Divide and Conquer

Lecture 13

by Marina Barsky

https://www.khanacademy.org/computing/computer-science/algorithms/sorting-algorithms/a/sorting

Warmup: Recursive array max

- Design a recursive algorithm called *findArrayMax* that returns the maximum value in an array

- Formally:

```
Input: array A of length n >= 1
Output: max value of A
```

- Examples:

```
Input: A = {4, 13, 21, 5, 2})
Output: 21

Input: A ={-1, -3, -8, -5, -12}
Output: -1

Input: A = {5}
Output: 5
```

Recursive array max: stop condition

```
Algorithm findArrayMax(A):
    input: a NONEMPTY array, A
    output: A's maximum element
```

Stop condition?

A

```
if A.length == 1
    return 0
```

B

```
if A.length == 1
    return 1
```

C

```
if A.length == 1
    return A[0]
```

- A
- B
- C
- None of the above
- More than one is correct

Recursive array max: recursive step

```
Algorithm findArrayMax(A):
    if  A.length == 1:
        return A[0]
    *
```

A *

```
if  A[0] < A[1]:
    A = A - A[0]
    return findArrayMax(A)
else:
    A = A - A[1]
    return findArrayMax(A)
```

B *

```
if  A[0] < A[1]:
    A = A - A[1]
    return findArrayMax(A)
else:
    A = A - A[0]
    return findArrayMax(A)
```

- A
- B
- None of the above

Recursive array max: solution

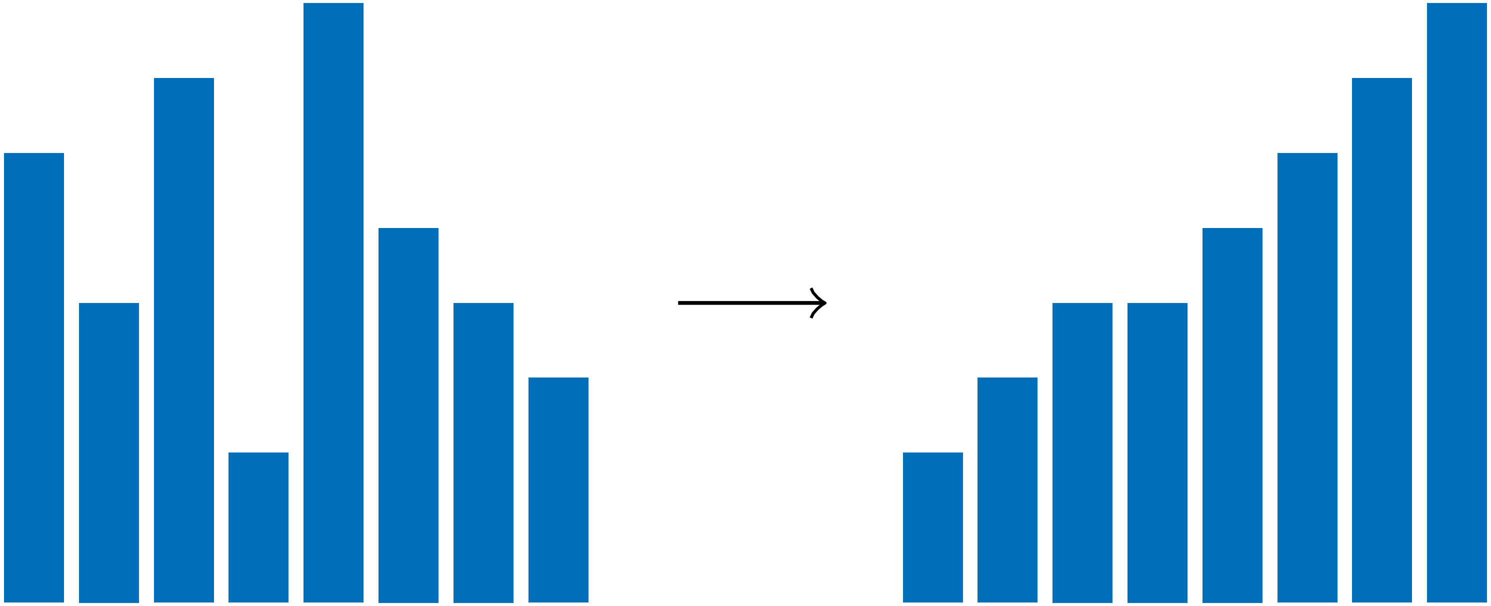
```
Algorithm findArrayMax(A):
   input: a NONEMPTY array, A
      output: A's maximum element
   if  A.length == 1:
      return A[0]

   if  A[0]<A[1]:
      A = A - A[0]
      return findArrayMax(A)
   else:
      A = A - A[1]
      return findArrayMax(A)
```

Sorting Problem

Input: Sequence A of n elements

Output: Permutation A' of elements in A such that all elements of A' are in non-decreasing order.

Sorting Problem



https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html

https://www.toptal.com/developers/sorting-algorithms

Why Sorting?

- Sorting data is an important step of many efficient algorithms

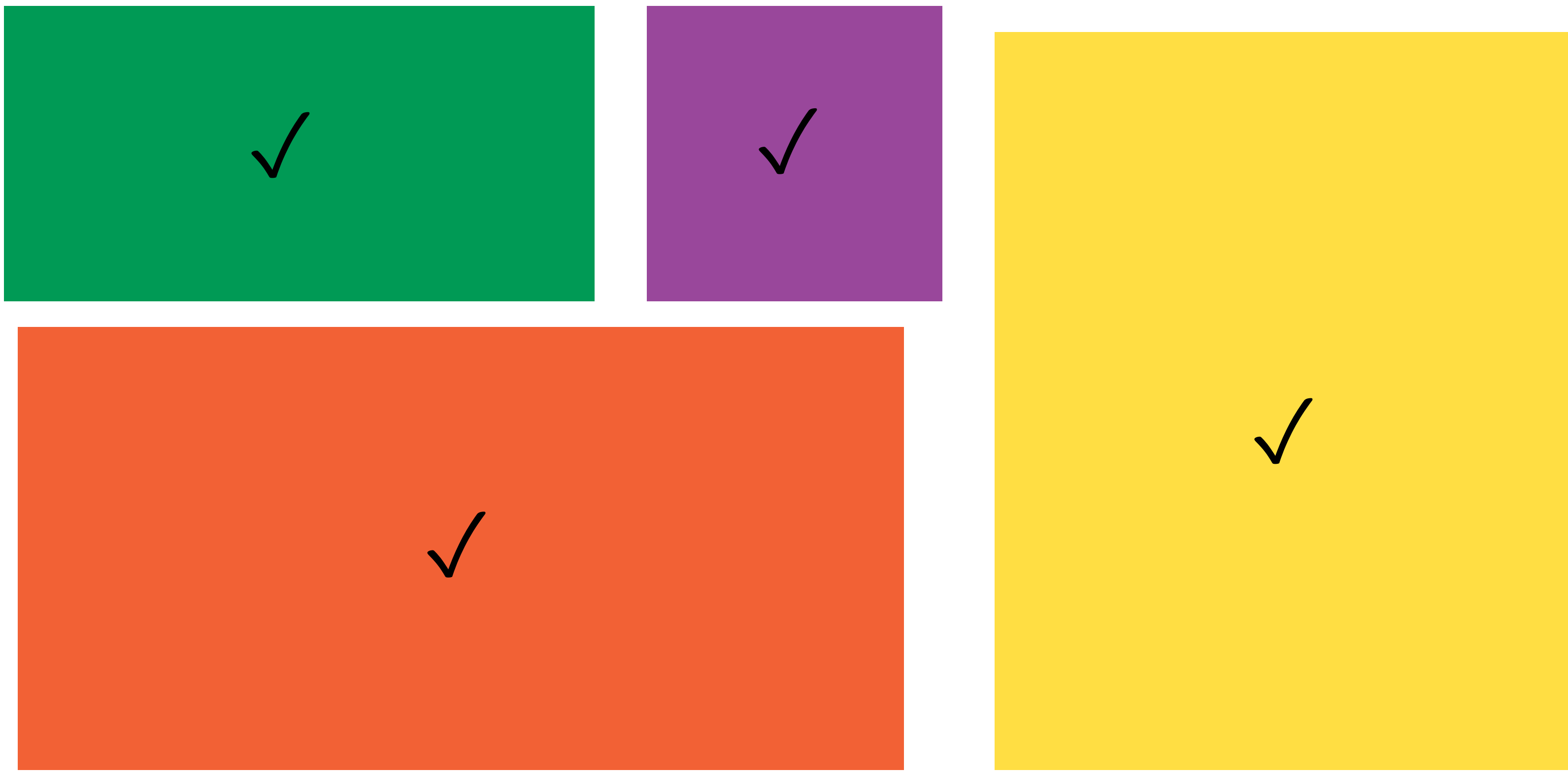
- Sorted data allows for more efficient queries (binary search)

We will use Divide-and-conquer technique

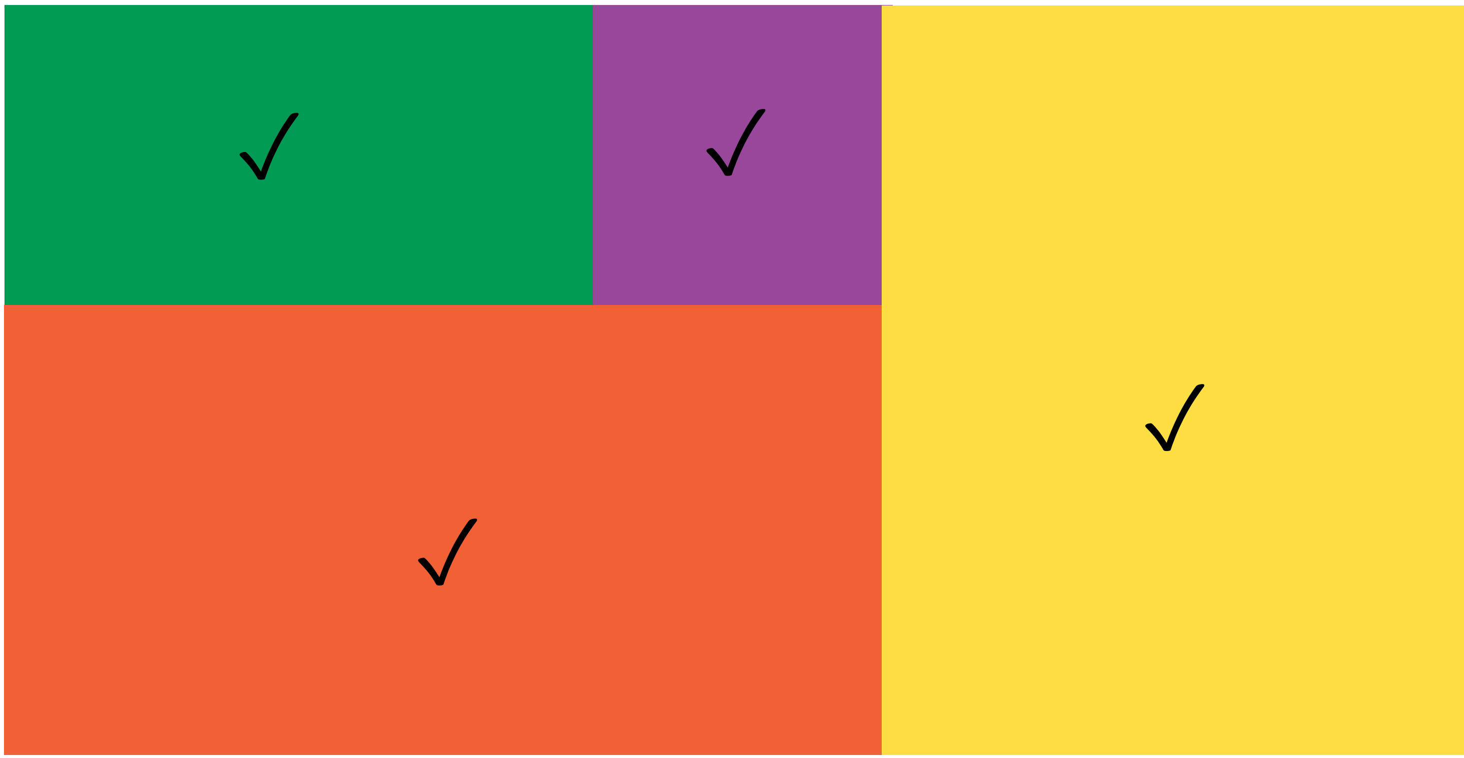
1. Break into *non-overlapping* subproblems *of the same type*
2. Solve subproblems
3. Combine results

Divide: break apart

Conquer: solve subproblems



Combine



✓

Idea: merge sort

7	2	5	3	7	13	1	6

split the array into two halves

7	2	5	3

7	13	1	6

Idea: merge sort

7	2	5	3	7	13	1	6

split the array into two halves

7	2	5	3

7	13	1	6

sort the halves recursively

2	3	5	7

1	6	7	13

Idea: merge sort

7	2	5	3	7	13	1	6

split the array into two halves

7	2	5	3

7	13	1	6

sort the halves recursively

2	3	5	7

1	6	7	13

merge the sorted halves into one array

1	2	3	5	6	7	7	13

Algorithm MergeSort (array $A[1...n]$)

if $n = 1$: return A ***# already sorted***

$m \leftarrow \lfloor n/2 \rfloor$

$B \leftarrow \text{MergeSort}(A[1 \ldots m])$

$C \leftarrow \text{MergeSort}(A[m+1 \ldots n])$

$A' \leftarrow \text{merge}(B, C)$

return A'

Merging Two Sorted Arrays

Algorithm Merge($B[1...\ p]$, $C\ [1...\ q]$)

 # *B* and *C* are sorted

D ← empty array of size $p + q$

while *B* and *C* are both non-empty:

    b ← the first element of *B*

    c ← the first element of *C*

    if $b \leq c$:

        move *b* from *B* to the end of *D*

    else:

        move *c* from *C* to the end of *D*

move what remains of *B* or *C* to the end of *D*

return *D*

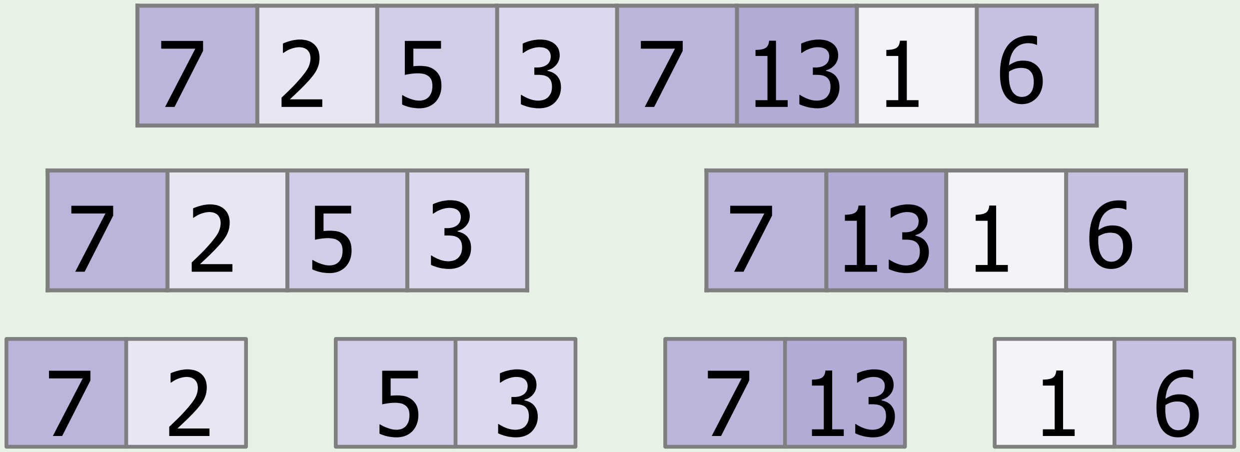
Merge sort: example

7	2	5	3	7	13	1	6

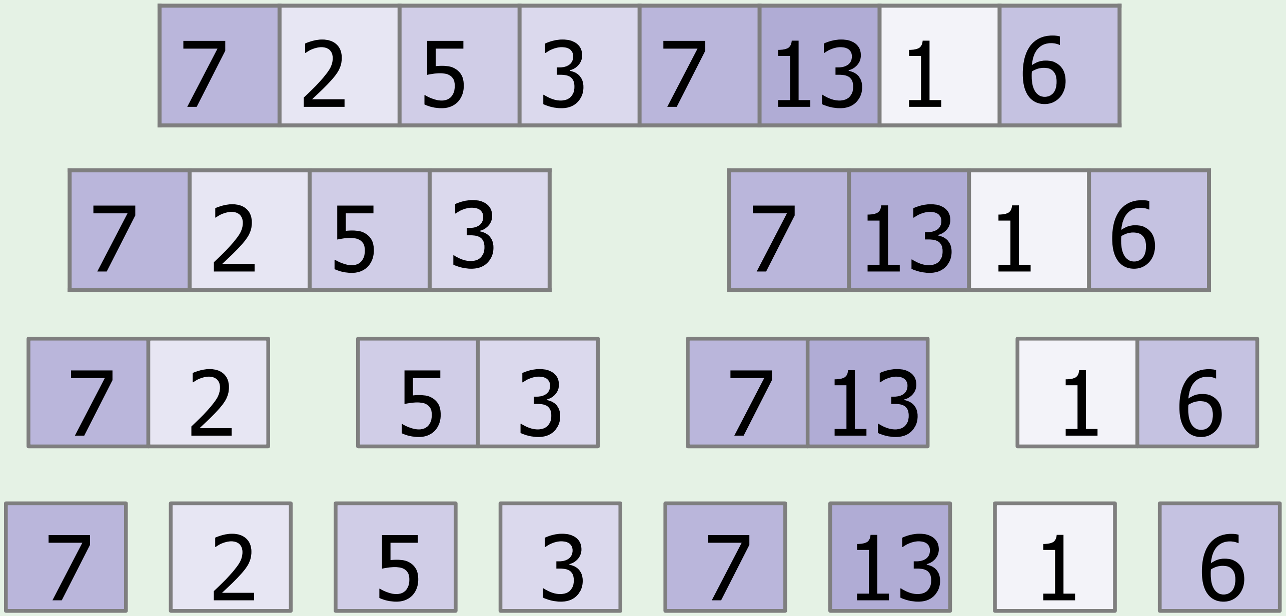
7	2	5	3

7	13	1	6

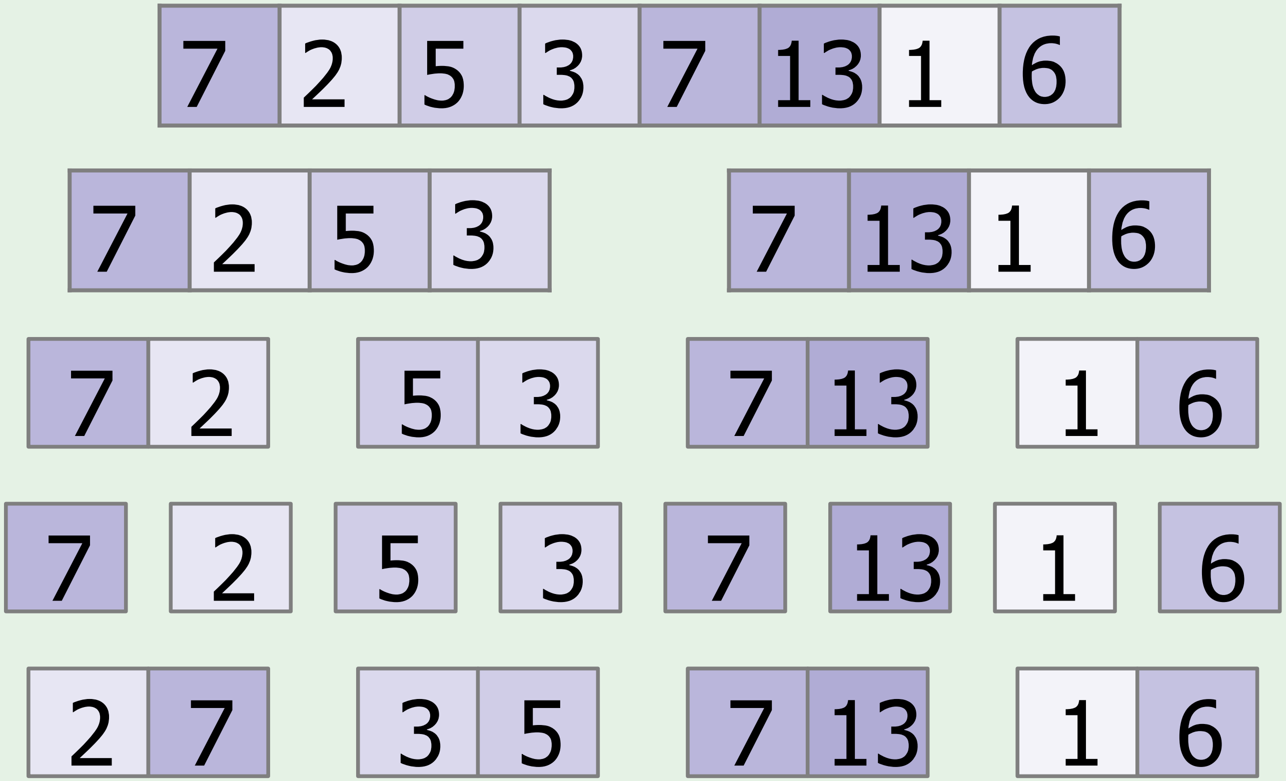
Merge sort: example



Merge sort: example

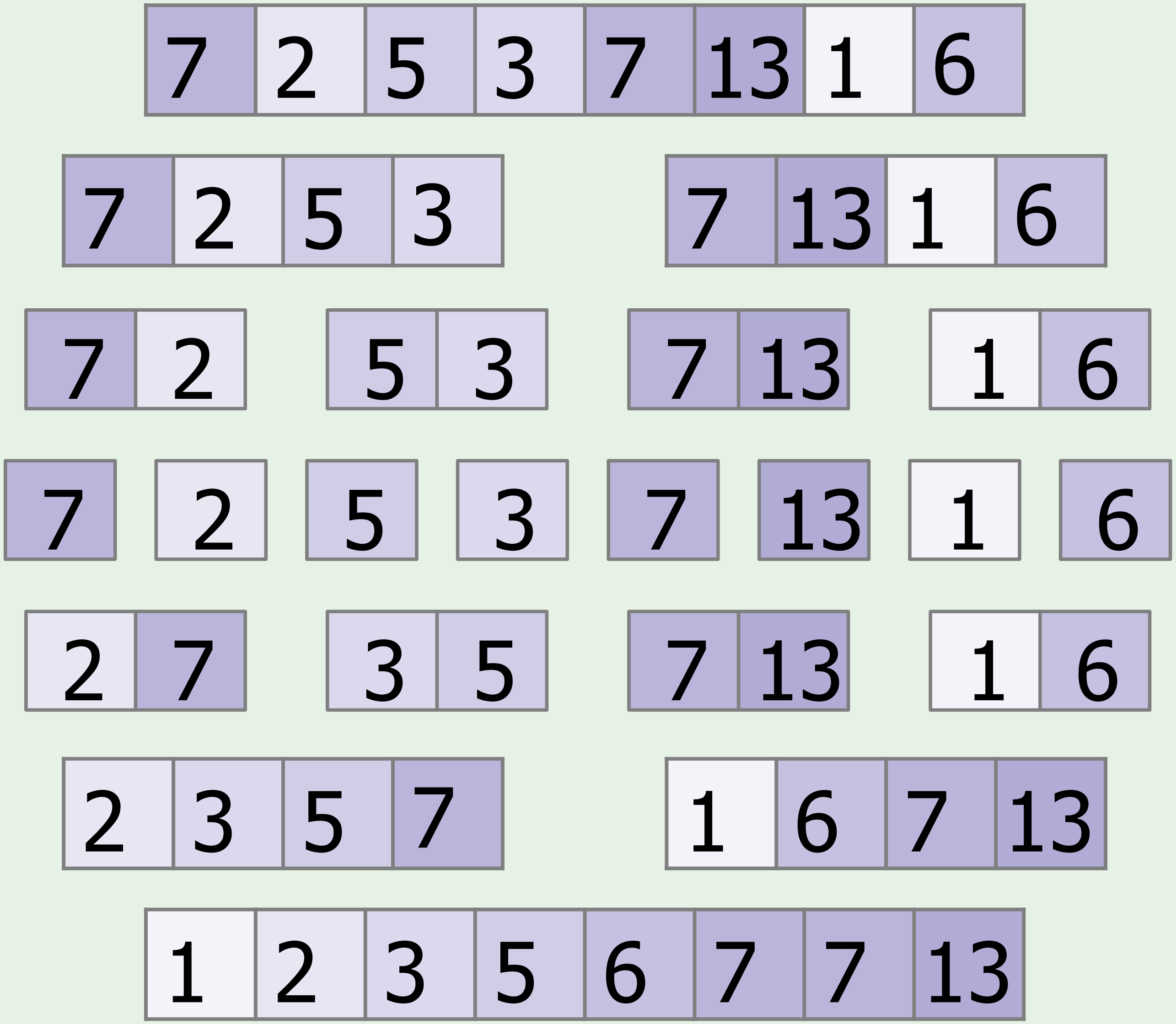


Merge sort: example

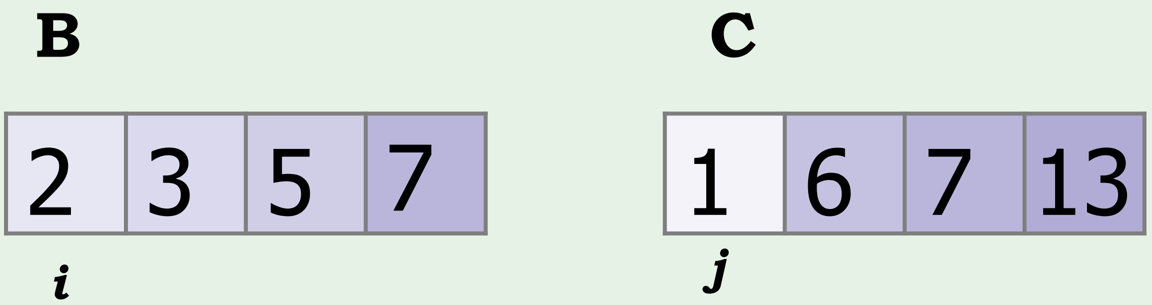


Merge sort: example

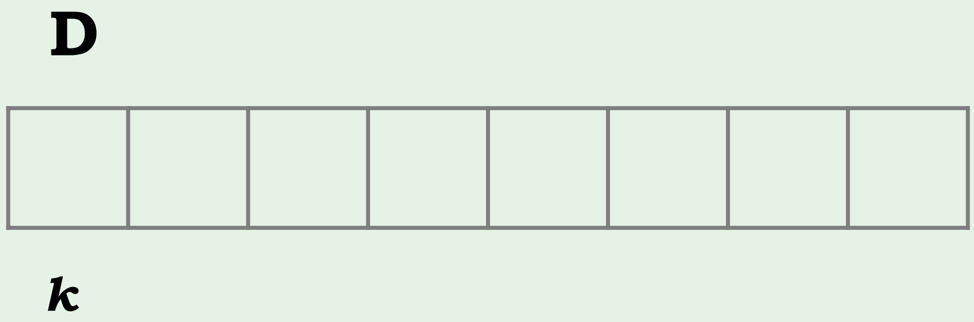
Merge sort: example



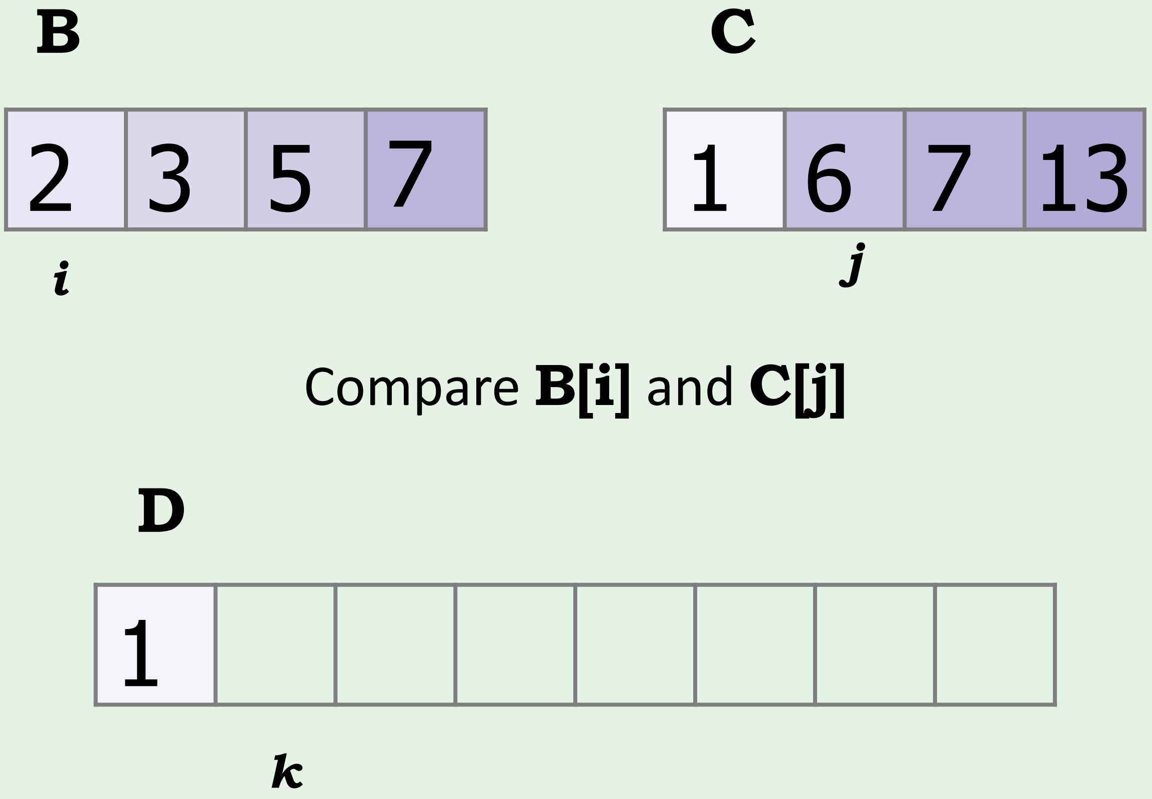
Merge: example



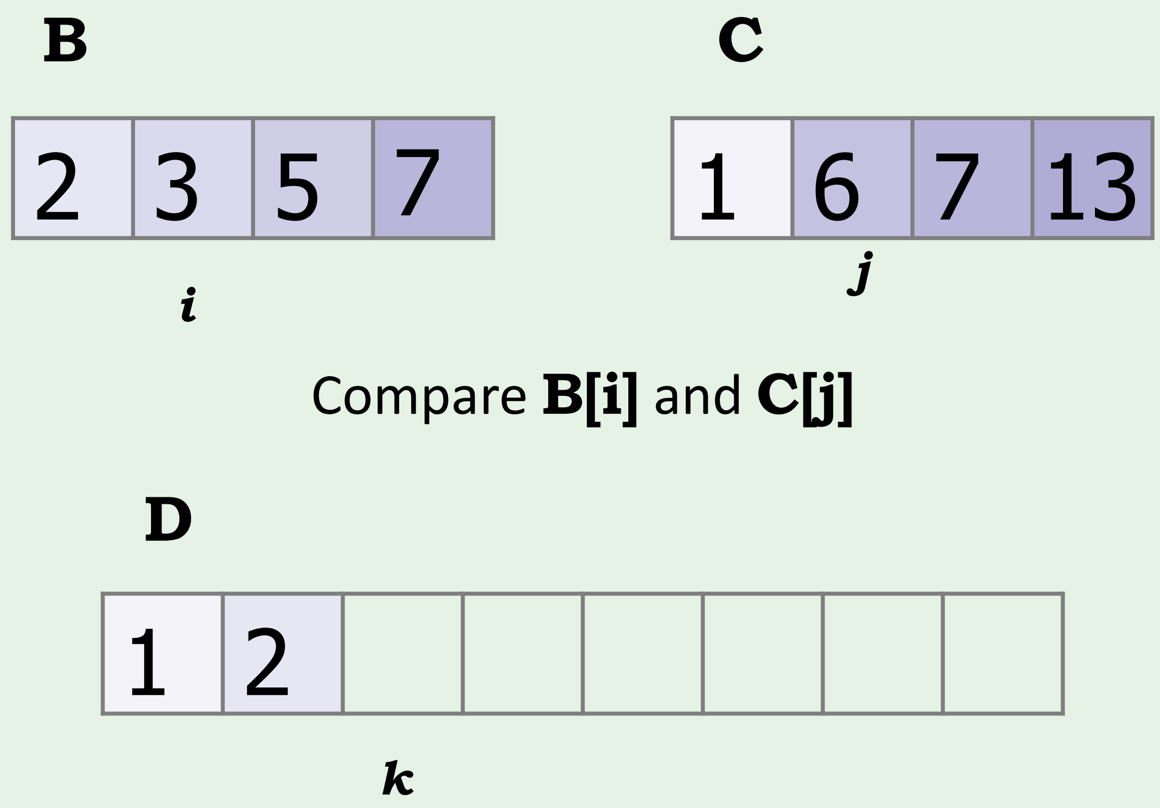
Compare **B[i]** and **C[j]**



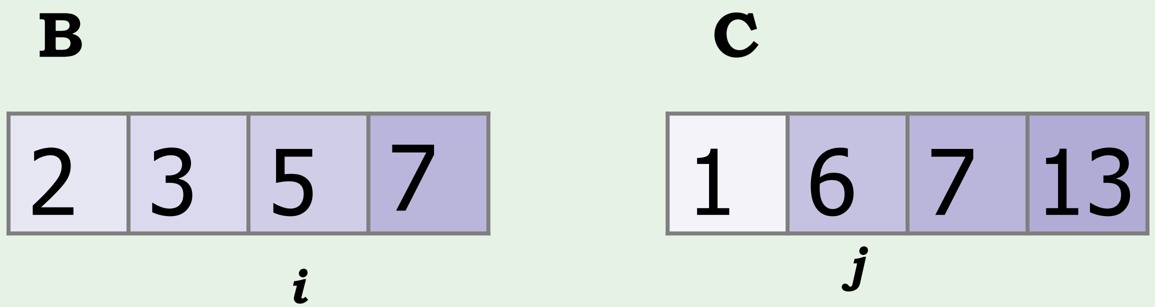
Merge: example



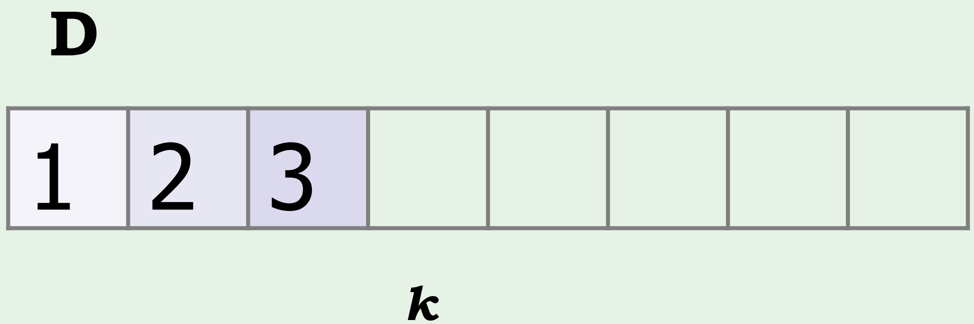
Merge: example



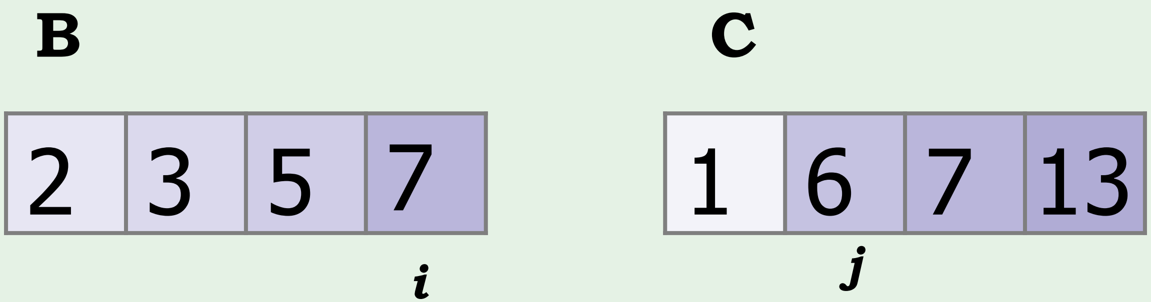
Merge: example



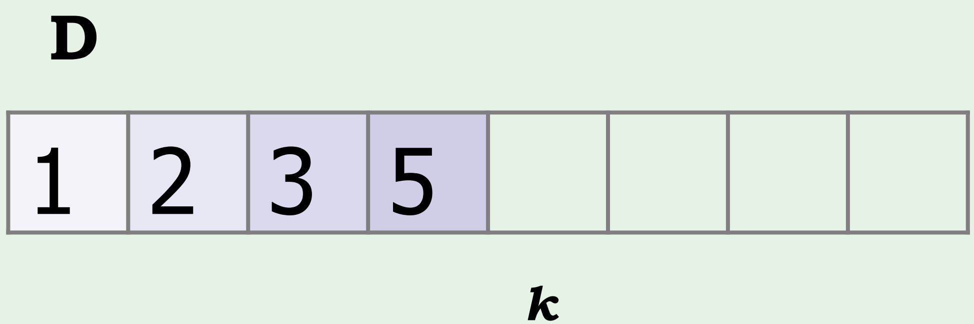
Compare **B[i]** and **C[j]**



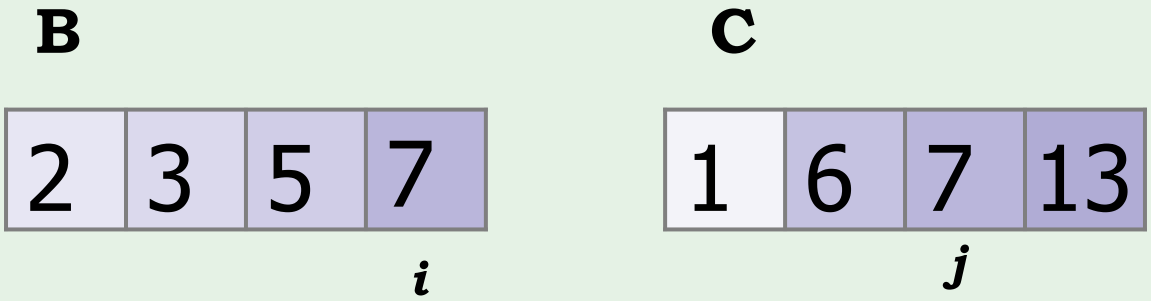
Merge: example



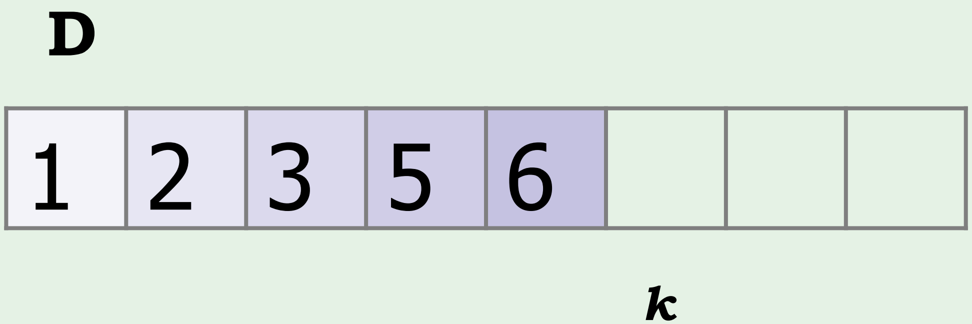
Compare **B[i]** and **C[j]**



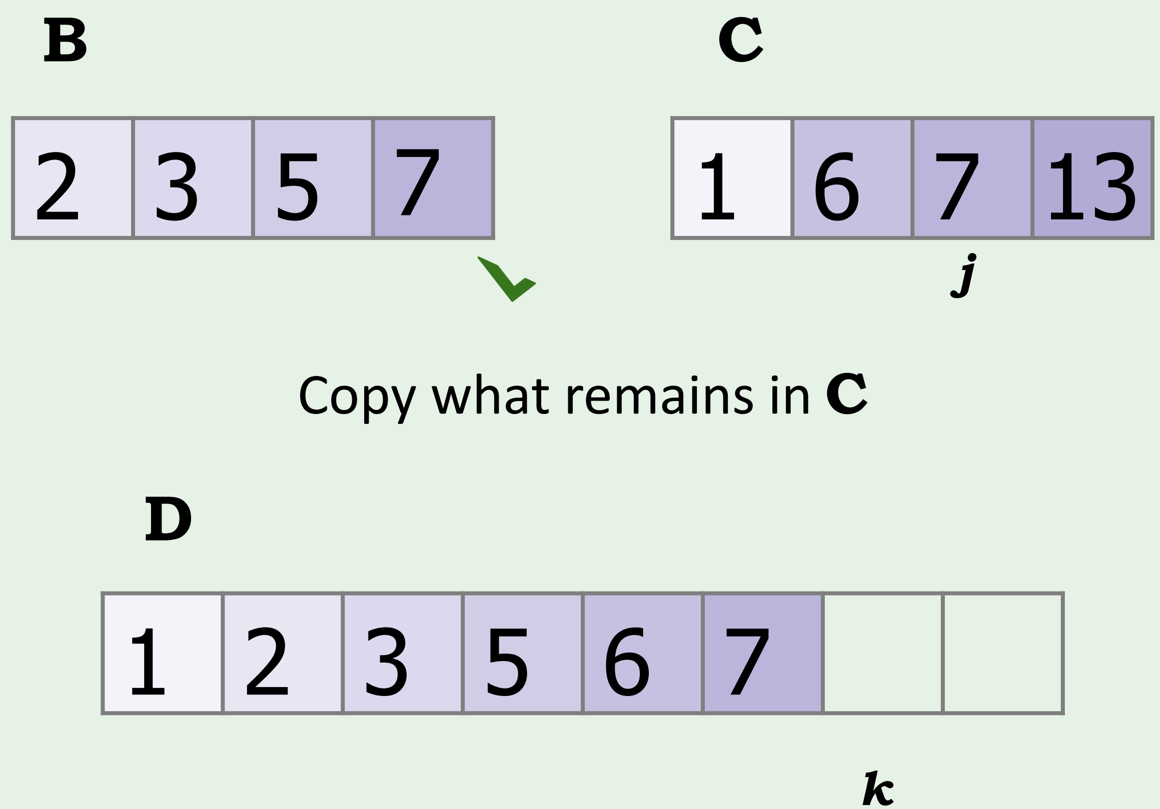
Merge: example



Compare **B[i]** and **C[j]**



Merge: example



Merge: example

B

2	3	5	7

C

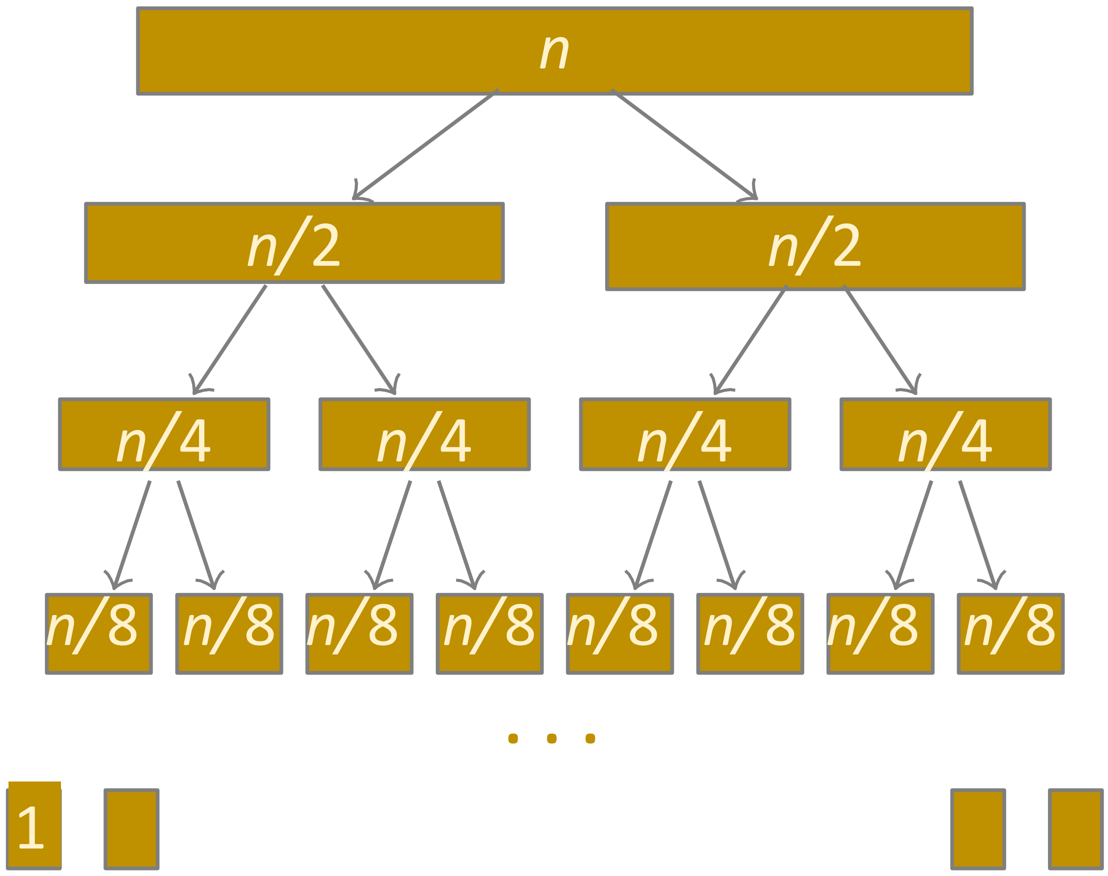
1	6	7	13

D

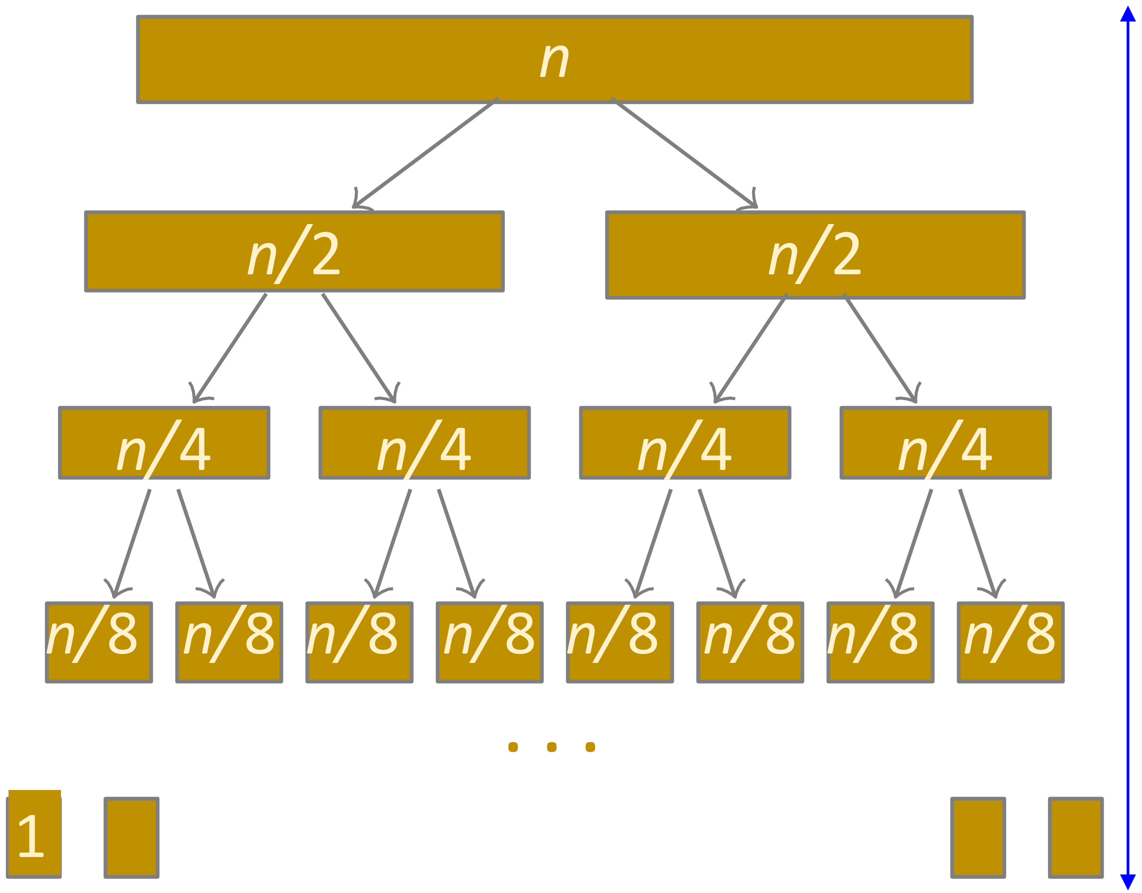
1	2	3	5	6	7	7	13

Merge sort: running time

Subproblem size at each level

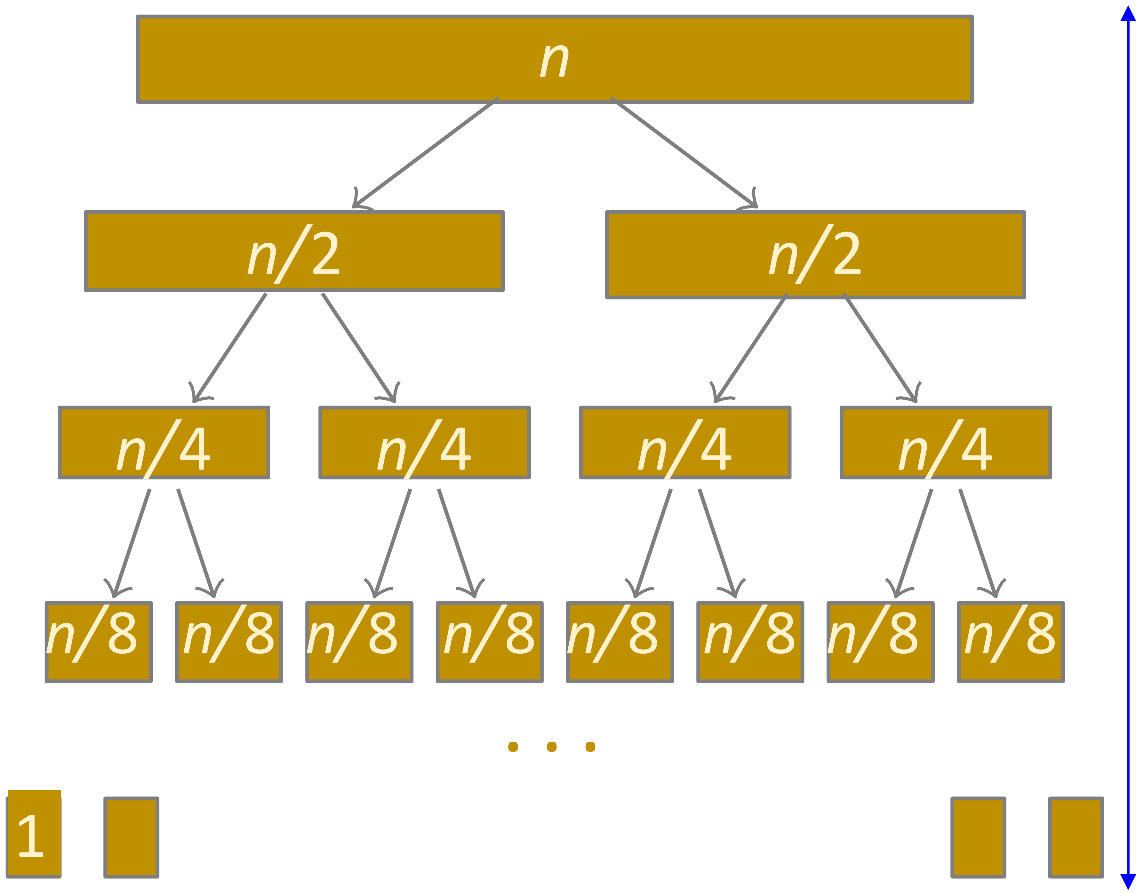


Merge sort: recursion tree



The height of the tree is...

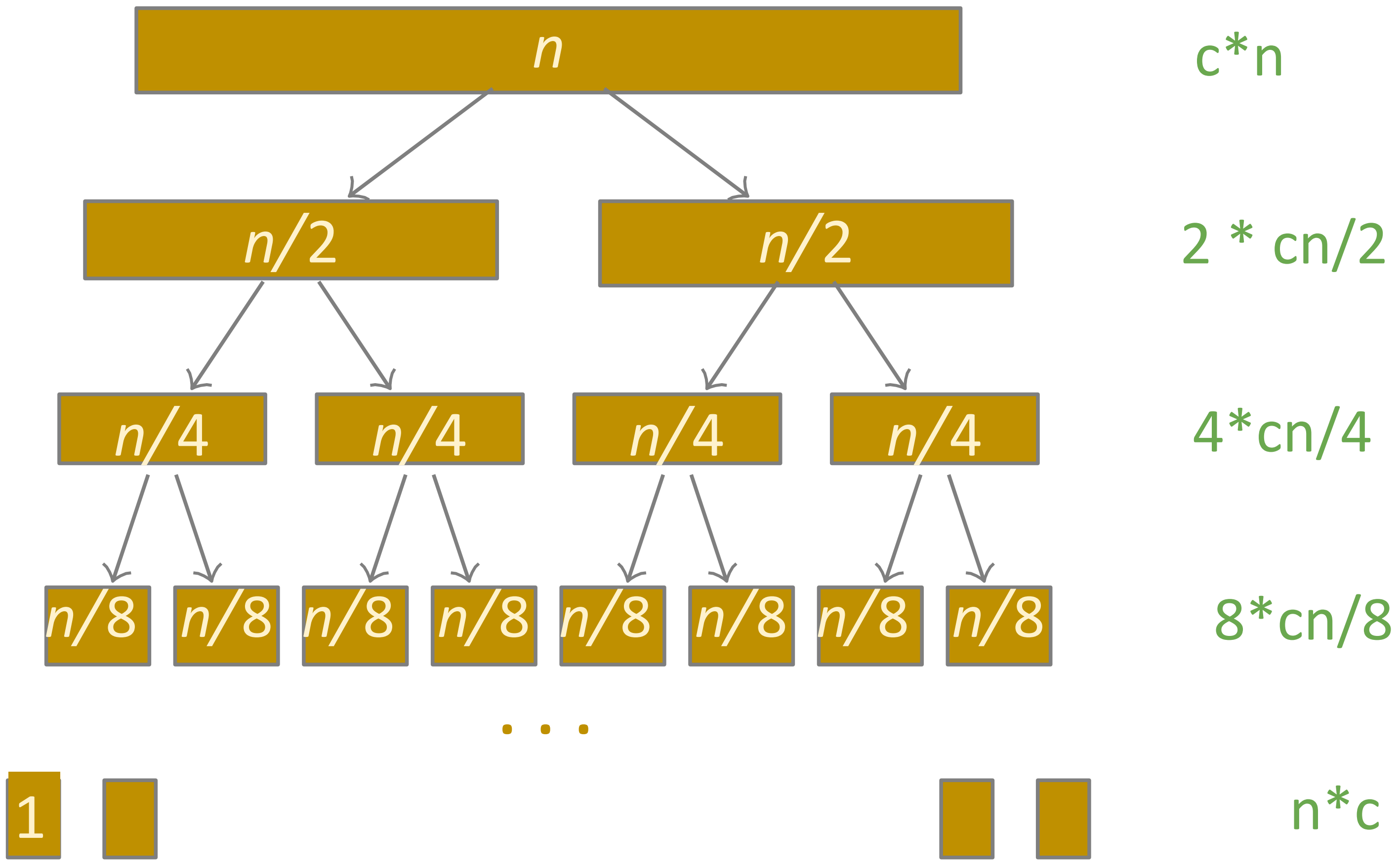
Merge sort: recursion tree



The height of the tree is *log n*

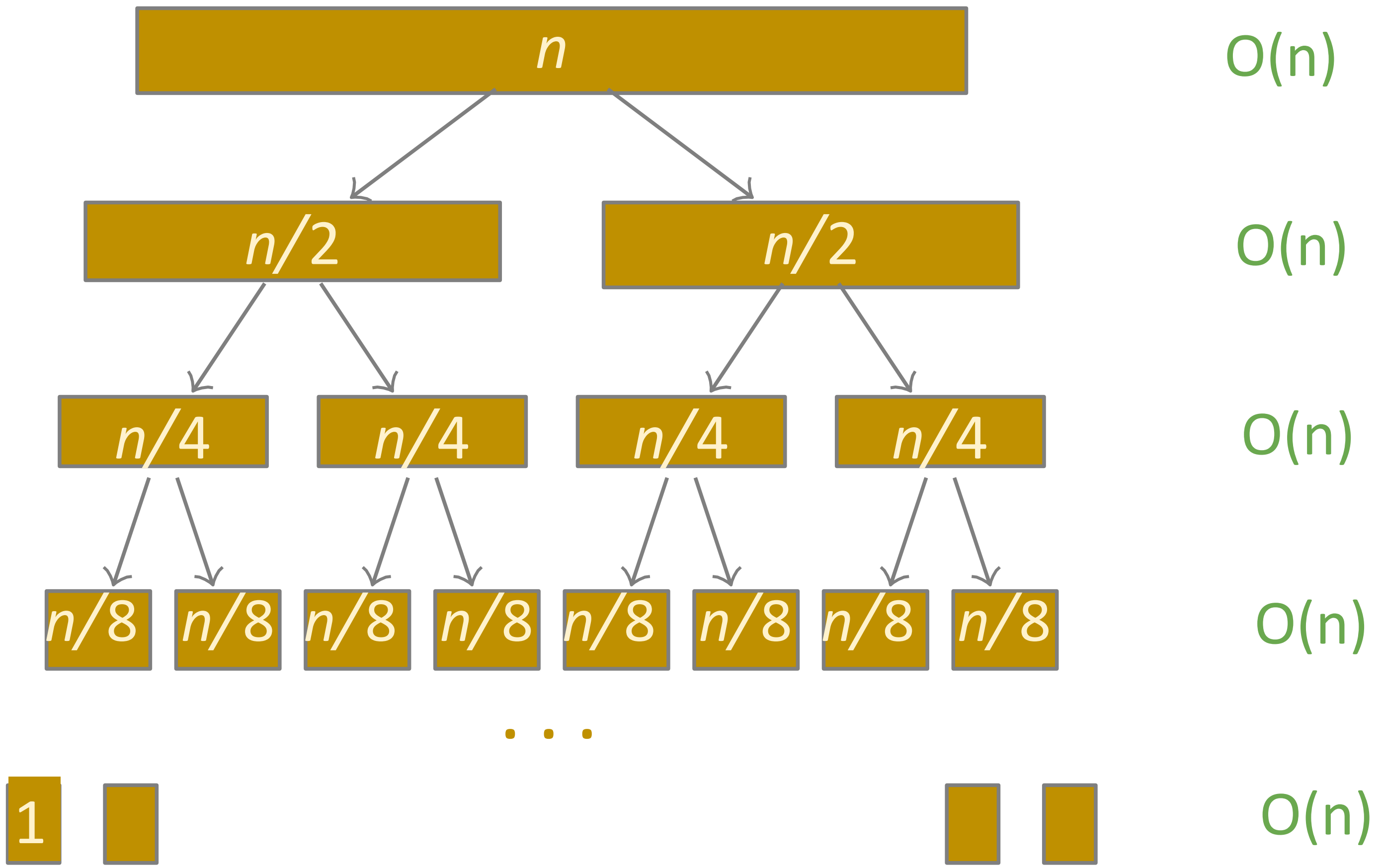
Merge sort: recursion tree

Work at each level: **all the work during *merge***



Merge sort: recursion tree

Work at each level: **O(n)**



Total: $O(n) * \log n = O(n \log n)$

Merge Sort: running time

The running time of `MergeSort`$(A[1 \,.\,.\,.\, n])$ is $O(n \log n)$.

We can prove that this running time is ***optimal*** if we consider **sorting based on comparing pairs of numbers**

We can **not** do (asymptotically) faster.

Can we do better in practice?

Idea: Quicksort

- ❑ Divide array A into 2 subarrays

  divide

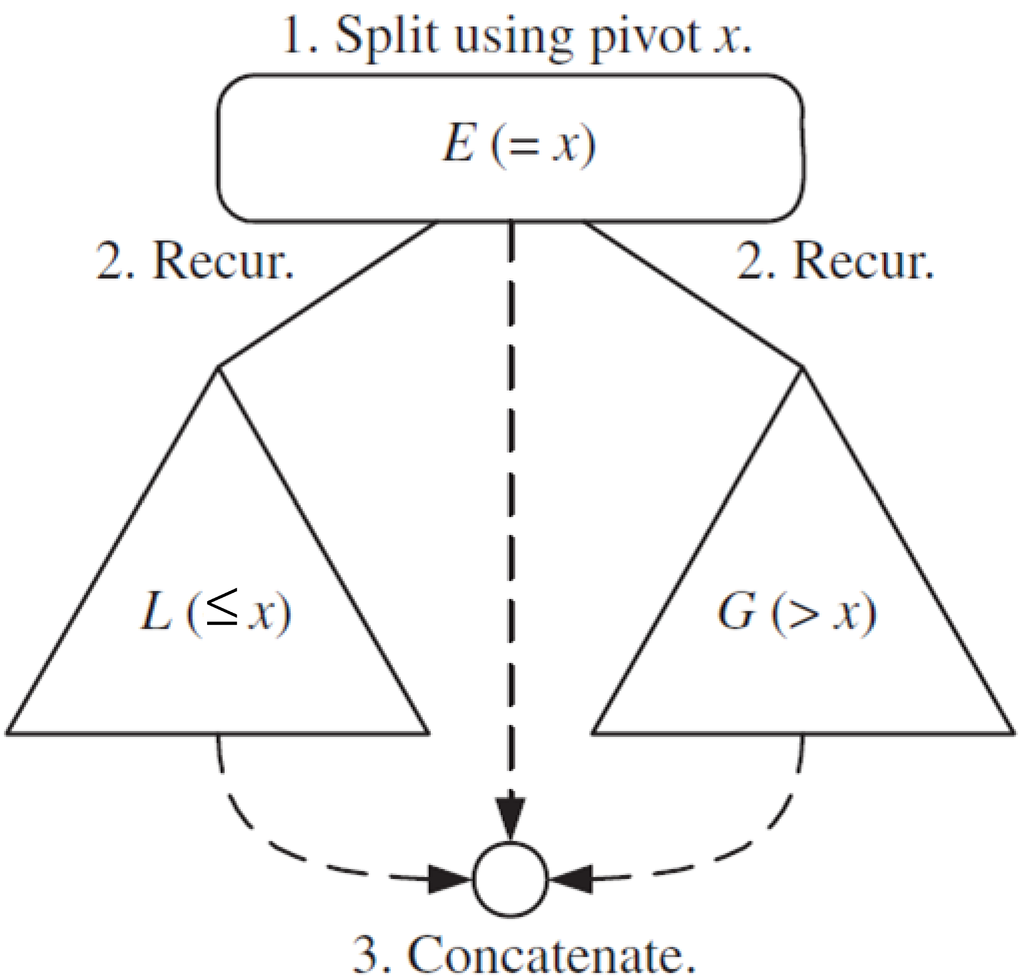
- ❑ Recursively fully sort each subarray

  conquer

- ❑ Combine the sorted subarrays by a simple concatenation

  combine

Quicksort



Select an element called ***pivot***

1. Divide elements into 2 groups *L* (less or equal), and *G* (greater than pivot)
2. Conquer: recursively sort *L* and *G*
3. Combine: concatenate L→E→R

Example: quick sort

6	4	8	2	9	3	9	4	7	6	1

Example: quick sort

6	4	8	2	9	3	9	4	7	6	1

Rearrange elements with respect to

$x = A[0]$

1	4	2	3	4	6	6	9	7	8	9

≤ 6 > 6

Example: quick sort

6	4	8	2	9	3	9	4	7	6	1

6 is in its final position

1	4	2	3	4	6	6	9	7	8	9

sort the two parts recursively

1	2	3	4	4	6	6	7	8	9	9

QuickSort(A, ℓ, r)

if $\ell \geq r$:

    return

$m \leftarrow$ Partition(A, ℓ, r)

$A[m]$ is in the final position

QuickSort(A, ℓ, $m - 1$)

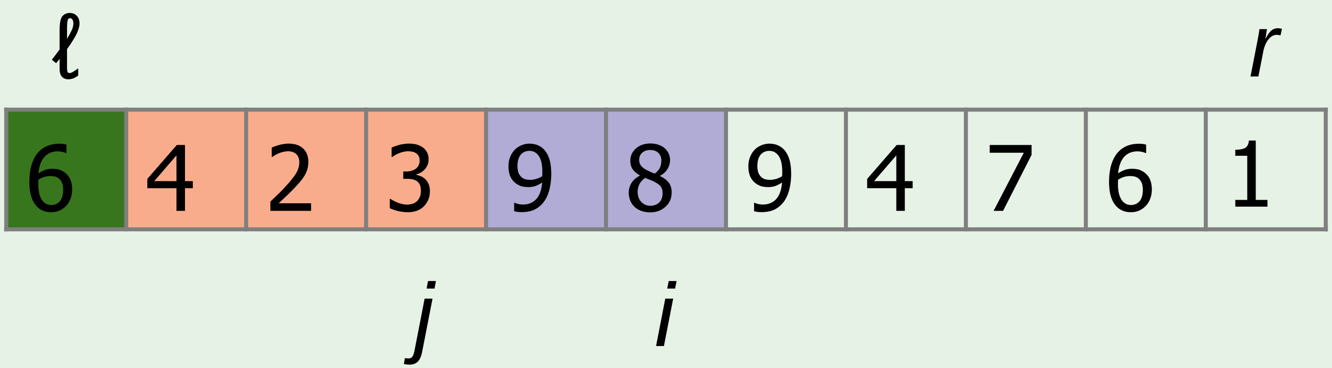
QuickSort(A, $m + 1$, r)

m is the final position of element **A[ℓ]**

PIVOT

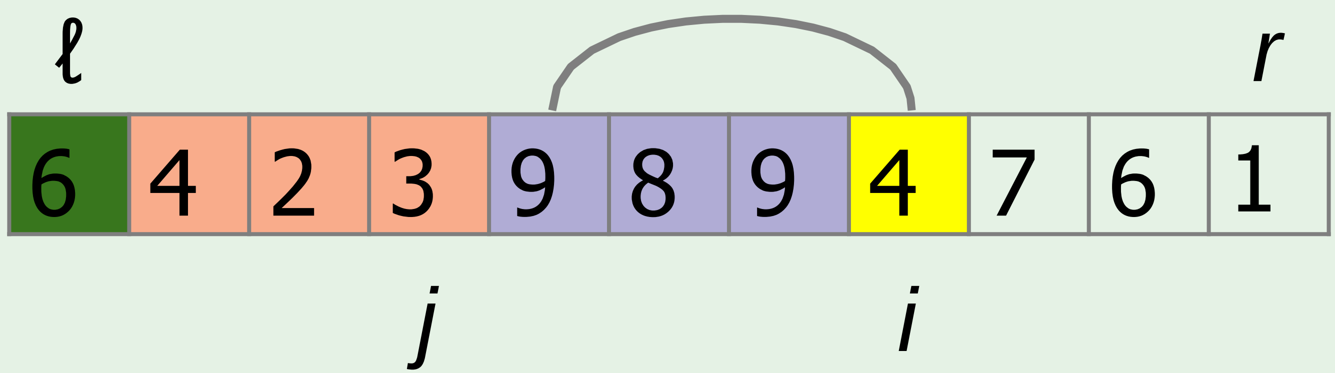
Partitioning: example

- the **pivot** is $x = A[\ell]$
- loop i from ℓ+1 to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$



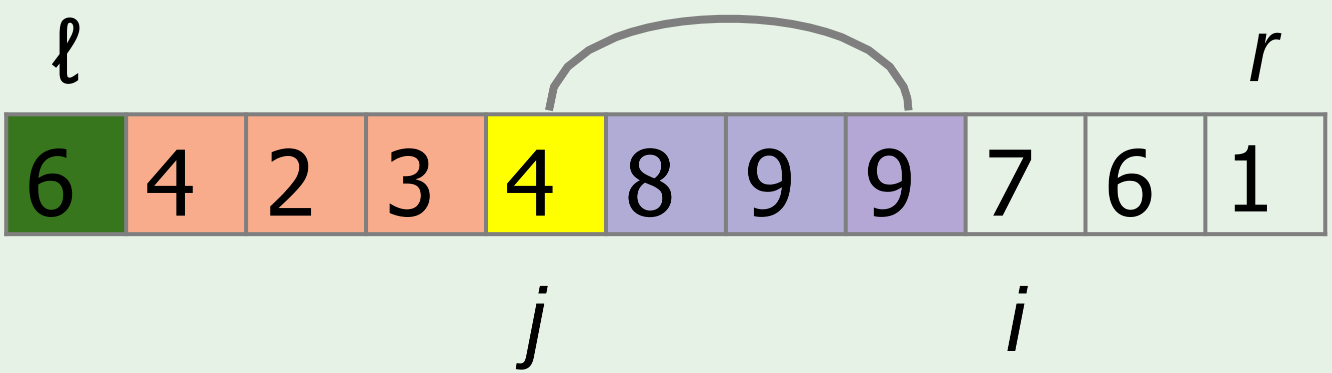
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from ℓ+1 to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



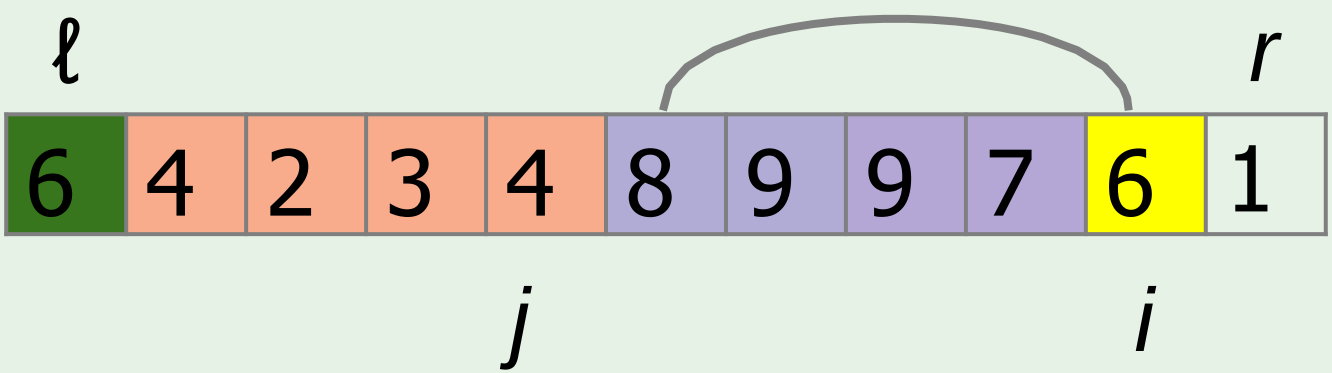
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



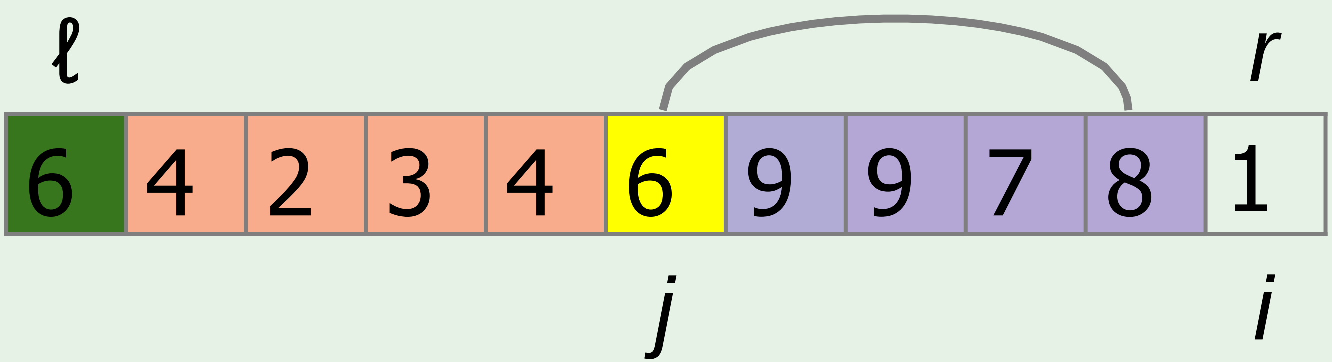
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \le x$ for all $\ell + 1 \le k \le j$
  - $A[k] > x$ for all $j + 1 \le k \le i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



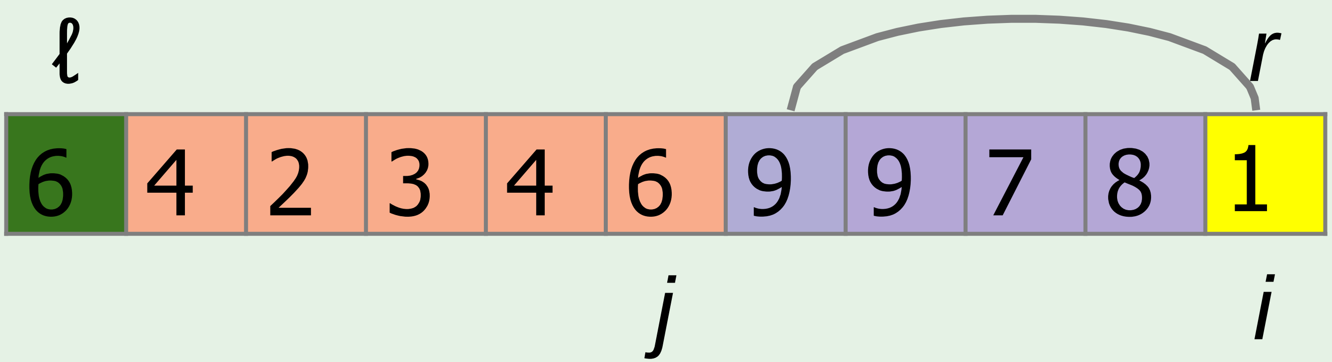
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



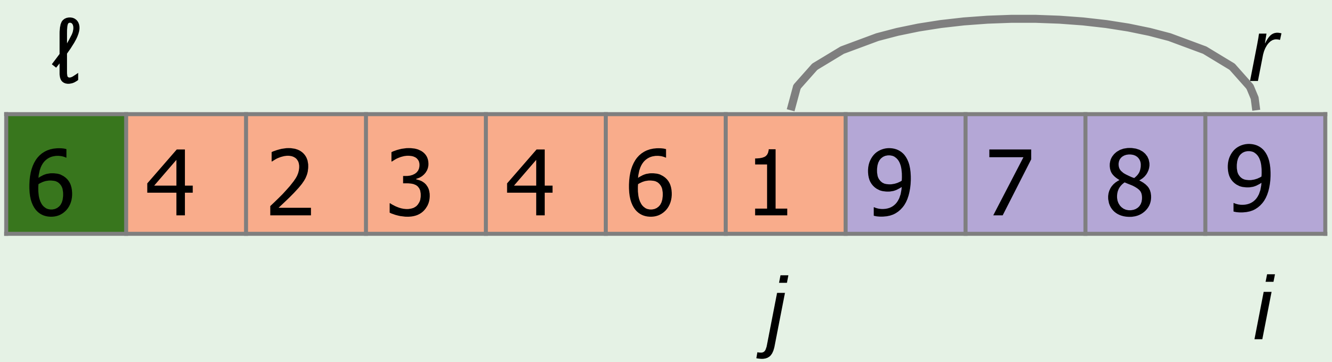
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \le x$ for all $\ell + 1 \le k \le j$
  - $A[k] > x$ for all $j + 1 \le k \le i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



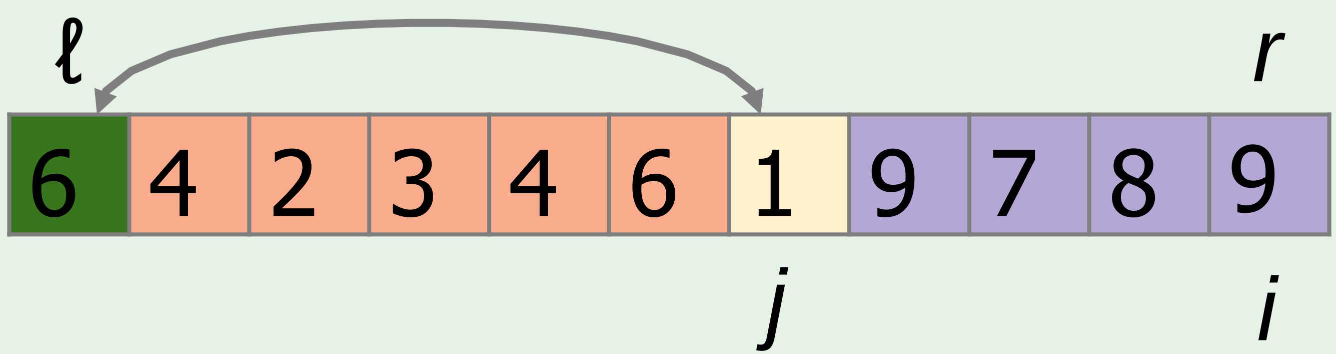
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- if encounter an out-of-order element: swap $A[i]$ with $A[j+1]$



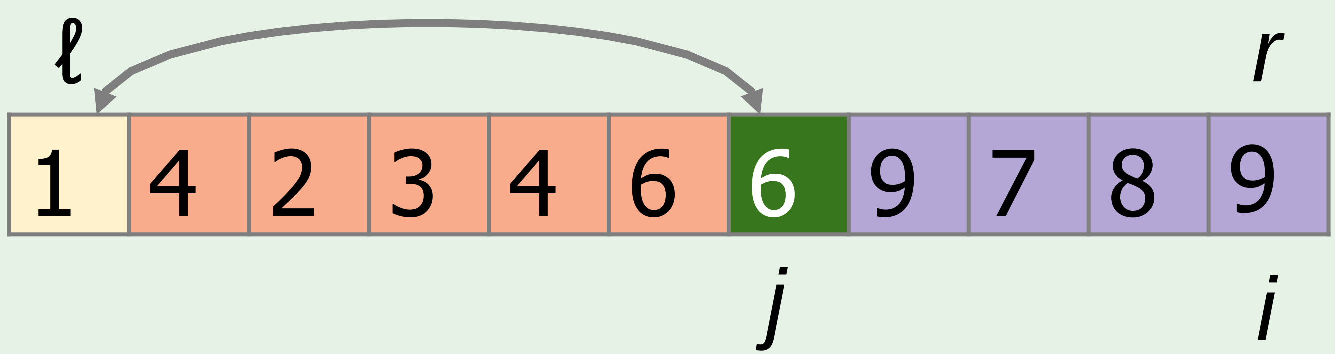
Partitioning: example

- the pivot is $x = A[\ell]$
- move i from ℓ+1 to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- in the end, move $A[\ell]$ to its final place j



Partitioning: example

- the pivot is $x = A[\ell]$
- move i from $\ell+1$ to r maintaining the following invariant:
  - $A[k] \leq x$ for all $\ell + 1 \leq k \leq j$
  - $A[k] > x$ for all $j + 1 \leq k \leq i$
- in the end, move $A[\ell]$ to its final place j



Algorithm Partition(A, ℓ, r)

$x \leftarrow A[\ell]$ # pivot

$j \leftarrow \ell$

for i from $\ell + 1$ to r :

    if $A[i] \leq x$:

        $j \leftarrow j + 1$

        swap $A[j]$ and $A[i]$

swap $A[\ell]$ and $A[j]$

return j

$A[\ell + 1 \dots j] \leq x$, $A[j + 1 \dots i] > x$

Running time of Quick Sort

If we happen to choose the pivot *x* in such a way that after the partitioning the array *A* is split into even halves:

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

This is the same as in Merge sort, only here *n* comes from *partitioning*, and in merge sort *n* comes from combine (*merge)*

⇓

The running time of Quicksort is $O(n \log n)$

Quick Sort: summary

- Simple
- Comparison-based
- Very fast in practice

Which choice of pivot would yield an optimal partitioning of *A*?

A									
	7	2	5	3	7	13	1	6	3

A. 7
B. 6
C. 5
D. 1
E. None of the above

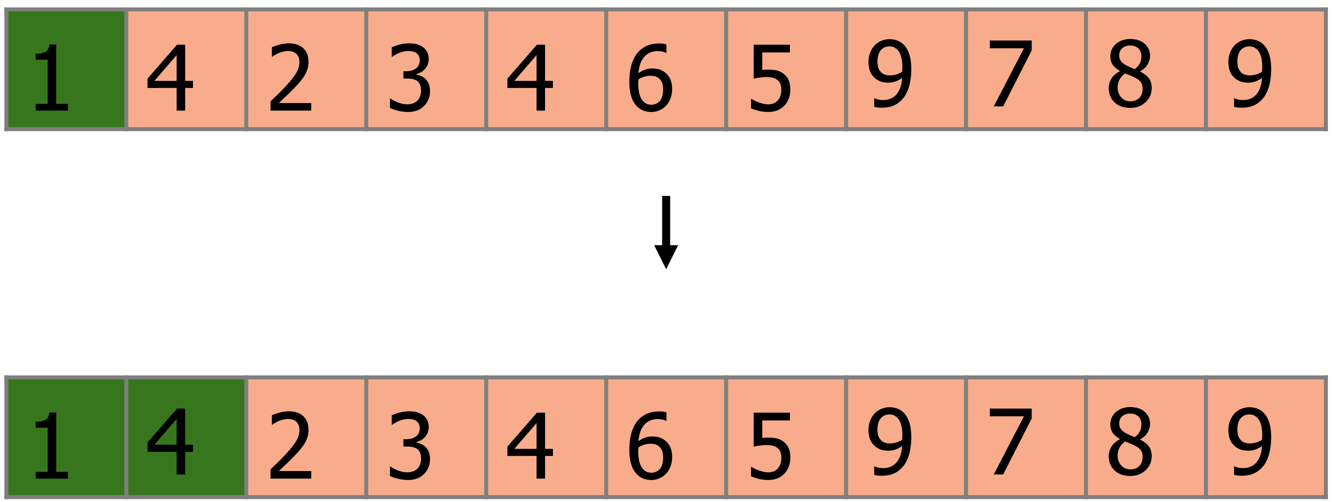
Which choice of pivot would yield the **worst** partitioning of A?

A									
	7	2	5	3	7	13	1	6	3

A. 7
B. 6
C. 5
D. 1
E. None of the above

Unlucky choice of pivot

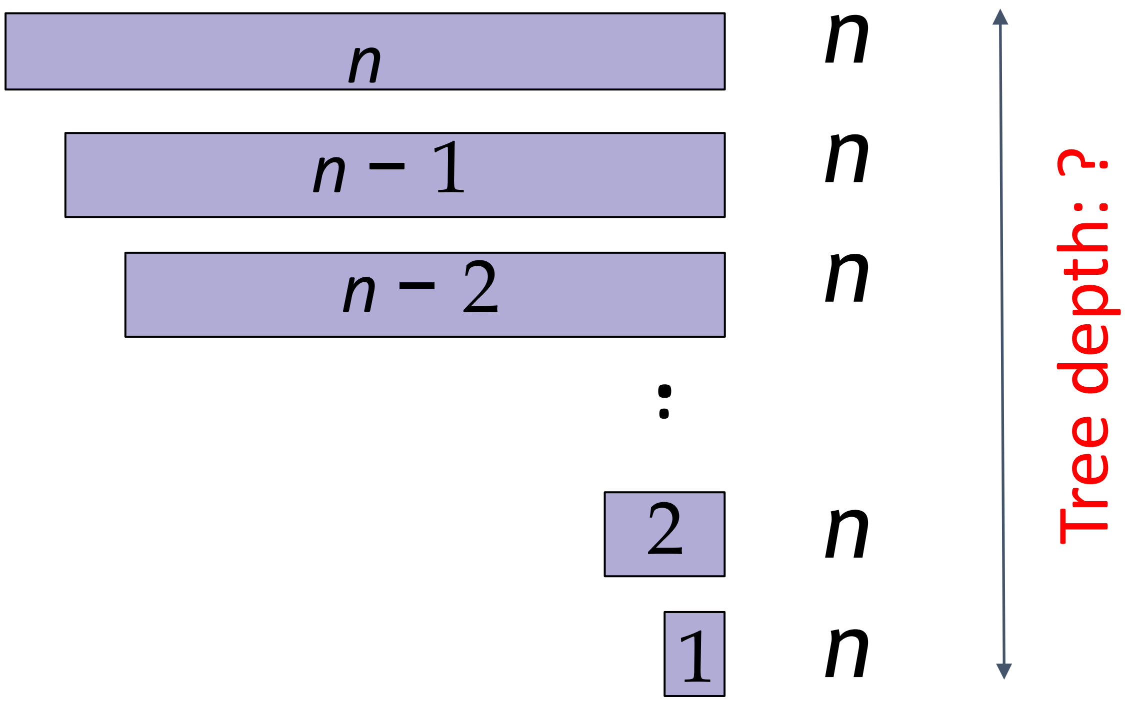
If we choose a pivot in such a way that **all values are greater than it**, then in each recursive step we decrement a size of the problem only by 1:



$T(n) = O(n) + T(n-1)$

Quick Sort: worst case complexity

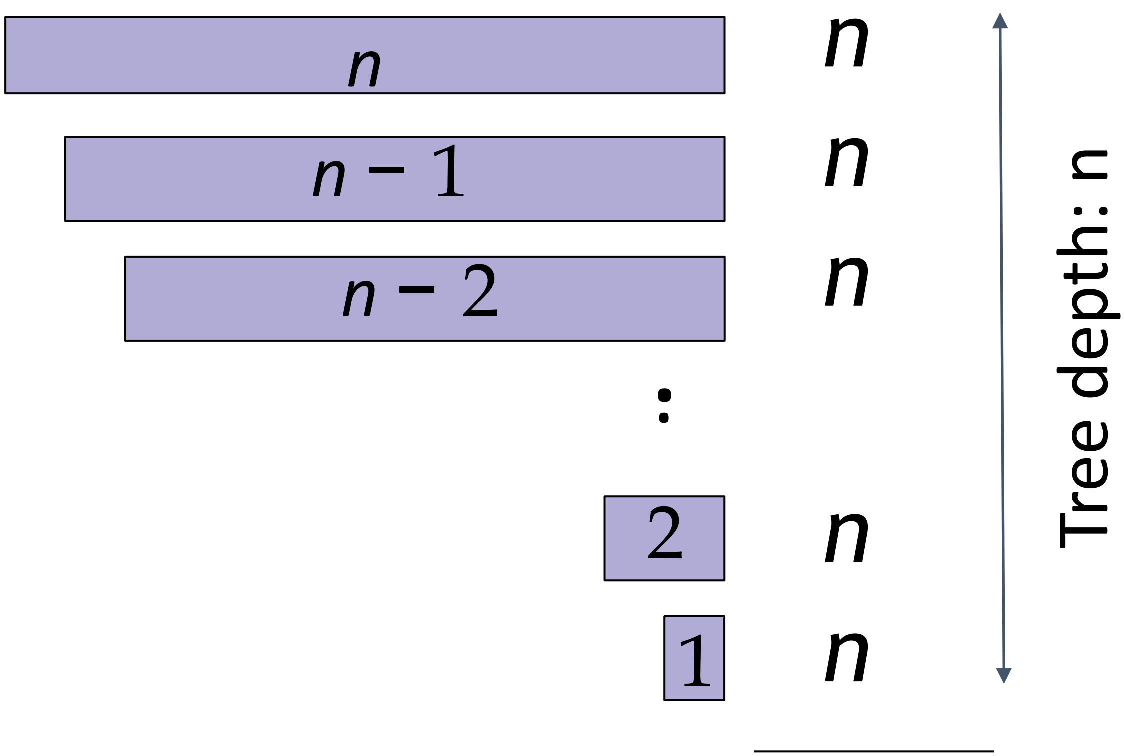
Input size at each level | Work at each level | $T(n) = n + T(n-1)$



Total: $n*n = O(n^2)$

Quick Sort: worst case complexity

Input size at each level Work at each level $T(n) = n + T(n-1)$



Total: $n*n = O(n^2)$

Pathological case

$$T\ (n) = \mathrm{O}(n^2)$$

1	2	4	5	6	6	8	9	9	9	9

It requires $O(n^2)$ time to process the already sorted array which seems very inefficient since the array is already sorted!

Choosing random pivot

- We can show that if we choose *x* randomly there is at least 50% chance that a good pivot will be chosen!

  We can prove this using the expectation and the probabilities of random events

- If we choose all pivots at random, then half the times we do decrease the input sizes by a factor

- This implies that the height of the recursive tree will be (2 log n) and the running time becomes O(n log n)

RandomizedQuickSort(A, ℓ, r)

if $\ell \geq r$:

    return

$k \leftarrow$ random number between ℓ and r

swap $A[\ell]$ and $A[k]$

$m \leftarrow$ Partition(A, ℓ, r)

$A[m]$ is in the final position

RandomizedQuickSort(A, ℓ, $m - 1$)

RandomizedQuickSort(A, $m + 1$, r)

Randomized Quick sort: Summary

- Randomized Quick sort is a comparison-based algorithm based on random partitioning
- Expected running time: $O(n \log n)$
- Still $O(n^2)$ in the worst case
- Very fast in practice