DSC 40B

Lecture 10 : Sorting

HW3: change git

Sorting

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 (and Friday)

- How do we sort?
- How fast can we sort?
- How do we use sorted structure to write faster algorithms?
- **Also**: how to understand complex loops with **loop invariants.**

Selection Sort

Selection Sort

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

Example

Example: `arr = [7, 8, 5, 3, 1]`

Example

Example: `arr = [7, 8, 5, 3, 1]`

- `[                    ]`

Example

Example: `arr = [7, 8, 5, 3, 1]`

- [1,]

Example

Example: arr = [7, 8, 5, 3, ~~1~~]

- [1,   ]

Example

Example: arr = [7, 8, 5, 3, ~~1~~]

- [1, 3,   ]

Example

Example: `arr = [7, 8, 5, ~~3~~, ~~1~~]`

- [1, 3,   ]

Example

Example: `arr = [7, 8, 5, ~~3~~, ~~1~~]`

- [1, 3, 5,   ]

Example

Example: `arr = [7, 8, ~~5~~, ~~3~~, ~~1~~]`

- [1, 3, 5,]

Example

Example: `arr = [7, 8, ~~5~~, ~~3~~, ~~1~~]`

- `[1, 3, 5, 7,     ]`

Example

Example: `arr = [~~7~~, 8, ~~5~~, ~~3~~, ~~1~~]`

- [1, 3, 5, 7,   ]

Example

Example: `arr = [`~~7~~, ~~8~~, ~~5~~, ~~3~~, ~~1~~`]`

- [1, 3, 5, 7, 8]

Example

Example: `arr = [~~7~~, ~~8~~, ~~5~~, ~~3~~, ~~1~~]`

- **[1, 3, 5, 7, 8]**

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: sorting in-place

Example: `arr = [7, 8, 5, 3, 1]`

Example: sorting in-place

Example: `arr = [7, 8, 5, 3, 1]`

`[7, 8, 5, 3, 1]`

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[7, 8, 5, 3, 1]

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: `arr = [7, 8, 5, 3, 1]`

`[7, 8, 5, 3, 1]` *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: `arr = [7, 8, 5, 3, 1]`

`[|7, 8, 5, 3, 1]` *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[**1**, 8, 5, 3, 7] *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[1, | 8, 5, 3, 7] *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[1, | 8, 5, **3**, 7]   *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[1, | **3**, 5, 8, 7] *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[1, 3, | 5, 8, 7] *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[1, 3, | 5, 8, 7] *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[**1, 3, 5,** | 8, 7] *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[1, 3, 5, | 8, 7] *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[1, 3, 5, | 7, 8] *#swap, move the barrier*

Example: sorting in-place

Left: sorted.

Right: needs to be sorted

Example: arr = [7, 8, 5, 3, 1]

[1, 3, 5, 7, 8] *#swap, move the barrier*

```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 = [7, 8, 5, 3, 1]`

Loop Invariant(s)

- After the αth iteration of selection sort, each of the first α elements is ≤ each of the remaining elements.
- α = 0: `arr = [7, 8, 5, 3, 1] #empty set. True`
- α = 1: `arr = [1, 8, 5, 3, 7] # 1 is ≤`
- α = 2: `arr = [1, 3, 5, 8, 7] # 1, 3 are ≤`

Loop Invariant(s)

- After the αth iteration of selection sort, each of the first α elements is ≤ each of the remaining elements.
- $\alpha = 0$: `arr = [7, 8, 5, 3, 1] #empty set. True`
- $\alpha = 1$: `arr = [1, 8, 5, 3, 7] # 1 is ≤`
- $\alpha = 2$: `arr = [1, 3, 5, 8, 7] # 1, 3 are ≤`

Is it enough to claim that the algorithm works properly?

A: Yes

B: No

Loop Invariant(s)

- After the αth iteration of selection sort, each of the first α elements is ≤ each of the remaining elements.
- α = 0: `arr = [|7, 8, 5, 3, 1] #empty set. True`
- α = 1: `arr = [1,| 8, 5, 3, 7] # 1 is ≤`
- α = 2: `arr = [3, 1,| 5, 8, 7] # 1, 3 are ≤`

Loop Invariant(s)

- After the αth iteration, the first α elements are sorted.

Loop Invariants

- Plug the total number of iterations into the loop invariant to learn about the results
  - `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 ≤ each of the remaining elements.

```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]
        )
```

(n-1) +

```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]
        )
```

(n-1) + (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]
        )
```

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

```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]
        )
```

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

```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]
        )
```

(n-1) + (n-2) + (n-3) +...+ 1 =

$\Theta(n^2)$

Time Complexity

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

Time Complexity

- How about sorted array input?

A: $\Theta(1)$ time.

B: $\Theta(n)$ time.

C: $\Theta(n^2)$ time.

D: Something else

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]
        )
```

MergeSort

mic

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

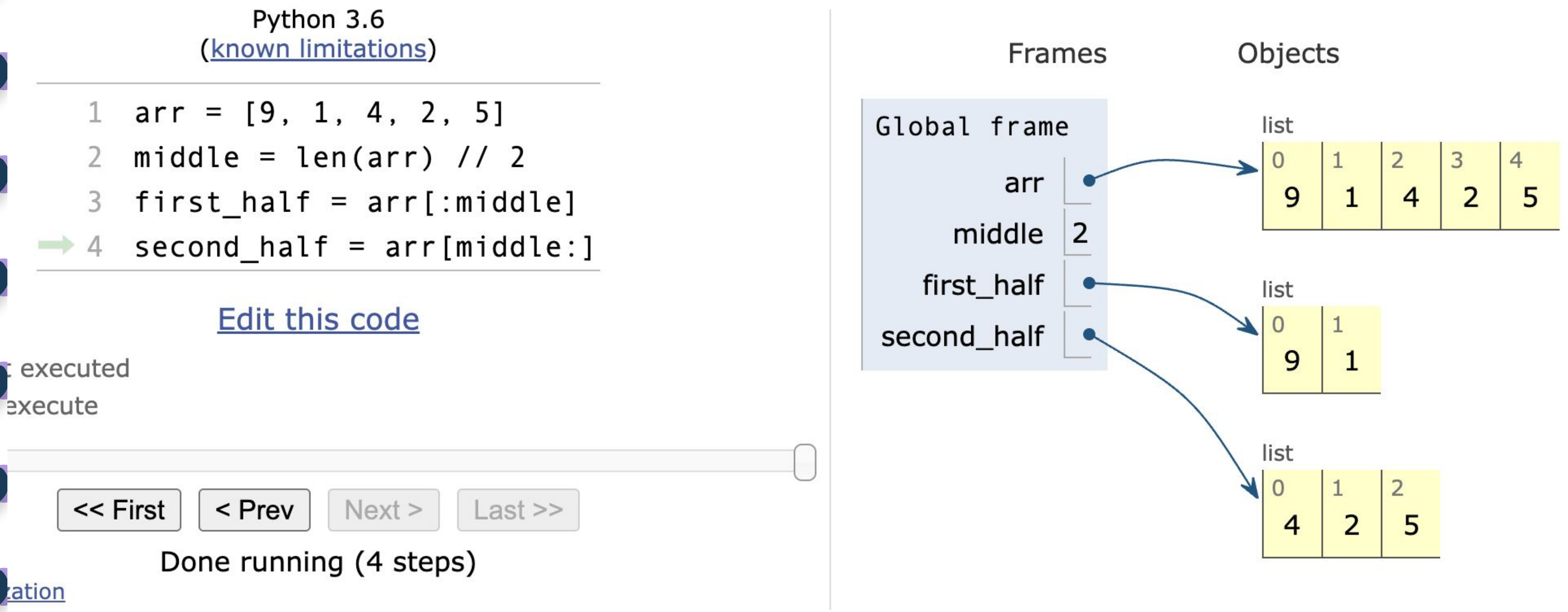
- **Divide**: split the array into halves
  - [6,1,9,2,4,3] → [6,1,9], [2,4,3]
- **Conquer**: sort each half, recursively
  - [6,1,9] → [1,6,9] and [2,4,3] → [2,3,4]
- **Combine**: merge sorted halves together
  - [1,6,9], [2,3,4] → [1,2,3,4,6,9]

Aside: splitting arrays

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

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

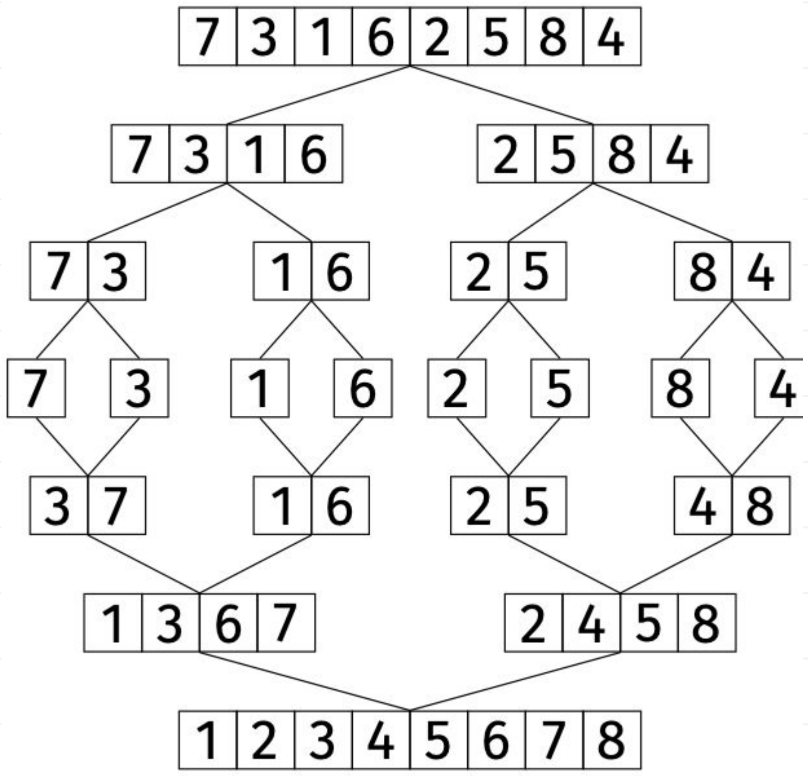
- **Warning! Creates a copy!**


Python 3.6
(known limitations)
arr = [9, 1, 4, 2, 5]
middle = len(arr) // 2
first_half = arr[:middle]
second_half = arr[middle:]
Edit this code
executed
execute
<< First
< Prev
Next >
Last >>
Done running (4 steps)
Frames
Objects
Global frame
arr
middle
2
first_half
second_half
list
0 1 2 3 4
9 1 4 2 5
list
0 1
9 1
list
0 1 2
4 2 5

Mergesort

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

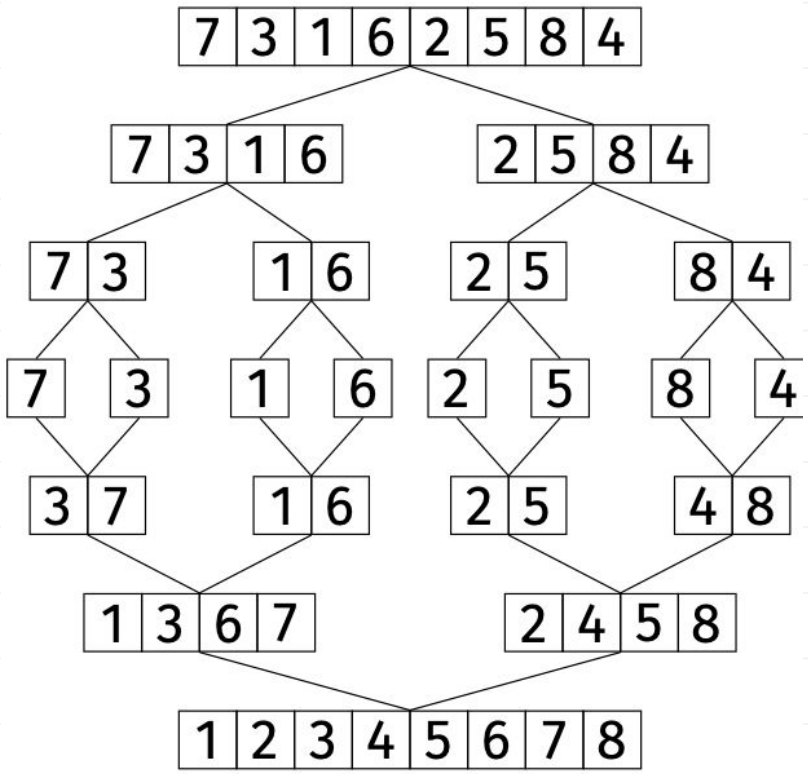
The Idea




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

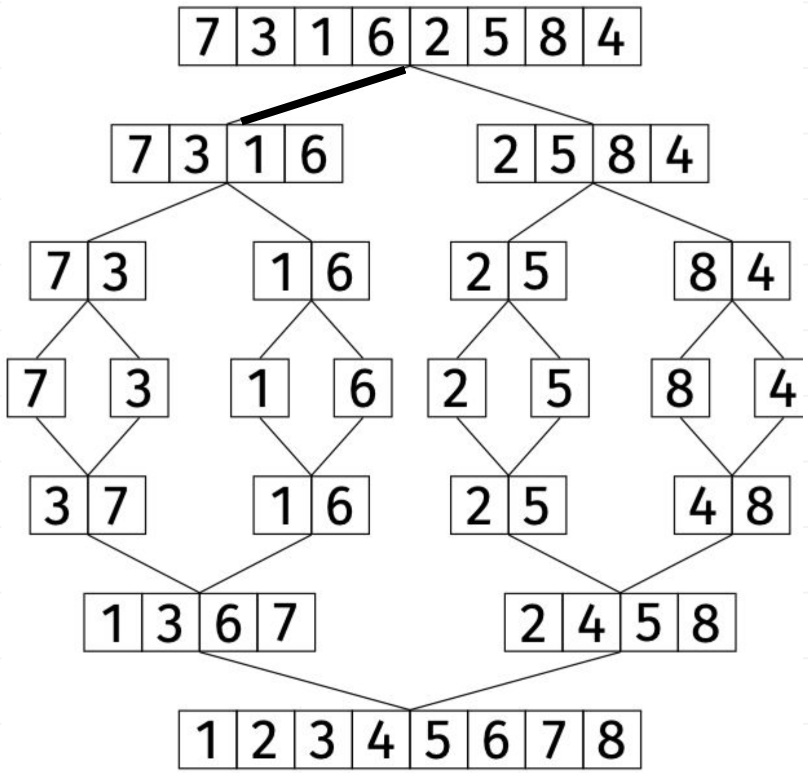
slidesmania.com

The Idea



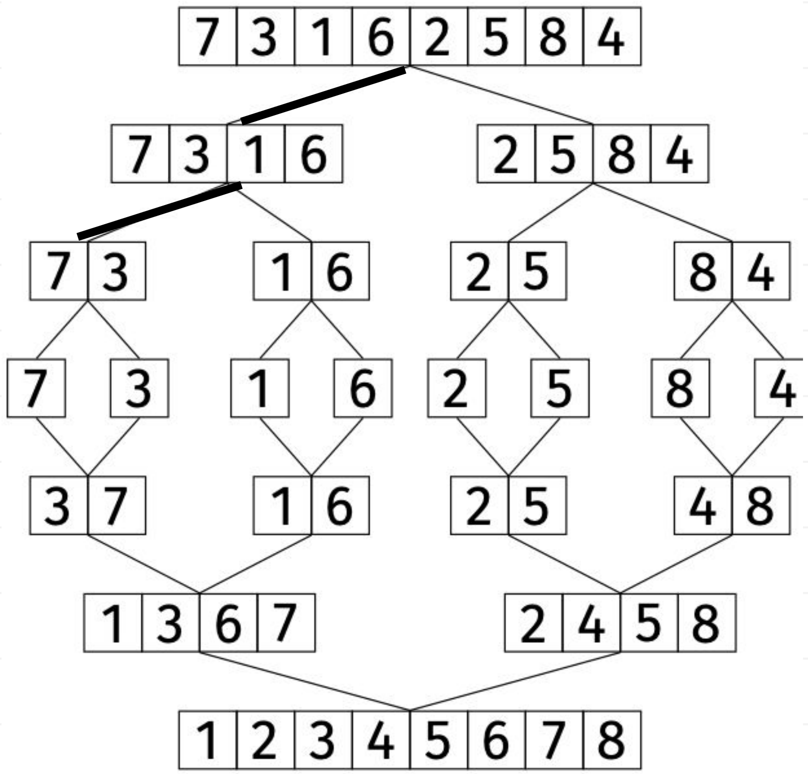

```
ms([7, 3, 1, 6, 2, 5, 8, 4])
```

The Idea



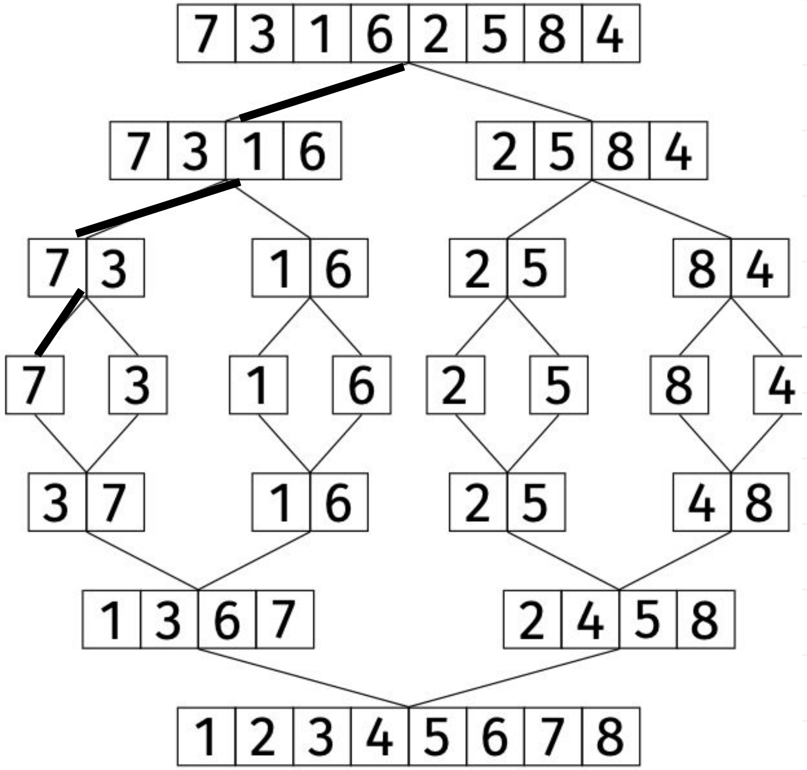
```
ms([7, 3, 1, 6, 2, 5, 8, 4])
    ms([7, 3, 1, 6])
```

The Idea



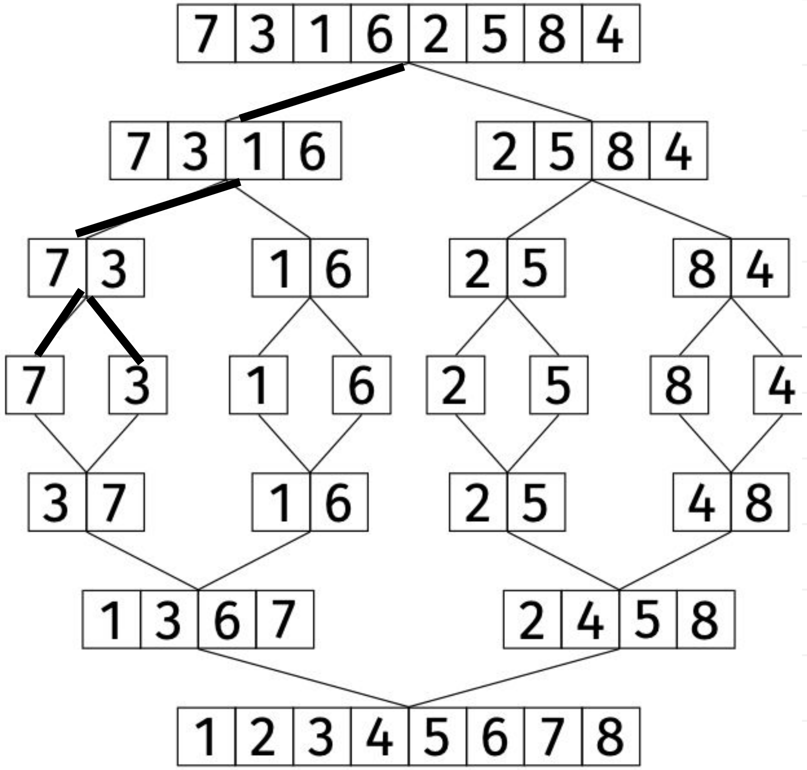

```
ms([7, 3, 1, 6, 2, 5, 8, 4])
  ms([7, 3, 1, 6])
    ms([7, 3])
```

The Idea



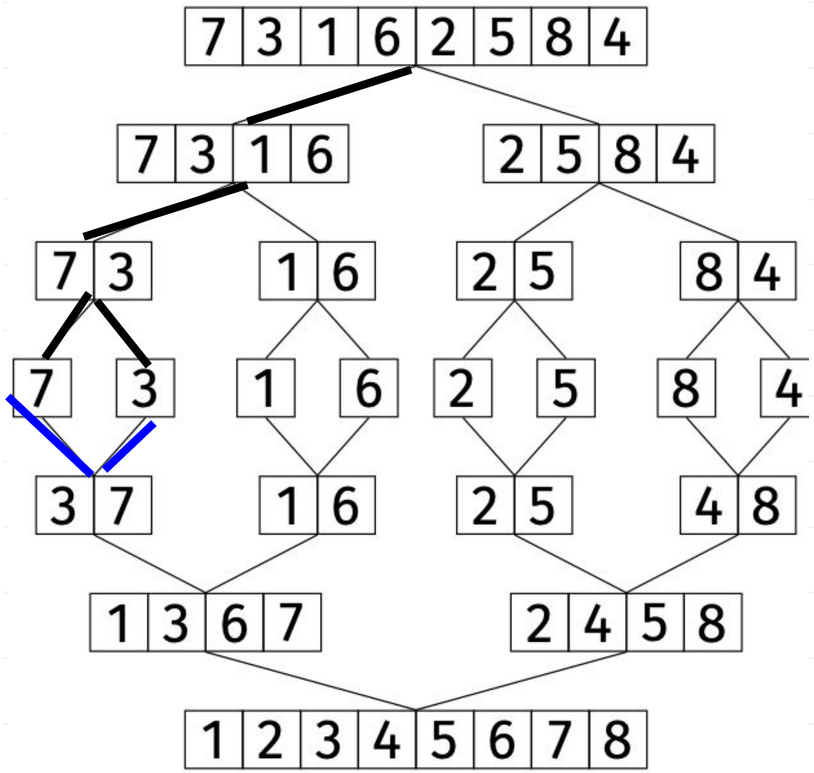

```
ms([7, 3, 1, 6, 2, 5, 8, 4])
  ms([7, 3, 1, 6])
    ms([7, 3])
      ms([7])
```

The Idea




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

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])
```

Mergesort: tip for tracing

```
def mergesort(arr):
    print(arr)
    if len(arr) > 1:
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]
        mergesort(left)
        mergesort(right)
        merge(left, right, arr)
```

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`.
- **Correct!**

3. Does it Work?

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

Mergesort

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

Does it work properly?

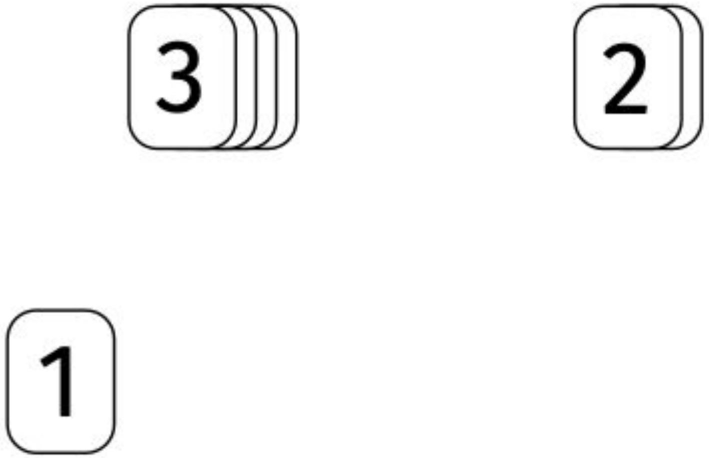
Merge

Merging

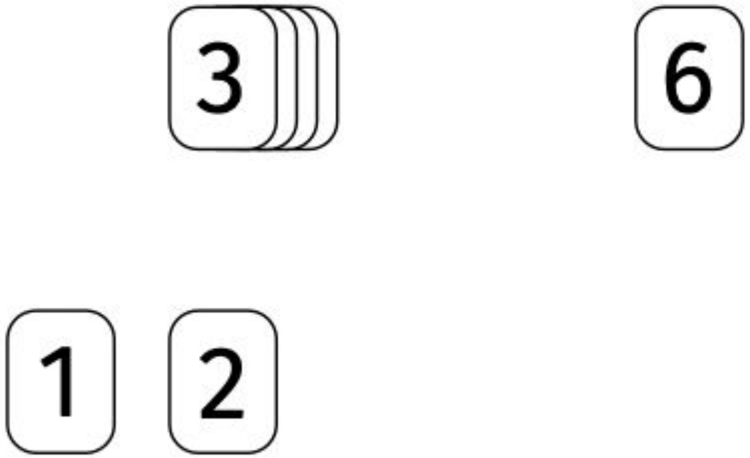
- We have sorted each half.
- Now we need to merge together into a sorted array.
- **Note**: this is an example of a problem that is made easier by sorting.

Merge. Stack of cards in SORTED order

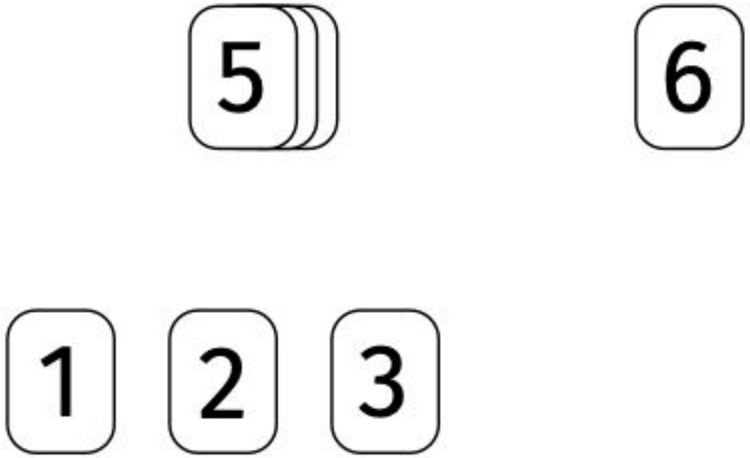
merge



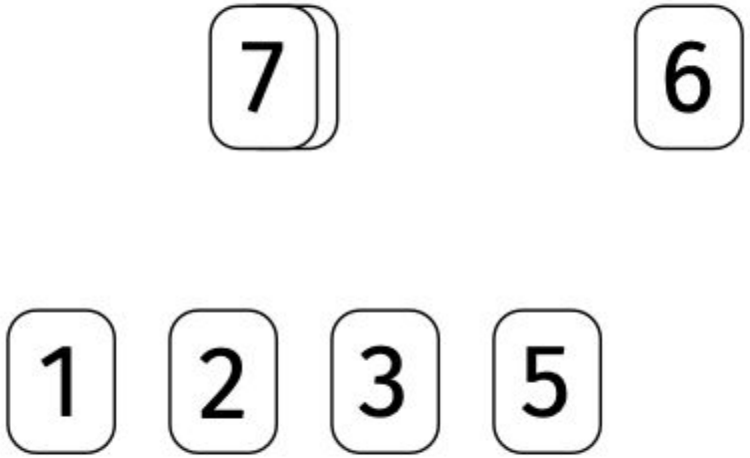
merge



merge



merge



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
```

merge

[5, 7, 9, 10]

```
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
```

merge

[5, 7, 9, 10, **'inf'**]

```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
```

merge

left = [5, 7, 9, 10, ‘**inf**’]

```
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
```

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
```

left = [5, 7, 9, 10, '**inf**']

right = [2, 4, 6, '**inf**']

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
```

left = [5, 7, 9, 10, ‘**inf**’]

right = [2, 4, 6, ‘**inf**’]

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
```

left = [5, 7, 9, 10, ‘**inf**’]

right = [2, 4, 6, ‘**inf**’]

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
```

left = [5, 7, 9, 10, '**inf**']

right = [2, 4, 6, '**inf**']

out = [, , , , , ,]

merge

```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
```

left = [5, 7, 9, 10, **'inf'**]

right = [2, 4, 6, **'inf'**]

out = [2, , , , , ,]

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
```

left = [5, 7, 9, 10, ‘**inf**’]

right = [2, 4, 6, ‘**inf**’]

out = [2, 4,   ,   ,   ,   ,   ]

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
```

left = [5, 7, 9, 10, **'inf'**]

right = [2, 4, 6, **'inf'**]

out = [2, 4, 5, , , ,]

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
```

left = [5, 7, 9, 10, **'inf'**]

right = [2, 4, 6, **'inf'**]

out = [2, 4, 5, 6, , ,]

merge

```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
```

left = [5, 7, 9, 10, **'inf'**]

right = [2, 4, 6, **'inf'**]

out = [2, 4, 5, 6, 7, ,]

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
```

left = [5, 7, 9, 10, **'inf'**]

right = [2, 4, 6, **'inf'**]

out = [2, 4, 5, 6, 7, 9,]

merge

```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
```

left = [5, 7, 9, 10, ‘**inf**’]

right = [2, 4, 6, ‘**inf**’]

out = [2, 4, 5, 6, 7, 9, 10]

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 Mergesort

Mergesort: time complexity

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

T(n) = non_recursive + recursive

Mergesort: time complexity

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

T(n) = **non_recursive** + recursive

Mergesort: time complexity

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

T(n) = **non_recursive** + recursive

$\Theta(n)$

Mergesort: time complexity

$T(n) = \Theta(n)$ + **recursive**

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

Mergesort: time complexity

$T(n) = \Theta(n)$ + **recursive**

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

Mergesort: time complexity

$T(n) = \Theta(n)$ + **recursive**

```
def mergesort(arr):
    if len(arr) > 1:
        middle = math.floor(len(arr) / 2)
        left = arr[:middle]
        right = arr[middle:]
        mergesort(left)        <- T(n/2)
        mergesort(right)       <- T(n/2)
        merge(left, right, arr)
```

Mergesort: time complexity

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

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

$T(n/2)$

$T(n/2)$

Solving the Recurrence

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

Solving the Recurrence. Step 2

T(n) = 2 T(n/2) + n # ***k=1***

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

Solving the Recurrence. Step 2

T(n) = 2 T(n/2) + n # ***k=1***

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

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

Solving the Recurrence. Step 2

T(n) = 2 T(n/2) + n # ***k=1***

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

$$T(n) = 2 \cdot \left[2 \cdot T(\tfrac{n}{4}) + \frac{n}{2}\right] + n$$ **# simplify**

Solving the Recurrence. Step 2

T(n) = 2 T(n/2) + n # ***k=1***

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

$$T(n) = 2 \cdot \left[2 \cdot T(\tfrac{n}{4}) + \frac{n}{2}\right] + n$$ **# simplify**

$$T(n) = 4 \cdot T(\tfrac{n}{4}) + 2n$$ # ***k=2***

Solving the Recurrence. Step 2

T(n) = 2 T(n/2) + n # ***k=1***

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

$$T(n) = 2 \cdot \left[2 \cdot T(\tfrac{n}{4}) + \frac{n}{2}\right] + n$$ **# simplify**

$$T(n) = 4 \cdot T(\tfrac{n}{4}) + 2n$$ # ***k=2***

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

Solving the Recurrence. Step 2

T(n) = 2 T(n/2) + n # ***k=1***

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

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

$$T(n) = 4 \cdot T(\frac{n}{4}) + 2n$$ # ***k=2***

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

$$T(n) = 4 \cdot \left[2 \cdot T(\frac{n}{8}) + \frac{n}{4}\right] + 2n$$

Solving the Recurrence. Step 2

T(n) = 2 T(n/2) + n # ***k=1***

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

$$T(n) = 2 \cdot \left[2 \cdot T(\tfrac{n}{4}) + \frac{n}{2}\right] + n$$ **# simplify**

$$T(n) = 4 \cdot T(\tfrac{n}{4}) + 2n$$ # ***k=2***

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

$$T(n) = 4 \cdot \left[2 \cdot T(\tfrac{n}{8}) + \frac{n}{4}\right] + 2n$$ **# simplify**

Solving the Recurrence. Step 2

$T(n) = 2\,T(n/2) + n$ # **k=1**

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

$$T(n) = 2 \cdot \left[2 \cdot T(\tfrac{n}{4}) + \frac{n}{2}\right] + n$$ # **simplify**

$$T(n) = 4 \cdot T(\tfrac{n}{4}) + 2n$$ # **k=2**

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

$$T(n) = 4 \cdot \left[2 \cdot T(\tfrac{n}{8}) + \frac{n}{4}\right] + 2n$$ # **simplify**

$$T(n) = 8 \cdot T(\tfrac{n}{8}) + 3n$$ # **k=3**

$$T(n) = 2^k\, T(\tfrac{n}{2^k}) + kn$$

Step 3. Number of steps to Base case.

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

Step 3. Number of steps to Base case.

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

$$\frac{n}{2^k} = 1 \quad n = 2^k \quad => k = log_2 n$$

Step 4. Substitute

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

$$\frac{n}{2^k} = 1 \quad n = 2^k \quad => k = log_2 n$$

Step 4. Substitute

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

$$\frac{n}{2^k} = 1 \quad n = 2^k \quad => k = log_2 n$$

$$T(n) = 2^{log_2 n} \cdot T(\frac{n}{2^{log_2 n}}) + n \ log_2 n$$

Step 4. Substitute

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

$$\frac{n}{2^k} = 1 \quad n = 2^k \quad => k = \log_2 n$$

$$T(n) = 2^{\log_2 n} \cdot T(\frac{n}{2^{\log_2 n}}) + n \log_2 n$$

$$T(n) = n \cdot T(1) + n \log_2 n$$

Step 4. Substitute

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

$$\frac{n}{2^k} = 1 \quad n = 2^k \quad => \quad k = \log_2 n$$

$$T(n) = 2^{\log_2 n} \cdot T\left(\frac{n}{2^{\log_2 n}}\right) + n \log_2 n$$

$$T(n) = n \cdot T(1) + n \log_2 n$$

$\Theta(n$ 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 $\boldsymbol{\Theta(n \log n)}$.
- 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.
    - 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

```python
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)
```

Think!

What is the time complexity of this algorithm?

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:

Example: with brute force only half

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

	60	80	90	120	130
60					
80	x				
90	x	x			
120	x	x	x		
130	x	x	x	x	

Best pair:

Example: with brute force only half

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

	60	80	90	120	130
60	x				
80	x	x			
90	x	x	x		
120	x	x	x	x	
130	x	x	x	x	x

Best pair:

Example:

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

First step: take **shortest** + **longest**.

	60	80	90	120	130
60	x				+35
80	x	x			
90	x	x	x		
120	x	x	x	x	
130	x	x	x	x	x

Best pair: 60, 130

+35 above the target

Example:

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

First step: take **shortest** + **longest**.

	60	80	90	120	130
60	x			+25	+35
80	x	x			x
90	x	x	x		x
120	x	x	x	x	x
130	x	x	x	x	x

Best pair: 60, 120

+25 above the target

Example:

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

First step: take **shortest** + **longest**.

	60	80	90	120	130
60	x			+25	+35
80	x	x		x	x
90	x	x	x	x	x
120	x	x	x	x	x
130	x	x	x	x	x

Best pair: 60, 120

+25 above the target

Example:

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

First step: take **shortest** + **longest**.

	60	80	90	120	130
60	x		-5	+25	+35
80	x	x		x	x
90	x	x	x	x	x
120	x	x	x	x	x
130	x	x	x	x	x

Best pair: 60, 90
-5 below the target

Example:

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

First step: take **shortest** + **longest**.

	60	80	90	120	130
60	x	x	-5	+25	+35
80	x	x		x	x
90	x	x	x	x	x
120	x	x	x	x	x
130	x	x	x	x	x

Best pair: 60, 90

-5 below the target

The Algorithm

- Keep *index* of shortest and longest remaining movies.
- 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

```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.

Thank you!

Do you have any questions?

CampusWire!