SCC.121: Fundamentals of Computer Science
Sorting, Trees and Graphs

Fabien Dufoulon

Plan for Weeks 16-20

Sorting, Trees and Graphs

- Week 16: Sorting Algorithms
- Week 17: Algorithms on Trees
- Week 18: Algorithms on Graphs
- Weeks 19-20: Quick Tour of Combinatorial Optimization

Quizz on Week 20

Lecturer for this Part

Fabien Dufoulon

Research Field: **Theory of Distributed Computing**

- Distributed **Graph Algorithms**
- **Fault-Tolerant** Distributed **Algorithms**

Today’s Lecture

Aim:

- Introduce the sorting problem
- Describe two basic sorting algorithms
  - Insertion sort
  - Selection sort
- Analyse their time complexity

Unstructured Data and Sorting

- Sorting is so common you don't know you even use it!
- Because most data in real-life comes in a **somewhat unstructured, hard-to-use form**:
  - Course marks,
  - Salaries and population counts,
  - The CPU/Memory usage of your programs,
  - News or social media,

Applications of Sorting

In short, fundamental subroutine for Data Analysis:

- **Ranking answers to a web query** by **relevance** value, or "**freshness**", or some **mix of both**,…
  - **Find X top (least)** scoring students (e.g., salaries, populated regions, energy-intensive industries),
  - Find cheap airplane tickets, also with few stops.

- **Searching** in a phone directory, dictionary, list of friends, …
  - **Sort** data first **into structured form,**
  - **Then find** an **arbitrary item** fast using **binary search** on structured data.

The Sorting Problem

- Input: **list/array of data** (integers or names) in an **arbitrary order**
- Output: **list/array in a sorted order** (alphabetical or numerical ordering)

- The data may consist of **duplicates**, and we do not have any information in advance about the **distribution of the data** (e.g., its highest and lowest values, or the median value).

The Sorting Problem

Numerical ordering, lowest to highest:

Input: → **Output:**

5	4	2	5	6	1	8	7	→	1	2	4	5	5	6	7	8
10	54	4	15	6	80	8	1	→	1	4	6	8	10	15	54	80
8	7	6	5	1	2	3	4	→	1	2	3	4	5	6	7	8

And for other orderings:

a	c	f	i	k	a	b	s	→	a	a	b	c	f	i	k	s
5	4	2	5	6	1	8	7	→	8	7	6	5	5	4	2	1

Sorting Algorithms: Context Summary

- **Sorting data** is fundamental to a wide range of computational problems
  - **Ranking** data, or **finding top X** items,
  - **Searching** (binary search) and other operations are **faster on sorted datasets.**
  - If many searches are done on the same dataset, sorting the dataset first can be an efficient **preprocessing** step.

- <u>There are many different algorithms for sorting:</u>
  - Some have interesting properties relative to particular data distributions
  - Some are efficient in terms of memory or CPU usage

A Simple Sorting Algorithm: Selection Sort

Input

5	4	2	5	6	1	8	7

Secondary sorted array

<u>How do we sort?</u>

5	4	2	5	6		8	7

Find minimum value

Remove minimum value from input array

Add minimum value to sorted array

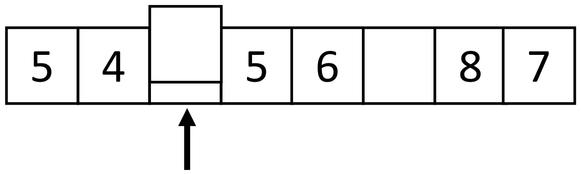
A Simple Sorting Algorithm: Selection Sort

Input

5	4	2	5	6	1	8	7

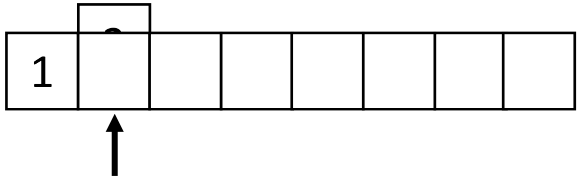
Secondary sorted array

<u>How do we sort?</u>



Find minimum value

Remove minimum value from input array



Add minimum value to sorted array

A Simple Sorting Algorithm: Selection Sort

Input

5	4	2	5	6	1	8	7

Secondary sorted array

How do we sort?

5			5	6		8	7

Find minimum value

Remove minimum value from input array

1	2						

Add minimum value to sorted array

A Simple Sorting Algorithm: Selection Sort

Input

5	4	2	5	6	1	8	7

Secondary sorted array

<u>How do we sort?</u>

Remove minimum value from input array

Find minimum value

1	2	4	5	5	6	7	

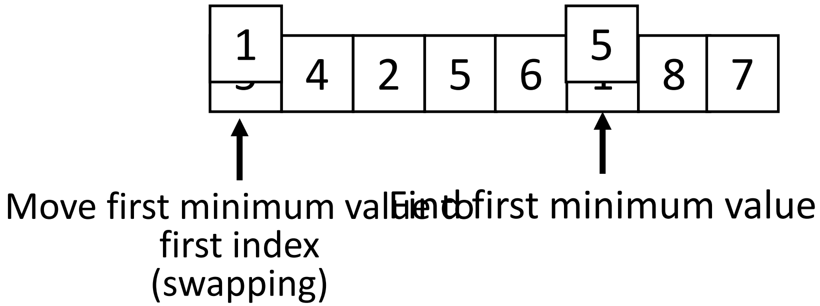
Add minimum value to sorted array

A Simple Sorting Algorithm: Selection Sort

In-place algorithm:

Uses little additional space
For example, $O(1)$ space

How do we sort in-place?

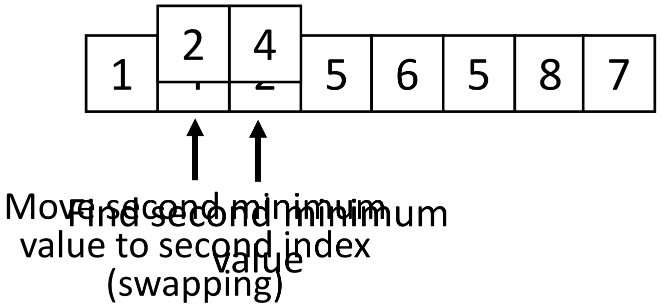


A Simple Sorting Algorithm: Selection Sort

In-place algorithm:

Uses little additional space
For example, $O(1)$ space

How do we sort in-place?



A Simple Sorting Algorithm: Selection Sort

In-place algorithm:

Uses little additional space
For example, $O(1)$ space

How do we sort in-place?

1	2	4	5	6	5	8	7

Find third minimum value

Third minimum value is
already at third index
(no swapping)

A Simple Sorting Algorithm: Selection Sort

Lancaster University

In-place algorithm:

Uses little additional space
For example, $O(1)$ space

How do we sort in-place?

1	2	4	5	5	6	7	8

Find eighth minimum value

Eighth minimum value is
already at eighth index
(no swapping)

Selection Sort: Code

```
class SelectionSort {
    static void selectionSort(int[] array){
        for (int i = 0; i < array.length; i++) {
            int iMin = i;
            for (int j = i + 1; j < array.length; j++) {  //find index of minimum
                if (array[j] < array[iMin]) iMin = j;
            }
            if (iMin != i) {                    //swap array[i] and array[iMin]
                int tmp= array[i];
                array[i] = array[iMin];
                array[iMin] = tmp;
            }
        }
    } }
```

Selection Sort: Correctness Analysis

```
class SelectionSort {
    static void selectionSort(int[] array){
        for (int i = 0; i < array.length; i++) {
            int iMin = i;
            for (int j = i + 1; j < array.length; j++) {  //find index of minimum
                if (array[j] < array[iMin]) iMin = j;
            }
            if (iMin != i) {                      //swap array[i] and array[iMin]
                int tmp= array[i];
                array[i] = array[iMin];
                array[iMin] = tmp;
            }
        }
    } }
```

Invariant (for $0 \leq i < length$):

At the end of phase i, the first $i + 1$ entries of the array are sorted and contain the minimum $i + 1$ values

Correctness:

- Prove invariant
- Invariant for $i = length - 1$ implies that the algorithm is correct

Selection Sort: Correctness Analysis

<u>Invariant (for $0 \leq i < length$):</u>

At the end of phase i, the first $i + 1$ entries of the array are sorted and contain the minimum $i + 1$ values

<u>Proof (by induction):</u>

- <u>Base step (for $i = 0$, or the first phase):</u>
  - In the first phase, we find the minimum value (by linear search) and if it is not the first index in the array, we move it to the first index.

- <u>Induction step (for $0 \leq i < length - 1$):</u>
  - By the induction hypothesis (holding for phase i), the first $i + 1$ entries in the (entire) array are sorted and contain the minimum $i + 1$ values.
  - During phase $i + 1$, we compute the minimum value in the sub-array from index $i + 1$ to $length - 1$. By the induction hypothesis, this is the $(i + 2)$th minimum value.
  - We move that value to the $(i + 1)$th index in the array (if it is not already there).
  - As a result, the first minimum $i + 2$ entries of the array are sorted and contain the minimum i +2 values.

Selection Sort: Worst-case Time Complexity

```
class SelectionSort {
    static void selectionSort(int[] array){
        for (int i = 0; i < array.length; i++) {
            int iMin = i;
            for (int j = i + 1; j < array.length; j++) {  //find index of minimum
                if (array[j] < array[iMin]) iMin = j;
            }
            if (iMin != i) {                       //swap array[i] and array[iMin]
                int tmp= array[i];
                array[i] = array[iMin];
                array[iMin] = tmp;
            }
        }
    } }
```

<u>Operation counting:</u>

In each phase i (for $0 \leq i < length$):

- At most length $- \ i$ comparisons
- At most $O(length - i)$ elementary operations

$$T_1(n) = O\left(\sum_{i=0}^{length-1} (length - i)\right) = O\left(\sum_{i=1}^{length} i\right) = O(n^2)$$

Selection Sort: Analysis Summary

	Selection Sort
Best case	$O(n^2)$
Average case	$O(n^2)$
Worst case	$O(n^2)$
In-place	Yes

More Efficient Sorting: Insertion Sort

Input

5	4	2	5	6	1	8	7

Secondary sorted array

How do we sort?

	4	2	5	6	1	8	7

Take first value from input array

Insert first value into sorted array

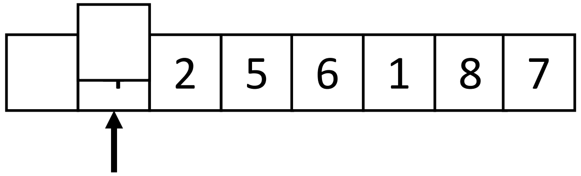
More Efficient Sorting: Insertion Sort

Input

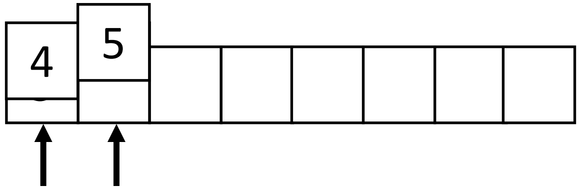
5	4	2	5	6	1	8	7

Secondary sorted array

<u>How do we sort?</u>



Take second value
from input array



First value greater than
(swapping)

Insert second value into
second sorted array

More Efficient Sorting: Insertion Sort

Input

5	4	2	5	6	1	8	7

Secondary sorted array

How do we sort?

		2	5	6	1	8	7

Take third value
from input array

2	4	5					

First value greater than
second (swapped)

Second value greater than
third (swapped)

More Efficient Sorting: Insertion Sort

Input

5	4	2	5	6	1	8	7

Secondary sorted array

How do we sort?

			5	6	1	8	7

Take fourth value
from input array

2	4	5	5				

Third value equal or smaller
to fourth (no swapping)

Insert fourth value into
sorted array

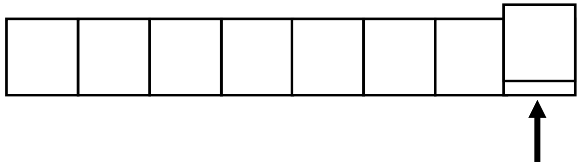
More Efficient Sorting: Insertion Sort

Input

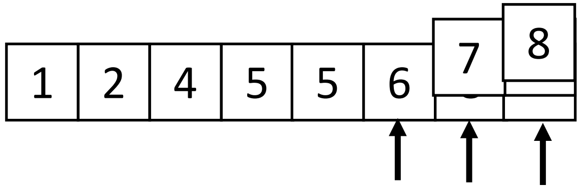
5	4	2	5	6	1	8	7

Secondary sorted array

How do we sort?



Take eighth value from input array



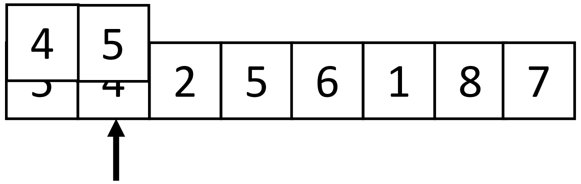
Insert eighth value into sorted array

Insertion Sort In-place

5	4	2	5	6	1	8	7

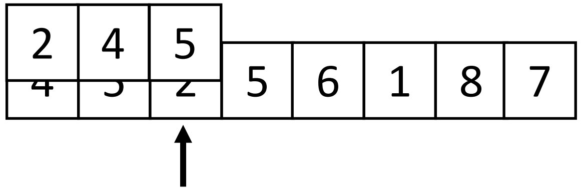
Insert first value into
sorted sub-array

Insertion Sort In-place



Insert second value into
sorted sub-array

Insertion Sort In-place



Insert third value into
sorted sub-array

Insertion Sort In-place

2	4	5	5	6	1	8	7

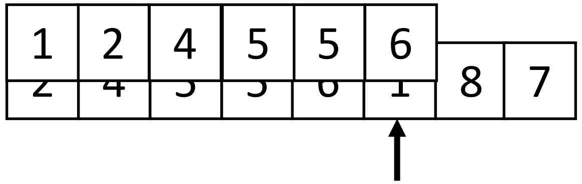
Insert fourth value into
sorted sub-array

Insertion Sort In-place

2	4	5	5	6	1	8	7

Insert fifth value into
sorted sub-array

Insertion Sort In-place



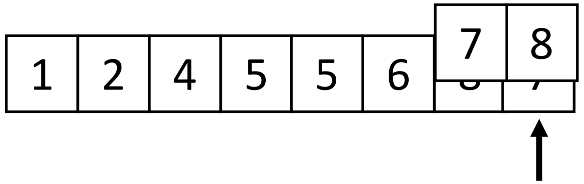
Insert sixth value into
sorted sub-array

Insertion Sort In-place

1	2	4	5	5	6	8	7

Insert seventh value into
sorted sub-array

Insertion Sort In-place



Insert eighth value into
sorted sub-array

Insertion Sort: Code

```
class InsertionSort {
    static void insertionSort(int[] array){
        for (int i = 1; i < array.length; i++) {
            int j = i;
            while (j > 0 && array[j-1] > array[j]) { //swap array[j-1] & array[j]
                int tmp = array[j-1];
                array[j-1] = array[j];
                array[j] = tmp;
                j--;
            }
        }
    }
}
```

Insertion Sort: Correctness Analysis

```
class InsertionSort {
    static void insertionSort(int[] array){
        for (int i = 1; i < array.length; i++) {
            int j = i;
            while (j > 0 && array[j-1] > array[j]) { //swap array[j-1] & array[j]
                int tmp = array[j-1];
                array[j-1] = array[j];
                array[j] = tmp;
                j--;
            }
        }
    }}
```

Invariant (for $1 \le i < length$):

At the end of phase i, the first $i+1$ entries of the array are sorted (i.e., in increasing order)

Correctness:

- Prove invariant
- Invariant for $i = length - 1$ implies that the algorithm is correct

Insertion Sort: Worst-case Time Complexity

```
class InsertionSort {
    static void insertionSort(int[] array){
        for (int i = 1; i < array.length; i++) {
            int j = i;
            while (j > 0 && array[j-1] > array[j]) { //swap array[j-1] & array[j]
                int tmp = array[j-1];
                array[j-1] = array[j];
                array[j] = tmp;
                j--;
            }
        }
    }}
```

<u>Operation counting:</u>

In each phase i (for $1 \leq i < length$):

- At most i comparisons
- At most $O(i)$ elementary operations

$$T_2(n) = O\left(\sum_{i=1}^{length-1} i\right) = O(n^2)$$

Selection Sort

	Selection Sort	Insertion Sort
Best case	$O(n^2)$	$O(n)$
Average case	$O(n^2)$	$O(n^2)$
Worst case	$O(n^2)$	$O(n^2)$
In-place	Yes	Yes

Summary

Today's lecture:

- Introduced two basic sorting algorithms
  - Insertion sort
  - Selection sort
- Proved their correctness and time complexity

- **Next Lecture:** More efficient sorting algorithms (merge sort)
- **Any questions?**