 0,\ldots,n-1, j>i} \frac{2}{j-i+1}$

# Average-case analysis of quick-sort

- For the expected time complexity of Quicksort, we have:

$$
\begin{aligned}
E[X_n] &= \sum_{i,j \in 0,\ldots,n-1, j>i} \frac{2}{j-i+1} \\
&= \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} \frac{2}{j-i+1} \\
&= \sum_{i=0}^{n-2} \sum_{k=2}^{n-i} \frac{2}{k} < \sum_{i=0}^{n-2} \sum_{k=2}^{n} \frac{2}{k} \\
&= \sum_{i=0}^{n-2} \Theta(\log n) = \Theta(n \log n)
\end{aligned}
$$

- So, $E[X_n]$ belongs to $\Theta(n \log n)$. Note that we used the fact that the sum of Harmonic series belongs to $\Theta(\log n)$.

# Selection Problem

- Can we improve the worst-case running time $\Theta(n^2)$ of Quick-sort to $\Theta(n \log n)$?
    - This relates to the **selection problem**

# Selection & order statistics

- The $i$'th order statistic of a set of comparable elements is the $i$'th smallest value in the set.
    - The $\lceil n/2 \rceil$'th order statistic among $n$ items is called **median**.
    - The $\lceil n/4 \rceil$'th order statistic among $n$ items is called **quartile**.
- How can we find the 0'th or $(n-1)$'th order statistic in $\Theta(n)$.

# Selection & order statistics

- The $i$'th order statistic of a set of comparable elements is the $i$'th smallest value in the set.
  - The $\lceil n/2 \rceil$'th order statistic among $n$ items is called **median**.
  - The $\lceil n/4 \rceil$'th order statistic among $n$ items is called **quartile**.
- How can we find the 0'th or $(n-1)$'th order statistic in $\Theta(n)$.
  - Finding min/max → a linear scan is sufficient!

## Selection & order statistics

- The $i$'th order statistic of a set of comparable elements is the $i$'th smallest value in the set.
    - The $\lceil n/2 \rceil$'th order statistic among $n$ items is called **median**.
    - The $\lceil n/4 \rceil$'th order statistic among $n$ items is called **quartile**.
- How can we find the 0'th or $(n-1)$'th order statistic in $\Theta(n)$.
    - Finding min/max → a linear scan is sufficient!
- **Selection problem:** find the $i$'th order statistics:
    - The input is a set of $n$ comparable objects (e.g., integers) and an integer $i$
    - The output is the element at index $i$ of the sorted array ($i+1$'th smallest item)

# Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
    - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.

# Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
  - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.
  - Can we do better?

# Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
    - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.
    - Can we do better?
- Attempt II: apply **heapify** on $A$ and **extract-min** $i+1$ times (we assume indices start at 0).
    - Heapify takes $\Theta(n)$ and each extract-min operation takes $\Theta(\log n)$
    - Select takes $\Theta(n + i \log n)$, which is $\Theta(n \log n)$ when $i \in \Theta(n)$.

# Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
    - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.
    - Can we do better?
- Attempt II: apply **heapify** on $A$ and **extract-min** $i + 1$ times (we assume indices start at 0).
    - Heapify takes $\Theta(n)$ and each extract-min operation takes $\Theta(\log n)$
    - Select takes $\Theta(n + i \log n)$, which is $\Theta(n \log n)$ when $i \in \Theta(n)$.
- What is the minimum time required for selection?

# Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
  - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.
  - Can we do better?
- Attempt II: apply **heapify** on $A$ and **extract-min** $i+1$ times (we assume indices start at 0).
  - Heapify takes $\Theta(n)$ and each extract-min operation takes $\Theta(\log n)$
  - Select takes $\Theta(n + i \log n)$, which is $\Theta(n \log n)$ when $i \in \Theta(n)$.
- What is the minimum time required for selection?
  - We need to read the whole input, i.e., the running time of **any** algorithm is $\Omega(n)$.

# Selection algorithms

- Attempt I: sort $A$ and return the element at index $i$ in the sorted array.
    - E.g., use Merge-sort; sorting takes $\Theta(n \log n)$ and accessing the element in sorted array takes $\Theta(1)$.
    - Can we do better?
- Attempt II: apply **heapify** on $A$ and **extract-min** $i + 1$ times (we assume indices start at 0).
    - Heapify takes $\Theta(n)$ and each extract-min operation takes $\Theta(\log n)$
    - Select takes $\Theta(n + i \log n)$, which is $\Theta(n \log n)$ when $i \in \Theta(n)$.
- What is the minimum time required for selection?
    - We need to read the whole input, i.e., the running time of **any** algorithm is $\Omega(n)$.
    - Can we select in $\Theta(n)$?

# Selection algorithms

- Quick-select: similar to Quick-sort, but for selection
- Select a pivot, partition around it, and recurs on the **one side** that contains the *i*'th element

# QuickSelect

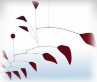
```
$quickSelect1(A, i)$
$A$: array of size $n$,    $i$: integer s.t. $0 \leq i < n$
    $p \leftarrow choosePivot1(A)$
    $j \leftarrow Partition(A, p)$
    **if** $j = i$ **then**
        **return** $A[j]$
    **else if** $j > i$ **then**
        **return** $quickSelect1(A[0, 1, \ldots, j-1], i)$
    **else if** $j < i$ **then**
        **return** $quickSelect1(A[j+1, j+2, \ldots, n-1], i-j-1)$
```

- If pivot is at position $j$, the cost of recursive call parameters will be:
  - None if $j = i$.
  - $(j, i)$ if $j > i$ (recursing on the left subarray).
  - $(n - j - 1, i - j - 1)$ if $j < i$ (recursing on the right subarray).

# Average-case analysis of quick-select1

- Assume all $n!$ permutations are equally likely.
    - Define $T(n, i)$ as average cost for selecting $i$th item from size-$n$ array The cost for recursive calls (RC) is

$$RC = \begin{cases} 0 & j = i \\ T(j, i), & j > i \\ T(n-j-1, i-j-1) & j < i \end{cases}$$



# Average-case analysis of quick-select1

- Assume all $n!$ permutations are equally likely.
    - Define $T(n,i)$ as average cost for selecting $i$th item from size-$n$ array The cost for recursive calls (RC) is

$$RC = \begin{cases} 0 & j = i \\ T(j,i), & j > i \\ T(n-j-1, i-j-1) & j < i \end{cases}$$

- Shuffled input → it is equally likely for the pivot to be at any position:

$$T(n,i) = \underbrace{cn}_{partition} + \frac{1}{n}\Big(\text{(RC if j=0) + (RC if j=1) + } \ldots + \text{ (RC if j=n-1)}\Big)$$

$$= \underbrace{cn}_{partition} + \frac{1}{n}\left(\sum_{j=0}^{i-1} T(n-j-1, i-j-1) + \sum_{j=i+1}^{n-1} T(j,i)\right)$$

- For simplicity, define $T(n) = \max_{0 \le k < n} T(n,k)$.

# Average-case analysis of quick-select1

$$T(n) \leq \underbrace{cn}_{partition} + \frac{1}{n}\left(\sum_{j=0}^{i-1} T(n-j-1) + \sum_{j=i+1}^{n-1} T(j)\right)$$

# Average-case analysis of quick-select1

$$T(n) \le \underbrace{cn}_{partition} + \frac{1}{n}\left(\sum_{j=0}^{i-1} T(n-j-1) + \sum_{j=i+1}^{n-1} T(j)\right)$$

- We say that a pivot is **good** if the arrays on both sides have size at least $n/4$
  - This happens when pivot index $j$ is in $[n/4, 3n/4)$.
  - Half of possible pivots are good and the rest are bad.

# Average-case analysis of quick-select1

$$T(n) \leq \underbrace{cn}_{partition} + \frac{1}{n}\left(\sum_{j=0}^{i-1} T(n-j-1) + \sum_{j=i+1}^{n-1} T(j)\right)$$

- We say that a pivot is **good** if the arrays on both sides have size at least $n/4$
  - This happens when pivot index $j$ is in $[n/4, 3n/4)$.
  - Half of possible pivots are good and the rest are bad.
- The recursive cost for a good pivot is at most $T(3n/4)$.
- The recursive cost for a bad pivot is at most $T(n)$.
- The average cost is then given by:

$$T(n) \leq \begin{cases} cn + \frac{1}{2}\Big(\underbrace{T(n)}_{bad\ \ pivot} + \underbrace{T(3n/4)}_{good\ \ pivot}\Big), & n \geq 2 \\ d & n = 1 \end{cases}$$

## Average-case analysis of quick-select1

- The average cost is then given by:

$$T(n) \leq \begin{cases} cn + \frac{1}{2}\Big(T(n) + T\big(3n/4\big)\Big), & n \geq 2 \\ d, & n = 1 \end{cases}$$

# Average-case analysis of quick-select1

- The average cost is then given by:

$$T(n) \leq \begin{cases} cn + \frac{1}{2}\Big(T(n) + T\big(3n/4\big)\Big), & n \geq 2 \\ d, & n = 1 \end{cases}$$

- Rearranging gives:

$$\begin{aligned} T(n) &\leq T\big(3n/4\big) + 2cn \leq 2cn + 2c(3n/4) + 2c(9n/16) + \cdots + d \\ &\leq d + 2cn \sum_{i=0}^{\infty} \left(\frac{3}{4}\right)^i \in O(n) \end{aligned}$$

- Since $T(n)$ **must** be $\Omega(n)$ (why?), $T(n) \in \Theta(n)$.

# Linear-time selection

- Although Quick-select runs in $O(n)$ on average, in the worst-case it is still super-linear.
    - Recurrence given by $T(n) = \begin{cases} T(n-1) + cn, & n \geq 2 \\ d, & n = 1 \end{cases}$

# Linear-time selection

- Although Quick-select runs in $O(n)$ on average, in the worst-case it is still super-linear.
  - Recurrence given by $T(n) = \begin{cases} T(n-1) + cn, & n \geq 2 \\ d, & n = 1 \end{cases}$
- Is there any selection algorithm that runs in $O(n)$ in the worst-case?
  - The answer is Yes; **Median of medians** algorithms!
  - It is a twist to Quick-select in which the pivot is selected a bit smarter!

# Median of five algorithm

- A variant of Quick-select in which the pivot is selected more carefully.
- The input is an array $A$ of $n$ objects (assume $n$ is divisible by 5).

# Median of five algorithm

- A variant of Quick-select in which the pivot is selected more carefully.
- The input is an array $A$ of $n$ objects (assume $n$ is divisible by 5).
- Divide $A$ into $n/5$ blocks of size 5.

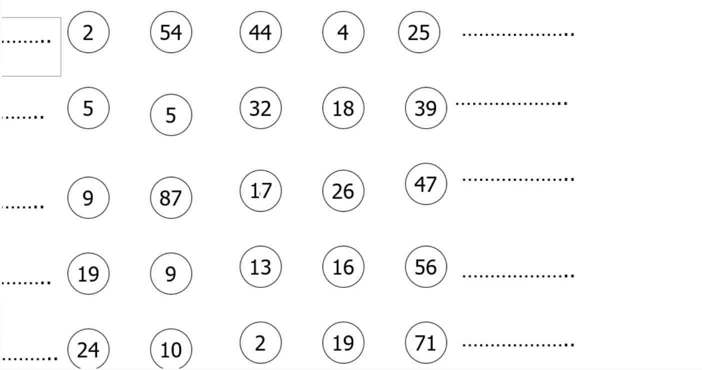
# Median of five algorithm

- A variant of Quick-select in which the pivot is selected more carefully.
- The input is an array $A$ of $n$ objects (assume $n$ is divisible by 5).
- Divide $A$ into $n/5$ blocks of size 5.
- Recursively find the median of the medians; denote it by $x$.
    - $x$ will be the pivot for quick-select

# Median of five algorithm

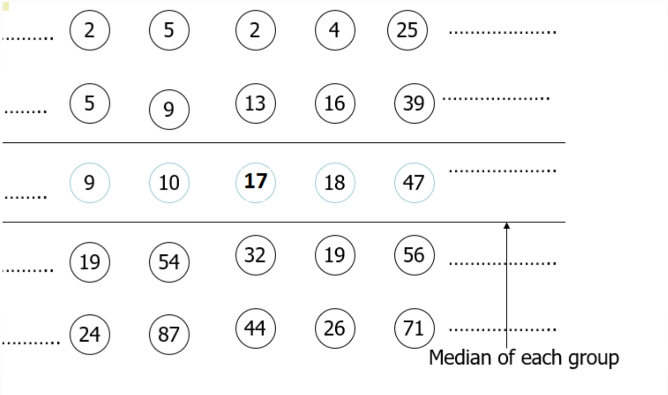
- A variant of Quick-select in which the pivot is selected more carefully.
- The input is an array $A$ of $n$ objects (assume $n$ is divisible by 5).
- Divide $A$ into $n/5$ blocks of size 5.
- Recursively find the median of the medians; denote it by $x$.
    - $x$ will be the pivot for quick-select
- Partition the whole array using $x$ as the pivot
- Recurs on the corresponding subarray as in Quick-select

# Median of five example



| | | | | | | |
|---|---|---|---|---|---|---|
| ......... | 2 | 54 | 44 | 4 | 25 | .................... |
| ........ | 5 | 5 | 32 | 18 | 39 | .................... |
| ........ | 9 | 87 | 17 | 26 | 47 | .................... |
| ......... | 19 | 9 | 13 | 16 | 56 | .................... |
| .......... | 24 | 10 | 2 | 19 | 71 | .................... |

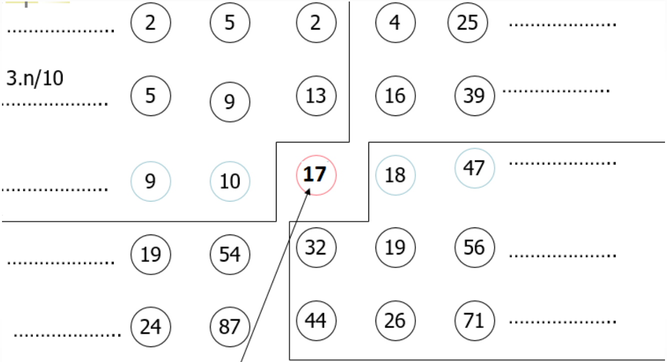
# Median of five example



| | | | | |
|---|---|---|---|---|
| 2 | 5 | 2 | 4 | 25 |
| 5 | 9 | 13 | 16 | 39 |
| 9 | 10 | **17** | 18 | 47 |
| 19 | 54 | 32 | 19 | 56 |
| 24 | 87 | 44 | 26 | 71 |

Median of each group

# Median of five example



X ,the median of medians

# Median of five algorithm

- Pivot $x$ is median of medians → half of blocks have median $< x$.
  - This implies half of blocks include at least 3 elements $< x$.
  - So, there will be at least $n/5 \cdot 1/2 \cdot 3 = 3n/10$ elements smaller than $x$
- Similarly, there will be at least $3n/10$ elements larger than $x$.
- We assume distinct items; when pivot is equal to multiple items, you can update the partition algorithm so that the pivot is the 'best' among items with the same key
- Hence, the size of recursive call is always in the range $(3n/10, 7n/10)$.
  - $x$ is always a 'good' pivot

## Median of five algorithm

- Pivot $x$ is median of medians → half of blocks have median $< x$.
  - This implies half of blocks include at least 3 elements $< x$.
  - So, there will be at least $n/5 \cdot 1/2 \cdot 3 = 3n/10$ elements smaller than $x$
- Similarly, there will be at least $3n/10$ elements larger than $x$.
- We assume distinct items; when pivot is equal to multiple items, you can update the partition algorithm so that the pivot is the 'best' among items with the same key
- Hence, the size of recursive call is always in the range $(3n/10, 7n/10)$.
  - $x$ is always a 'good' pivot
- In the worst case, the size of recursive call is always $7n/10$.

$$T(n) \leq \begin{cases} \underbrace{T(n/5)}_{find\ \ x} + \underbrace{cn}_{partition\ \ around\ \ x} + \underbrace{T(7n/10)}_{recursive\ \ call}, & n \geq 2 \\ d, & n = 1 \end{cases}$$

# Median of five algorithm

$$T(n) \leq \begin{cases} \underbrace{T(n/5)}_{find\ \ x} + \underbrace{cn}_{partition\ \ around\ \ x} + \underbrace{T(7n/10)}_{recursive\ \ call}, & n \geq 2 \\ d, & n = 1 \end{cases}$$

- We **guess** that $T(n) \in O(n)$ and use **strong** induction to prove it.
- We prove there is a value $M$ s.t. $T(n) \leq Mn$ for all $n \geq 1$.
- For the base we have $T(1) = d \leq M$ as long as $M \geq d$.
- For any value of $n$ we can state:

$$\begin{aligned} T(n) &\leq T(n/5) + T(7n/10) + cn \quad \text{(definition)} \\ &\leq M \cdot n/5 + M \cdot 7n/10 + cn \quad \text{(induction hypothesis)} \\ &= (9M/10 + c)n \\ &\leq M \cdot n \quad \text{as log as } M \geq 9M/10 + c, i.e., M \geq 10c \end{aligned}$$

- so, we showed for $M = \max\{10c, d\}$ we have $T(n) \leq M \cdot n$ for $n \geq 1$. So, $T(n) \in O(n)$.

# Quick-sort revisit

**Theorem**

*It is possible to select the i'th smallest item in a list of n numbers in time* $\Theta(n)$

# Quick-sort revisit

**Theorem**

*It is possible to select the $i$'th smallest item in a list of n numbers in time $\Theta(n)$*

- Quick-sort in $O(n \log n)$ time:
    - Using select algorithm to choose the pivot as the **median** of $n$ items in $O(n)$ time
    - Partition around pivot in $O(n)$ time (selecting pivot as $n/c$'th smallest item for constant $c$ gives the same result)
    - Sort the two sides of pivot recursively in time $2T(n/2)$.
- The cost will be $T(n) = 2T(n/2) + \Theta(n)$, which gives $T(n) = \Theta(n \log n)$ [case II of Master theorem]

**Theorem**

*A smart selection of pivot, using linear-time select, results in quick-sort running in $\Theta(n \log n)$*

# Lower Bound for Comparison-Based Sorting

- In **comparison-based sorting**, we have a set of objects (e.g., a bag of potato) and an operator that can tell us whether an object is smaller than the other (e.g., a scale for comparing weights of potatoes)

# Lower Bound for Comparison-Based Sorting

- In **comparison-based sorting**, we have a set of objects (e.g., a bag of potato) and an operator that can tell us whether an object is smaller than the other (e.g., a scale for comparing weights of potatoes)
    - No other assumption is made for the objects (e.g., they are not necessarily numbers).
    - Algorithms like Bubble-sort, Quick-sort, Merge-sort, and Heap-sort are all comparison-based.

# Lower Bound for Comparison-Based Sorting

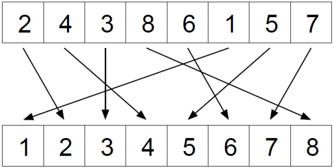
- In **comparison-based sorting**, we have a set of objects (e.g., a bag of potato) and an operator that can tell us whether an object is smaller than the other (e.g., a scale for comparing weights of potatoes)
  - No other assumption is made for the objects (e.g., they are not necessarily numbers).
  - Algorithms like Bubble-sort, Quick-sort, Merge-sort, and Heap-sort are all comparison-based.
- Does a comparison-base sorting with time asymptotically less than $O(n \log n)$ exist?
  - We show the answer is No!

# Lower Bound for Comparison-Based Sorting

- Consider a set of $n$ distinct objects $a_1, \ldots, a_n$.
- Any permutation of thee objects forms an ordering (a possible input array).
- How many ways they can be ordered?
  - $n$! ways, that is, there are $n$! possible inputs!

# Lower Bound for Comparison-Based Sorting

- Consider a set of $n$ distinct objects $a_1, \ldots, a_n$.
- Any permutation of thee objects forms an ordering (a possible input array).
- How many ways they can be ordered?
    - $n!$ ways, that is, there are $n!$ possible inputs!
- Sorting corresponds to identifying the permutation of a sequence of elements. Once the permutation is known, the position of each item can be restored.

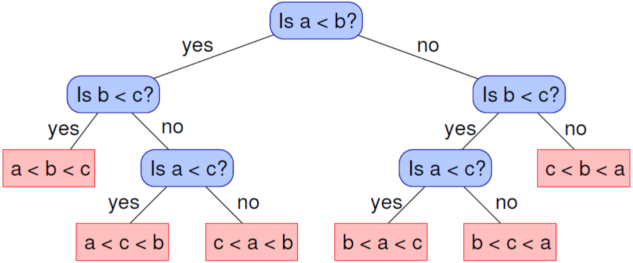


| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 4 | 3 | 8 | 6 | 1 | 5 | 7 |

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

# Decision Tree for Sorting an Array

- Suppose an array of $A$ of three elements $[a, b, c]$ is to be sorted.
- Any algorithm can be described with a **decision tree** for determining the correct sorted order (i.e., the array's permutation).
  - The number of comparisons made by the algorithm equals to the height of the tree!
  - The number of leaves is $n!$
- Here is the decision tree for one algorithm:



# Lower Bound using Decision Tree

- The minimum height of a binary tree with $n$ nodes is at least $\log n$.
- Intuitively, whenever two elements are compared (is $x < y$?) this eliminates a number of possible permutations. In the worst case, at most half of the remaining possible permutations are eliminated.

# Lower Bound using Decision Tree

- The minimum height of a binary tree with $n$ nodes is at least $\log n$.
- Intuitively, whenever two elements are compared (is $x < y$?) this eliminates a number of possible permutations. In the worst case, at most half of the remaining possible permutations are eliminated.
- If there are $n!$ permutations and half are eliminated on each step, it takes at least $\log_2(n!)$ comparisons to identify the correct permutation of the items.

# Lower Bound using Decision Tree

- The minimum height of a binary tree with $n$ nodes is at least $\log n$.
- Intuitively, whenever two elements are compared (is $x < y$?) this eliminates a number of possible permutations. In the worst case, at most half of the remaining possible permutations are eliminated.
- If there are $n!$ permutations and half are eliminated on each step, it takes at least $\log_2(n!)$ comparisons to identify the correct permutation of the items.
- So, any algorithm has to make at least $\log n!$ comparisons.

$$
\begin{aligned}
\log_2(n!) &= \log_2(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2 \cdot 1) \\
&> \log_2(n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot (n/2+1) \cdot (n/2)) \\
&> \log_2(\underbrace{(n/2) \cdot (n/2) \cdot (n/2) \cdot \ldots \cdot (n/2) \cdot (n/2)}_{n/2 \text{ times}}) \\
&= \log_2\left((n/2)^{n/2}\right) \\
&= \frac{n}{2}\log_2(n/2) \quad \in \Omega(n \log n)
\end{aligned}
$$

# Lower Bound for Comparison-Based Sorting

**Theorem**

*Any comparison-based sorting has time complexity of $\Omega(n \log n)$*

- We use $\Omega(n \log n)$ to denote the time complexity that is asymptotically **at least** $n \log n$

# Non-comparison based algorithms

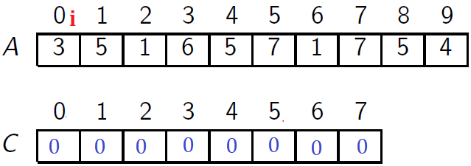
- The lower bound of $\Omega(n \log n)$ applies to algorithms that determine a sorted ordering by comparing pairs of elements.
- If the set of elements to be sorted has specific characteristics, then faster sorting algorithms may be possible.
- Such algorithms must use techniques other than comparison alone to determine the sorted order.
- Examples: counting sort, bucket sort, radix sort.

# Counting Sort

- Let A be an array of n integers in the range $\{0, 1, \ldots, k\}$.
- For each value $i$ in $\{0, 1, \ldots, k\}$, count the number of occurrences of $i$ in $\{0, 1, \ldots, k\}$ and store it in $C[i]$.
- Overwrite $A[0..n-1]$ with the number of occurrences of each value $\{0, 1, \ldots, k\}$ in sorted order.

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

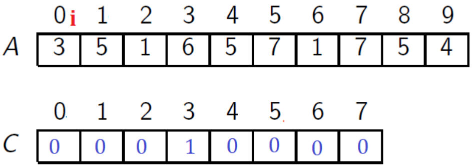


|  | 0 i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| C | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.



|  | 0 i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

| | 0 | 1 i | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|   | 0 | 1 i | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 i | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 |

## Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 i | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 i | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 0 |

## Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 i | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 i | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 i | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 1 | 0 | 1 | 0 | 2 | 1 | 0 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 i | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 1 | 0 | 1 | 0 | 2 | 1 | 0 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|   | 0 | 1 | 2 | 3 | 4 | 5 i | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 1 | 0 | 1 | 0 | 2 | 1 | 1 |

## Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 i | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 1 | 0 | 1 | 0 | 2 | 1 | 1 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 i | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 0 | 2 | 1 | 1 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 i | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 0 | 2 | 1 | 1 |

## Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7i | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 0 | 2 | 1 | 2 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 i | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 0 | 2 | 1 | 2 |

## Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 i | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 0 | 3 | 1 | 2 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 i |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 0 | 3 | 1 | 2 |

## Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 i |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

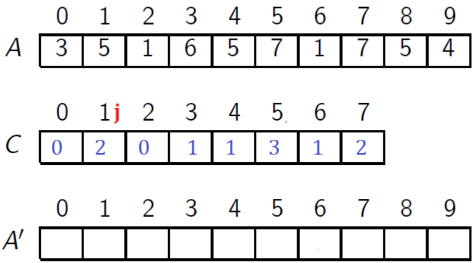
|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ |  |  |  |  |  |  |  |  |  |  |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.



|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 j | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ |  |  |  |  |  |  |  |  |  |  |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 j | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 |  |  |  |  |  |  |  |  |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2j | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 |  |  |  |  |  |  |  |  |

## Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3j | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 |  |  |  |  |  |  |  |  |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3j | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 | 3 |  |  |  |  |  |  |  |

## Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4j | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 | 3 | 4 |  |  |  |  |  |  |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4j | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 | 3 | 4 |  |  |  |  |  |  |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5**j** | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 | 3 | 4 |  |  |  |  |  |  |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

| | 0 | 1 | 2 | 3 | 4 | 5j | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 | 3 | 4 | 5 | 5 | 5 | | | |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 j | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 | 3 | 4 | 5 | 5 | 5 |  |  |  |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 j | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 | 3 | 4 | 5 | 5 | 5 | 6 |  |  |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7j |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

|  | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 | 3 | 4 | 5 | 5 | 5 | 6 |  |  |

# Counting Sort Example

Consider the following array $A$ of size $n = 10$ with items in the range $\{1..k\}$ where $k = 7$.

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A$ | 3 | 5 | 1 | 6 | 5 | 7 | 1 | 7 | 5 | 4 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| $C$ | 0 | 2 | 0 | 1 | 1 | 3 | 1 | 2 |

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| $A'$ | 1 | 1 | 3 | 4 | 5 | 5 | 5 | 6 | 7 | 7 |

# Counting Sort Running Time

- Counting Sort runs in $O(n+k)$ times.
    - Initializing elements of $C$ to 0 takes

# Counting Sort Running Time

- Counting Sort runs in $O(n+k)$ times.
    - Initializing elements of $C$ to 0 takes $O(k)$ times

# Counting Sort Running Time

- Counting Sort runs in $O(n + k)$ times.
    - Initializing elements of $C$ to 0 takes $O(k)$ times
    - Looping over $A$ and setting values of $C$ takes

# Counting Sort Running Time

- Counting Sort runs in $O(n+k)$ times.
    - Initializing elements of $C$ to 0 takes $O(k)$ times
    - Looping over $A$ and setting values of $C$ takes $O(n)$ times

# Counting Sort Running Time

- Counting Sort runs in $O(n+k)$ times.
    - Initializing elements of $C$ to 0 takes $O(k)$ times
    - Looping over $A$ and setting values of $C$ takes $O(n)$ times
    - Looping over $C$ and adding sorted elements back to array $A$ takes

# Counting Sort Running Time

- Counting Sort runs in $O(n+k)$ times.
    - Initializing elements of $C$ to 0 takes $O(k)$ times
    - Looping over $A$ and setting values of $C$ takes $O(n)$ times
    - Looping over $C$ and adding sorted elements back to array $A$ takes $O(n+k)$ times.
- The worst-case, average-case, or best-case times are the same

# Counting Sort Running Time

- Counting Sort runs in $O(n+k)$ times.
  - Initializing elements of $C$ to 0 takes $O(k)$ times
  - Looping over $A$ and setting values of $C$ takes $O(n)$ times
  - Looping over $C$ and adding sorted elements back to array $A$ takes $O(n+k)$ times.
- The worst-case, average-case, or best-case times are the same
- If $k \in O(n \log n)$, then counting sort is at least as efficient as a comparison-based sorting algorithm in the worst case.
- If k $k \notin O(n \log n)$, then counting sort is slower than an efficient comparison-based sorting algorithm in the worst case.

# Augmented Counting Sort

- Suppose the value of k is initially unknown.
- We can compute $k$ via a linear scan in $O(n)$ to find the largest element in the array.
- If $k$ is too large, call another algorithm (e.g., quick-sort or bucket-sort), otherwise continue with counting sort.

# Augmented Counting Sort

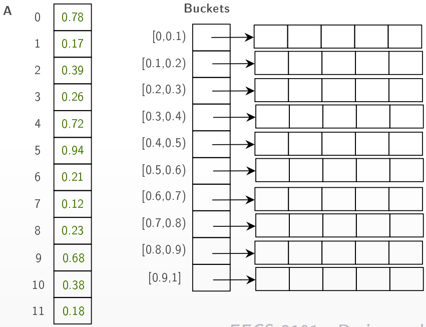
- Suppose the value of k is initially unknown.
- We can compute $k$ via a linear scan in $O(n)$ to find the largest element in the array.
- If $k$ is too large, call another algorithm (e.g., quick-sort or bucket-sort), otherwise continue with counting sort.
- What if array $A$ contains both positive and negative values?
    - Add a fixed value to all elements to make them non-negative; apply counting sort, and after sorting deduce the added element.

## Bucket Sort

- BucketSort is useful when the input is generated by a random process that distributes elements uniformly, e.g., over $[0, 1)$.
- BucketSort steps:
    - Divide $[0, 1)$ into $k$ equal-sized **buckets** (we generally assume $k = \Theta(n)$)
    - Distribute the $n$ input values into the buckets
    - Sort each bucket (e.g., using quicksort)
    - Go through the buckets in order, listing elements in each one

# Bucket Sort

- We start by distributing items into buckets; this takes $\Theta(n)$.

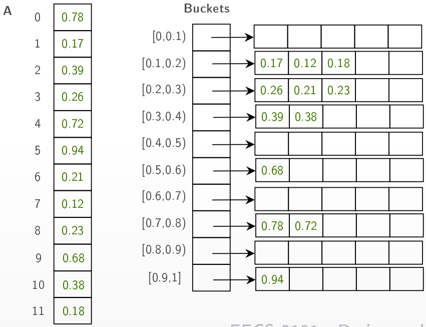


A

| | |
|---|---|
| 0 | 0.78 |
| 1 | 0.17 |
| 2 | 0.39 |
| 3 | 0.26 |
| 4 | 0.72 |
| 5 | 0.94 |
| 6 | 0.21 |
| 7 | 0.12 |
| 8 | 0.23 |
| 9 | 0.68 |
| 10 | 0.38 |
| 11 | 0.18 |

# Bucket Sort

- We start by distributing items into buckets; this takes $\Theta(n)$.

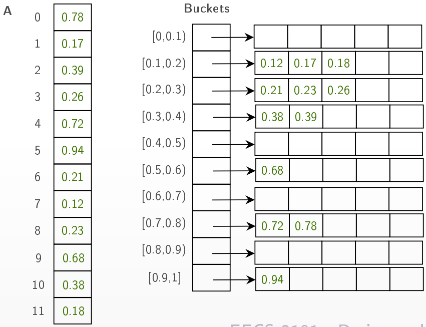


| A |  |
|---|---|
| 0 | 0.78 |
| 1 | 0.17 |
| 2 | 0.39 |
| 3 | 0.26 |
| 4 | 0.72 |
| 5 | 0.94 |
| 6 | 0.21 |
| 7 | 0.12 |
| 8 | 0.23 |
| 9 | 0.68 |
| 10 | 0.38 |
| 11 | 0.18 |

| Buckets |  |  |  |  |  |
|---|---|---|---|---|---|
| [0,0.1) |  |  |  |  |  |
| [0.1,0.2) | 0.17 | 0.12 | 0.18 |  |  |
| [0.2,0.3) | 0.26 | 0.21 | 0.23 |  |  |
| [0.3,0.4) | 0.39 | 0.38 |  |  |  |
| [0.4,0.5) |  |  |  |  |  |
| [0.5,0.6) | 0.68 |  |  |  |  |
| [0.6,0.7) |  |  |  |  |  |
| [0.7,0.8) | 0.78 | 0.72 |  |  |  |
| [0.8,0.9) |  |  |  |  |  |
| [0.9,1] | 0.94 |  |  |  |  |

# Bucket Sort

- We start by distributing items into buckets; this takes $\Theta(n)$.
- Sort within each of the $k$ buckets



**A**

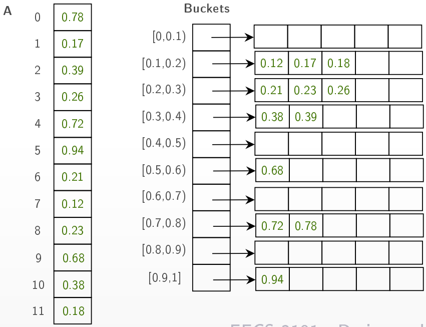
| | |
|---|---|
| 0 | 0.78 |
| 1 | 0.17 |
| 2 | 0.39 |
| 3 | 0.26 |
| 4 | 0.72 |
| 5 | 0.94 |
| 6 | 0.21 |
| 7 | 0.12 |
| 8 | 0.23 |
| 9 | 0.68 |
| 10 | 0.38 |
| 11 | 0.18 |

**Buckets**

| | | | | | |
|---|---|---|---|---|---|
| [0,0.1) | | | | | |
| [0.1,0.2) | 0.12 | 0.17 | 0.18 | | |
| [0.2,0.3) | 0.21 | 0.23 | 0.26 | | |
| [0.3,0.4) | 0.38 | 0.39 | | | |
| [0.4,0.5) | | | | | |
| [0.5,0.6) | 0.68 | | | | |
| [0.6,0.7) | | | | | |
| [0.7,0.8) | 0.72 | 0.78 | | | |
| [0.8,0.9) | | | | | |
| [0.9,1] | 0.94 | | | | |

# Bucket Sort

- We start by distributing items into buckets; this takes $\Theta(n)$.
- Sort within each of the $k$ buckets
    - If the input is uniformly distributed, each bucket contains $\Theta(n/k)$ numbers; and the sort in each bucket takes $\Theta(n/k \log(n/k))$, and for all buckets, it takes $\Theta(n \log(n/k)$. This is $\Theta(n)$, assuming $k = \Theta(n)$.

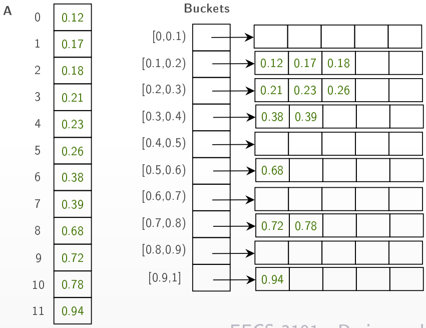


| A | |
|---|---|
| 0 | 0.78 |
| 1 | 0.17 |
| 2 | 0.39 |
| 3 | 0.26 |
| 4 | 0.72 |
| 5 | 0.94 |
| 6 | 0.21 |
| 7 | 0.12 |
| 8 | 0.23 |
| 9 | 0.68 |
| 10 | 0.38 |
| 11 | 0.18 |

| Buckets | | | | | |
|---|---|---|---|---|---|
| [0,0.1) | | | | | |
| [0.1,0.2) | 0.12 | 0.17 | 0.18 | | |
| [0.2,0.3) | 0.21 | 0.23 | 0.26 | | |
| [0.3,0.4) | 0.38 | 0.39 | | | |
| [0.4,0.5) | | | | | |
| [0.5,0.6) | 0.68 | | | | |
| [0.6,0.7) | | | | | |
| [0.7,0.8) | 0.72 | 0.78 | | | |
| [0.8,0.9) | | | | | |
| [0.9,1] | 0.94 | | | | |

# Bucket Sort

- We start by distributing items into buckets; this takes $\Theta(n)$.
- Sort within each of the $k$ buckets
  - If the input is uniformly distributed, each bucket contains $\Theta(n/k)$ numbers; and the sort in each bucket takes $\Theta(n/k\log(n/k))$, and for all buckets, it takes $\Theta(n\log(n/k)$. This is $\Theta(n)$, assuming $k = \Theta(n)$.
- Concatenate the buckets together, in order; this takes $\Theta(n)$.



| A | |
|---|---|
| 0 | 0.12 |
| 1 | 0.17 |
| 2 | 0.18 |
| 3 | 0.21 |
| 4 | 0.23 |
| 5 | 0.26 |
| 6 | 0.38 |
| 7 | 0.39 |
| 8 | 0.68 |
| 9 | 0.72 |
| 10 | 0.78 |
| 11 | 0.94 |

| Buckets | | | | | |
|---|---|---|---|---|---|
| [0,0.1) | | | | | |
| [0.1,0.2) | 0.12 | 0.17 | 0.18 | | |
| [0.2,0.3) | 0.21 | 0.23 | 0.26 | | |
| [0.3,0.4) | 0.38 | 0.39 | | | |
| [0.4,0.5) | | | | | |
| [0.5,0.6) | 0.68 | | | | |
| [0.6,0.7) | | | | | |
| [0.7,0.8) | 0.72 | 0.78 | | | |
| [0.8,0.9) | | | | | |
| [0.9,1] | 0.94 | | | | |

# BucketSort

- If the input is uniformly distributed, BucketSort is expected to run in $\Theta(n)$.
    - If the input is Not uniformly distributed, it could be that all numbers belong to one bucket, and the algorithm runs in $\Theta(n \log n)$.

# BucketSort

- If the input is uniformly distributed, BucketSort is expected to run in $\Theta(n)$.
  - If the input is Not uniformly distributed, it could be that all numbers belong to one bucket, and the algorithm runs in $\Theta(n \log n)$.
- RadixSort is a variant of BucketSort for sorting integers with the same number $b$ of digits:
  - At the first level, items are put in 10 buckets based on their most-significant digits (e.g., $102, 141, 123$ in one bucket, $218, 231, 250, 253$ in one bucket, and so on). To sort items within each bucket, the same procedure is used, except that this time items are placed in buckets based on their second most significant digit. This continues for all $b$ digits.
  - RadixSort runs in $O(n \cdot b)$, which is often useful for sorting integers.

## Sorting Summary

- We have examined a few sorting algorithms.
- When a comparison-based sorting algorithm is necessary, we often use QuickSort or HeapSort; they run in optimal $O(n \log n)$ time and are in-place.
- When items are coming from a set of $k$ possible values, for $k \in o(n \log n)$, use CountingSort.
- For sorting items which are uniformly distributed, apply BucketSort.
- For sorting integers with small number of digits, use RadixSort.