

# Selection Sort

09114319: Data Structures and Algorithms

R. Promkam, Dr. rer. nat.

Department of Mathematics and Computer Science, RMUTT

- ✎ Memory and Data Storage
  - ✎ How memory works
  - ✎ Arrays and linked lists
  - ✎ Comparisons of data structures
- ✎ Introduction to Selection Sort
  - ✎ Problem setup: Sorting a list
  - ✎ Algorithm description
- ✎ Performance Analysis
  - ✎ Big  $O$  analysis of Selection Sort
- ✎ Code Implementations
- ✎ Applications of Selection Sort
- ✎ Recap

# How Memory Works

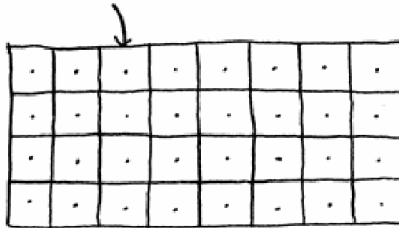
📎 Computer memory is like a giant set of drawers.



# How Memory Works

- Each drawer has a unique address, used to store items.

ADDRESS: fe0ffeeb



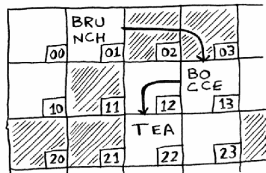
# How Memory Works

- ✎ Use an array: all items are stored contiguously.




To store multiple items:






- ✎ Use a linked list: items can be scattered but linked by addresses.



## Arrays



-  Items stored next to each other in memory.
-  Fast random access:  $O(1)$  for reads.
-  Slow inserts:  $O(n)$  if shifting or resizing is needed.

## Linked Lists



-  Items scattered in memory.
-  Fast inserts and deletes:  $O(1)$  if the pointer is known.
-  Slow reads: Sequential access required ( $O(n)$ ).

# When to Use Arrays vs. Linked Lists?

 Use arrays when:


-  Fast random access is needed.
-  Memory is contiguous and resizing is minimal.


 Use linked lists when:


-  Frequent inserts and deletes are required.
-  Memory does not need to be contiguous.

# Common Operations


## Arrays:


 Reading:  $O(1)$


 Inserting:  $O(n)$

 Deleting:  $O(n)$

## Linked Lists:




 Reading:  $O(n)$

 Inserting:  $O(1)$

 Deleting:  $O(1)$






# Memory and Data Storage Recap

-  Arrays are great for fast reads and situations requiring random access.
-  Linked lists are ideal for frequent inserts and deletes.
-  Both structures are foundational for implementing algorithms like sorting.


# Problem Setup: Sorting a List

- ✎ Suppose you have a list of music artists with play counts.
- ✎ Goal: Sort the list from most played to least played.
- ✎ How can you achieve this?
  - ✎ Start by finding the most played artist.
  - ✎ Add that artist to a new list.
  - ✎ Repeat for the remaining artists.
- ✎ This approach is the basis of the **Selection Sort** algorithm.

# What is Selection Sort?


-  Selection Sort is a simple algorithm for sorting a list.
-  Process:
  1. Find the smallest (or largest) element in the list.
  2. Add it to a new sorted list.
  3. Repeat for all remaining elements.
-  Result: A fully sorted list.

# Selection Sort in Action





 Example: Sort the list [5, 3, 6, 2, 10].

 Steps:

1. Find the smallest number (2).
2. Add 2 to the new list.
3. Remove 2 from the original list.
4. Repeat until the original list is empty.

 Output: [2, 3, 5, 6, 10]


# Key Characteristics of Selection Sort


-  **Deterministic Algorithm:** Same input produces the same output.
-  **Comparison-Based Sorting:** Involves pairwise comparisons of elements.
-  **Not Adaptive:** Does not optimize for partially sorted lists.
-  **In-Place Sorting:** Requires no extra space beyond the original list (when implemented in-place).


# Use Cases for Selection Sort


- ✎ Simple scenarios with small datasets.
- ✎ Educational purposes: Demonstrates sorting mechanics.
- ✎ Limited memory environments: Can be implemented in-place.
- ✎ Not ideal for large datasets due to its  $O(n^2)$  time complexity.

# Performance of Selection Sort

 Selection Sort has a time complexity of  $O(n^2)$ .


 Why  $O(n^2)$ ?

 To find the smallest element, you must check all  $n$  elements.

 Repeat the process for the remaining  $n - 1$ ,  $n - 2$ , ..., 1 elements.

 Total comparisons:

$$n + (n - 1) + (n - 2) + \dots + 1 = \frac{n(n + 1)}{2}.$$

 This simplifies to  $O(n^2)$  as constants are ignored in Big O notation.




# Space Complexity of Selection Sort

- ✎ Space complexity is  $O(1)$ .
- ✎ Selection Sort operates in-place:
  - ✎ Does not require additional memory for a new list.
  - ✎ Sorting happens within the original list.
- ✎ Efficient in terms of memory usage.






# Selection Sort vs. Other Algorithms


## Advantages

-  Simple to implement.
-  Works well on small datasets.
-  Requires minimal memory.

## Disadvantages


-  Poor performance on large datasets due to  $O(n^2)$ .
-  Not adaptive: No advantage for partially sorted data.
-  Slow compared to more advanced algorithms like Quicksort or Merge Sort.

# Analyzing $O(n^2)$ Complexity

 Example: Sorting a list of size  $n = 10$ :


 Comparisons:  $10 + 9 + 8 + \dots + 1 = 55$ .

 Doubling the input size to  $n = 20$ :

 Comparisons:  $20 + 19 + 18 + \dots + 1 = 210$ .

 Observations:



 Number of comparisons grows quadratically.

 Doubling the input size increases work by 4 times ( $O(n^2)$ ).

# When is Selection Sort Practical?

- ✎ Small datasets where  $O(n^2)$  is acceptable.
- ✎ When memory usage is a critical concern ( $O(1)$  space complexity).
- ✎ Educational purposes: Explaining sorting mechanics and  $O(n^2)$  complexity.
- ✎ When simplicity of implementation is preferred over performance.



## Step 1: Finding the Smallest Element

-  Write a function to find the smallest element in an array.
-  This helper function will be used repeatedly in Selection Sort.

```
1 def findSmallest(arr):  
2     smallest = arr[0] # Stores the smallest value  
3     smallest_index = 0 # Stores the index of the  
        ↪ smallest value  
4     for i in range(1, len(arr)):  
5         if arr[i] < smallest:  
6             smallest = arr[i]  
7             smallest_index = i  
8     return smallest_index
```

# Code Implementation: Selection Sort Function


## Step 2: Sorting the Array

-  Use the 'findSmallest' function to implement Selection Sort.
-  Iteratively find the smallest element and build a new sorted array.

```
1 def selectionSort(arr):  
2     newArr = []  
3     for i in range(len(arr)):  
4         smallest = findSmallest(arr) # Find the  
           ↪ smallest element  
5         newArr.append(arr.pop(smallest)) # Add it  
           ↪ to the new array  
6     return newArr
```



# Example: Running Selection Sort

## Input Example:

 Sort the array [5, 3, 6, 2, 10] using Selection Sort.



```
1 print(selectionSort([5, 3, 6, 2, 10]))  
2 # Output: [2, 3, 5, 6, 10]
```

## Explanation:

-  Finds the smallest element (2) and adds it to the new array.
-  Repeats the process for all elements until the array is sorted.

# In-Place Implementation (Optional)

## Sorting Without a New Array

-  Modify Selection Sort to sort the original array in place.
-  Reduces space complexity to  $O(1)$ .

```
1 def selectionSortInPlace(arr):
2     for i in range(len(arr)):
3         smallest_index = i
4         for j in range(i + 1, len(arr)):
5             if arr[j] < arr[smallest_index]:
6                 smallest_index = j
7         arr[i], arr[smallest_index] =
            ↪ arr[smallest_index], arr[i]
8     return arr
```



# Discussion Points

- ✎ What are the advantages of the in-place implementation?
- ✎ How does the `findSmallest` function work in tandem with the main sorting logic?
- ✎ Compare the in-place version with the version that creates a new array:
  - ✎ Space complexity
  - ✎ Code clarity





# Advantages of the In-Place Implementation

## **Memory Efficiency:**

-  Reduces space complexity from  $O(n)$  to  $O(1)$ .
-  No need to create and manage an additional array.

## **Practical Use:**

-  Preferred for large datasets where memory usage is a concern.
-  Common in real-world applications where overhead must be minimized.

# How the findSmallest Function Works

- Identifies the smallest element in the unsorted portion of the array.

- Non-In-Place Version:**

- The smallest element is removed from the original array.
- It is appended to a new array, progressively building the sorted list.

- In-Place Version:**



- Logic is integrated into the nested loop to find the smallest element.
- The smallest element is swapped with the current index, avoiding the creation of a new array.

# In-Place vs. Non-In-Place Implementations



Feature	Non-In-Place	In-Place
Space Complexity	$O(n)$ : Requires a new array	$O(1)$ : No additional memory
Time Complexity	$O(n^2)$	$O(n^2)$
Code Simplicity	Easier to understand	Slightly more complex
Use Case	Learning or small datasets	Large datasets, memory-constrained environments

# Summary of the Comparison


## **Non-In-Place Version:**

-  Simple and intuitive for educational purposes.
-  Suitable for small-scale applications.


## **In-Place Version:**

-  More efficient for large datasets.
-  Requires a deeper understanding of memory management.


## **Small Datasets:**

-  Selection Sort is simple and efficient for small input sizes.


## **Memory-Constrained Environments:**

-  Minimal space requirements make it practical for limited-memory systems.

## **Educational Purposes:**

-  Ideal for teaching sorting mechanics and complexity analysis.



## **Sorting Data by Priority:**

-  Useful when processing data with explicit priority order (e.g., finding top scores).




# Limitations of Selection Sort

- ✎ Poor performance on large datasets ( $O(n^2)$ ).
- ✎ Not adaptive: Does not leverage partial sorting.
- ✎ Slower compared to advanced algorithms like Quicksort ( $O(n \log n)$ ).



## **Next Steps in Sorting:**

-  Learn faster sorting algorithms, such as Quicksort and Merge Sort.
-  Understand when to use different sorting approaches.



## **Practical Use Cases:**

-  Sorting names in a phone book.
-  Organizing data by timestamps (e.g., emails or events).
-  Ranking items (e.g., playlists, scores, priorities).





## **Broader Applications:**

-  Sorting is a fundamental operation in numerous algorithms and systems.
-  Real-world optimizations often combine sorting with other techniques (e.g., search).

## **Memory and Data Structures:**

-  Arrays: Contiguous memory, fast random access ( $O(1)$ ).
-  Linked Lists: Scattered memory, efficient inserts/deletes ( $O(1)$ ).

## **Selection Sort Algorithm:**

-  Iteratively finds the smallest (or largest) element.
-  Builds a sorted list step by step.
-  Time complexity:  $O(n^2)$ .
-  Space complexity:  $O(1)$  (in-place implementation).