

# DSC 40B

*Theoretical Foundations II*

Lecture 6 | Part 1

**Sorting Algorithms**

# Previously

- ▶ Binary search operation in an array
- ▶ Require that the array is already **sorted!**
- ▶ Answering queries is much faster on sorted data; remember the examples from the previous lecture.

# Motivation

- ▶ There are many reasons why we want to solve the sorting problem
  - ▶ scheduling, e.g, CPU may want to process tasks in decreasing order of priority
- ▶ Sorting can also make other problems easy
  - ▶ answering queries e.g. the search problem discussed last lecture
  - ▶ or more generally, range search in multidimensional databases etc.

# Today

- ▶ How do we sort?
- ▶ How fast can we sort?
- ▶ How do we use sorted structure to write faster algorithms?

# Today

- ▶ **Also:** how to understand complex loops with **loop invariants**.

# DSC 40B

## *Theoretical Foundations II*

Lecture 6 | Part 2

### **Selection Sort and Loop Invariants**

# Selection Sort (Simple Idea)

- ▶ Start with input array
- ▶ At each iteration, identify the smallest number in the remainder unsorted portion of the array
- ▶ Put it at the end of the already-sorted portion
- ▶ Iterate till the end

**Example:** arr = [5, 6, 3, 2, 1]



# In-place Selection Sort

- ▶ We don't need a separate list.
  - ▶ Meaning that it will only operate on the same array
  - ▶ We can swap elements until sorted.
- ▶ separate “good” / “bad” part of the array by a **barrier-id**

**Example:** arr = [5, 6, 3, 2, 1]

```
def selection_sort(arr):  
    """In-place selection sort."""  
    n = len(arr)  
    if n <= 1:  
        return  
    for barrier_ix in range(n-1):  
        # find index of min in arr[start:]  
        min_ix = find_minimum(arr, start=barrier_ix)  
        #swap  
        arr[barrier_ix], arr[min_ix] = (  
            arr[min_ix], arr[barrier_ix]  
        )
```

```
def find_minimum(arr, start):  
    """Finds index of minimum. Assumes non-empty."""  
    n = len(arr)  
    min_value = arr[start]  
    min_ix = start  
    for i in range(start + 1, n):  
        if arr[i] < min_value:  
            min_value = arr[i]  
            min_ix = i  
    return min_ix
```

# Loop Invariants

- ▶ How we understand an iterative algorithm?
- ▶ A **loop invariant** is a statement that is true after every iteration.
  - ▶ And before the loop begins!

# Loop Invariant(s)

After the  $\alpha$ th iteration of selection sort, each of the first  $\alpha$  elements is  $\leq$  each of the remaining elements.

**Example:** `arr = [5, 6, 3, 2, 1]`

# Loop Invariant(s)

After the  $\alpha$ th iteration, the first  $\alpha$  elements are sorted.

**Example:** `arr = [5, 6, 3, 2, 1]`

# Loop Invariants

- ▶ Plug the total number of iterations into the loop invariant to learn about the result.
  - ▶ `selection_sort` makes  $n - 1$  iterations:
  - ▶ After the  $(n - 1)$ th iteration, the first  $(n - 1)$  elements are sorted.
  - ▶ After the  $(n - 1)$ th iteration, each of the first  $(n - 1)$  elements is  $\leq$  each of the remaining elements.



# Loop Invariants (Summary)

- ▶ Base case:  $\alpha = 0$  loop invariant holds trivially.
- ▶ Inductive step:
  - ▶ If it holds for  $\alpha - 1$
  - ▶ then, we identify the smallest from the remainder  $n - \alpha$  numbers, which must be the  $\alpha$ -th smallest of the original array
  - ▶ so after this  $\alpha$ -th iteration, the loop invariant holds for  $\alpha$ .
- ▶ Thus the algorithm is correct in the end
  - ▶ it returns sorted array after  $n - 1$  iterations

# Time Complexity

```
def selection_sort(arr):  
    """In-place selection sort."""  
    n = len(arr)  
    if n <= 1:  
        return  
    for barrier_ix in range(n-1):  
        # find index of min in arr[barrier_ix:]  
        min_value = arr[barrier_ix]  
        min_ix = barrier_ix  
        for i in range(barrier_ix + 1, n):  
            if arr[i] < min_value:  
                min_value = arr[i]  
                min_ix = i  
  
        #swap  
        arr[barrier_ix], arr[min_ix] = (  
            arr[min_ix], arr[barrier_ix]  
        )
```

# Time Complexity

- ▶ Selection sort takes  $\Theta(n^2)$  time.
- ▶ Essentially nested loops
- ▶  $T(n) = cn + c(n - 1) + c(n - 2) + \dots + c \cdot 1 = \Theta(n^2)$

## Exercise

Modify `selection_sort` so that it computes a **median** of the input array. What is the time complexity?

```
def selection_sort(arr):  
    """In-place selection sort."""  
    n = len(arr)  
    if n <= 1:  
        return  
    for barrier_ix in range(n-1):  
        # find index of min in arr[start:]  
        min_ix = find_minimum(arr, start=barrier_ix)  
        #swap  
        arr[barrier_ix], arr[min_ix] = (  
            arr[min_ix], arr[barrier_ix]  
        )
```

# DSC 40B

## *Theoretical Foundations II*

Lecture 6 | Part 3

**Mergesort**

# Can we sort faster?

- ▶ The tight theoretical lower bound for **comparison** sorting is  $\Theta(n \log n)$ .
- ▶ Selection sort is quadratic.
- ▶ How do we sort in  $\Theta(n \log n)$  time?

# Mergesort

- ▶ Mergesort is a fast sorting algorithm.
- ▶ Has **best possible** (worst-case) time complexity:  $\Theta(n \log n)$ .
- ▶ Implements **divide/conquer/recombine** strategy.

# The Idea

- ▶ **Divide:** split the array into halves
  - ▶  $[6, 1, 9, 2, 4, 3] \rightarrow [6, 1, 9], [2, 4, 3]$
- ▶ **Conquer:** sort each half, recursively
  - ▶  $[6, 1, 9] \rightarrow [1, 6, 9]$  and  $[2, 4, 3] \rightarrow [2, 3, 4]$
- ▶ **Combine:** merge sorted halves together
  - ▶  $[1, 6, 9], [2, 3, 4] \rightarrow [1, 2, 3, 4, 6, 9]$



## Aside: splitting arrays

- ▶ Splitting an array in half by **slicing**:

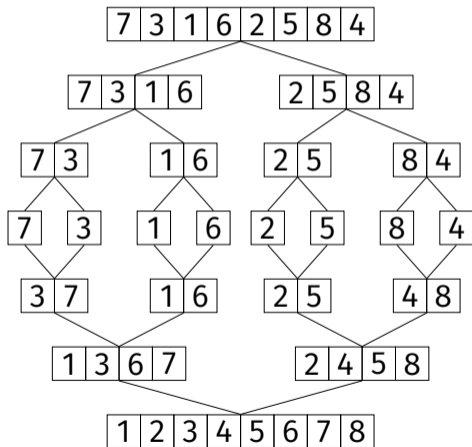
```
>>> arr = [9, 1, 4, 2, 5]
>>> middle = math.floor(len(arr) / 2)
>>> arr[:middle]
[9, 1]
>>> arr[middle:]
[4, 2, 5]
```

- ▶ **Warning!** Creates a copy!

# Mergesort

```
def mergesort(arr):  
    """Sort array in-place."""  
    if len(arr) > 1:  
        middle = math.floor(len(arr) / 2)  
        left = arr[:middle]  
        right = arr[middle:]  
        mergesort(left)  
        mergesort(right)  
        merge(left, right, arr)
```

# The Idea



# Understanding Mergesort

1. What is the base case?
2. Are the recursive problems smaller?
3. Assuming the recursive calls work, does the whole algorithm work?

## 1. Base Case: $n = 1$

- ▶ Arrays of size one are trivially sorted.
- ▶ Returns immediately. **Correct!**

## 2. Smaller Problems?

- ▶ Are `arr[:middle]` and `arr[middle:]` always smaller than `arr`?
- ▶ Try it for `len(arr) == 2`.

### 3. Does it Work?

- ▶ Assume mergesort works on arrays of size  $< n$ .
- ▶ Does it work on arrays of size  $n$ ?

# Mergesort

```
def mergesort(arr):  
    """Sort array in-place."""  
    if len(arr) > 1:  
        middle = math.floor(len(arr) / 2)  
        left = arr[:middle]  
        right = arr[middle:]  
        mergesort(left)  
        mergesort(right)  
        merge(left, right, arr)
```



# DSC 40B

## *Theoretical Foundations II*

Lecture 6 | Part 4

**Merge**

# Merging

- ▶ We have sorted each half.
- ▶ Now we need to **merge** together.

# Merging

- ▶ We have sorted each half.
- ▶ Now we need to **merge** together.
- ▶ **Note:** this is an example of a problem that is made easier by sorting.

merge

3

1

# merge



3



2



1

# merge

3

6

1

2

# merge

5

6

1

2

3

# merge

7

6

1

2

3

5



# merge

7

1

2

3

5

6

# merge

8

1 2 3 5 6 7

merge

1 2 3 5 6 7 8

# merge

```
def merge(left, right, out):  
    """Merge sorted arrays, store in out."""  
    left.append(float('inf'))  
    right.append(float('inf'))  
    left_ix = 0  
    right_ix = 0  
  
    for ix in range(len(out)):  
        if left[left_ix] < right[right_ix]:  
            out[ix] = left[left_ix]  
            left_ix += 1  
        else:  
            out[ix] = right[right_ix]  
            right_ix += 1
```

# Loop Invariant

- ▶ Assume `left` and `right` are sorted.
- ▶ **Loop invariant:** After  $\alpha$ th iteration,  
`out[: $\alpha$ ] == sorted(left + right)[: $\alpha$ ]`

## Key of mergesort

- ▶ merge is where the **actual sorting** happens.
- ▶ Example: `merge([3], [1], ...)` results in `[1,3]`

# Time Complexity of merge

```
def merge(left, right, out):  
    """Merge sorted arrays, store in out."""  
    left.append(float('inf'))  
    right.append(float('inf'))  
    left_ix = 0  
    right_ix = 0  
  
    for ix in range(len(out)):  
        if left[left_ix] < right[right_ix]:  
            out[ix] = left[left_ix]  
            left_ix += 1  
        else:  
            out[ix] = right[right_ix]  
            right_ix += 1
```

# DSC 40B

## *Theoretical Foundations II*

Lecture 6 | Part 5

### **Time Complexity of Mergesort**



# Time Complexity

```
def mergesort(arr):  
    """Sort array in-place."""  
    if len(arr) > 1:  
        middle = math.floor(len(arr) / 2)  
        left = arr[:middle]  
        right = arr[middle:]  
        mergesort(left)  
        mergesort(right)  
        merge(left, right, arr)
```

## Aside: Copying

- ▶ What is `arr[:middle]` doing “under the hood”?
- ▶ What is the time complexity?

# The Recurrence

```
def mergesort(arr):  
    """Sort array in-place."""  
    if len(arr) > 1:  
        middle = math.floor(len(arr) / 2)  
        left = arr[:middle]  
        right = arr[middle:]  
        mergesort(left)  
        mergesort(right)  
        merge(left, right, arr)
```

# Solving the Recurrence

$$T(n) = 2T(n/2) + \Theta(n)$$

# Optimality

- ▶ **Theorem:** Any (comparison) sorting algorithm's worst-case time complexity must be  $\Omega(n \log n)$ .
- ▶ Mergesort is **optimal!**

# Be Careful!

- ▶ It is possible for a sorting algorithm to have a **best case** time complexity smaller than  $n \log n$ .
  - ▶ Insertion sort, for example.
- ▶ Mergesort has best case time complexity of  $\Theta(n \log n)$ .
- ▶ Mergesort is **sub-optimal** in this sense!

# Be Careful!

- ▶ The  $\Theta(n \log n)$  lower-bound is for **comparison sorting**.
- ▶ It is possible to sort in worst-case  $\Theta(n)$  time without comparing.<sup>1</sup>

---

<sup>1</sup>Bucket sort, radix sort, etc.

# What if?

- ▶ **Divide:** split the array into halves
- ▶ **Conquer:** sort each half **using selection sort**
- ▶ **Combine:** merge sorted halves together



# mergeselectionsort

```
def mergeselectionsort(arr):  
    """Sort array in-place."""  
    if len(arr) > 1:  
        middle = math.floor(len(arr) / 2)  
        left = arr[:middle]  
        right = arr[middle:]  
        selection_sort(left)  
        selection_sort(right)  
        merge(left, right, arr)
```

## Exercise

What is the time complexity of this algorithm?

# DSC 40B

*Theoretical Foundations II*

Lecture 6 | Part 6

**Using Sorted Structure**

# Sorted structure is useful

- ▶ Some problems become **much easier** if input is sorted.
  - ▶ For example, median, minimum, maximum.
- ▶ Sorting is useful as a **preprocessing** step.

## Recall: The Movie Problem

- ▶ You're on a flight that will last  $D$  minutes.
- ▶ You want to pick two movies to watch.
- ▶ You want the total time of the two movies to be **as close as possible** to  $D$ .

# The Movie Problem

- ▶ Brute force algorithm:  $\Theta(n^2)$
- ▶ We can do better, if movie times are **sorted**.

# Example

- ▶ Flight duration  $D = 155$
- ▶ Movie times: 60, 80, 90, 120, 130

	60	80	90	120	130
60					
80					
90					
120					
130					

Best pair:

# The Algorithm

- ▶ Keep index of shortest and longest remaining.
- ▶ On every iteration, pair the shortest and longest.
- ▶ If this pair is too long, remove longest movie; otherwise remove shortest.
  - ▶ If times are **sorted**, finding new longest/shortest movie takes  $\Theta(1)$  time!

60, 80, 90, 120, 130



# The Algorithm

```
def optimize_entertainment(times, target):  
    """assume times is sorted."""  
    shortest = 0  
    longest = len(times) - 1  
  
    best_pair = (shortest, longest)  
    best_objective = None  
  
    for i in range(len(times) - 1):  
        total_time = times[shortest] + times[longest]  
  
        if abs(total_time - target) < best_objective:  
            best_objective = abs(total_time - target)  
            best_pair = (shortest, longest)  
  
        if total_time == target:  
            return (shortest, longest)  
        elif total_time < target:  
            shortest += 1  
        else: # total_time > target  
            longest -= 1  
  
    return best_pair
```

## Main Idea

Sorted structure allows you to rule out possibilities without explicitly checking them. But, it requires you to spend the time sorting first.

Tip: when designing an algorithm, think about sorting the input first.