DSC 40B
Theoretical Foundations II

Lecture 6 | Part 1

Selection Sort and Loop Invariants

Sorting

- Sorting is a very common operation.
- But why is it important?

Sorting

- Sorting is a very common operation.
- But why is it important?
- A e s t h e t i c reasons?

Sorting

- Sorting is a very common operation.
- But why is it important?
- `A e s t h e t i c` reasons?
- Sorting makes some problems easier to solve.

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**.

Selection Sort

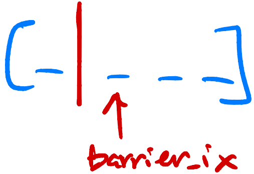
- Repeatedly remove smallest element.
- Put it at beginning of new list.

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

[1, 2, 3, 5, 6]

In-place Selection Sort

- We don’t need a separate list.
  - We can swap elements until sorted.



- Store “new” list at the beginning of input list.

- Separate the old and new with a **barrier**.

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

[1, 2, 3, 5, 6]

barrier_ix

a, b = b, a

```python
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]
        )
```

```python
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 do 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 αth iteration of selection sort, each of the first α elements is ≤ each of the remaining elements.

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

[1, 2, 3, 6, 5] $\alpha = 2$

Loop Invariant(s)

After the αth iteration, the first α 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.

Time Complexity

$\frac{n}{2} + \left(\frac{n}{2} - 1\right) + \cdots$

$\frac{\frac{n}{2}\left(\frac{n}{2}+1\right)}{2} = \Theta(n^2)$

$(n-1) + (n-2) + (n-3)$

$+ 3 + 2 + 1 = \Theta(n^2)$

```python
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]
        )
```

execs?

Time Complexity

- Selection sort takes $\Theta(n^2)$ time.

Exercise

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

```python
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]
        )
```

$\Theta(n^2)$

n//2

DSC 40B

Theoretical Foundations II

Lecture 6 | Part 2

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]` → `[6,1,9]`, `[2,4,3]`
- **Conquer**: sort each half, recursively
  - `[6,1,9]` → `[1,6,9]` and `[2,4,3]` → `[2,3,4]`
- **Combine**: merge sorted halves together
  - `[1,6,9]`, `[2,3,4]` → `[1,2,3,4,6,9]`

Aside: splitting arrays

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

n

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

$\Theta(n)$

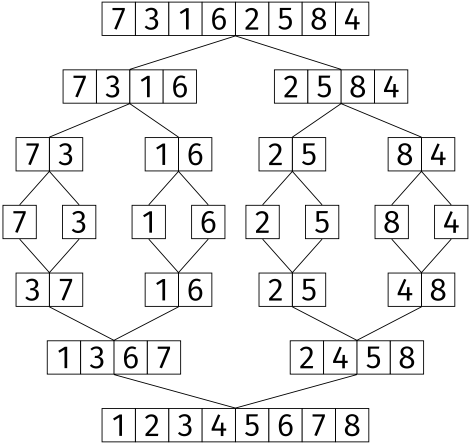
- **Warning!** Creates a copy!

Mergesort

print(arr)

```python
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



ms([7,3,1,6,2,5,8,4])

ms(7,3,1,6)

ms(7,3)

ms(7)

ms(3)

merge(7,3)

⋮

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

```python
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 3

Merge

Merging

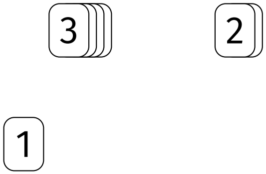
- We have sorted each half.
- Now we need to **merge** together.

Merging

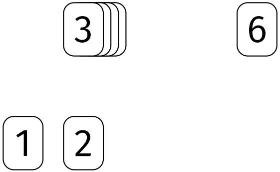
- We have sorted each half.
- Now we need to **merge** together.
- **Note:** this is an example of a problem that is made easier by sorting.

merge

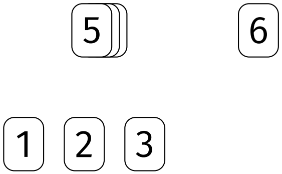
`merge`



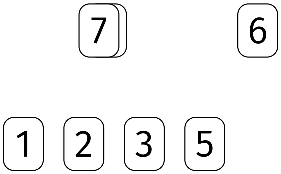
merge



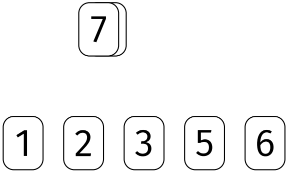
merge



merge



merge

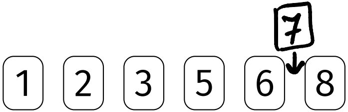


`merge`

8

1 2 3 5 6 7

merge



merge

left = [1, 4, 5, 8, ∞]

l_ix

right = [3, 6, 12, 15, ∞]

r_ix

out = [1, 3, 4, 5, 6, 8, 12, 15]

```python
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 αth iteration, first α elements of `out` are the smallest α elements of those in `left` and `right`, in sorted order.
- That is, after αth iteration,
  `out[:α] == sorted(left + right)[:α]`

Key of `mergesort`

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

Time Complexity of `merge`

n

len(left) + len(right) = n

```
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
```

$\Theta(n)$

DSC 40B

Theoretical Foundations II

Lecture 6 | Part 4

Time Complexity of Mergesort

Time Complexity

```python
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?

$\Theta(n)$

The Recurrence

$$T(n) = \Theta(n) + 2T(\tfrac{n}{2})$$

```python
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)
```

$\Theta(n)$

$T(n/2)$

$T(n/2)$

$2T(n/2)$

Solving the Recurrence

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

$= 2T(\frac{n}{2}) + n \quad k=1$

$T(\frac{n}{2}) = \underbrace{2T(\frac{n}{4}) + \frac{n}{2}}$

$= 2\left[2T(\frac{n}{4}) + \frac{n}{2}\right] + n$

$T(\frac{n}{4}) = 2T(\frac{n}{8}) + \frac{n}{4}$

$= 4T(\frac{n}{4}) + \frac{2n}{2} + n$

$= 4T(\frac{n}{4}) + 2n \quad k=2$

$= 8T(\frac{n}{8}) + 3n \quad k=3$

$$\boxed{T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn}$$

on step k

Solving the Recurrence

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

Solve $\frac{n}{2^k} = 1$ for k:

$$n = 2^k \Rightarrow k = \log_2 n$$

$$T(n) = 2^k T\left(\frac{n}{2^k}\right) + kn$$

$$T(n) = \cancelto{n}{2^{\log_2 n}}\, T\left(\cancel{\frac{n}{2^{\log_2 n}}}\right) + (\log_2 n) \times n$$

$\Theta(1)$ $\quad$ $n/n = 1$

$$= n\,T(1) + n \log_2 n = n \times \Theta(1) + n \log n$$

$$= \Theta(n \log n)$$

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.[1]

[1] 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)
```

$\Theta(n^2)$

Exercise

What is the time complexity of this algorithm?

DSC 40B

Theoretical Foundations II

Lecture 6 | Part 5

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	✗	✗	−5	+25	+35
80	✗			✗	✗
90	✗	✗		✗	✗
120	✗	✗	✗	✗	✗
130	✗	✗	✗	✗	✗

Best pair:

~~(60, 130) +35~~

~~(60, 120) +25~~

(60, 90) −5

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

```python
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.