DSC40B: Theoretical Foundations of Data Science II

Lecture 6: *Sorting, and more on recurrences*

Instructor: Yusu Wang

Previously

- Binary search operation in an array
  - Require that the array is already sorted!

- Today: the sorting problem
  - Input: given an arbitrary array of numbers
  - Output: convert them into an array where all elements are either in non-decreasing or non-increasing order.
    - from now on, unless otherwise specified, in this class, we will assume a sorted array is in non-decreasing order.

Motivation

- There are many reasons why we want to solve the sorting problem
  - Given a list of tasks with different priority values, the CPU may want to process them in decreasing order of priority
  - Sorting can also make other problems easy
    - E.g, the search problem discussed last lecture,
    - or more generally, range search in multidimensional databases etc.

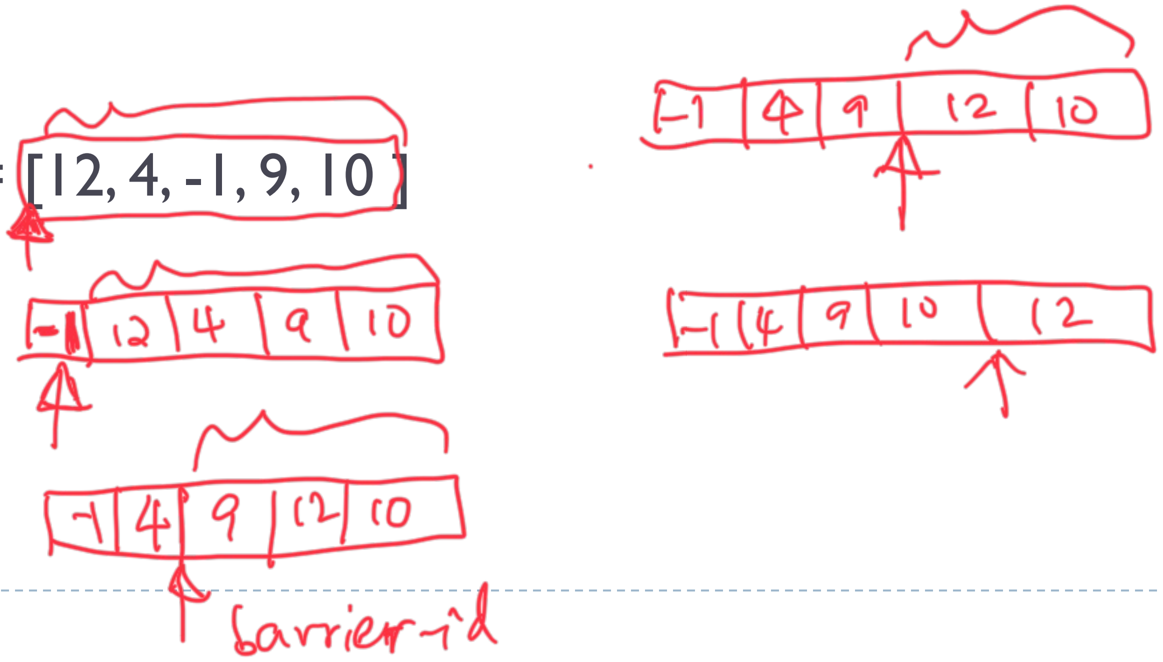
- But we will just focus on the simplest version
  - where the input is just a list of real numbers stored in an array.

Part A:
(1) A simple sorting algorithm: Selection sort
(2) Correctness of algorithm via loop invariants

A 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:
  - Input array A = [12, 4, -1, 9, 10]



- How to implement this idea using an algorithm

- How to prove the correctness of the algorithm

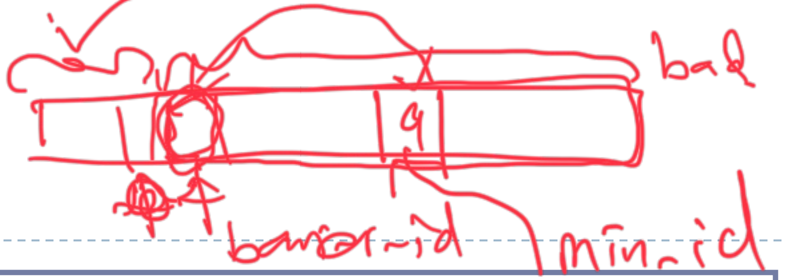
- Time complexity

- How to implement this idea using an algorithm
  - *in-place* selection sort
    - meaning that it will only operate on the same array
  - separate "good" / "bad" part of the array by a barrier-id

- How to prove the correctness of the algorithm

- Time complexity

Algorithm selection_sort



```
def selection_sort(A):
    n = len(A)
    if n <= 1:
        return
    for barrier_id in range(n-1):
        # find index of min in A[start:]
        min_id = find_minimum(A, start=barrier_id)
        #swap
        A[barrier_id], A[min_id] = (
                    A[min_id], A[barrier_id]
        )
```

Subroutine find_minimum

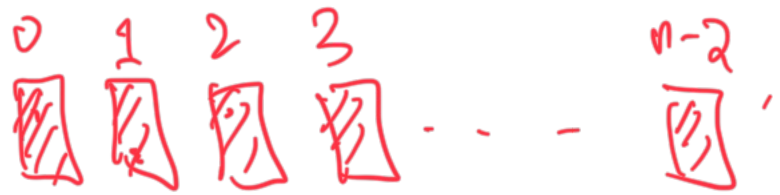
```
def find_minimum(A, start):
"""Finds index of minimum from [start, len(A)). Assumes non-empty."""
    n = len(A)
    min_value = A[start]
    min_id = start
    for i in range(start + 1, n):
        if A[i] < min_value:
            min_value = A[i]
            min_id = i
    return min_id
```

Note that instead of using this sub-routine, selection_sort can be written by using a nested loop.

Correctness

- How to convince us that this algorithm is correct?
  - Using loop invariants
    - Similar to the inductive idea mentioned earlier

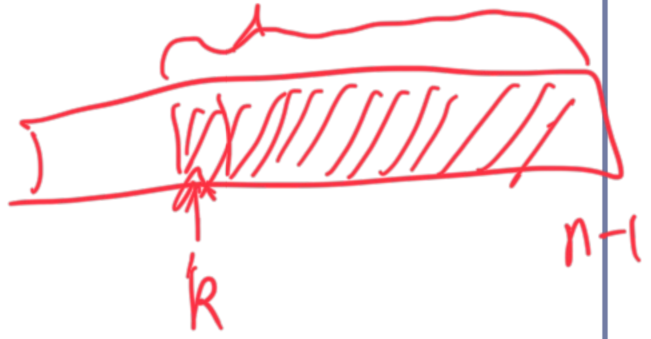
Correctness



- How to convince us that this algorithm is correct?
  - Using loop invariants
    - Similar to the inductive idea mentioned earlier
  - A loop invariant is a statement that holds at the end of each iteration
    - to show that it holds for each iteration, we first show it holds for the base case
    - then we argue that if it holds at the end of $(i- 1)$-th iteration, which is the beginning of the i-th iteration, then it will also hold at the end of i-th iteration.
  - Using appropriate loop invariants, we can then argue the algorithm is correct after all iterations.

Algorithm selection_sort

```
def selection_sort(A):
    n = len(A)
    if n <= 1:
        return
    for barrier_id in range(n-1):
        # find index of min in A[start:]
        min_id = find_minimum(A, start=barrier_id)
        #swap
        A[barrier_id], A[min_id] = (
                A[min_id], A[barrier_id]
        )
```



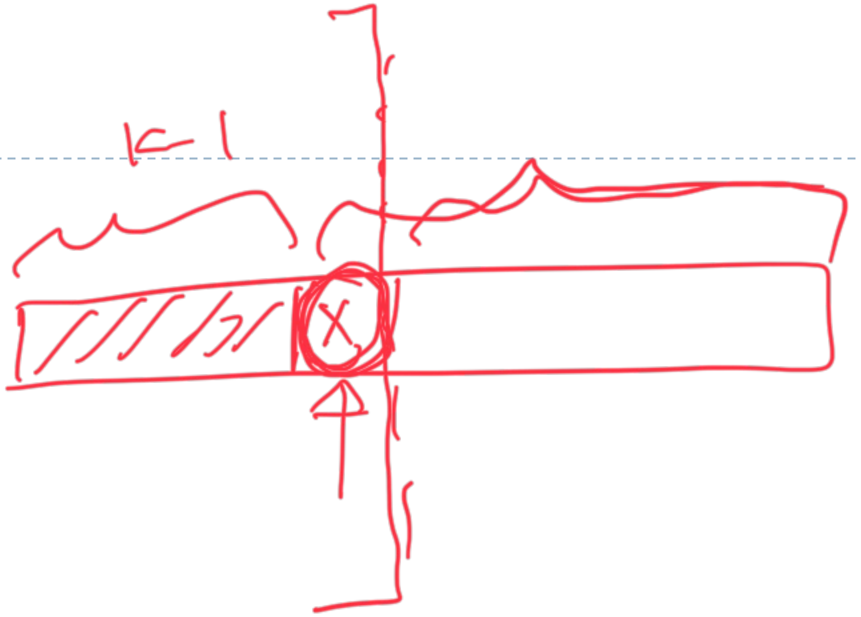
$\to c$

Loop invariants for selection_sort

- Loop invariant: after $\boxed{\text{k}}$ iterations,
  - The first k numbers in A are sorted, and are smaller than all the remainder $n - k$ numbers.
    - k = barrier_id+1 in the code

- If this statement holds for any k, then after $k = n - 1$ iterations, we will get a sorted array
  - as by the loop invariant, the first $n - 1$ numbers are sorted, and the last one is the largest, meaning that all n numbers are sorted.

- Base case:
  - $k = 0:$ loop invariant holds trivially



- Base case:
  - $k = 0$: loop invariant holds trivially
- Inductive step:
  - if it holds for $k - 1$
  - then, we identify the smallest from the remainder $\mathrm{n} - k + 1$ numbers, which must be the k -th smallest of the original array
  - so after this k -th iteration, the loop invariant holds for k.

- Base case:
  - $k = 0:$ loop invariant holds trivially

- Inductive step:
  - if it holds for $k-1$
  - then, we identify the smallest from the remainder $\mathrm{n}-k+1$ numbers, which must be the k -th smallest of the original array
  - so after this k -th iteration, the loop invariant holds for k.

- Thus the algorithm is correct in the end
  - i.e., it returns sorted array after $n-1$ iterations.

Time complexity

```
for i=1 to n
    for j=i to n
```

- The algorithm is essentially nested for-loops

$$T(n) \simeq \overset{\text{1st iteration}}{\boxed{cn}} + \overset{\text{2nd}}{\boxed{c(n-1)}} + \cdots + \boxed{c(n-i+1)} + \cdots + c \cdot 1$$

$$= c(\underline{1+2+3+\cdots+n}) = \Theta(n^2)$$

Time complexity

- The algorithm is essentially nested for-loops
  - $T(n) = cn + c(n-1) + c(n-2) + \ \dots c \cdot 1$
    $= \Theta(n^2)$

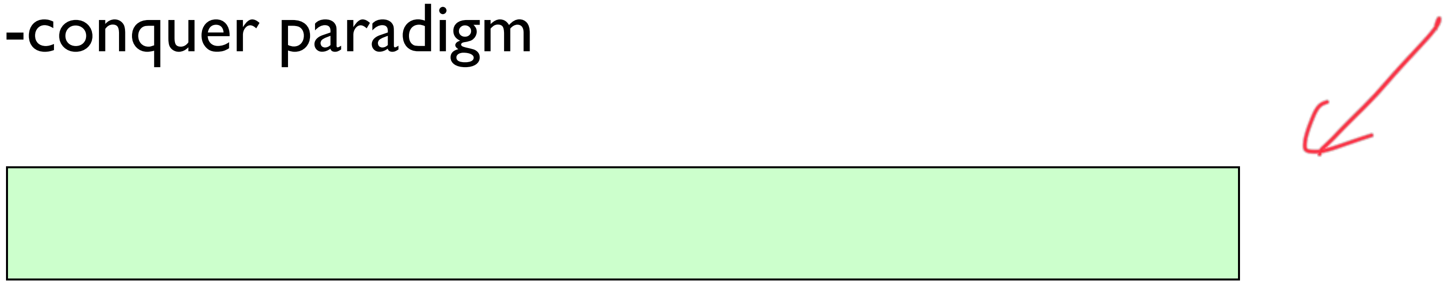
Part B: A more efficient sorting algorithm: Merge sort

MergeSort

- A faster sorting algorithm
  - has the optimal worst-case time complexity under the so-called comparison model.
- Use an idea called
  - divide-and-conquer to solve problems, which naturally leads to recursive algorithms.

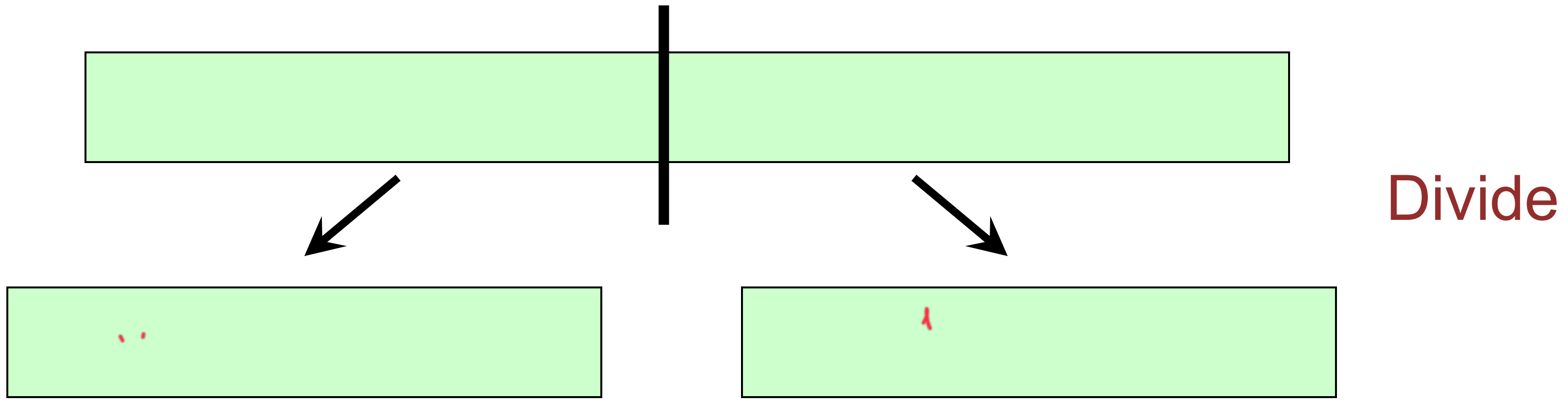
Merge sort

- Use divide-and-conquer paradigm



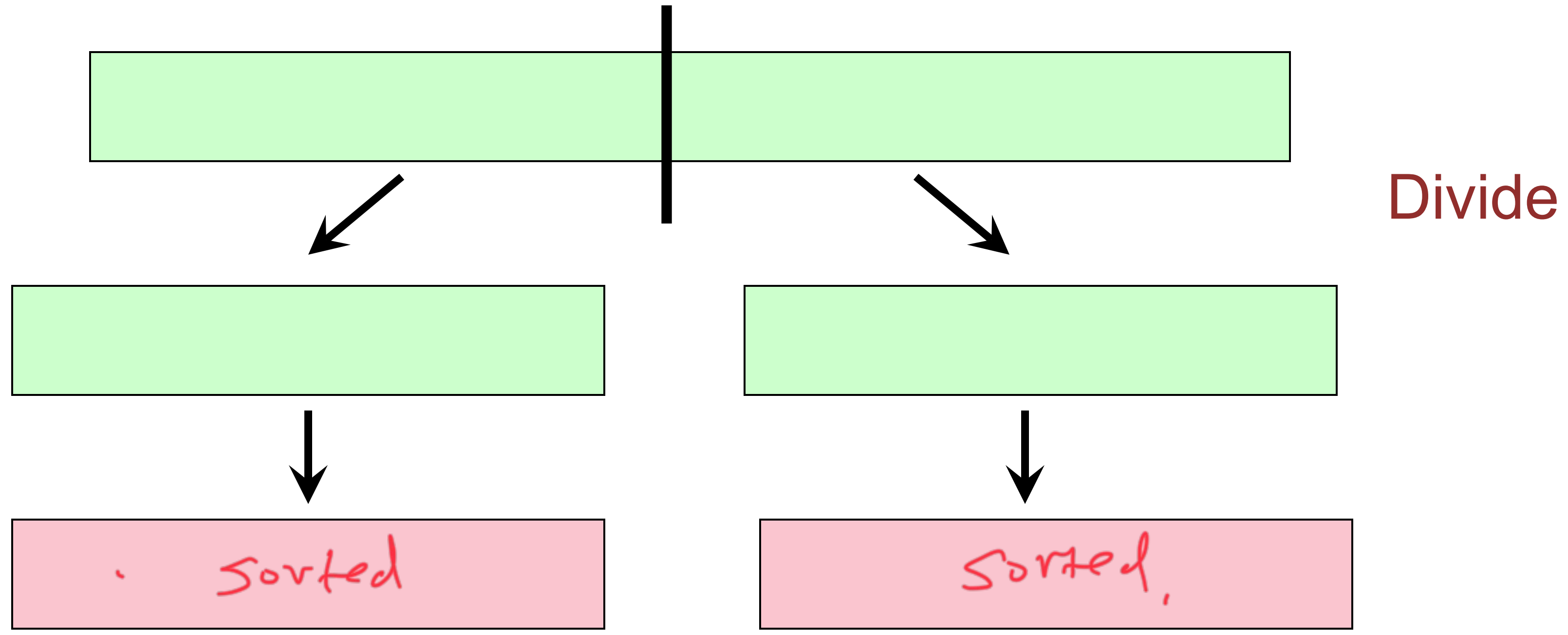
Merge sort

- Use divide-and-conquer paradigm



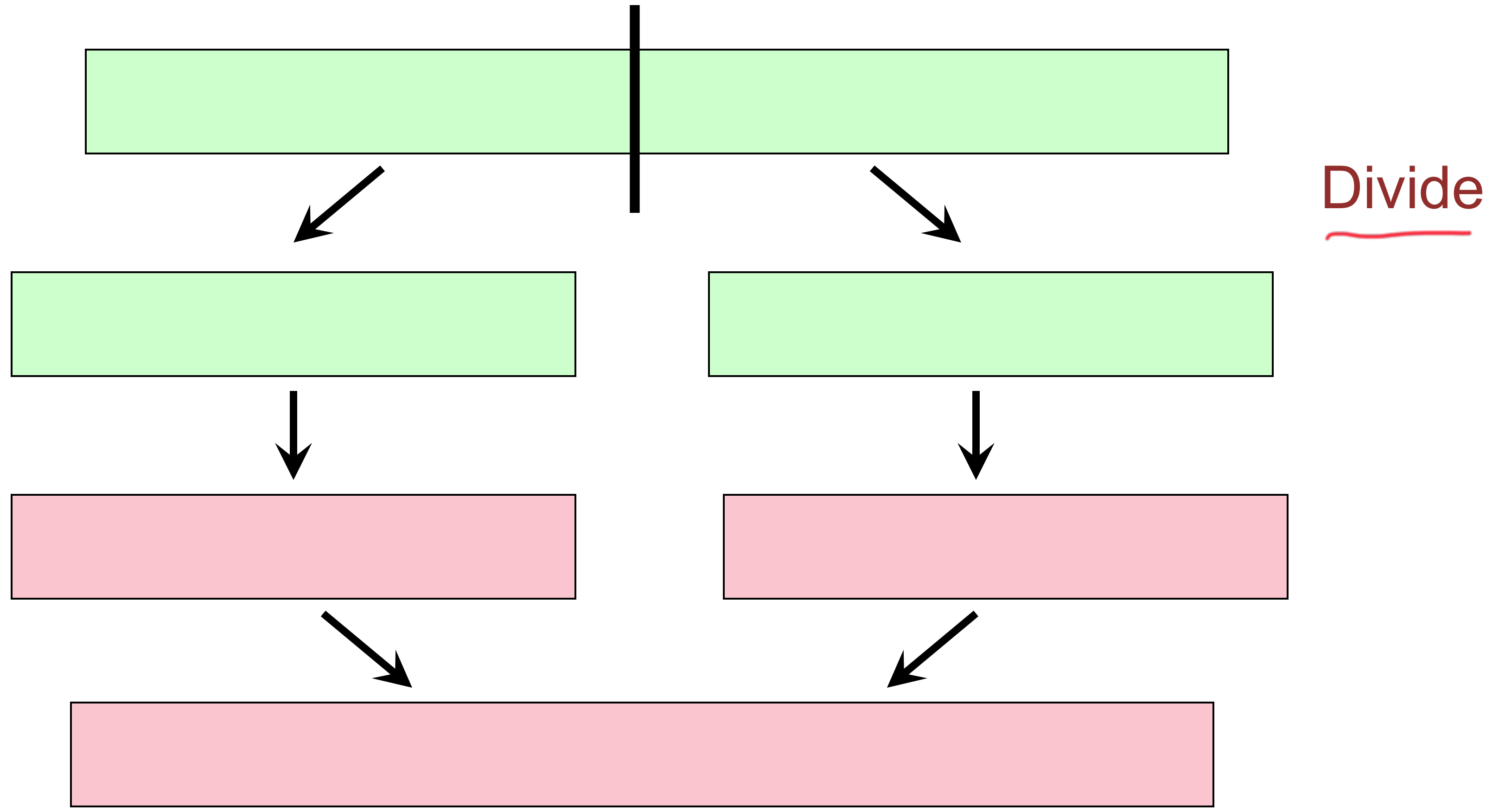
Merge sort

- Use divide-and-conquer paradigm



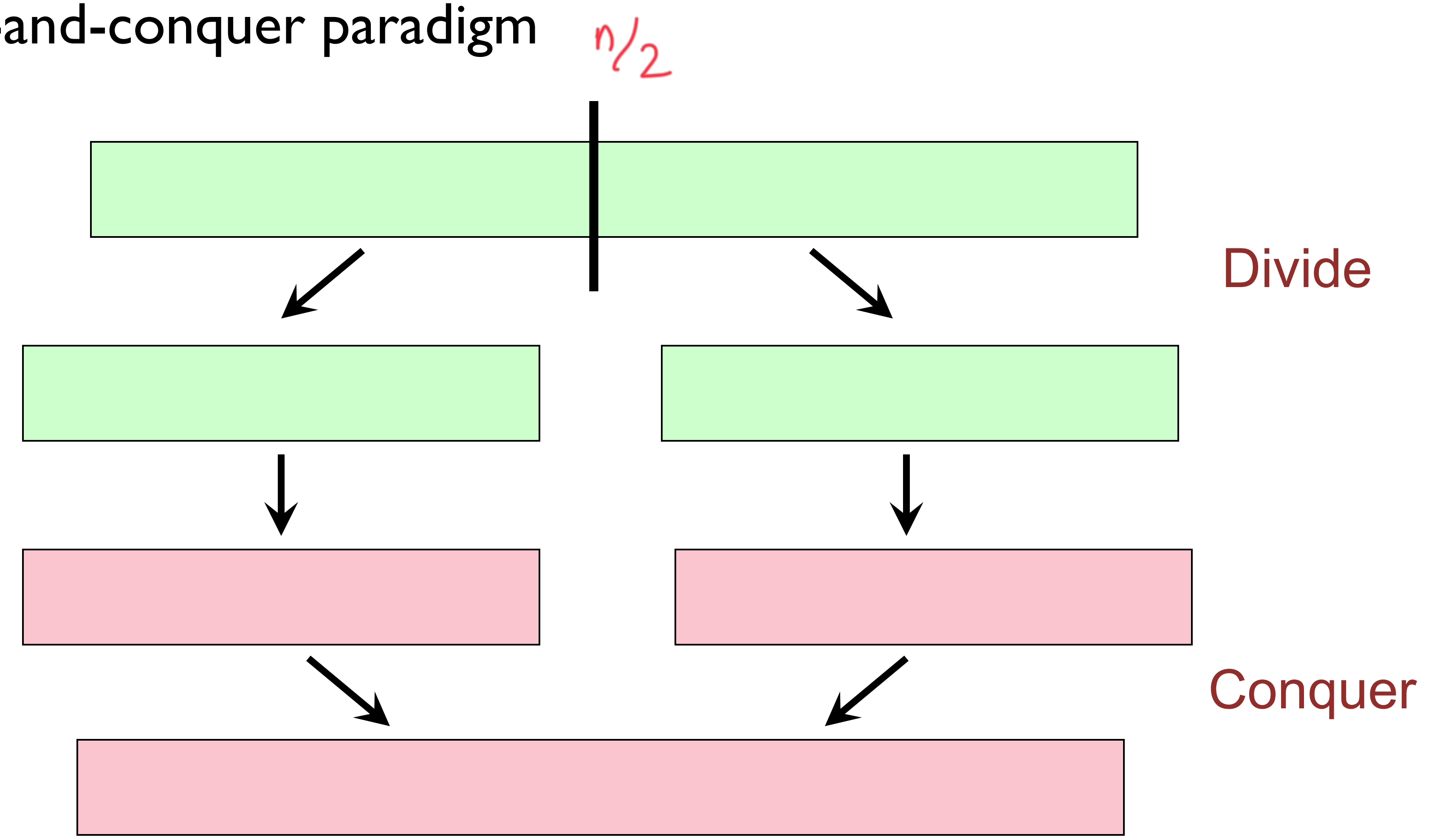
Merge sort

- Use divide-and-conquer paradigm



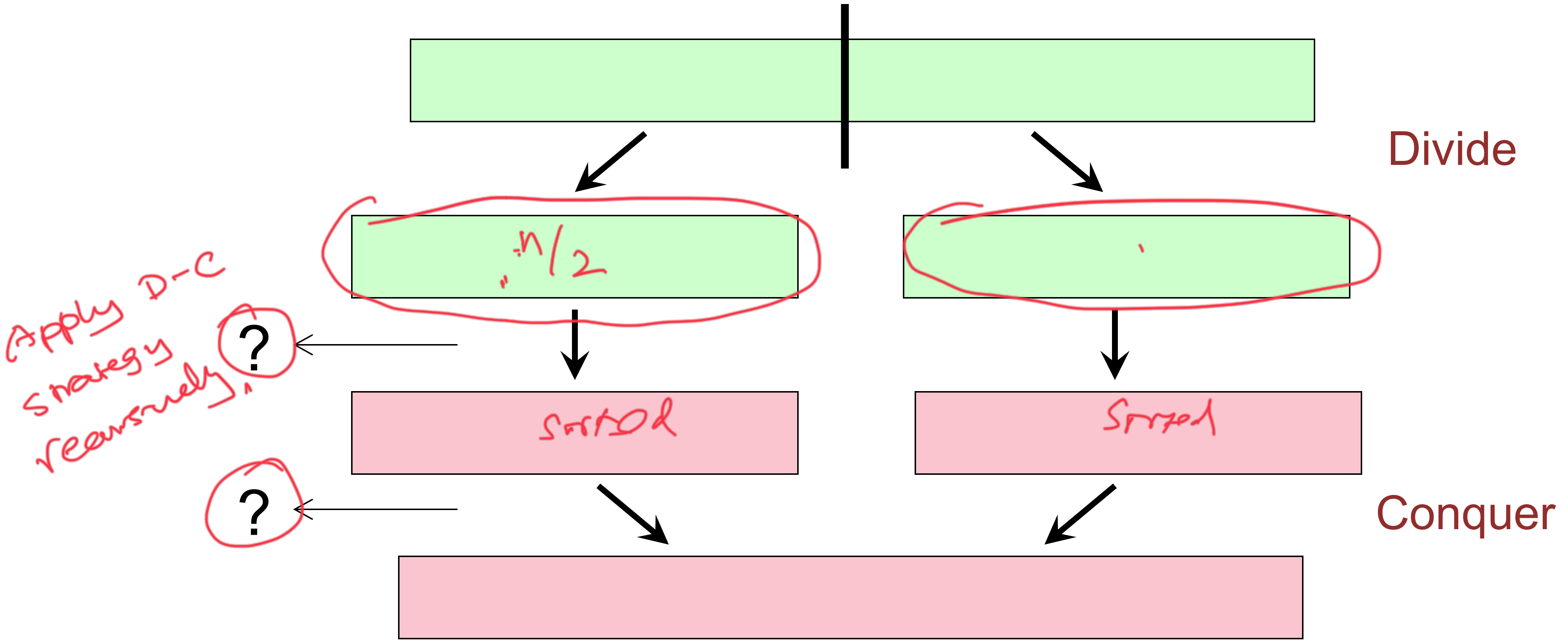
Merge sort

- Use divide-and-conquer paradigm



Merge sort

- Use divide-and-conquer paradigm



Pseudo-code

```
MergeSort ( A, l, r )
    if (l ≥ r)  return;
    mid  = ⌊(l + r) / 2 ⌋;
    LeftA = MergeSort ( A, l, mid );
    RightA = MergeSort ( A, mid+1, r );
    B = Merge (LeftA, RightA);
    return B;
```

- MergeSort (A, ℓ, r) sorts the subarray $A[\ell, r\ ]$
- Input: an array A of length n
- Output: a new sorted array
- Call: MergeSort$(A, 0, n - 1)$

Pseudo-code

```
MergeSort ( $A, l, r$ )
    if $(l \geq r)$  return;
    $mid = \lfloor (l + r) / 2 \rfloor$;
    $LeftA$ = MergeSort ( $A, l, mid$ );
    $RightA$ = MergeSort ( $A, mid+1, r$ );
    $B$ = Merge ($LeftA, RightA$);
    return $B$;
```

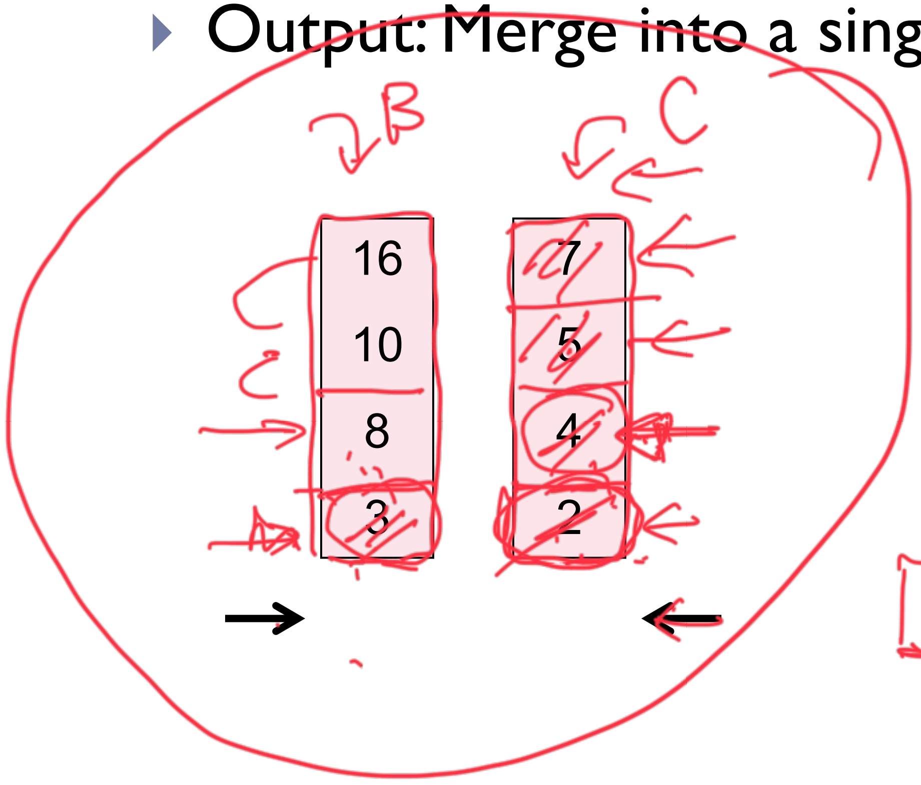
Use recursive calls!

This is NOT in-place sorting!

- MergeSort (A, ℓ, r) sorts the subarray $A[\ell, r]$
- Input: an array A of length n
- Output: a new sorted array
- Call: MergeSort$(A, 0, n-1)$

Conquer: Merge(B, C)

- Input: Given two sorted arrays B and C
- Output: Merge into a single sorted array



$s = |B|$

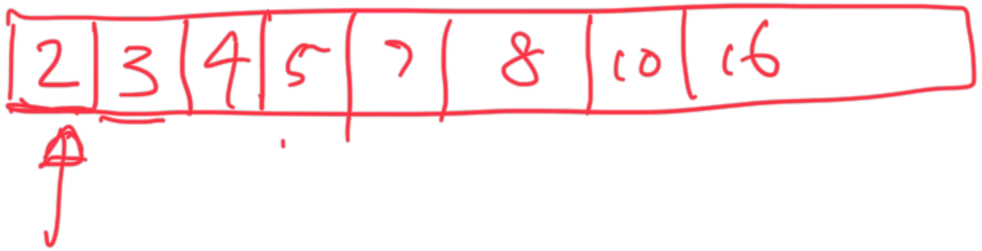
$\boxed{B} = \boxed{3, 8, 10, 16}$

$\boxed{C} = \boxed{2, 4, 5, 7}$

$t = |C|$

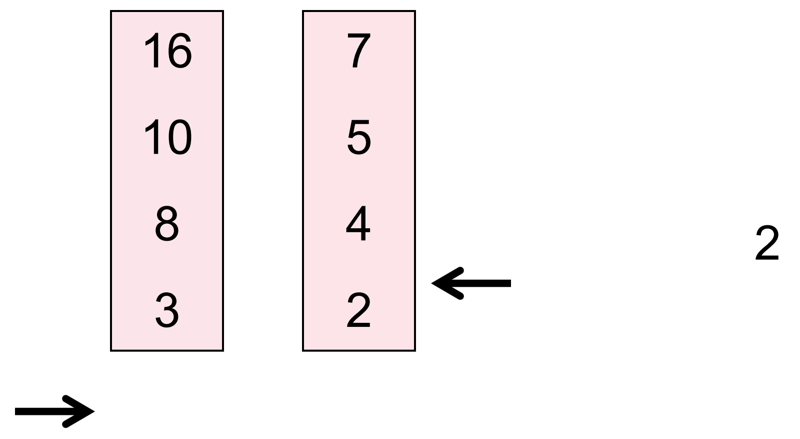
$\Theta(\underline{s+t})$

merged . sorted array. $= \Theta(|B| + |C|)$



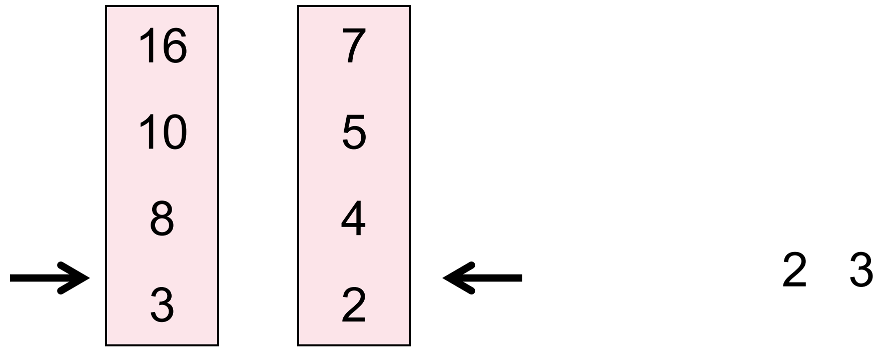
Conquer: Merge(B, C)

- Input: Given two sorted arrays B and C
- Output: Merge into a single sorted array



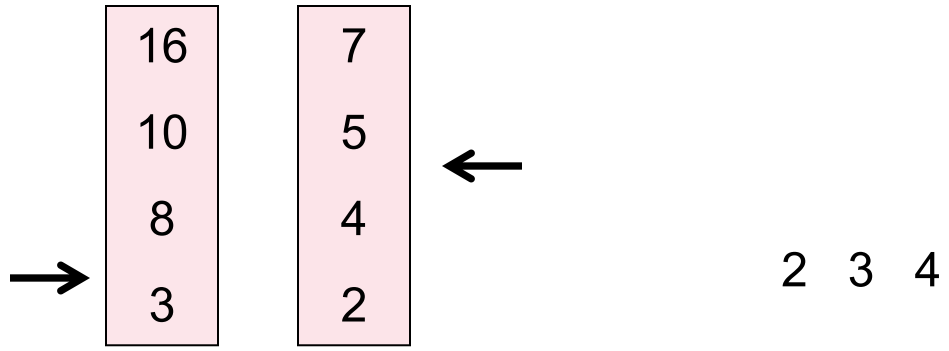
Conquer: Merge(B, C)

- Input: Given two sorted arrays B and C
- Output: Merge into a single sorted array



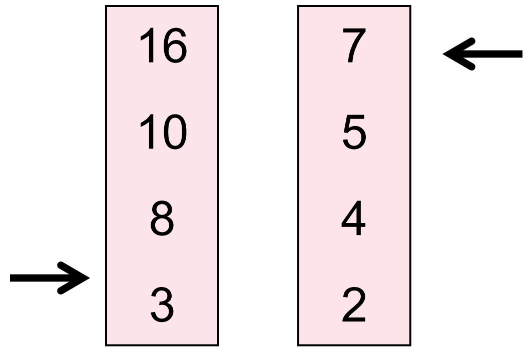
Conquer: Merge(B, C)

- Input: Given two sorted arrays B and C
- Output: Merge into a single sorted array



Conquer: Merge(B, C)

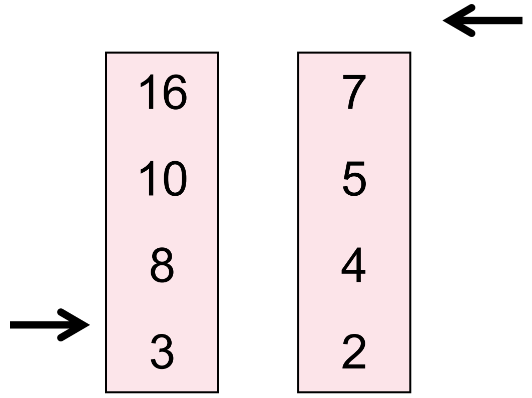
- Input: Given two sorted arrays B and C
- Output: Merge into a single sorted array



2 3 4 5

Conquer: Merge(B, C)

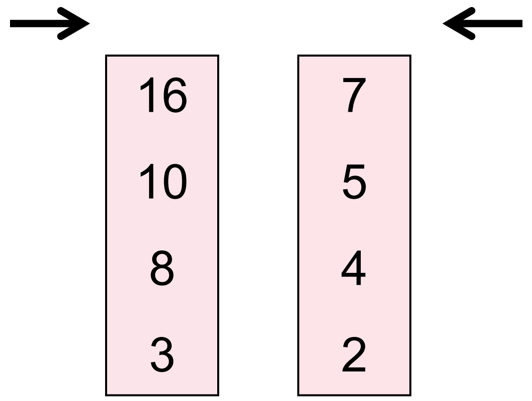
- Input: Given two sorted arrays B and C
- Output: Merge into a single sorted array



2 3 4 5 7

Conquer: Merge(B, C)

- Input: Given two sorted arrays B and C
- Output: Merge into a single sorted array



2 3 4 5 7 8 10 16

Pseudo-code

```
Merge ( $B, C$ )
    $n_b = len(B); n_c = len(C);\ \ n_o = n_b + n_c;$
    init $(outA, n_o)$;        //initialize $outA$ to be an array of size $n_o$
    $id_b = 0;\ \ id_c = 0;$
    for $(i = 0; i < n_o; i++)$ {
        if $(B[\underline{id_b}] > C[id_c])$ or $(id_b \geq n_b)$
                $outA[i] = C[\underline{id_c}];$
                $id_c++;$
        else
                $outA[i] = B[id_b];$
                $id_b++;$
    }
    return $outA$;
```

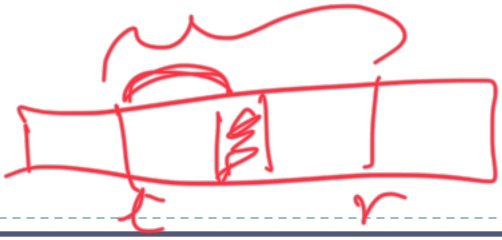
Time complexity analysis

- First: worst case time complexity for Merge(B, C)
  - Let $n_b = len(B); n_c = len(C)$

Time complexity analysis

- First: worst case time complexity for Merge(B, C)
  - Let $n_b = len(B); n_c = len(C)$
  - Then the time $T_{merge(B,C)} = \Theta(\underline{n_b + n_c})$

Pseudo-code



```
MergeSort ( A, l, r )
    if (l ≥ r)  return;
    mid = ⌊(l + r) / 2 ⌋;        ← c
    LeftA = MergeSort ( A, l, mid );        ← T(n/2)
    RightA = MergeSort ( A, mid+1, r );     ← T(n/2)
    B = Merge (LeftA, RightA);              ← Θ(n)
    return B;
```

$T(n)$: worst case time of MergeSort ~~for an~~ to sort a portion of array of size n.

$$T(n) = \boxed{c} + \underline{T(\tfrac{n}{2}) + T(\tfrac{n}{2})} + \boxed{c' \cdot n}$$

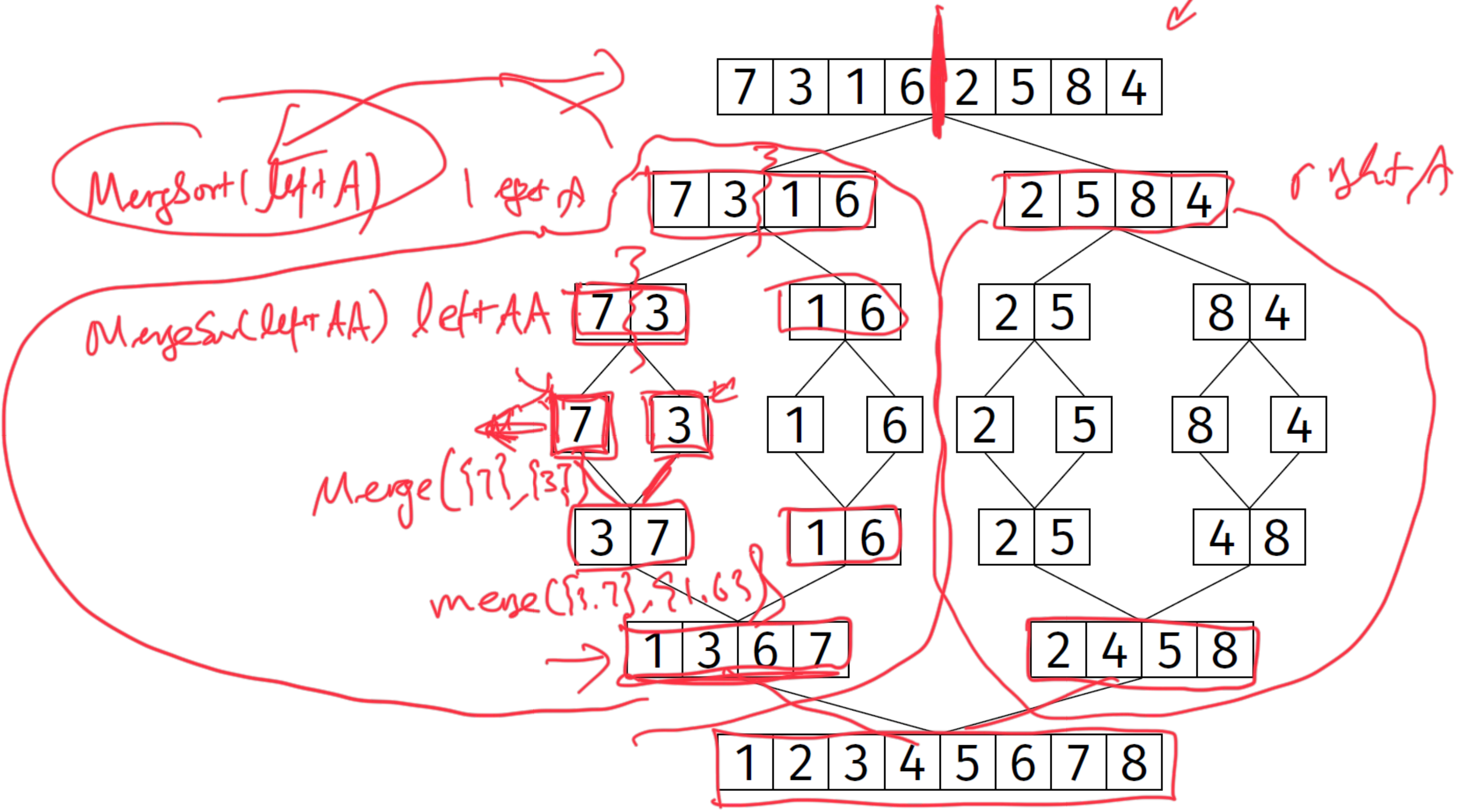
$$= 2T(\tfrac{n}{2}) + \boxed{c'' n}$$

- MergeSort (A, ℓ, r) sorts the subarray $A[\ell, r\,]$

- Input: an array A of length n
- Output: a new sorted array
- Call: MergeSort$(A, 0, n-1)$

$$c + c' n = \underline{\Theta(n)}$$

Example



MergeSort A
7 3 1 6 2 5 8 4
MergeSort(left A)
left A
7 3 1 6
2 5 8 4
right A
MergeSort(left AA)
left AA
7 3
1 6
2 5
8 4
7
3
1
6
2
5
8
4
Merge({7},{3})
3 7
1 6
2 5
4 8
merge({3,7},{1,6})
1 3 6 7
2 4 5 8
1 2 3 4 5 6 7 8

Correctness

- Recall for a recursive algorithm:
  - (1) Make sure algorithm works in the base case.
  - (2) Check that all recursive calls are on smaller problems, and that it terminates
  - (3) Assuming that the recursive calls work, does the whole algorithm work?

- (1) Base case:
  - Portion of array to be inspected is of size at most 1
  - Obviously already sorted!
- (2) Work on smaller subproblems? Terminate?
  - Yes
- (3) If recursive calls return correct output, does the entire algorithm works ?
  - Yes, as long as Merge (B, C) is correct.

Pseudo-code

```
MergeSort ( $A, l, r$ )
    if $(l \geq r)$ return;
    $mid = \lfloor (l + r) / 2 \rfloor$;
    $LeftA$ = MergeSort ( $A, l, mid$ ); ← ✓
    $RightA$ = MergeSort ( $A, mid+1, r$ ); ← ✓
    $B$ = Merge ($LeftA, RightA$); ← ✓
    return $B$;
```

- $T(n)$:
  - the worst case time complexity of MergeSort performed on a subarray of size n

Pseudo-code

```
MergeSort ( $A, l, r$ )
    if $(l \geq r)$ return;
    $mid = \lfloor (l + r) / 2 \rfloor$;
    $LeftA$ = MergeSort ( $A, l, mid$ );
    $RightA$ = MergeSort ( $A, mid+1, r$ );
    $B$ = Merge ($LeftA, RightA$);
    return $B$;
```

- $\underline{T(n)}$:
  - the $\underline{\text{worst case time complexity of MergeSort}}$ performed on a subarray of size n

- $$\underline{T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + cn = 2T\left(\frac{n}{2}\right) + cn}$$

$T(n') = 2T\left(\frac{n'}{2}\right) + n'$

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

Solving Recurrence relations

- $T(n) = 2T\left(\frac{n}{2}\right) + cn$

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

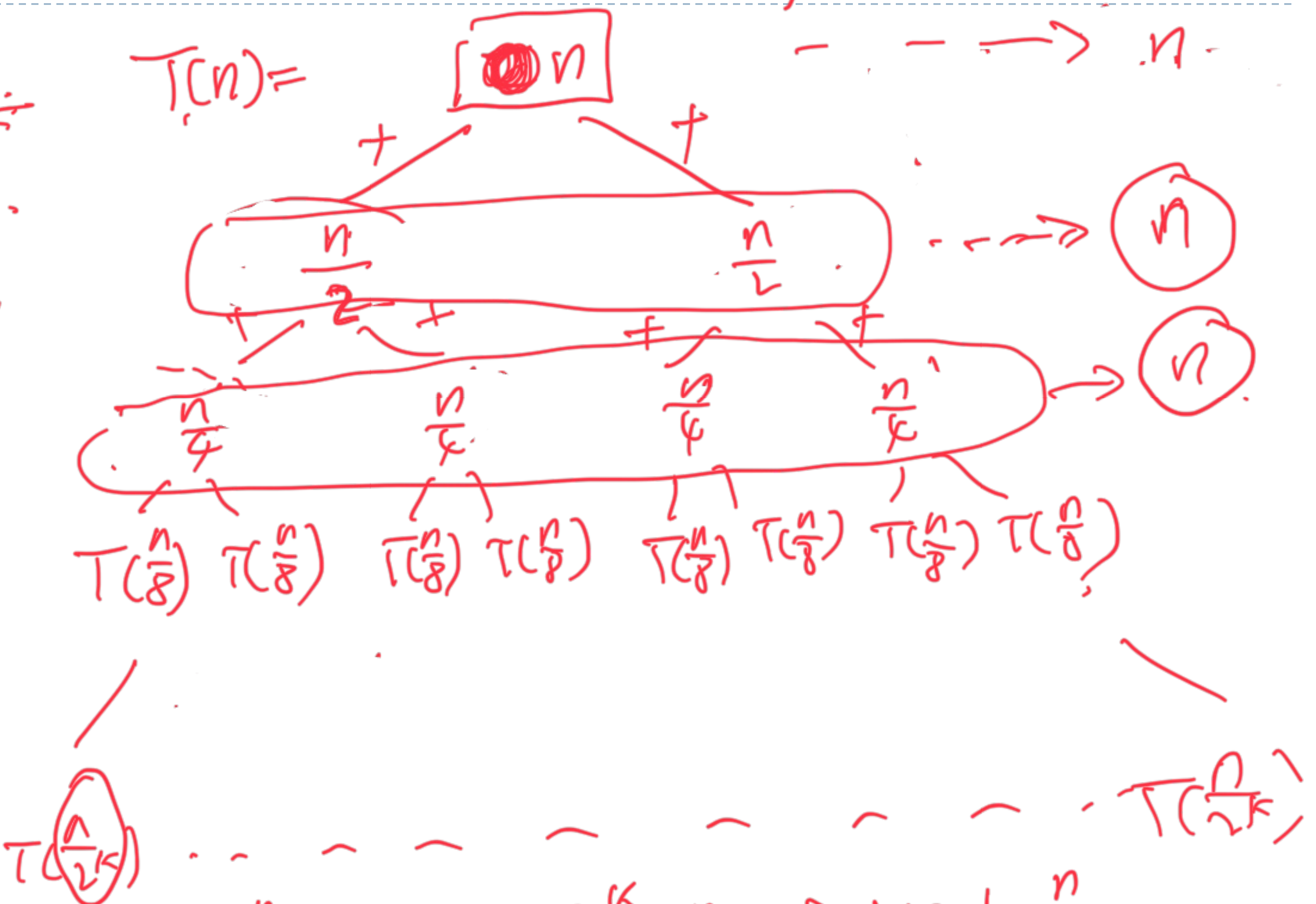
$= 4T\left(\frac{n}{4}\right) + n + n$

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

$= 8T\left(\frac{n}{8}\right) + n + n + n$

$\vdots$

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



Stop when $\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$

- $= n \cdot T(1) + n \cdot \log_2 n = \Theta(n \lg n)$

Solving Recurrence

binary search

$$\underline{T(n)} = T\left(\frac{n}{2}\right) + \underline{c}$$

$$= \Theta(\lg n)$$

- One way is via the following strategy:
  - 1. "Unroll" several times to find a pattern.
  - 2. Write general formula for *k*th unroll.
  - 3. Solve for # of unrolls needed to reach base case.
  - 4. Plug this number into general formula.

$$\underline{T(n)} = T\left(\frac{n}{2}\right) + \underline{cn}.$$

$$= \Theta(n)$$

Solving Recurrence relations

- $T(n) = 2T\left(\frac{n}{2}\right) + cn$

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

$$= 4\left(2T\left(\frac{n}{8}\right) + \frac{cn}{4}\right) + 2cn = 8T\left(\frac{n}{8}\right) + 3cn$$

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

Terminates when $\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$

Thus: $T(n) = 2^k T\left(\frac{n}{2^k}\right) + kcn = n\,T(1) + cn\log_2 n$

$$= \Theta(n \lg n)$$

Sorting problem

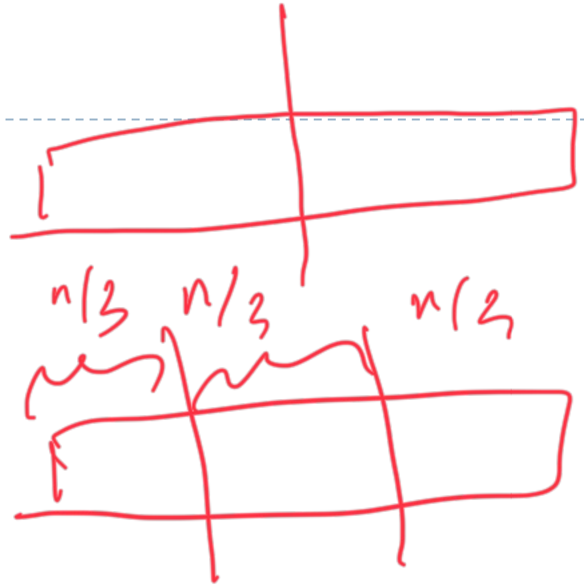
- The sorting problem can be solved in $\Theta(n \lg n)$ worst-case time.
- It has the optimal asymptotic time complexity
  - if we assume the so-called comparison model.
  - So under the comparison model, we cannot have an asymptotically faster algorithm than the merge sort.
- This algorithm is $\underline{\text{not in-place}}$.
  - in practice, quicksort tends to be rather popular

Part C:
Three-way MergeSort, and more on solving recurrences

Another MergeSort

```
MergeSort ( A, l, r )  // sorting subarray $A[\ell, r]$
    if ( $l \geq r$ ) return;
    $m_1 = l + (r - l) / 3$;
    $m_2 = l + 2(r - l) / 3$;
    A1 = MergeSort ( A, l, $m_1$ );
    A2 = MergeSort ( A, $m_1$ +1, $m_2$ );
    A3 = MergeSort ( A, $m_2$ +1, r );
    Merge (A1,A2,A3);
```

linear.



$$T(n) = 3T\left(\frac{n}{3}\right) + cn$$

- Recurrence relation for MergeSort(A,ℓ, r) when $r - \ell + 1 = n$

Another MergeSort

```
MergeSort ( A, l, r )  // sorting subarray $A[\ell, r]$
    if ( $l \geq r$ )  return;
    $m_1 = l + (r - l) / 3$;
    $m_2 = l + 2(r - l) / 3$;
    A1 = MergeSort ( A, l, $m_1$ );
    A2 = MergeSort ( A, $m_1 + 1$, $m_2$ );
    A3 = MergeSort ( A, $m_2 + 1$, r );
    Merge (A1,A2,A3);
```

$\sqrt{n}$
$\sqrt{n}$
$\sqrt{n}$

- Recurrence relation for MergeSort(A,ℓ, r) when $r - \ell + 1 = n$
  - $T(n) = 3T(\frac{n}{3}) + cn$

Solving recurrence

- $T(n) = 3T(\frac{n}{3}) + cn$

$= 3\left(3T(\frac{n}{3^2}) + c\cdot\frac{n}{3}\right) + cn = 3^2 T(\frac{n}{3^2}) + \underline{cn + cn}$

$= 3^2\left(3T(\frac{n}{3^3}) + c\cdot\frac{n}{3^2}\right) + cn + cn = 3^3 T(\frac{n}{3^3}) + cn + cn + cn$

$= 3^k T(\frac{n}{3^k}) + cnk$

stop when $\frac{n}{3^k} = 1 \Rightarrow 3^k = n$

$\Rightarrow k = \log_3^n$

$= \underbrace{n\cdot T(1)}_{\Theta(1)} + cn\cdot\log_3^n = \Theta(n \lg n)$

Another example

- $T(n) = T(\frac{n}{3}) + \cancel{c}n$

$= \left(T(\frac{n}{9}) + \frac{n}{3}\right) + \cancel{c}n$

$= T(\frac{n}{3^3}) + \frac{n}{9} + \frac{n}{3} + n$

$\vdots$

$= T(\frac{n}{3^k}) + \underline{n + \frac{n}{3} + \frac{n}{3^2} + \dots + \frac{n}{3^{k-1}}}$

stop $\frac{n}{3^k} = 1 \Rightarrow 3^k = n \Rightarrow k = \log_3^n$

$= T(1) + n(\underbrace{1 + \frac{1}{3} + \frac{1}{3^2} + \dots + \frac{1}{3^{k-1}}}_{= \Theta(1)})$

$T(n) = n + 1 + \frac{n}{3} + 1 + \frac{n}{3^2} + 1 + \frac{n}{3^3} \vdots \frac{n}{3^{k-1}} + T(\frac{n}{3^k})$ $T(1)$

$\log_3^n$

$= T(1)\,\Theta(1) + n \cdot \Theta(1) = \Theta(n)$

Part D:
The Movie problem revisited

Recall

- The Movie problem
  - Input: Given a list of length of movies available, stored in array $movies$, and a flight duration D
  - Output: Return two movies whose total length = D; None otherwise.

- Previously,
  - we gave an algorithm with worst-case time complexity $\Theta(n^2)$
- Can we do better?
  - Yes, if we first sort the input array of movie times. $\rightarrow \Theta(n \lg n)$
- Example: $D = 175$
  - Flight time: ~~170 = D~~
  - Movie times (sorted): 60, 80, 95, 110, 130

min min max max

Code

```
def optimize_entertainment(times, target):
    n = len(times)
    MergeSort(times, 0, n-1)
    shortest = 0
    longest = n - 1
    for i in range(n - 1):
        total_time = times[shortest] + times[longest]
        if total_time == target:
            return (shortest, longest)
        elif total_time < target:
            shortest += 1
        else: # total_time > target
            longest -= 1
    return None
```

$\leftarrow \Theta(n \lg n)$

$\Theta(n)$

$\Theta(n \lg n)$

Code

```
def optimize_entertainment(times, target):
    n = len(times)
    MergeSort(times, 0, n-1)
    shortest = 0
    longest = n - 1
    for i in range(n - 1):
        total_time = times[shortest] + times[longest]
        if total_time == target:
            return (shortest, longest)
        elif total_time < target:
            shortest += 1
        else: # total_time > target
            longest -= 1
    return None
```

Worst-case time complexity:

$$T(n) = \Theta(n \lg n) + \Theta(n)$$
$$= \Theta(n \lg n)$$

Take-home messages

- Sorting can be done in $\Theta(n \lg n)$ time
- More examples on solving recurrences
- Using sorted structures can sometimes help solve other problems more efficiently
  - e.g, binary search, and the movie problems.

FIN