UVA DATA SCIENCE

Foundation of Computer Science for Data Science

Linear data structure: Array, Dynamic Array, Linked list

Mai Dahshan

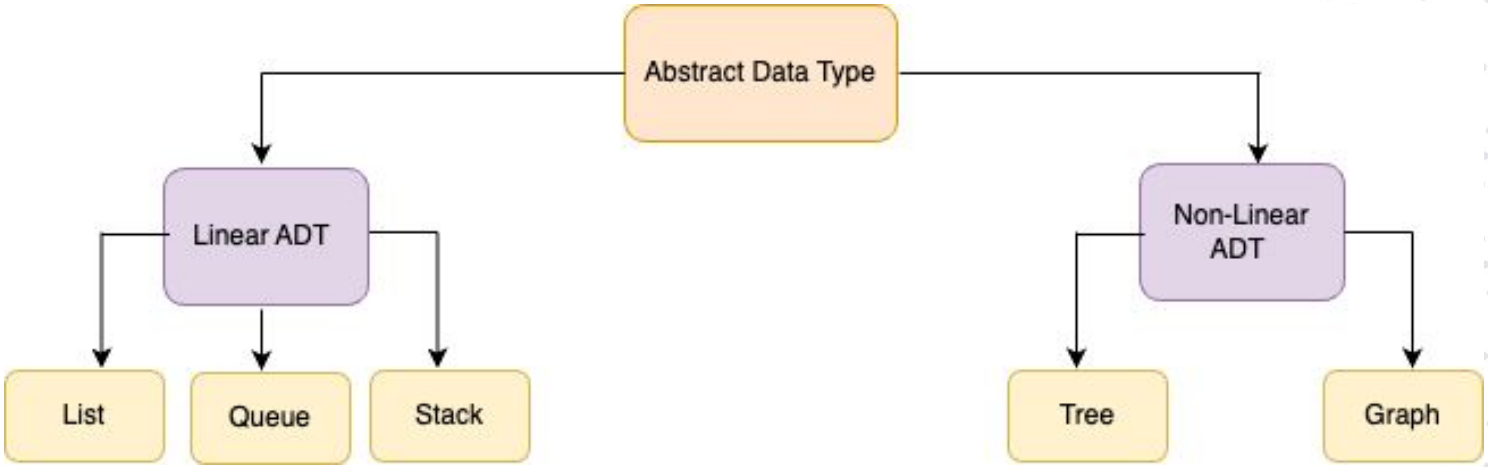
September 9, 2024

UVA DATA SCIENCE

Learning Objectives

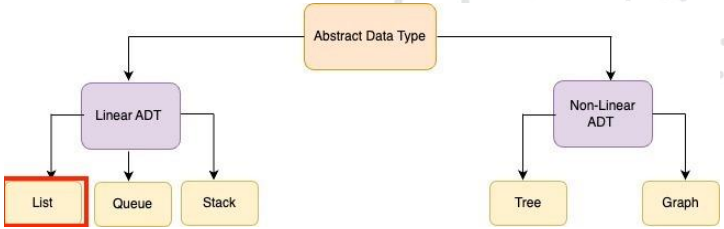
- Implement Linear ADTs
- Understand linear data structures
- Analyze the performance of linear data structures
- Implement linear data structures in Python
- Apply linear data structures to Data Science Problems
- Understand and implement searching algorithms for linear data structures

ADTs


Abstract Data Type
Linear ADT
Non-Linear ADT
List
Queue
Stack
Tree
Graph

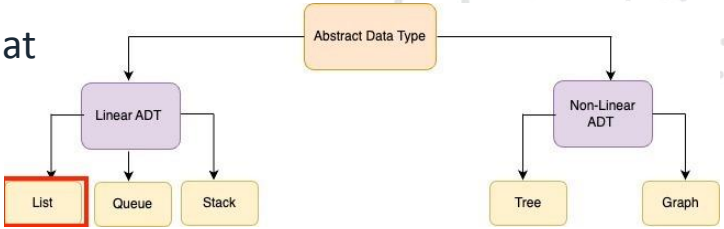
ADTs

- **List Abstract Data Type (ADT)** defines a collection of elements arranged in a sequential order. It abstracts the operations that can be performed on a list, without specifying the details of how these operations are implemented



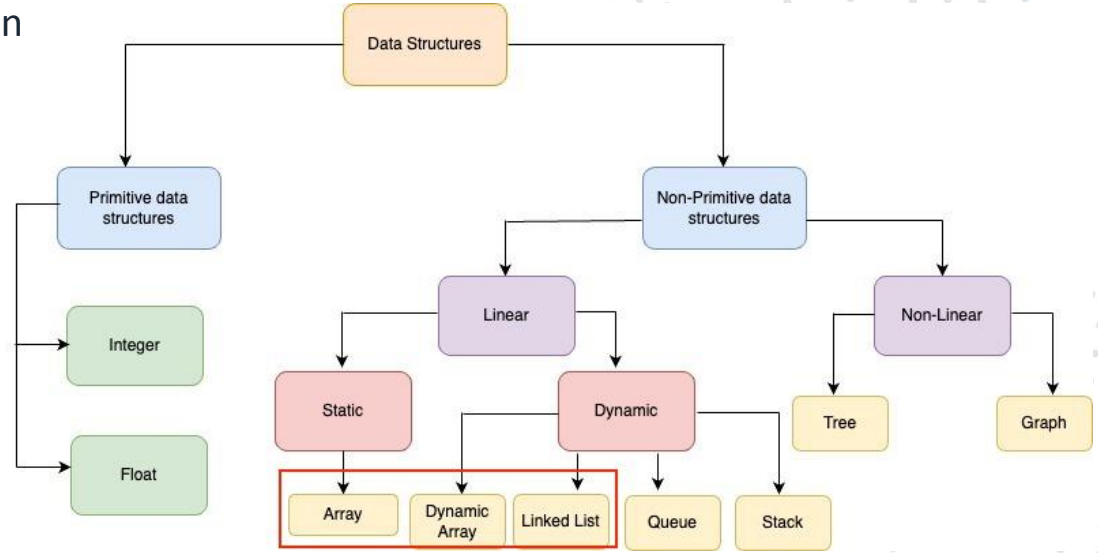
ADTs

- Key Operations of a List ADT
  - **Insertion**: Add an element at a specific position, at the beginning, or at the end of the list
  - **Deletion**: Remove an element from a specific position or based on the element's value
  - **Traversal**: Access each element in the list in order.
  - **Search**: Find an element in the list that matches a given value
  - **Sorting:-**Arranging the elements in logical order
  - **Update**: Modify the value of an element at a specific position



ADTs

- List ADT Implementations
  - Array-based List Implementation
  - Linked List Implementation

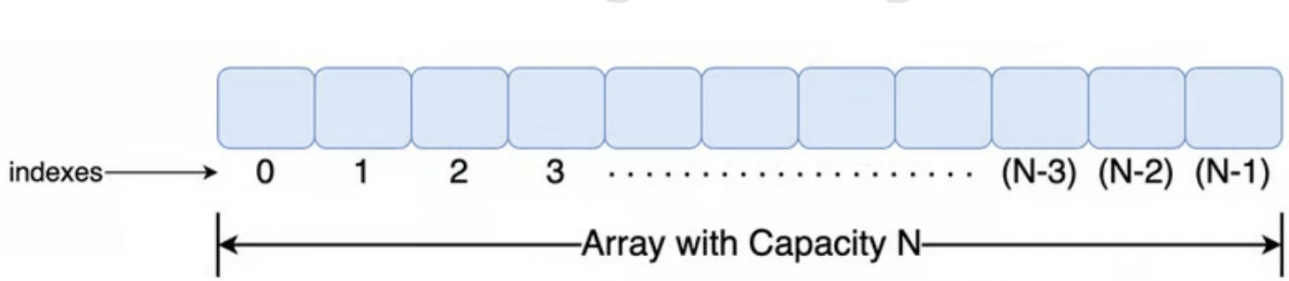


Linear Data Structures

- Data elements are arranged in a linear sequence.
- Iterate over all elements in a single run
- Elements can be accessed in a specific, linear order
- Data elements are organized in a single level, without any hierarchical or nested structure

Array

- An array is a static linear data structure with a fixed-size, indexed sequence of homogeneous data items
- Array elements are stored in consecutive memory locations
- The array length is fixed upon creation and cannot be altered



Example of 1D array of size n

Array Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access			
Insertion at the end			
Insertion at beginning/middle			
Deletion at the end			
Deletion at beginning/middle			
Update value at given index			

Array Operations Visualization: https://visualgo.net/en/array

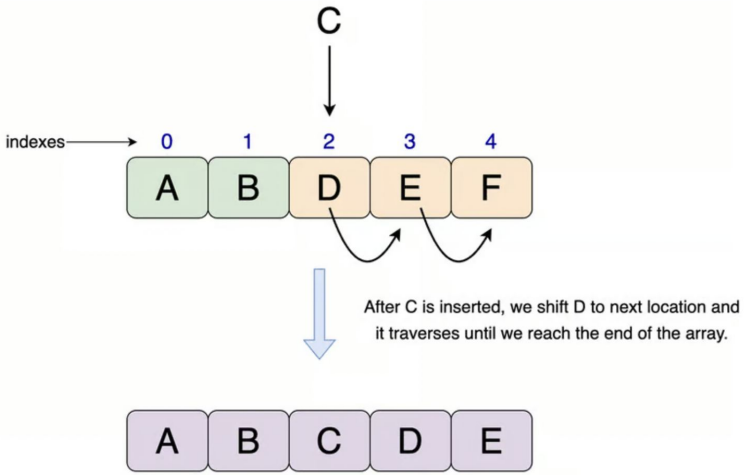
Array Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access	O(1)	O(1)	O(1)
Insertion at the end	O(1)	O(1)	O(1)
Insertion at beginning/middle	O(n)	O(n)	O(n)
Deletion at the end	O(1)	O(1)	O(1)
Deletion at beginning/middle	O(n)	O(n)	O(n)
Update value at given index	O(1)	O(1)	O(1)

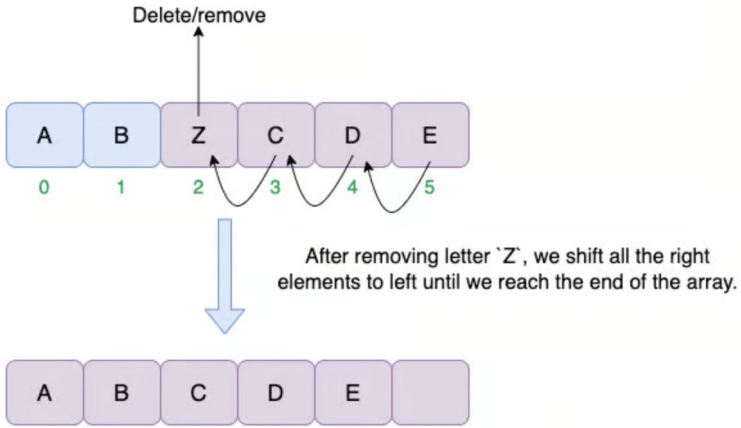
Array Operations Visualization: https://visualgo.net/en/array

Array Insertion and Deletion

Inserting an element in middle



Deleting an element in middle



Pros and Cons of Array

- Pros:
  - Fast lookups (random access)
  - Fast appends
  - Cache friendliness
- Cons:
  - Fixed size.
  - Memory unused or wasted.
  - Costly inserts
  - Costly deletes

Python Implementations of Array

- NumPy Array
- Array Module
- Tuple
- Tensorflow, Pytorch tensor

Python Implementation of Array

Feature	NumPy Array	Array Module	Tuple	Tensor
Mutability	Mutable elements can be changed	Mutable elements	Immutable	Mutable or immutable depending on the framework (e.g., TensorFlow, PyTorch)
Data Type Support	Supports multiple homogeneous data types, including complex numbers	Supports homogeneous data types	Supports heterogeneous data types	Supports homogeneous data types
Operations	Supports a wide range of mathematical and statistical operations	Limited operations; mainly basic array manipulations	Limited operations; mostly indexing and slicing	Supports a wide range of mathematical operations, including gradient computation in deep learning frameworks

Python Implementation of Array

Feature	NumPy Array	Array Module	Tuple	Tensor
Performance	Optimized for numerical operations with efficient performance	Less optimized suitable for basic needs	Not optimized for numerical operations	Optimized for high-performance computations in deep learning frameworks
Use Cases	Scientific computing, data analysis, numerical computations	Basic array manipulations, simple data storage	Data storage, fixed collections of elements	Machine learning, deep learning, high-performance computing

Uses of Array in Data Science

- **Text analysis**: Storing text data as arrays of word vectors, enabling operations like sentiment analysis for small/medium datasets
- **Regression analysis**: Using arrays to store both feature vectors and target values for training a regression model
- **Gradient Calculations**: In backpropagation of deep learning models, gradients of the loss function with respect to network parameters (weights and biases) are stored as arrays for efficient computation and updates
- **Deep learning libraries** like TensorFlow and PyTorch leverage the power of array operations to perform computations across entire arrays simultaneously, significantly improving processing speed compared to element-wise operations

Array

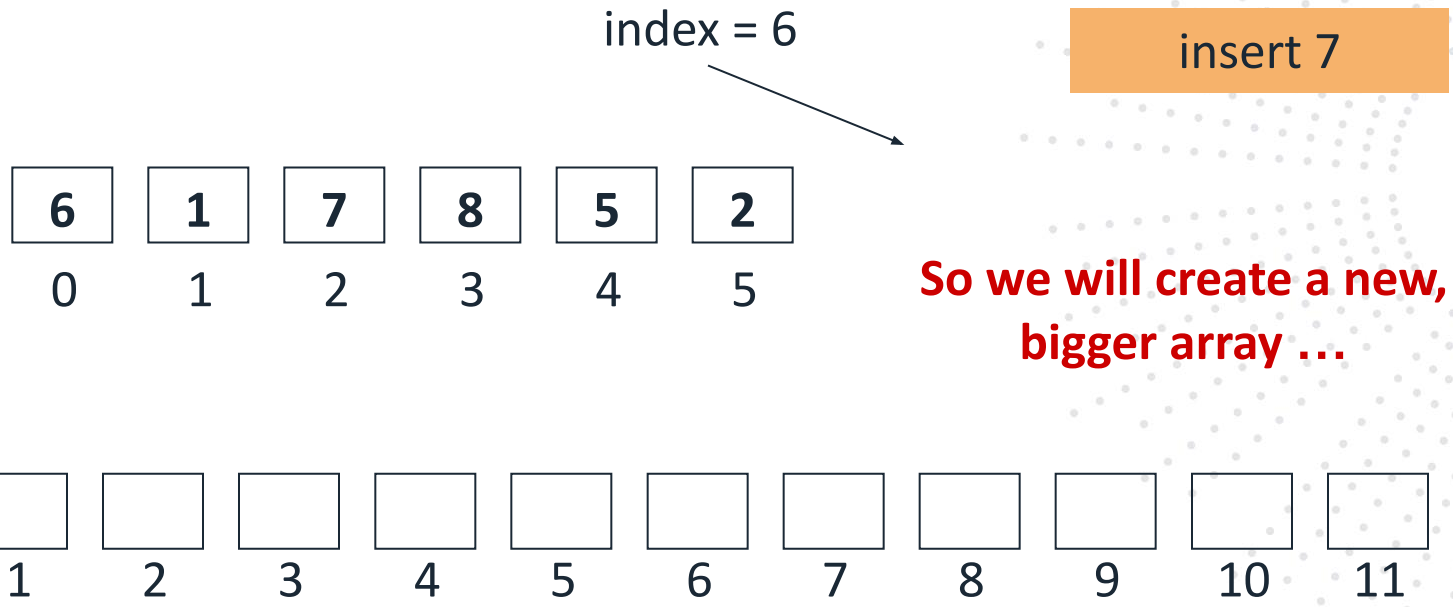
insert 2

Array

insert 7

The array is full and there is no room for a new item!

Array


index = 6
insert 7
6
1
7
8
5
2
0
1
2
3
4
5
So we will create a new, bigger array …
0
1
2
3
4
5
6
7
8
9
10
11

Dynamic Array

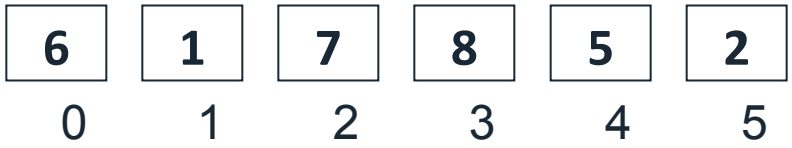
- A dynamic arrays is a linear data structure that allows resizing during runtime
- They can grow or shrink as needed, making them more flexible than static arrays
- Like static arrays, elements are stored in contiguous memory locations

Dynamic Array

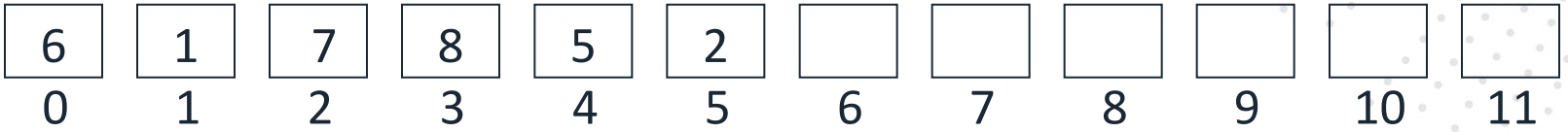
- How Dynamic Arrays Work
  - Typically implemented using a static array that doubles in size when capacity is exceeded.
  - When full, a new array is created, and all elements are copied over.
    - Usually doubles the size (growth factor of 2), but can vary.
    - It’s good, but… copying an array is computationally burdensome: O(n)

Dynamic Array

insert 7



… copy the elements of the old array into it new old

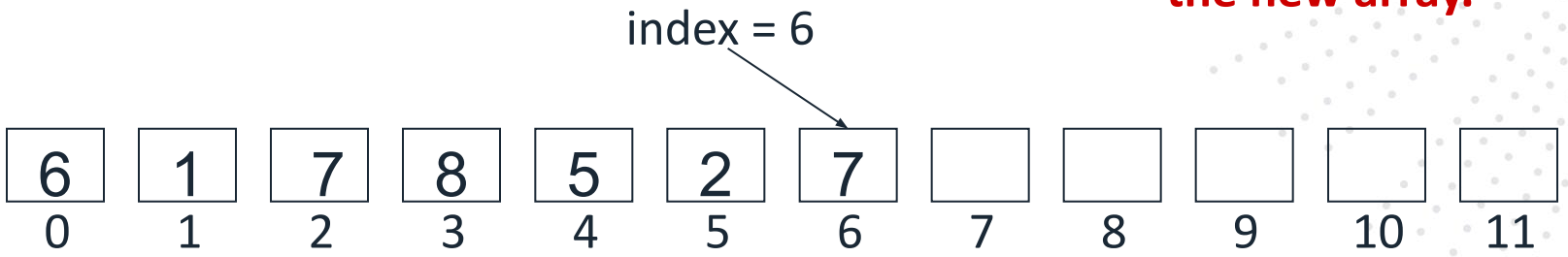


Dynamic Array

The old array will eventually be deleted by garbage collector

insert 7

… and finally insert 7 into the new array.



Dynamic Array

- How to minimize the number of copies
  - Difficult to do without knowing the max capacity needed
- Exponential Growth Scheme
  - When attempting to add element and capacity is full, allocate a new array with double capacity and copy.
  - How many copies are needed if we follow the above re-sizing rule?
    - Assume the max capacity is (which is unknown apriori)
    - Assume array is initialized to size *i*.
    - Total number of allocation-copy events: $i * log_2(N - i)$

Dynamic Array Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access			
Insertion at the end			
Insertion at beginning/middle			
Deletion at the end			
Deletion at beginning/middle			
Update value at given index			
Resizing			

Dynamic Array Operations Visualization: https://visualgo.net/en/array

Dynamic Array Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access	O(1)	O(1)	O(1)
Insertion at the end	O(1)	O(1) amortized cost	O(n)
Insertion at beginning/middle	O(n)	O(n)	O(n)
Deletion at the end	O(1)	O(1)	O(1)
Deletion at beginning/middle	O(n)	O(n)	O(n)
Update value at given index	O(1)	O(1)	O(1)
Resizing	O(1)	O(1) amortized cost	O(n)

Dynamic Array Operations Visualization: https://visualgo.net/en/array

Pros and Cons of Dynamic Array

- Pros:
  - Fast lookups (random access)
  - Flexible Size: Can dynamically adjust to accommodate new elements.
  - Cache friendliness
- Cons:
  - Resizing Cost
  - Memory Allocation

Python Implementation of Dynamic Array

- List
- Python using custom classes

Applications of Dynamic Array in Data Science

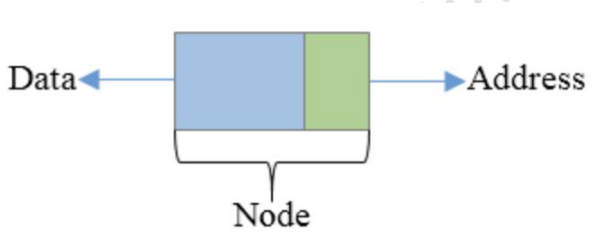
- In **online learning or real-time ML applications**, dynamic arrays can efficiently handle incoming data streams, allowing models to learn from data as it arrives.
- In **natural language processing (NLP) and time series analysis**, dynamic arrays store sequences of varying lengths, such as sentences or time series data points,

Dynamic Array

- What if we have a massive list and constantly need to add new data?
- The copying process when the list reaches capacity can become cumbersome.
  - Is there a way to avoid this issue?

Linked List

- A linked list is a linear data structure where elements, called nodes, are stored in non-contiguous memory locations.
- Each node contains a data value and a reference(s) (or pointer(s))

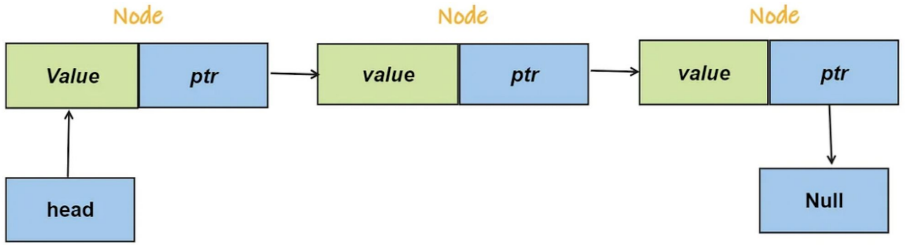


Types of Links List

- **Singly Linked List**
- **Doubly Linked List**
- **Circular Linked List**
- Skip List (Advanced)
- Unrolled Linked List (Advanced)

Singly Linked List

- A singly linked list is a linear data structure where each element (node) points to the next node in the sequence.
- It consists of a series of nodes connected in a single direction



Singly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access			
Insertion at the head			
Deletion at the head			

Singly linked list Operations Visualization: https://csvistool.com/LinkedList
https://visualgo.net/en/list

Singly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access	O(1)	O(n)	O(n)
Insertion at the head	O(1)	O(1)	O(1)
Deletion at the head	O(1)	O(1)	O(1)

Singly linked list Operations Visualization: https://csvistool.com/LinkedList
https://visualgo.net/en/list

Singly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle			
Insertion at the end			
Deletion at the middle			
Deletion at the end			

*Assume no direct reference

Singly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle	O(n)	O(n)	O(n)
Insertion at the end	O(n)	O(n)	O(n)
Deletion at the middle	O(n)	O(n)	O(n)
Deletion at the end	O(n)	O(n)	O(n)

*Assume no direct reference

- Inserting a new node is an O(1) operation if you have a reference to the node before the insertion point
- Deleting a node is O(1) if you have a reference to the node before the deletion point
- Finding the position in the middle or tail of the list dominates the time complexity, making it O(n)

Python Implementation of Singly Linked List

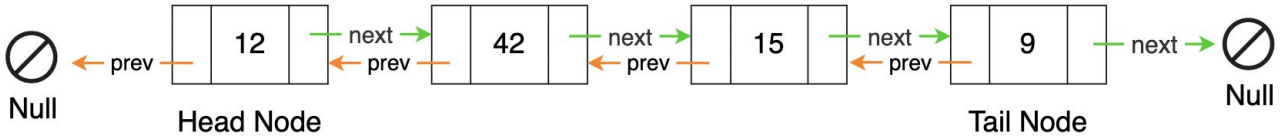
- Python using custom classes

Pros and Cons of Singly Linked List

- Pros:
  - Is able to grow in size as needed
  - Does not require the shifting of items during insertions and deletions
- Cons
  - Increased Memory Usage
  - Increased Overhead
  - Unidirectional Traversal
  - Accessing an element does not take a constant time

Doubly Linked List

- A doubly linked list is a linear data structure where each element (node) points to the next node and previous node in the sequence



Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access			
Insertion at the head			
Deletion at the head			

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Access	O(1)	O(n)	O(n)
Insertion at the head	O(1)	O(1)	O(1)
Deletion at the head	O(1)	O(1)	O(1)

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle			
Insertion at the end			
Deletion at the middle			
Deletion at the end			

*Assume tail pointer

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle	O(n)	O(n)	O(n)
Insertion at the end	O(1)	O(1)	O(1)
Deletion at the middle	O(n)	O(n)	O(n)
Deletion at the end	O(1)	O(1)	O(1)

*Assume tail pointer

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle			
Insertion at the end			
Deletion at the middle			
Deletion at the end			

*Assume no direct reference

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Doubly Linked List Operations Time Complexity

Operation	Best Case	Average Case	Worst Case
Insertion at the middle	O(n)	O(n)	O(n)
Insertion at the end	O(n)	O(n)	O(n)
Deletion at the middle	O(n)	O(n)	O(n)
Deletion at the end	O(n)	O(n)	O(n)

Insertion cost + Traversing cost

Deletion cost + Traversing cost

*Assume no direct reference

Doubly linked list Operations Visualization:https://csvistool.com/DoublyLinkedList
https://visualgo.net/en/list

Pros and Cons of Doubly Linked List

- Pros:
  - Is able to grow in size as needed
  - Does not require the shifting of items during insertions and deletions
  - Able to traverse the list in any direction
  - Reversing the list is simple
- Cons
  - Increased Memory Usage
  - Increased Overhead
  - Accessing an element does not take a constant time

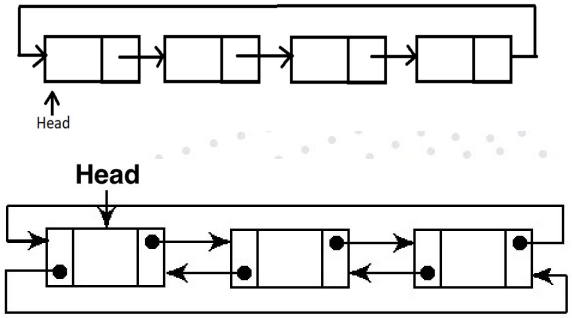
Python Implementation of Doubly Linked List

- Python using custom classes

Circular Linked List

- A circular linked list is a linear data structure where the last node points back to the first node, forming a circle

- Types of Circular Linked Lists
  - Singly Circular Linked List
  - Doubly Circular Linked List



Applications of Linked list in Data Science

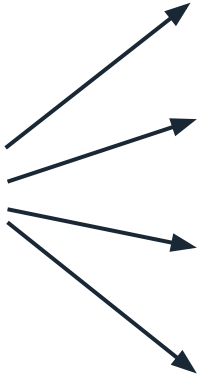
- **Text Processing:** Linked lists can be used to represent and manipulate strings or large texts where frequent insertions and deletions are required.

- **Replay Buffers:** In reinforcement learning, linked lists can be used to implement replay buffers that store past experiences or states for training.

Applications of Linear Data Structures in Data Science

- Q: Image classification, how many way to implement?

Is it a cat or dog?



1D Array: Feed forward net, Traditional ML

2D, 3D Array: CNN

Linked list: Recurrent Neural Net

Graph: Graph Neural Net

Linear Data Structure Activity

- For each case, identify the most suitable linear data structure to use
  - **Case 1:** You were asked to design an e-commerce application that allows users to freely modify their cart contents. The shopping cart should automatically update in real-time as items are added or removed. Which data structure would you use to store the cart contents?
  - **Case 2:** You were asked to design a weather monitoring system where data collection is restricted to a fixed timeframe. The system should store daily temperature readings for a specific city over the course of a week. Which data structure would you use to store the temperature?

Linear Data Structure Activity

- **Case 3:** You were asked to design an application that tracks visited URLs in the order they were accessed, allowing users to navigate backward or forward through their browsing history. Which data structure would you use to store visited URLs?
- **Case 4:** You are asked to design a web application that manages and displays items such search results, user comments, or product listings, which data structure would you use to store these items?
- **Case 5:** You are asked to design a music player program that handles a playlist, allowing users to frequently add and remove tracks, and also manages metadata for each track. which data structures would you use in this case?

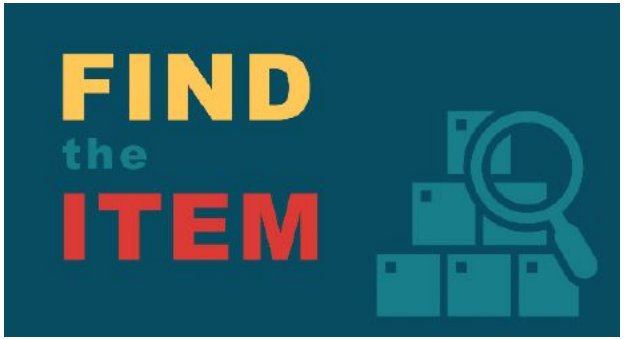
UVA DATA SCIENCE

Foundation of Computer Science for Data Science

Searching linear data structure

Searching

- As we gather and manage data, we want to search for specific items within a data container



Key Questions to Address Before Searching

- Before choosing a search algorithm, the following key questions need to be addressed:
  - What is the structure of the data (how is the data stored)?
  - What is to be searched: numbers, text or graphs?
  - How large is the set of elements to be searched?
  - Is data presorted in some order such as ascending or descending order?
  - What are the performance requirements? Are there constraints on time complexity (speed) or space complexity (memory usage)?

Why Learn Search Algorithms?

- Searching for data is one of the fundamental operations in computing
- Often, the efficiency of a search algorithm is what distinguishes a fast program from a slow one.
- In the age of big data, efficiently searching through data is crucial for maintaining a competitive edge

Searching Linear Data Structures

- Parameters that affect search algorithm selection:
  - Data size
  - Whether the list is **sorted**
  - Whether all the elements in the list are **unique** or have **duplicate** values
  - whether the search is performed frequency or not

Sequential/Linear Search

- Sequential/Linear search is a searching algorithm that finds a target value by sequentially checking each element in a list until the target is found or the end is reached.
    - **Input:** A collection of data (e.g., a list, array, or dataset) and a target value that needs to be located within this collection.
    - **Output:** The position or index of the target value in the collection if it is present; otherwise, a special value indicating that the target value is not found in the collection

Sequential/Linear Search

Algorithm

- Check if the list is Empty: if the list is empty, return -1 (sentinel value)
- Initialization: Start at the first element of the list (index 0)
- Iterate through list: Traverse each element of the list
  - Compare and check the current element with the target element.
    - If the current element matches the target element, return the index of the matching element
    - If the current element does not match the target element, move to the next element in the list
- If the end of the list is reached without finding the target element, the search is unsuccessful, and the algorithm returns a value indicating that the element is not in the list (-1 or 'None')

Search(A,n)
Input: An array $A[n]$, where $n \geq 1$; an item x
Output: Index where x occurs in A, or -1

for $i \leftarrow 0$ to $n-1$ do
    if $A[i] = x$ then return(i);
return(-1);

Sequential/Linear Search

- Examples

Sequential/Linear Search: Time Complexity

Operation	Best Case	Average Case	Worst Case
Item present			
Item not present			

Linear Search Visualization: https://www.cs.usfca.edu/~galles/visualization/Search.html

Sequential/Linear Search: Time Complexity

Operation	Best Case	Average Case	Worst Case
Item present	O(1)	O(n/2) = O(n)	O(n)
Item not present	O(n)	O(n)	O(n)

Linear Search Visualization: https://www.cs.usfca.edu/~galles/visualization/Search.html

Sequential/Linear Search

- Pros:
  - Linear search is easy to understand and implement.
  - Works on unsorted lists, so there's no need for pre-processing or sorting the data.
  - Can be used on any data structure that allows sequential access, such as arrays, linked lists, or files. Efficient for small lists
- Cons
  - The time complexity is O(n), which means the time taken grows linearly with the number of elements. This can be very slow for large lists.
  - Not Optimal for Repeated Searches

Binary Search

- Binary search is a searching algorithm for finding a target value within a sorted array or list by repeatedly dividing the search interval in half.
  - **Input:** A collection of sorted data (e.g., a list, array, or dataset) and a target value that needs to be located within this collection.
  - **Output:** The position or index of the target value in the collection if it is present; otherwise, a special value indicating that the target value is not found in the collection

Binary Search

Algorithm

- Check for Empty list: if the list is empty, return -1
- Initialize two variables: **low** set to index 0(lower bound of the search space), and **high** set to n-1 (upper bound of the search space).
- Traverse through the list while low<=high
  - Calculate the Midpoint (mid= floor((low+high)/2)
  - Compare the Midpoint Value with the Target
    - if target value == list[mid], then target found at index (mid). Return mid
    - if target value > list[mid], then low=mid+1
    - if target value < list[mid], then high=mid-1
- If low becomes greater than high, it means the target value is not present in the list. Return -1

```
BinarySearch(A[0..n − 1], K)
    //Implements nonrecursive binary search
    //Input: An array A[0..n − 1] sorted in ascending order and
    //       a search key K
    //Output: An index of the array's element that is equal to K
    //       or −1 if there is no such element
    l ← 0;   r ← n − 1
    while l ≤ r do
        m ← ⌊(l + r)/2⌋
        if K = A[m] return m  # Target found at index (mid)
        else if K < A[m] r ← m − 1 # Discard right part
        else l ← m + 1 # Discard left part
    return −1
```

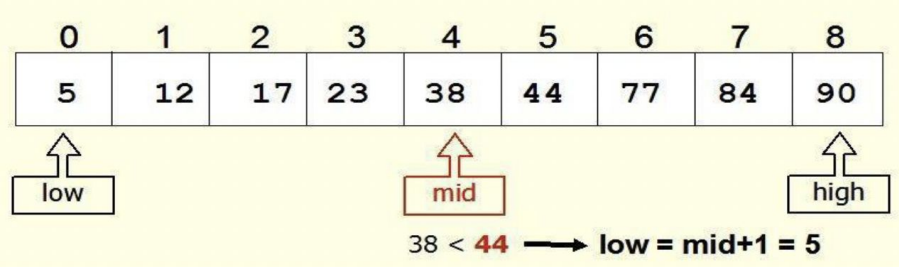
Binary Search

- How to find the midpoint?

$$\text{mid} = \left\lfloor \frac{\text{low}+\text{high}}{2} \right\rfloor$$

  - low: The starting index of the search range
  - high: The ending index of the search range

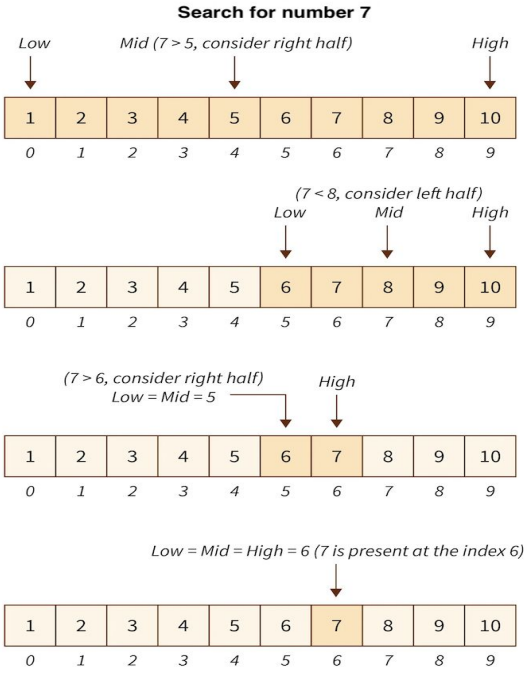
- Example



$$\text{mid} = \frac{0+8}{2} = \frac{8}{2} = 4$$

Binary Search

- Example



Binary Search: Time Complexity

Operation	Best Case	Average Case	Worst Case
Item present			
Item not present			

Binary Search Visualization: https://www.cs.usfca.edu/~galles/visualization/Search.html

Binary Search: Time Complexity

Operation	Best Case	Average Case	Worst Case
Item present	O(1)	O(log n)	O(log n)
Item not present	O(log n)	O(log n)	O(log n)

Binary Search Visualization: https://www.cs.usfca.edu/~galles/visualization/Search.html

Binary Search

- The data must be SORTED !
  - Before implementing Binary search, there is a non-optional pre-processing step that is required on the input data (in an array or list...) if not sorted
- Binary Search is also called half-interval search
  - No matter what is the size of the data, each iteration reduces the number of items to be searched by half
  - It’s a good example of the divide-and-conquer strategy (Complexity: O(lg n))

Binary Search

- Much more efficient than sequential search
  - Especially if search is done many times (sort once, search many times)
  - Might be worth the overhead (once) if you will use this algorithm multiple times afterwards

Binary Search Activity

- Some binary search inputs will be provided
  - List of int values, plus a **target** to find
- Respond with:
  - What **index** is returned, and
  - The **sequence of mid index values** that were compared to the target to get that answer

Binary Search Example #1

- Input: | -1 | 4 | 5 | 11 | 13 | and target 4

  Index returned is?

  mid indices (positions) compared to target-value:

1) -1 4
2) 0 1
3) 5 -1 4
4) 2 0 1
5) other

Binary Search Example #1 Solution

- Input: | -1 | 4 | 5 | 11 | 13 | and target 4

  Index returned is? **1**

  mid indices (positions) compared to target-value:

1) -1 4
2) 0 1
3) 5 -1 4
4) **2** **0** **1**
5) other

Binary Search Example #2

- Input:

-5	-2	-1	4	5	11	13	14	17	18

and target 3

Index returned is?

mid indices (positions) compared to target-value

1) 5 2 3 4
2) 4 1 2 3
3) 4 2 3 4
4) other

Binary Search Example #2 Solution

- Input:

-5	-2	-1	4	5	11	13	14	17	18

and target 3

Index returned is? **-1**

mid indices (positions) compared to target-value

1) 5 2 3 4
2) **4 1 2 3**
3) 4 2 3 4
4) other

Binary Search Example

- Input: | -1 | 4 | 5 | 11 | 13 | and target 13

  Index returned is?

  mid indices (positions) compared to target-value:

1) 0 1 2 3 4
2) 2 4
3) 2 3 4
4) other

Binary Search Example #3 Solution

- Input: | -1 | 4 | 5 | 11 | 13 | and target 13

  Index returned is? **4**

  mid indices (positions) compared to target-value:

1) 0 1 2 3 4
2) 2 4
3) **2** **3** **4**
4) other

Sorting linear data structure

Learning Objectives

- Understand and implement sorting algorithms for linear data structures
- Understand the complexity of sorting algorithms

Sorting

- Sorting is a process of arranging the values in a list or collection into a specific order (either ascending or descending order)
  - Problem: There are **n** comparable elements in a list and we want to rearrange them to be in increasing/decreasing order
  - Input: A sequence of n numbers, denoted as $\langle a_1, a_2, \ldots, a_n \rangle$
  - Output: A reordered sequence $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Applications of Sorting

- E-commerce
- Search Algorithms (i.e., binary search)
- Financial Systems (i.e., stock market)
- Genomics
- and more

Sorting Algorithms

- There are many sorting algorithms, with over a dozen being widely used
- New algorithms are frequently created by combining elements of existing ones

Sorting Algorithms

- Simple sorts
  - Insertion sort
  - Selection sort

Efficient sorts

Merge sort

- Heapsort
- Quicksort

- Bubble sort and variants
  - Bubble sort
  - Shell sort
  - Comb sort

Distribution sort

Counting sort

- Bucket sort
- Radix sort

- …Etc!

Insertion sort

Insertion sort

- Insertion sort is a simple and appropriate algorithm for small inputs

- It's commonly used by card players to sort cards in hand by inserting each new card into its proper position

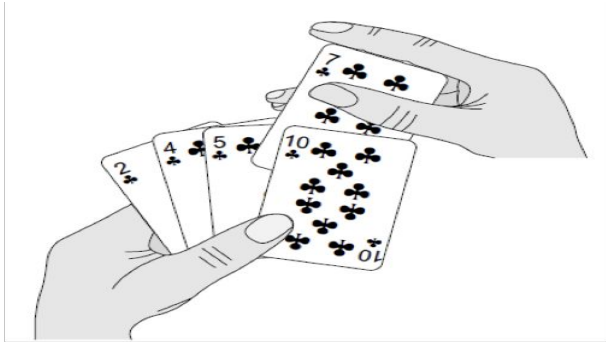
How Insertion Sort Works

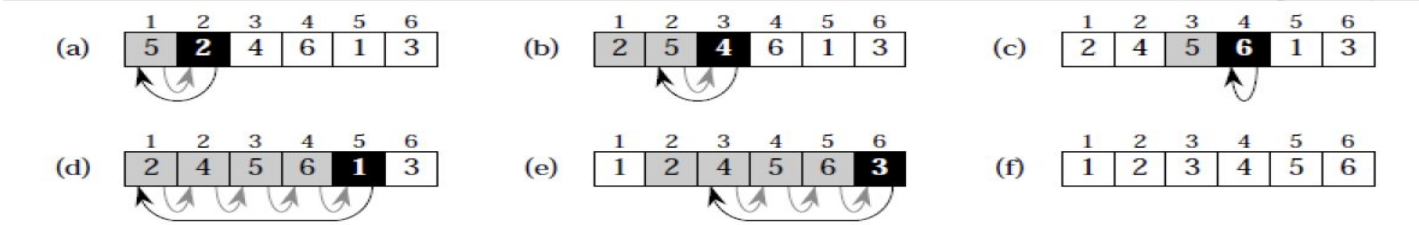
- The list is divided into two spaces: sorted and unsorted
- Start with the first element, treating it as a sorted sublist.
- Pick the next element from the unsorted portion.
- Compare it with elements in the sorted sublist.
- Shift larger elements in the sorted sublist to the right.
- Insert the picked element in its correct position.
- Repeat for all remaining elements.

A list of n elements will take at most n-1 passes to sort the data

this is because the first element is already sorted. This n-1 is really talking about each pass of an element from the unsorted list to the sorted list

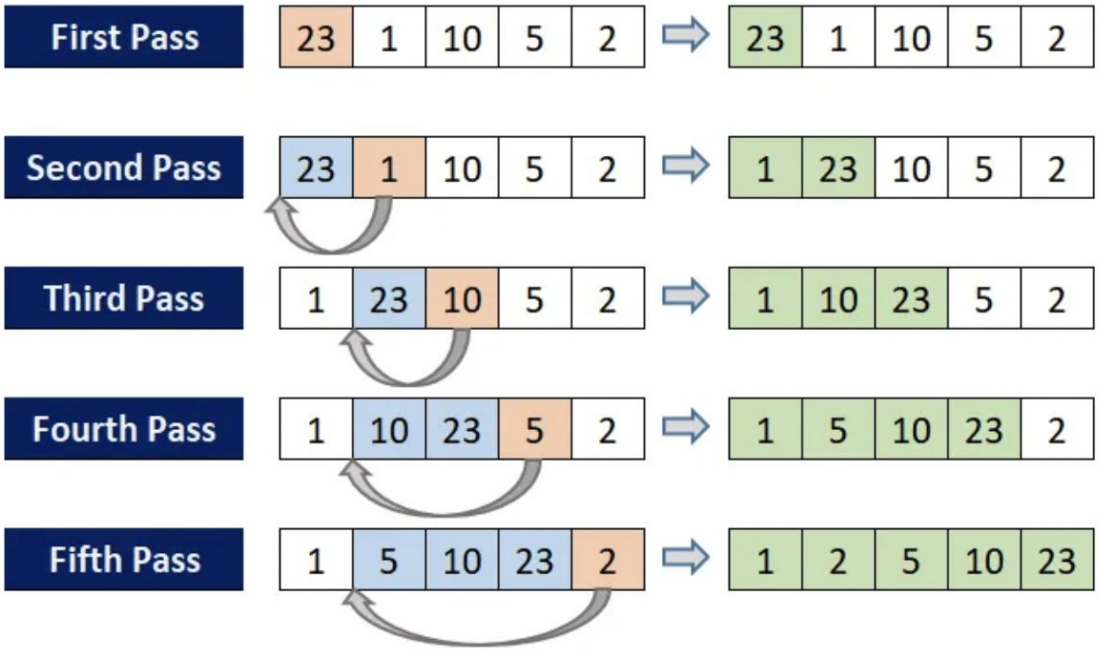
Insertion sort





Insertion Sort Visualization: https://visualgo.net/en/sorting

Insertion Sort Example



Advantages and Disadvantages Insertion Sort

Advantages

- It is useful for small sets of data
- It is easy to implement
- It does not require additional memory
- It can be fast if the data is almost nearly sorted

Disadvantages

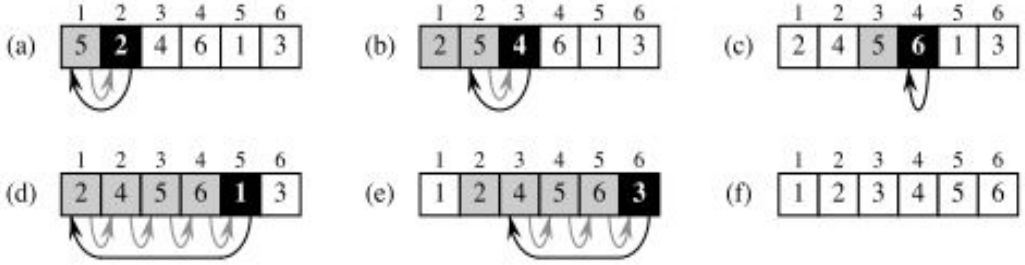
- Not suitable for large datasets due to $O(n^2)$ time complexity.
- Requires more comparisons and shifts for larger, unsorted lists

it's O(n^2) in the worst case.

Insertion Sort: Pseudocode

6 5 3 1 8 7 2 4

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1].
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```



Insertion Sort in Python

```python
def insertion_sort(arr):

    # Traverse the list starting from the second element
    for i in range(1, len(arr)):
        # The element to be positioned in the sorted part of the list
        key = arr[i] <-- key is the value you're moving from the unsorted array to the sorted array

        # Index of the last element in the sorted portion of the list
        j = i - 1

        # Move elements of arr[0..i-1], that are greater than 'key', to one position ahead
        # of their current position, making room for the 'key' element to be inserted
        while j >= 0 and key < arr[j]:
            # Shift the larger element to the right
            arr[j + 1] = arr[j]
            #The decrement happens each time the inner while loop executes
            j -= 1

        # Insert 'key' in its correct position in the sorted portion of the list
        arr[j + 1] = key

    # Return the sorted list
    return arr
```

Insertion Sort: Complexity Analysis- Best Case

```
def insertion_sort(arr):

    for i in range(1, len(arr)):          # O(n)
        key = arr[i]                      # O(1)

        j = i - 1                         # O(1)

        while j >= 0 and key < arr[j]:    # O(1)
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key                  # O(1)

    return arr                            # O(1)
```

Overall time Complexity

O(n)

(n-1)*O(1)

(n-1)*O(1)

(n-1)*O(1)

(n-1)*O(1)

O(1)

T(n) = O(n) +(n-1)*O(1) + (n-1)*O(1) + (n-1)*O(1) + (n-1)*O(1) + O(1)

= O(n) + O(n) + O(n) + O(n) + O (n) + O (1) = 5 O(n) + O(1) ⇒ **O(n)**

Insertion Sort: Complexity Analysis- Worst Case

```
def insertion_sort(arr):

    for i in range(1, len(arr)):          # O(n)
        key = arr[i]                      # O(1)

        j = i - 1                         # O(1)

        while j >= 0 and key < arr[j]:    # O(i)
            arr[j + 1] = arr[j]           # O(1)
            j -= 1                        # O(1)

        arr[j + 1] = key                  # O(1)

    return arr                            # O(1)
```

Overall time Complexity

Overall time Complexity

O(n)

(n-1)*O(1)

(n-1)*O(1)

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

$\sum_{i=1}^{n-1} i*O(1)$

$\sum_{i=1}^{n-1} i* O(1)$

(n-1)*O(1)

O(1)

$T(n) = O(n) + (n-1)*O(1) + (n-1)*O(1) + \sum_{i=1}^{n-1} O(i) + \sum_{i=1}^{n-1} i*O(1) + \sum_{i=1}^{n-1} i*O(1) + (n-1)*O(1) + O(1)$

$T(n) = 4\,O(n) + 3\,O(n^2) + O(1) \Rightarrow \mathbf{O(n^2)}$

Selection sort

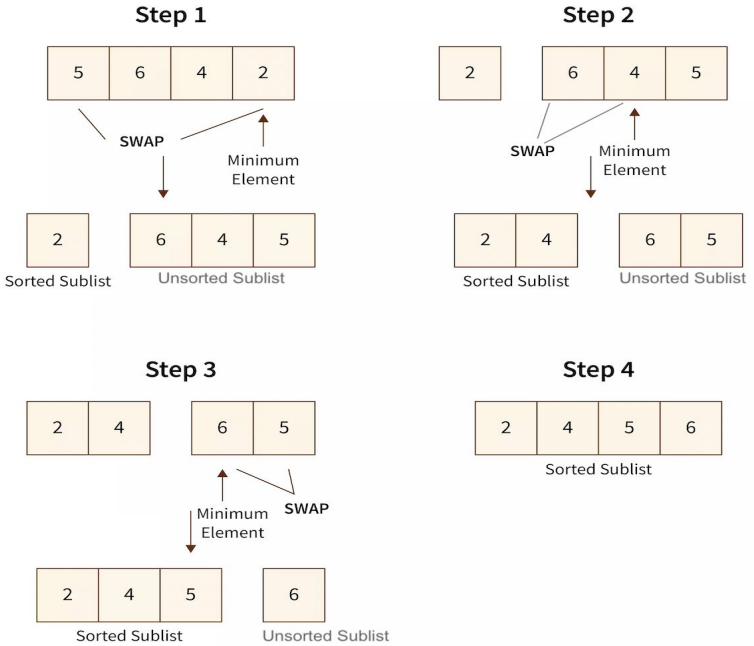
Selection sort

- Selection Sort is a comparison-based sorting algorithm that works by repeatedly finding the minimum (or maximum) element from the unsorted portion of the list and moving it to the sorted portion

How Selection Sort Works

- The list is conceptually divided into two parts: the sorted part, which is initially empty, and the unsorted part, which contains all the elements.
- Initialization: Start with an unsorted list.
- Find the Minimum: Iterate through the unsorted portion of the list to find the smallest (or largest) element
- Swap the found minimum element with the first element of the unsorted portion.
- Repeat steps 2 to 4 until the entire list is sorted.

Selection Sort Example



Selection Sort Visualization: https://visualgo.net/en/sorting

Advantages and Disadvantages Selection Sort

Advantages

- Simple to understand and implement
- No additional memory required

Disadvantages

- Inefficient for large lists

Selection Sort: Pseudocode

ALGORITHM *SelectionSort*$(A[0..n-1])$

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

Selection Sort in Python

```python
def selection_sort(arr):

    # Get the number of elements in the array
    n = len(arr)

    # Traverse through all array elements
    for i in range(n-1):
        # Assume the minimum element is at the current position i
        min_index = i  # O(1) => This is a constant-time operation. It only involves setting a variable

        # Find the index of the minimum element in the remaining unsorted portion
        for j in range(i+1, n):
            # If the element at index j is smaller than the current minimum
            if arr[j] < arr[min_index]:
                # Update min_index to the index of the new minimum element
                min_index = j

        # Swap the found minimum element with the element at index i
        # Only swap if min_index is different from i
        if min_index != i:
            arr[i], arr[min_index] = arr[min_index], arr[i]

    return arr
```

Selection Sort: Complexity Analysis (Worst/Average/Best)

```
def selection_sort(arr):

    n = len(arr)                                    # O(1)

    for i in range(n-1):                            # O(n)
        min_index = i                               # O(1)

        for j in range(i+1, n):                     # O(n-i-1)
            if arr[j] < arr[min_index]:             # O(1)
                min_index = j                       # O(1)

        if min_index != i:                          # O(1)
            arr[i], arr[min_index] = arr[min_index], arr[i] # O(1)

    return arr                                      # O(1)
```

Overall time Complexity

O(1)

O(n)

n* O(1)

$\sum_{i=0}^{n-2} O(n-i-1)$

$\sum_{i=0}^{n-2} (n-i-1)* O(1)$

$\sum_{i=0}^{n-2} (n-i-1)* O(1)$

n* O(1)

n* O(1)

O(1)

$T(n) = O(1) + O(n) + n*O(1) + \sum_{i=0}^{n-2} O(n-i-1) + \sum_{i=0}^{n-2} O(1)*(n-i-1) + \sum_{i=0}^{n-2} O(1)*(n-i-1) + n* O(1) + n* O(1) + O(1)$

$= 2\ O(1) + 4\ O(n) + 3\ O(n^2)$ => **$O(n^2)$**

Bubble Sort

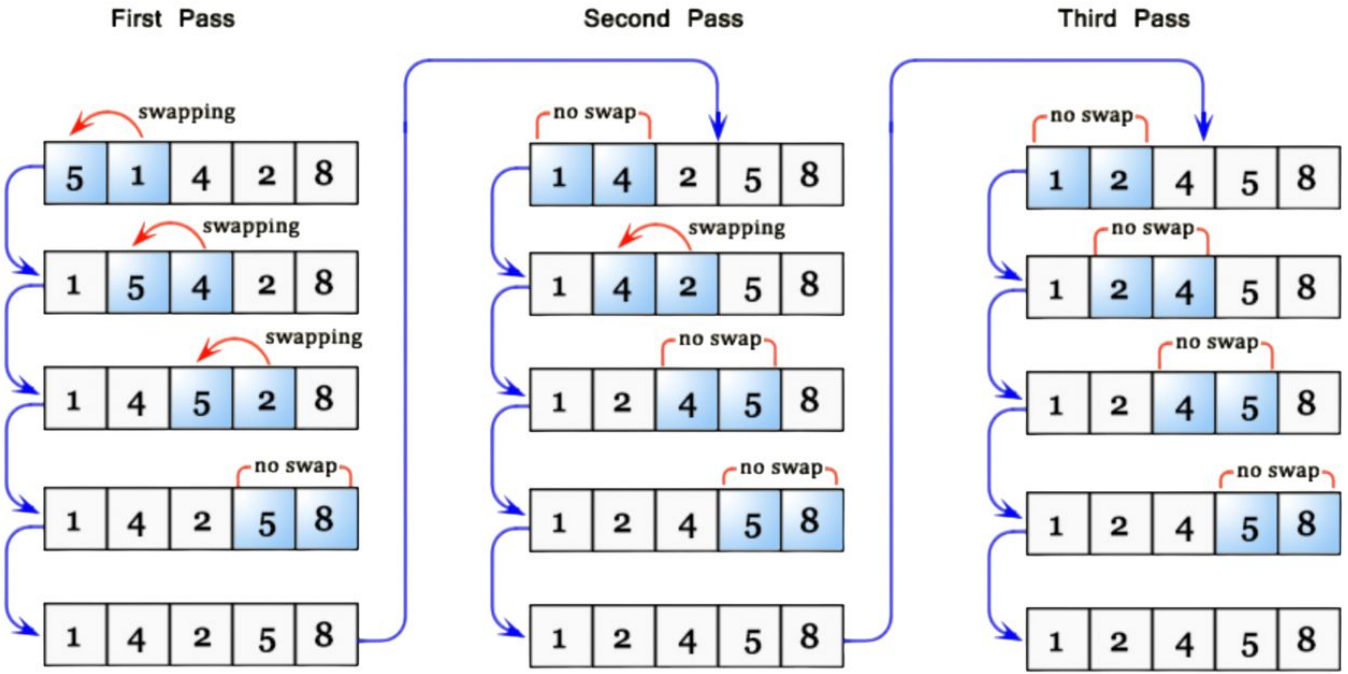
Bubble sort

- Bubble Sort is a simple, comparison-based sorting algorithm. It repeatedly steps through a list of elements, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the list is sorted.
- The algorithm gets its name because larger elements "bubble" to the top (end of the list) with each pass.

How Bubble Sort Works?

- Start with an unsorted list.
- Compare adjacent elements:
  - If the first element is greater than the second, swap them.
  - If not, leave them as they are.
- Repeat this process for every element in the list (this is one pass).
- After each pass, the largest unsorted element moves to its correct position at the end.
- Continue this process until no more swaps are needed.

Bubble Sort Example



Bubble Sort Visualization: https://visualgo.net/en/sorting

Bubble Sort: Pseudocode

```
Algorithm BubbleSort(aList, n)
Input:
    aList – List of n elements
    n – Number of elements in List

Output:
    aList – Sorted List in non-decreasing order

Begin:
      for i ← 0 to n – 1 do:
            swapped ← false

            for j ← 0 to n – i – 1 do:
                    if aList[j] > aList[j + 1] then:
                      // Swap aList[j] and aList[j + 1]
                      temp ← aList[j]
                      aList[j] ← aList[j + 1]
                      aList[j + 1] ← temp
                      swapped ← true

                    end if
                  end for

          if swapped = false then:
              break  // No swaps made, list is sorted
          end if
       end for
End
```

Advantages and Disadvantages Bubble Sort

Advantages

- Simple to understand and implement
- No additional memory required

Disadvantages

- Inefficient for large lists

Bubble Sort in Python

```python
def bubble_sort(arr):

    # Get the length of the list
    n = len(arr)

    # Outer loop runs n times where n is the length of the list
    # The i-th pass ensures the last i elements are sorted
    for i in range(n-1):
        # This flag keeps track of whether any swaps occurred in this pass
        swapped = False

        # Inner loop runs from the first element to the n-i-1 element
        # After each pass, the largest unsorted element is moved to its correct position at the end
        for j in range(n - i - 1):

            # Compare adjacent elements
            if arr[j] > arr[j + 1]:
                # Swap if the current element is greater than the next element
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

                # Set swapped to True indicating that a swap occurred
                swapped = True

        # If no swaps occurred during the pass, the listis already sorted
        # We can terminate the algorithm early to improve performance
        if not swapped:
            break

    # Return the sorted list
    return arr
```

Bubble Sort: Complexity Analysis- Worst Case

```
def bubble_sort(arr):
    n = len(arr)                                    # O(1)
    for i in range(n-1):                            # O(n)
        swapped = False                             # O(1)

        for j in range(n - i - 1):                  # O(n-i-1)

            if arr[j] > arr[j + 1]:                 # O(1)
                arr[j], arr[j + 1] = arr[j + 1], arr[j] # O(1)
                swapped = True                      # O(1)
        if not swapped:                             # O(1)
            break

    return arr                                      # O(1)
```

Overall time Complexity

O(1)

O(n)

n* O(1)

$\sum_{i=0}^{n-2} O(n-i-1)$

$\sum_{i=0}^{n-2} (n-i-1)* O(1)$

$\sum_{i=0}^{n-2} (n-i-1)* O(1)$

$\sum_{i=0}^{n-2} (n-i-1)* O(1)$

n* O(1)

O(1)

$$T(n) = O(1) + O(n) + n*O(1) + \sum_{i=0}^{n-2} O(n-i-1) + \sum_{i=0}^{n-2} O(1)*(n-i-1) + \sum_{i=0}^{n-2} O(1)*(n-i-1) + \sum_{i=0}^{n-2} O(1)*(n-i-1) + n* O(1) + O(1)$$

$T(n) = 2\ O(1) + 3\ O(n) + 4\ O(n^2)$ => **O(n^2)**

Bubble Sort: Complexity Analysis- Best Case

Quiz: What is the time complexity of Bubble sort in best case?

Merge Sort

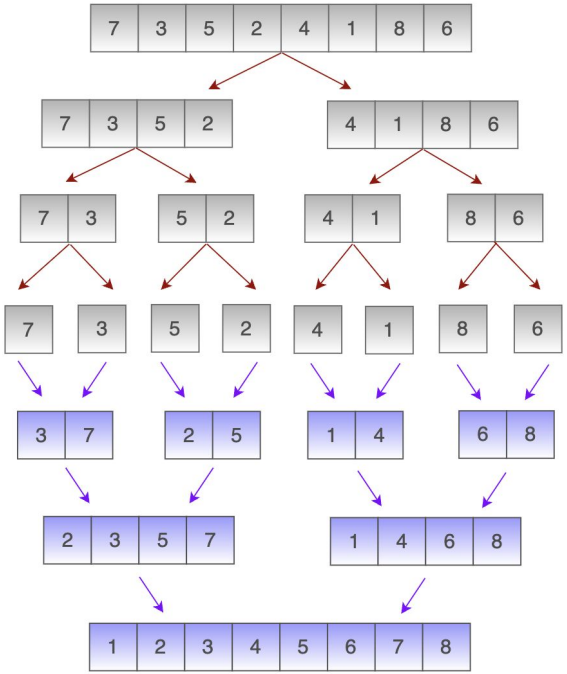
Merge sort

- Merge Sort is a divide-and-conquer algorithm used for sorting an array or list.
- It works by dividing the unsorted list into smaller sublists until each sublist contains a single element (which is inherently sorted), and then merging these sublists to produce a sorted list.

How Merge Sort Works?

- **Divide:** The list is divided into two halves until every sublist contains just one element. Single-element lists are considered sorted by default
- **Conquer**: Recursively sort each half.
- **Combine:** Merge the sorted halves into a single sorted list. The merging step compares elements from the sublists and arranges them in order.

Merge Sort Example



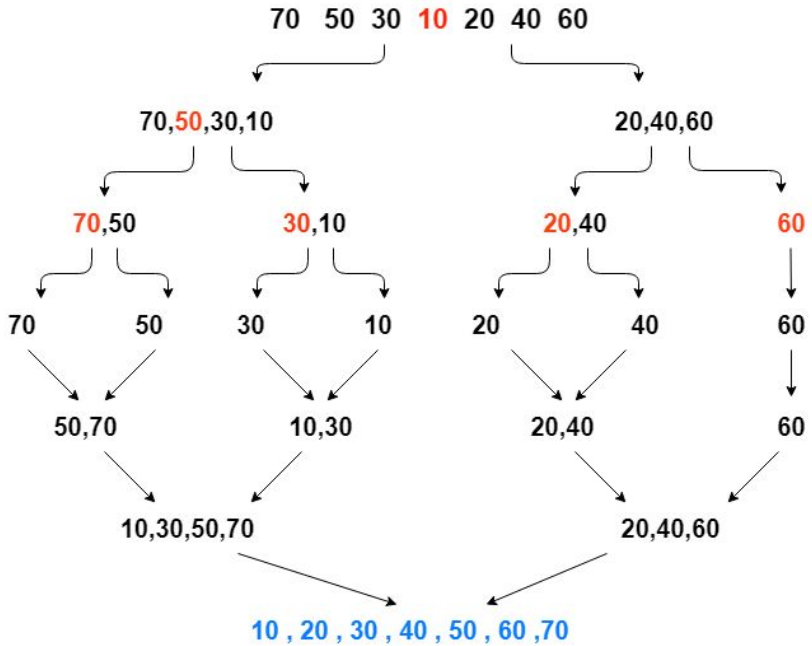
the time complexity of splitting is O(logn)

We have O(n), logn times, meaning that the total complexity is O(nlogn)

The time complexity of merging is O(n). We merge them with a merge function w/ a time complexity of O(n)

Merge Sort Visualization: https://opendsa-server.cs.vt.edu/embed/mergesortAV

Merge Sort Example



Merge Sort Visualization: https://opendsa-server.cs.vt.edu/embed/mergesortAV

Advantages and Disadvantages Merge Sort

Advantages

- Handles large datasets effectively

Disadvantages

- Goes through the whole process even if the list is sorted
- Uses more memory space to store the sublists

Merge Sort: Pseudocode

```
MERGE-SORT(A, p, r)
1  if p ≥ r                          // zero or one element?
2      return
3  q = ⌊(p + r)/2⌋                   // midpoint of A[p : r]
4  MERGE-SORT(A, p, q)               // recursively sort A[p : q]
5  MERGE-SORT(A, q + 1, r)           // recursively sort A[q + 1 : r]
6  // Merge A[p : q] and A[q + 1 : r] into A[p : r].
7  MERGE(A, p, q, r)
```

```
MERGE(A, p, q, r)
1  n_L = q − p + 1                   // length of A[p : q]
2  n_R = r − q                       // length of A[q + 1 : r]
3  let L[0 : n_L − 1] and R[0 : n_R − 1] be new arrays
4  for i = 0 to n_L − 1              // copy A[p : q] into L[0 : n_L − 1]
5      L[i] = A[p + i]
6  for j = 0 to n_R − 1              // copy A[q + 1 : r] into R[0 : n_R − 1]
7      R[j] = A[q + j + 1]
8  i = 0                             // i indexes the smallest remaining element in L
9  j = 0                             // j indexes the smallest remaining element in R
10 k = p                             // k indexes the location in A to fill
11 // As long as each of the arrays L and R contains an unmerged element,
   //     copy the smallest unmerged element back into A[p : r].
12 while i < n_L and j < n_R
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
18     k = k + 1
19 // Having gone through one of L and R entirely, copy the
   //     remainder of the other to the end of A[p : r].
20 while i < n_L
21     A[k] = L[i]
22     i = i + 1
23     k = k + 1
24 while j < n_R
25     A[k] = R[j]
26     j = j + 1
27     k = k + 1
```

Merge Sort in Python

```python
def merge_sort(arr):

    # Base case: If the list contains 0 or 1 element, it is already sorted
    if len(arr) <= 1:
        return arr

    # Find the middle point to divide the list into two halves
    mid = len(arr) // 2

    # Recursively sort the first half of the list
    left_half = merge_sort(arr[:mid])

    # Recursively sort the second half of the list
    right_half = merge_sort(arr[mid:])

    # Merge the two sorted halves and return the sorted list
    return merge(left_half, right_half)
```

```python
def merge(left, right):

    # List to store the merged sorted elements
    sorted_list = []
    # Indices to keep track of positions in left and right lists
    i = j = 0

    # Compare elements of both lists and merge them in sorted order
    while i < len(left) and j < len(right):
        if left[i] <= right[j]:
            # If the current element in the left list is smaller or equal,
            # append it to the sorted list and move the pointer in the left list
            sorted_list.append(left[i])
            i += 1
        else:
            # If the current element in the right list is smaller,
            # append it to the sorted list and move the pointer in the right list
            sorted_list.append(right[j])
            j += 1

    # If there are remaining elements in the left list, append them to the sorted list
    # This happens when all elements in the right list have been added
    sorted_list.extend(left[i:])

    # If there are remaining elements in the right list, append them to the sorted list
    # This happens when all elements in the left list have been added
    sorted_list.extend(right[j:])

    # Return the merged and sorted list
    return sorted_list
```

Merge Sort: Complexity Analysis (Best/Average/Worst)

```
def merge_sort(arr):

    if len(arr) <= 1:  # O(1) Checking if the list has 0 or 1 element takes constant time
        return arr     # O(1) => This is a constant-time operation. It does not depend on the size of the list.

    mid = len(arr) // 2 #O(1) Calculating the midpoint of the list is done in constant time.

    left_half = merge_sort(arr[:mid]) #   merge_sort(arr[:mid]) itself has a complexity of  T(n/2), which is part of the total Merge Sort process.
                                      #   Total Complexity: The entire Merge Sort algorithm, including all recursive calls and merging steps,
                                      #   has an overall time complexity of O(nlogn)


    right_half = merge_sort(arr[mid:]) #   merge_sort(arr[mid:]) itself has a complexity of T(n/2), which is part of the total Merge Sort process.
                                       #   Total Complexity: The entire Merge Sort algorithm, including all recursive calls and merging steps,
                                       #   has an overall time complexity of O(nlogn)


    return merge(left_half, right_half) #O(n) for merging two sorted halves of size  n/2
```

Merge Sort: Complexity Analysis (Best/Average/Worst)

```python
def merge(left, right):

    sorted_list = []   # O(1) Initializing an empty

    i = j = 0          # O(1) Initializing two indices (i and j) to 0 takes constant time


    while i < len(left) and j < len(right): # O(n)  where  n is the total number of elements in both sublists (left and right)
        if left[i] <= right[j]: # O(1)

            sorted_list.append(left[i]) # O(1) amortized
            i += 1 # O(1)
        else:

            sorted_list.append(right[j]) # O(1) amortized
            j += 1 # O(1)


    sorted_list.extend(left[i:])  # O(k), where k  is the number of remaining elements in left list
                                  # Extending the sorted_list with the remaining elements takes linear time wrt the number of remaining elements
                                  # In the worst case, this operation adds  O(n) elements, which still contributes to  O(n) time complexity


    sorted_list.extend(right[j:])  # O(k), where k  is the number of remaining elements in right list
                                   # Extending the sorted_list with the remaining elements takes linear time wrt the number of remaining elements
                                   # In the worst case, this operation adds  O(n) elements, which still contributes to  O(n) time complexity

    return sorted_list # O(1)  Returning the sorted list takes constant time
```

Merge Sort: Complexity Analysis (Best/Average/Worst)

- **Divide Phase**
  - **Recursive Splitting**: The list is divided into two halves recursively. At each level of recursion, the list is divided into smaller sublists until each sublist contains a single element.
  - **Depth of Recursion Tree**: For a list of size n, the list is split into halves, resulting in a binary tree of recursion depth log n. This is because the list size reduces from n to n/2, then n/4, and so forth, until it reaches 1.
  - **Time Complexity**:
    - The splitting itself does not involve significant computational effort beyond creating new sublist references. Thus, splitting is handled at O(logn) levels, but it is more about the structure of the recursion rather than computational cost.

Merge Sort: Complexity Analysis (Best/Average/Worst)

- **Merging Process**: Once the sublists are reduced to size 1, they are merged back together in a sorted manner. The merging operation combines two sorted sublists into a single sorted list. This process is done for each level of recursion.
- **Merging Complexity**:
  - **Number of Merges**: At each level of the recursion tree, every element of the list is involved in merging. Thus, the total merging cost at each level is O(n) for combining the sublists.
  - **Number of Levels**: There are log n levels in the recursion tree.
- **Total Merging Time Complexity**:
  - For each level of recursion, merging takes O(n). With log n levels, the total time complexity for merging across all levels is O(nlogn).

Quick Sort

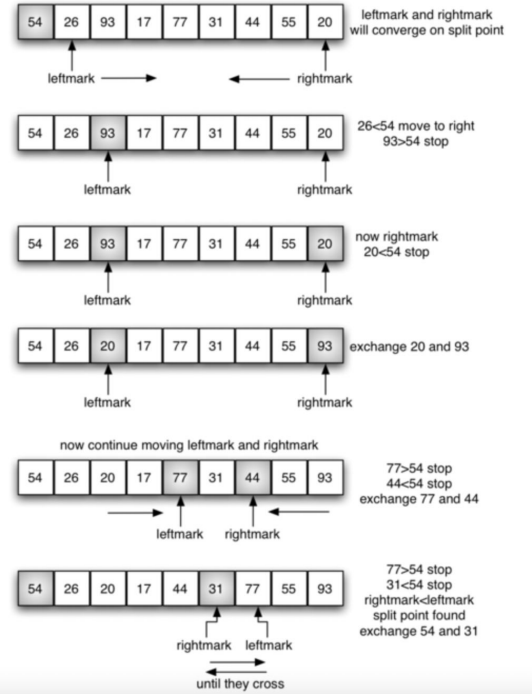
QuickSort

- **QuickSort** is a highly efficient, comparison-based **divide-and-conquer** sorting algorithm. It works by selecting a **pivot** element from the list and partitioning the other elements into two sub-lists based on whether they are less than or greater than the pivot. The process is repeated recursively on the sub-lists, eventually sorting the entire list.

How QuickSort Works?

- **Pivot Selection**
  - Quick Sort selects a pivot element from the list
  - There are various ways to choose the pivot (e.g., first element, last element, random element, or median)
- **Partitioning**: The list is partitioned such that:
  - All elements less than the pivot are placed to the left.
  - All elements greater than the pivot are placed to the right.
  - The pivot element is in its correct sorted position.
- **Recursive Sorting**: After partitioning, Quick Sort is applied recursively to the left and right sub-lists.

QuickSort Example



QuickSort Visualization: https://www.hackerearth.com/practice/algorithms/sorting/quick-sort/visualize/

Advantages and Disadvantages QuickSort

Advantages

- Handles large datasets effectively
- It doesn’t require additional storage like Merge Sort

Disadvantages

- It isn’t as effective when the pivot element is the largest or smallest, or when all of the components have the same size.

QuickSort: Pseudocode

```
QUICKSORT(A, p, r)
1  if p < r
2      // Partition the subarray around the pivot, which ends up in A[q].
3      q = PARTITION(A, p, r)
4      QUICKSORT(A, p, q − 1)  // recursively sort the low side
5      QUICKSORT(A, q + 1, r)  // recursively sort the high side
```

```
PARTITION(A, p, r)
1  x = A[r]                          // the pivot
2  i = p − 1                         // highest index into the low side
3  for j = p to r − 1                // process each element other than the pivot
4      if A[j] ≤ x                   // does this element belong on the low side?
5          i = i + 1                 // index of a new slot in the low side
6          exchange A[i] with A[j]   // put this element there
7  exchange A[i + 1] with A[r]       // pivot goes just to the right of the low side
8  return i + 1                      // new index of the pivot
```

QuickSort in Python

```python
def quickSort(array, low, high):
    """
    Quick Sort algorithm that sorts the elements of the list in ascending order.

    Parameters:
    array (list): The list of elements to be sorted.
    low (int): The starting index of the sublist to be sorted.
    high (int): The ending index of the sublist to be sorted.
    """

    if low < high:
        # Find the pivot element, such that elements smaller than pivot are on the left,
        # and elements greater than pivot are on the right.
        pi = partition(array, low, high)

        # Recursively apply Quick Sort to the left of the pivot
        quickSort(array, low, pi - 1)

        # Recursively apply Quick Sort to the right of the pivot
        quickSort(array, pi + 1, high)
```

```python
def partition(array, low, high):
    """
    This function takes the last element as pivot, places the pivot at its correct
    sorted position in the list, and places all elements smaller than pivot to the left
    and all greater elements to the right.
    Parameters:
    array (list): The list of elements to partition.
    low (int): The starting index of the sublist.
    high (int): The ending index of the sublist (pivot).

    Returns:
    int: The partition index where the pivot element is placed.
    """
    # Choose the rightmost element as pivot
    pivot = array[high]
    # Pointer for greater element
    i = low - 1
    # Traverse through all elements in the range [low, high)
    # Compare each element with the pivot
    for j in range(low, high):
        if array[j] <= pivot:
            # If an element smaller than the pivot is found,
            # increment the pointer for greater element (i) and
            # swap the current element with the element at pointer i.
            i = i + 1
            array[i], array[j] = array[j], array[i]
    # Swap the pivot element with the element at i + 1
    array[i + 1], array[high] = array[high], array[i + 1]
    # Return the partition index (where the pivot is placed)
    return i + 1
```

QuickSort: Complexity Analysis- Best Case/Average

Best Case is that array is balanced perfectly on either side as less than pivot to the left and greater than pivot to the right

Total Time Complexity O(n log n)

```python
def quickSort(array, low, high):

    if low < high: # O(1) - comparison operation

        pi = partition(array, low, high) # O(n) - partition call

        # Recursively apply Quick Sort to the left of the pivot
        quickSort(array, low, pi - 1) # Recursively called on a subarray of size n/2 (


        # Recursively apply Quick Sort to the right of the pivot
        quickSort(array, pi + 1, high) # Recursively called on a subarray of size n/2
```

```python
def partition(array, low, high):

    pivot = array[high]  # O(1)


    i = low - 1  # O(1)


    for j in range(low, high): # O(n) - this loop runs 'high - low' times
        if array[j] <= pivot:  # O(1) - comparison with the pivot

            array[i], array[j] = array[j], array[i]  # O(1) - swap operation


    array[i + 1], array[high] = array[high], array[i + 1] # O(1)


    return i + 1 # O(1)
```

QuickSort: Complexity Analysis- Worst Case

Total Time Complexity O(n^2)

this is b/c the recursion depth of the tree is O(n) and there are O(n) levels, therefore, the T(n) = O(n)*O(n) = O(n^2)

```
def quickSort(array, low, high):

    if low < high: # O(1) - comparison operation

        pi = partition(array, low, high) # O(n) - partition call

        # Recursively apply Quick Sort to the left of the pivot
        quickSort(array, low, pi - 1) # Recursively called on the left subarray (worst case)/ unbalanced partition

        # Recursively apply Quick Sort to the right of the pivot
        quickSort(array, pi + 1, high) # Recursively called on the right subarray (worst case)/ unbalanced partition
```

Worst Case: The array is unbalanced, with all the elements to the left being smaller OR all the elements to the right being greater

```
def partition(array, low, high):

    pivot = array[high]  # O(1)

    i = low - 1  # O(1)

    for j in range(low, high): # O(n) - this loop runs 'high - low' times
        if array[j] <= pivot:  # O(1) - comparison with the pivot

            array[i], array[j] = array[j], array[i]  # O(1) - swap operation

    array[i + 1], array[high] = array[high], array[i + 1] # O(1)

    return i + 1 # O(1)
```

Broader issues of Sorting Algorithms

Broader issues of Sorting Algorithms

- Can **all** data be loaded into our computer **memory** at one time?
- Do we want to sort data on our **single** computer?
- Do we have very **large** amount of data?
- What is the s**tability** of the sorting algorithm we intend to use?
- What is the **complexity** of the algorithm under consideration?

Internal – External Sorting

- The term **internal sort** is used when, at the start of sorting, all the data is already available in the computer memory or all the data can be loaded at once

- The term **external sort** is used when the whole array or vector cannot fit into the computer's memory. The data will have to be loaded from a storage device of the computer or the network

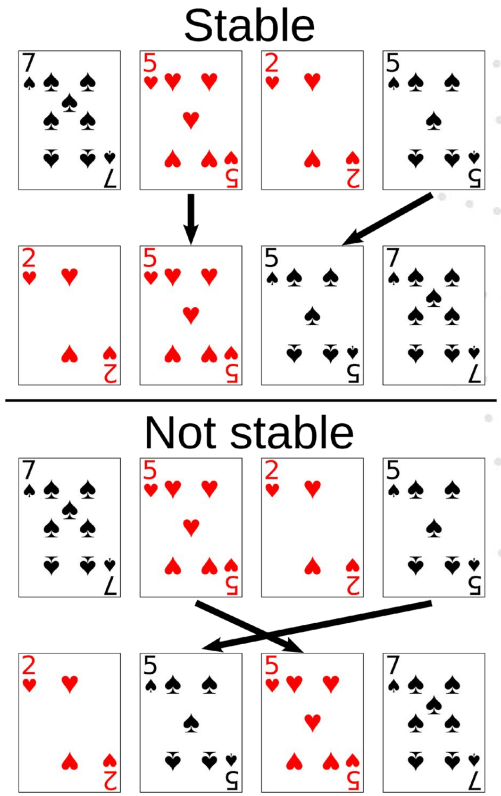
Local – Distributed Sorting

- Our discussion will be focused on Internal Sort algorithms in a non-distributed environment
- We need to remember that there are many sorting algorithms that can be used, if data has to be reloaded or streamed from an external source
- As an example, Hadoop can handle very large amounts of data in a distributed system. There are sorting algorithms to be used for such distributed systems

Sorting Stability

- A sorting algorithm is stable if elements with the same key appear in the output list in the same order as they do in the input list
- Let's have a data set S={(A,3),(B,5),(C,2),(D,5),(E,4)}
- If a stable sorting algorithm sorts S on the second value in each pair using the ≤ relation then the result is guaranteed to be {(C,2),(A,3),(E,4),(B,5),(D,5)}.
- However, if an algorithm is not stable, then (D,5) may come before (B,5) in the sorted output

Sorting Stability



Programming activities

Sort the People

- You are given a list of strings names, and a list of heights that consists of distinct positive integers. Both list are of length n. For each index i, names[i] and heights[i] denote the name and height of the ith person. Return names sorted in descending order by the people's heights.
- Example 1:
  - Input: names = ["Mary","John","Emma"], heights = [180,165,170]
  - Output: ["Mary","Emma","John"] Explanation: Mary is the tallest, followed by Emma and John.
- Example 2:
  - Input: names = ["Alice","Bob","Bob"], heights = [155,185,150]
  - Output: ["Bob","Alice","Bob"] Explanation: The first Bob is the tallest, followed by Alice and the second Bob.

Programming activities

Sort List by Increasing Frequency

- Given a list of integers nums, sort the list in increasing order based on the frequency of the values. If multiple values have the same frequency, sort them in decreasing order. Return the sorted list.
- Example 1:
  - Input: nums = [1,1,2,2,2,3]
  - Output: [3,1,1,2,2,2] Explanation: '3' has a frequency of 1, '1' has a frequency of 2, and '2' has a frequency of 3
- Example 2:
  - Input: nums = [2,3,1,3,2]
  - Output: [1,3,3,2,2] Explanation: '2' and '3' both have a frequency of 2, so they are sorted in decreasing order

Programming activities

Solution format:

```
Class Solution():
    def YourFunction(self,input):
        return output
```