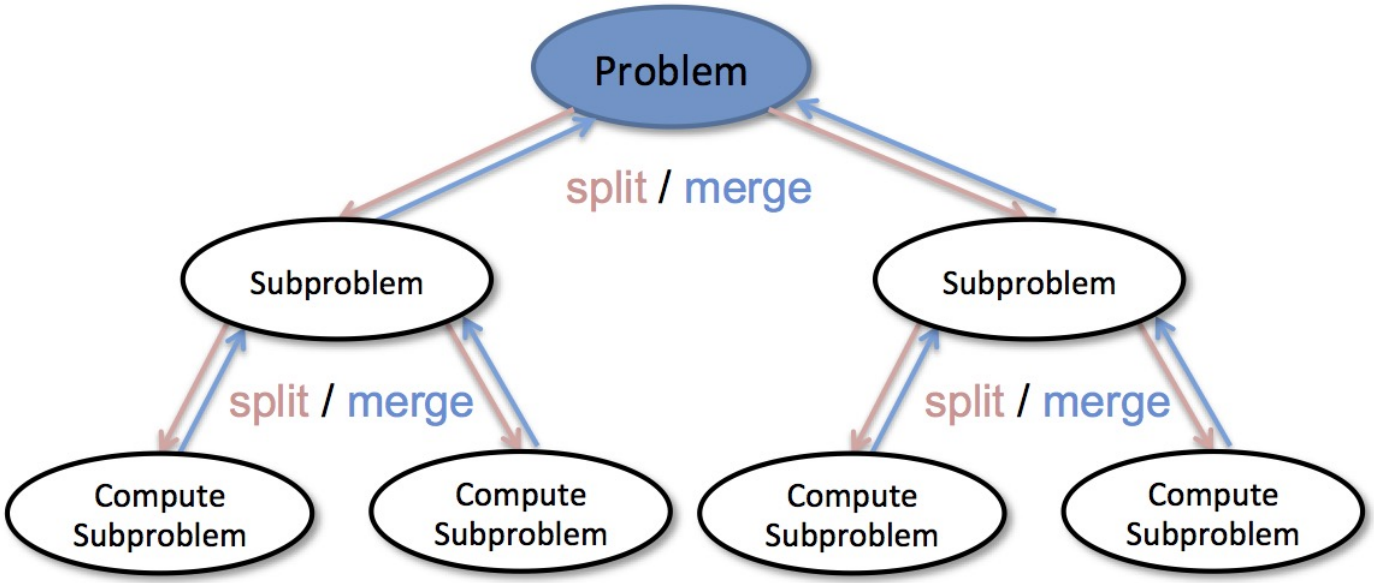
Divide and Conquer Algorithms



NYIT CSCI-651

Review

- Asymptotic analysis is analyzing what happens to the run time (or other performance metric) as the input size n goes to infinity.
  - Rate of growth ≠ fastness
- For large enough inputs, an algorithm whose running time has a lower order of growth , a $O(n^2)$ algorithm, for example, will take less time (run more quickly) in the worst case than a $O(n^3)$ algorithm.

- **Order Notation** – Big-O, Big-Theta and Big-Omega

Review - Asymptotic Analysis Hacks

- These are quick tricks to get big-Θ (or big-O) category.
- Eliminate low order terms (and hence constants)
  - $4n+5 \Rightarrow 4n$
  - $0.5\ n \log n - 2n + 7 \Rightarrow 0.5\ n \log n$
  - $2^n + n^3 + 3n \Rightarrow 2^n$
- Eliminate coefficients
  - $4n \Rightarrow n$
  - $0.5\ n \log n \Rightarrow n \log n$
  - $n \log (n^2) = 2\ n \log n \Rightarrow n \log n$

Solving Recurrences

- Substitution
  - Based on guesswork
  - Requires a lot of experience
- Recursion Tree
  - Can be be good if used done correctly
  - Or used to create an initial guess for the substitution method
- Master Method
  - Can only be used in certain cases
  - Requires memorization of the rules required to identify the three base cases

Solving Recurrences – Substitution Method

The most general method:

1. ***Guess*** the form of the solution.
2. ***Verify*** by induction.
3. ***Solve*** for constants.

Solving Recurrences – Substitution Method

Determine upper bound on recurrence

$$T(n) = 2T(\lfloor n/2 \rfloor) + n\,, \tag{4.19}$$

which is similar to recurrences (4.3) and (4.4). We guess that the solution is $T(n) = O(n \lg n)$. The substitution method requires us to prove that $T(n) \le cn \lg n$ for an appropriate choice of the constant $c > 0$. We start by assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \le c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)$. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\le 2(c \lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor)) + n \\ &\le cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\le cn \lg n\,, \end{aligned}$$

Solving Recurrences – Recursion Tree Method

- A recursion tree models the costs (time) of a recursive execution of an algorithm.
- The recursion-tree method can be unreliable, just like any method that uses ellipses (…).
- The recursion-tree method promotes intuition, however.
- The recursion tree method is good for generating guesses for the substitution method.

Solving Recurrences – Master Method

The master method applies to recurrences of the form

$$T(n) = a\,T(n/b) + f(n)\,,$$

where $a \geq 1$, $b > 1$, and f is asymptotically positive.

Solving Recurrences – Master Method

$$T(n) = a\,T(n/b) + f(n)$$

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$, constant $\varepsilon > 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$, and regularity condition
$\Rightarrow T(n) = \Theta(f(n))$.

$$T(n) = a\,T(n/b) + f(n)\ ,$$

Compare $f(n)$ with $n^{\log_b a}$:

1. $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially slower than $n^{\log_b a}$ (by an n^{ε} factor).

   Solution: $T(n) = \Theta(n^{\log_b a})$.

2. $f(n) = \Theta(n^{\log_b a} \lg^k n)$ for some constant $k \geq 0$.
   - $f(n)$ and $n^{\log_b a}$ grow at similar rates.

   Solution: $T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

Compare $f(n)$ with $n^{\log_b a}$:

3. $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$.
   - $f(n)$ grows polynomially faster than $n^{\log_b a}$ (by an n^{ε} factor),

   and $f(n)$ satisfies the ***regularity condition*** that $af(n/b) \le cf(n)$ for some constant $c < 1$.

   Solution: $T(n) = \Theta(f(n))$.

Master Method - Examples

Ex. $T(n) = 4T(n/2) + n$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n.$

Case 1: $f(n) = O(n^{2-\varepsilon})$ for $\varepsilon = 1$.

$\therefore\ T(n) = \Theta(n^2).$

Master Method - Examples

Ex. $T(n) = 4T(n/2) + n^2$

$a = 4, b = 2 \Rightarrow n^{\log_b a} = n^2; f(n) = n^2.$

Case 2: $f(n) = \Theta(n^2 \lg^0 n)$, that is, $k = 0$.

$\therefore\ T(n) = \Theta(n^2 \lg n).$

Master Method - Examples

Ex. $T(n) = 4T(n/2) + n^3$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^3.$

CASE 3: $f(n) = \Omega(n^{2+\varepsilon})$ for $\varepsilon = 1$

and $4(n/2)^3 \le cn^3$ (reg. cond.) for $c = 1/2$.

$\therefore\ T(n) = \Theta(n^3).$

Ex. $T(n) = 4T(n/2) + n^2/\lg n$

$a = 4,\ b = 2 \Rightarrow n^{\log_b a} = n^2;\ f(n) = n^2/\lg n.$

Master method does not apply. In particular, for every constant $\varepsilon > 0$, we have $n^\varepsilon = \omega(\lg n)$.

Merge Sort – Master Theorem

$$T(n) = a\,T(n/b) + f(n)$$

CASE 1: $f(n) = O(n^{\log_b a - \varepsilon})$, constant $\varepsilon > 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a})$.

CASE 2: $f(n) = \Theta(n^{\log_b a} \lg^k n)$, constant $k \geq 0$
$\Rightarrow T(n) = \Theta(n^{\log_b a} \lg^{k+1} n)$.

CASE 3: $f(n) = \Omega(n^{\log_b a + \varepsilon})$, constant $\varepsilon > 0$,
and regularity condition
$\Rightarrow T(n) = \Theta(f(n))$.

Merge sort: $a = 2$, $b = 2$ $\Rightarrow$ $n^{\log_b a} = n^{\log_2 2} = n$
$\Rightarrow$ CASE 2 ($k = 0$) $\Rightarrow$ $T(n) = \Theta(n \lg n)$.

Divide and Conquer

Divide and Conquer

- In politics and sociology is gaining and maintaining power by breaking up larger concentrations of power into pieces that individually have less power than the one implementing the strategy.

[wikipedia]

Divide and Conquer

In Computer Science:

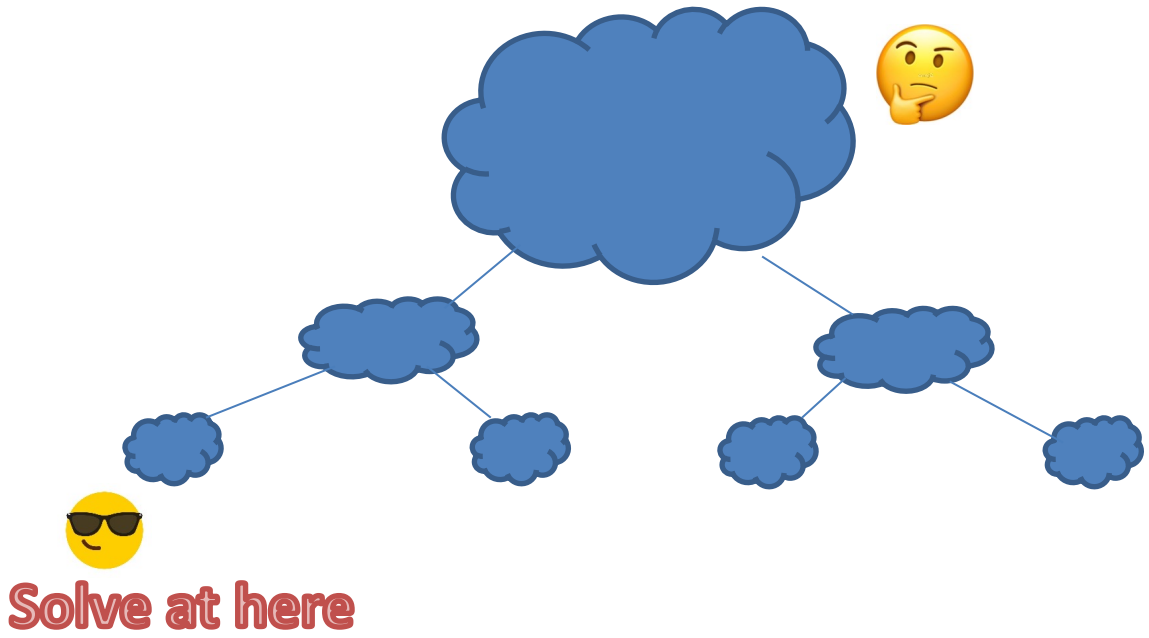
- A divide and conquer algorithm recursively breaks down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly.
- The solutions to the sub-problems are then combined to give a solution to the original problem.

Divide and Conquer

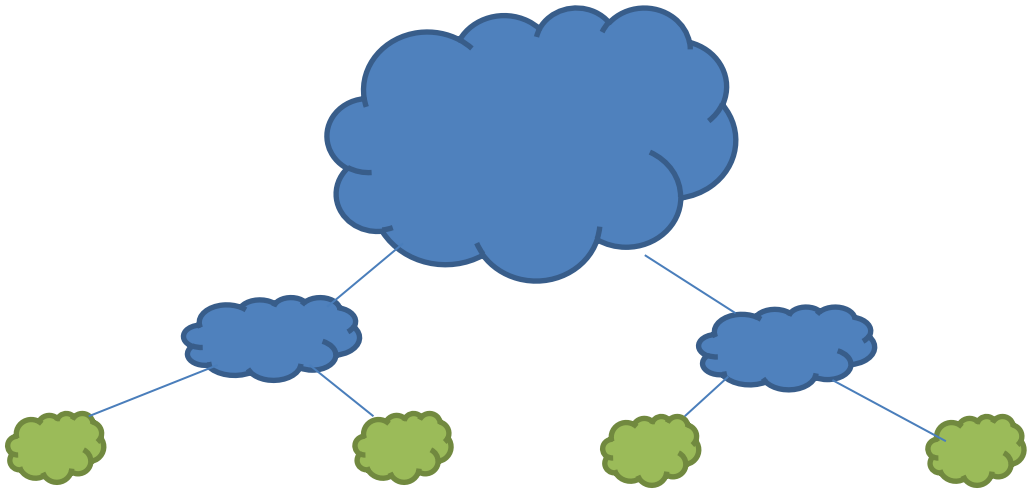
- Involves 3 steps

1. **Divide** the problem into a number of sub-problems that are smaller instances of the same problem
2. **Conquer** the sub-problems by solving them recursively. If the sub-problem is small enough, solve it in a straightforward way
3. **Combine** the solutions to the sub-problems into the solution of the original problem.

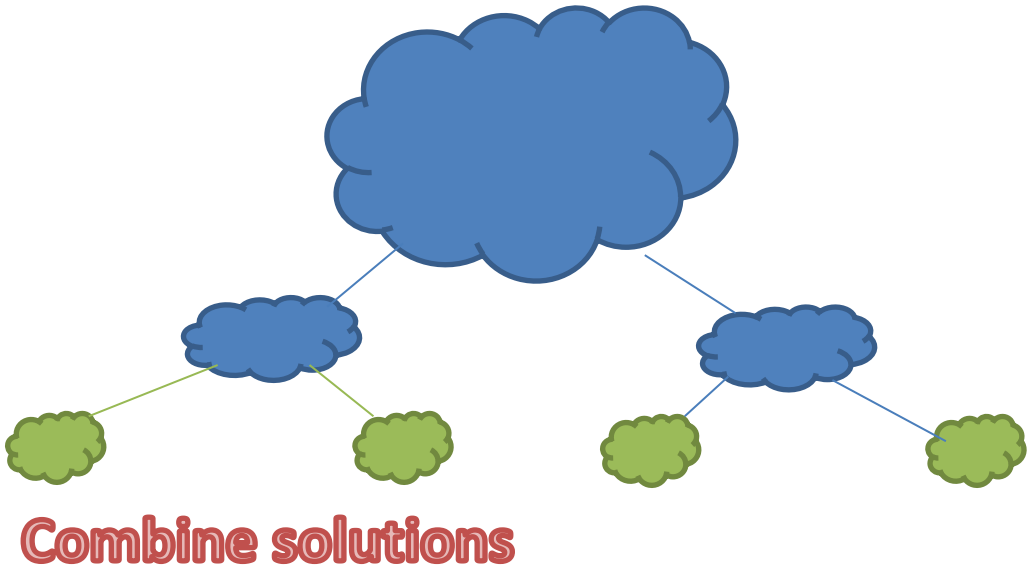
Divide and Conquer



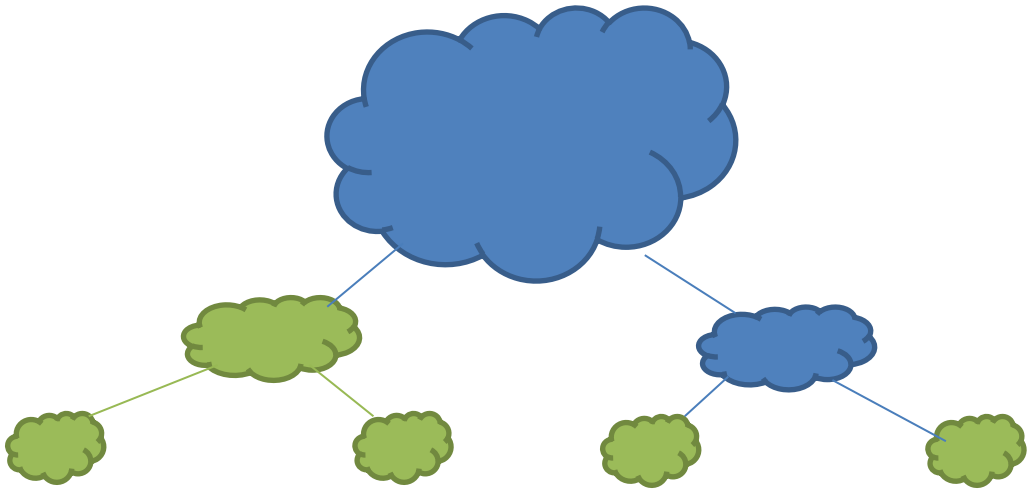
Divide and Conquer



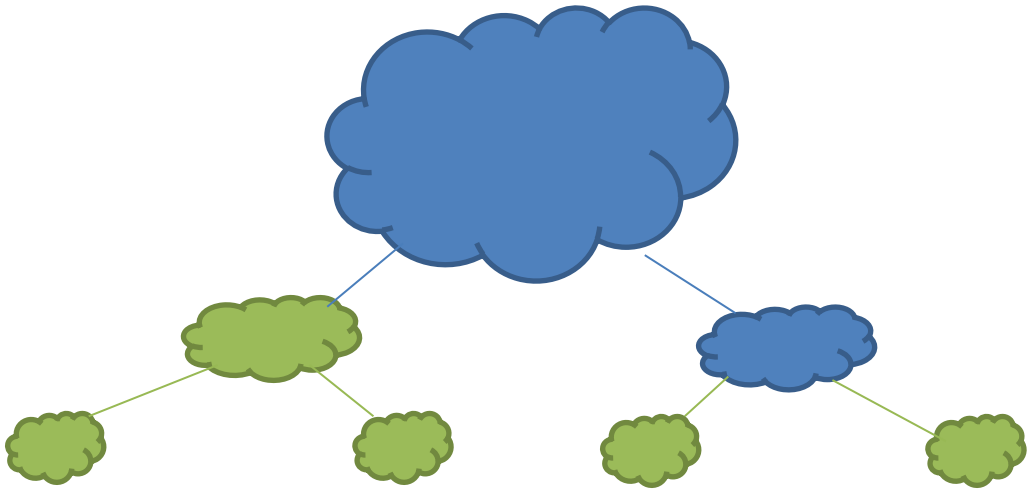
Divide and Conquer



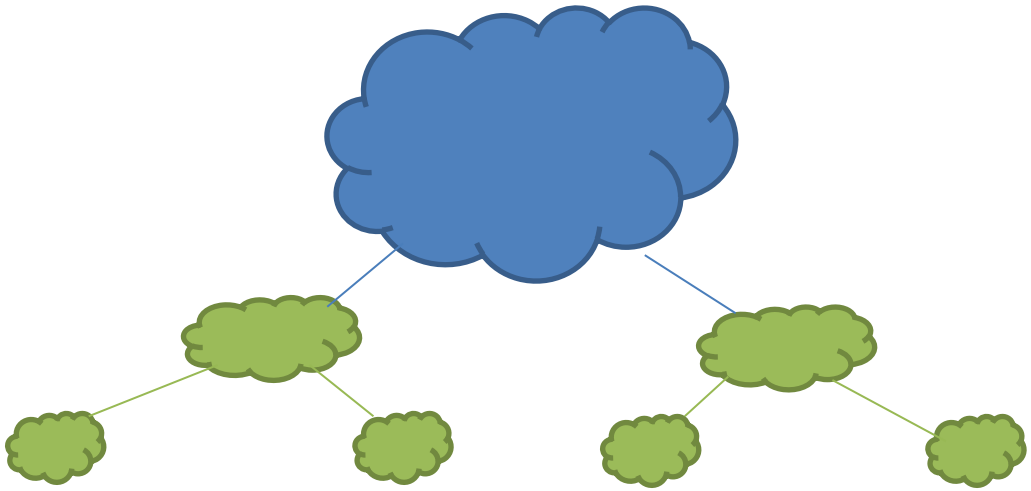
Divide and Conquer



Divide and Conquer



Divide and Conquer



Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3	5	7	8	9	12	15

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

Binary Search

Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

Example: Find 9

3 5 7 8 9 12 15

Binary Search

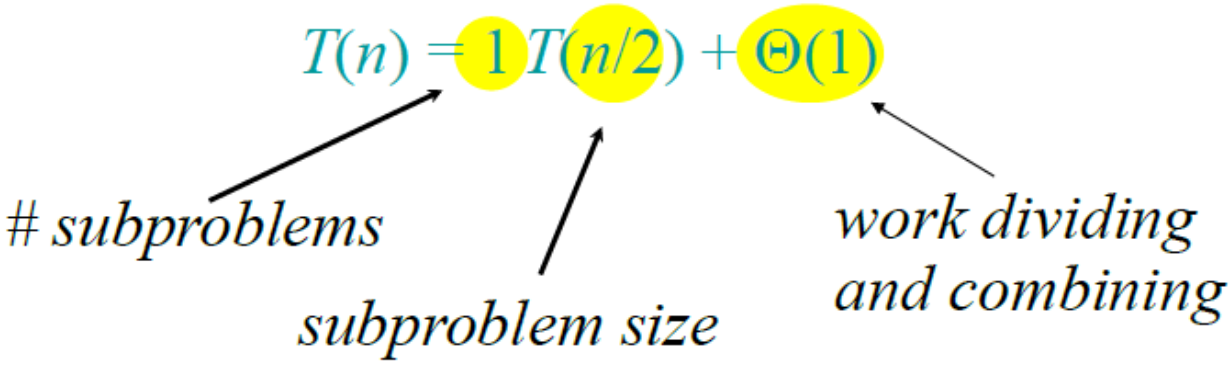
Find an element in a sorted array:

1. ***Divide:*** Check middle element.
2. ***Conquer:*** Recursively search 1 subarray.
3. ***Combine:*** Trivial.

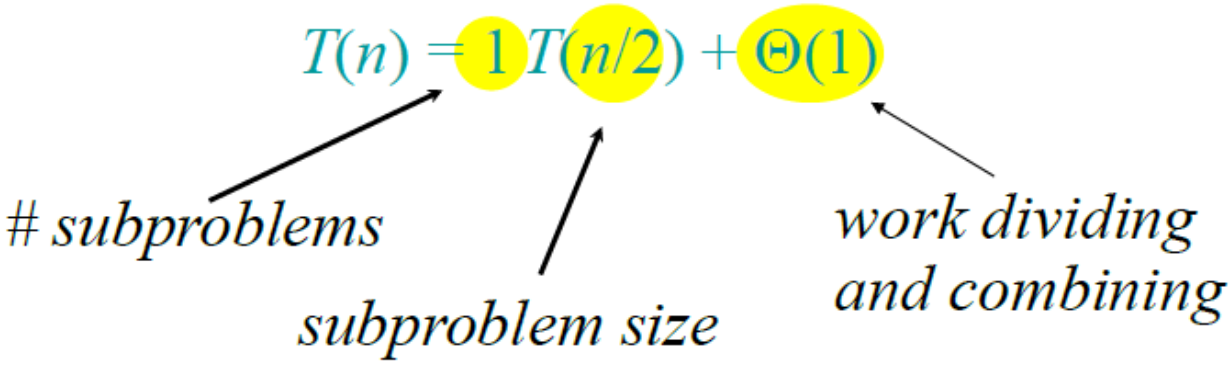
Example: Find 9

3 5 7 8 9 12 15

Binary Search - Recurrence



Binary Search - Recurrence



$n^{\log_b a} = n^{\log_2 1} = n^0 = 1 \Rightarrow$ CASE 2 ($k = 0$)

$\Rightarrow T(n) = \Theta(\lg n)$.

Algorithm – Sorting Problem

- **Computational Problem:** *Sort a sequence of numbers into nondecreasing order*
- **Input and Output:**

Input: sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of numbers.

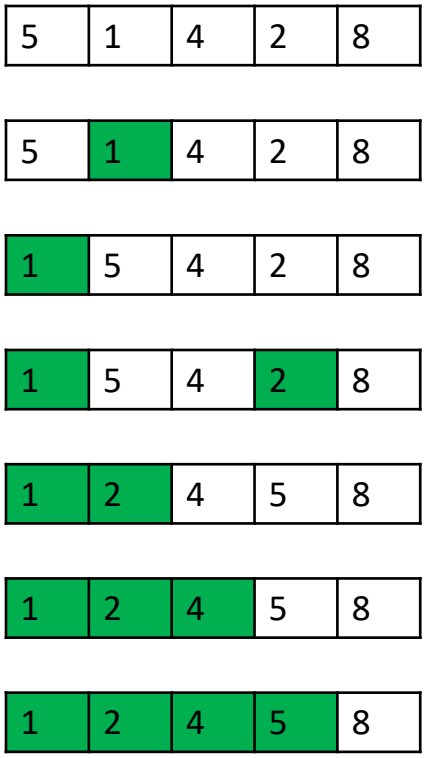
Output: permutation $\langle a'_1, a'_2, \ldots, a'_n \rangle$ such that $a'_1 \leq a'_2 \leq \cdots \leq a'_n$.

Example:

Input: 8 2 4 9 3 6

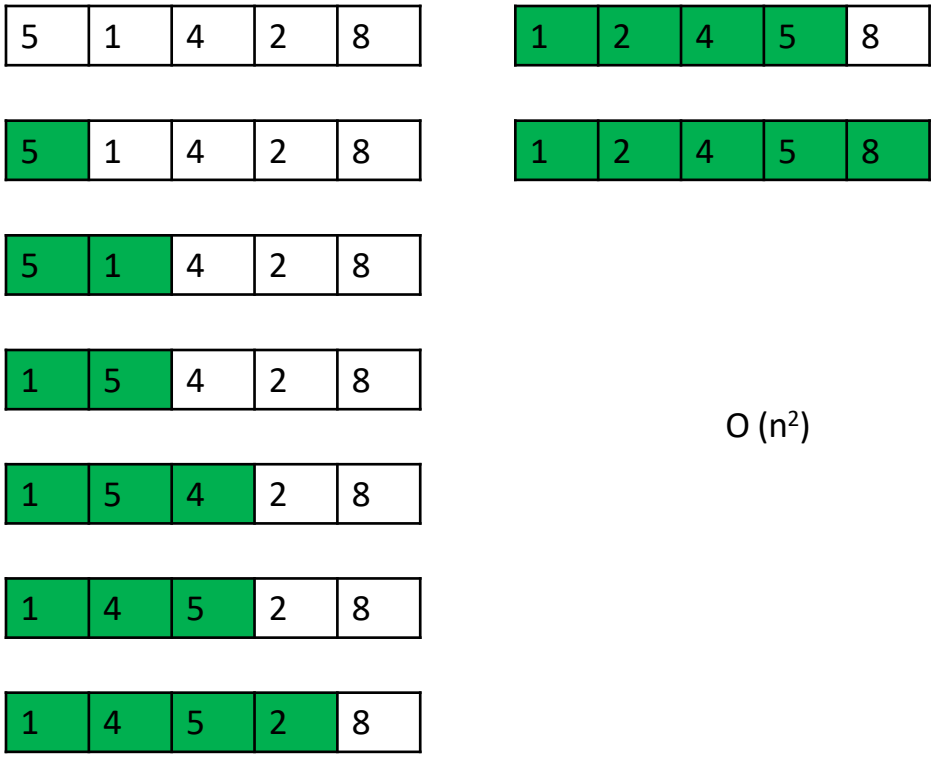
Output: 2 3 4 6 8 9

Selection Sort



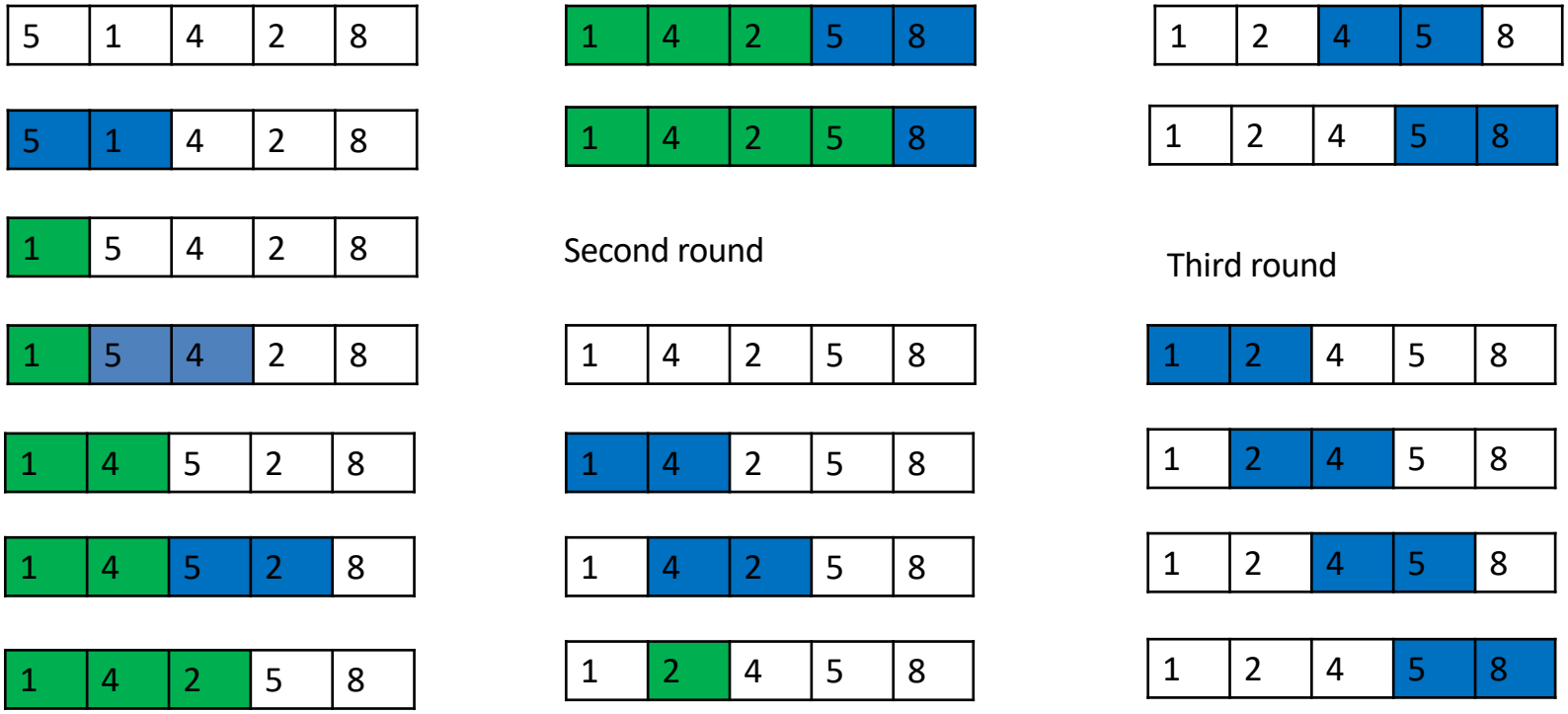
$O(n^2)$

Insertion Sort



$O(n^2)$

Bubble Sort


5 1 4 2 8
5 1 4 2 8
1 5 4 2 8
1 5 4 2 8
1 4 5 2 8
1 4 5 2 8
1 4 2 5 8
1 4 2 5 8
1 4 2 5 8
Second round
1 4 2 5 8
1 4 2 5 8
1 4 2 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
Third round
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8

Merge Sort

- Merge sort algorithm follows the divide and conquer paradigm
  - The problem space is continually split in half, recursively applying the algorithm to each half until the base case is reached.

- Merge sort improves over Selection, Insertion and Bubble sorts

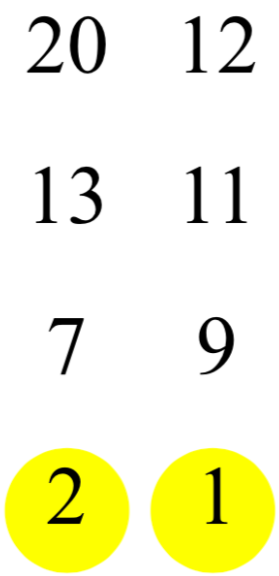
Merge Sort

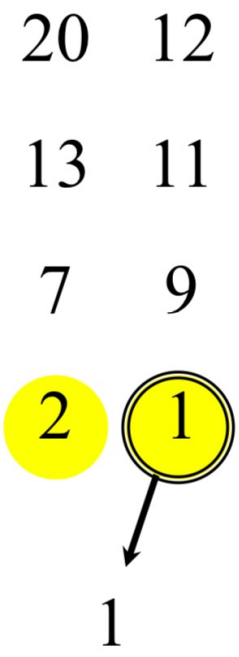
MERGE-SORT $A[1 \ldots n]$

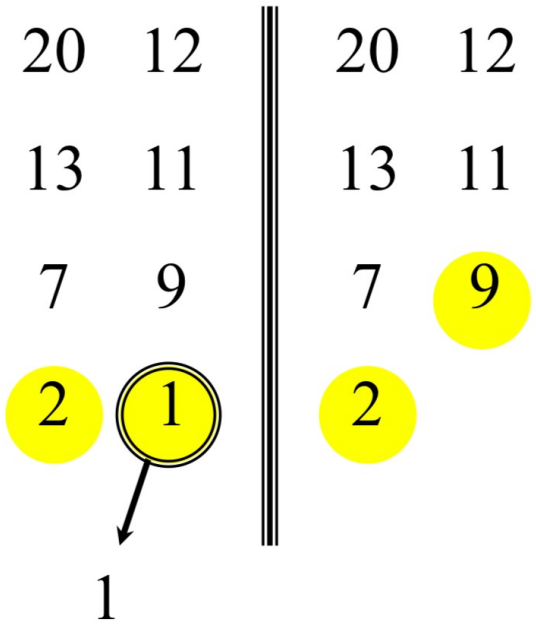
1. If $n = 1$, done.
2. Recursively sort $A[\,1 \ldots \lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1 \ldots n\,]$.
3. "***Merge***" the 2 sorted lists.

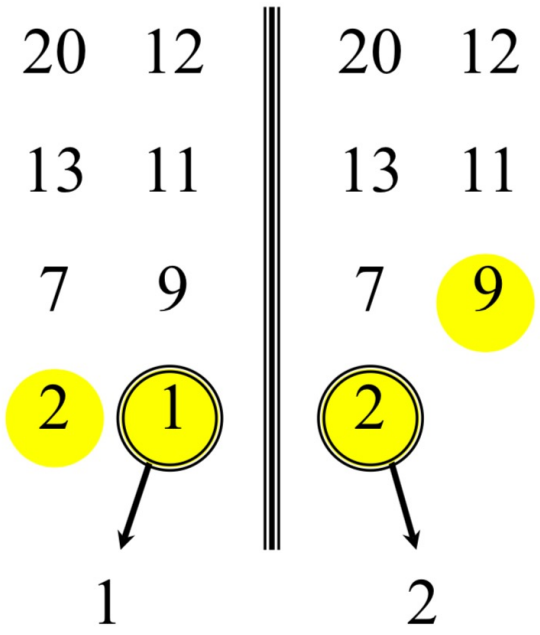
Key subroutine: **MERGE**

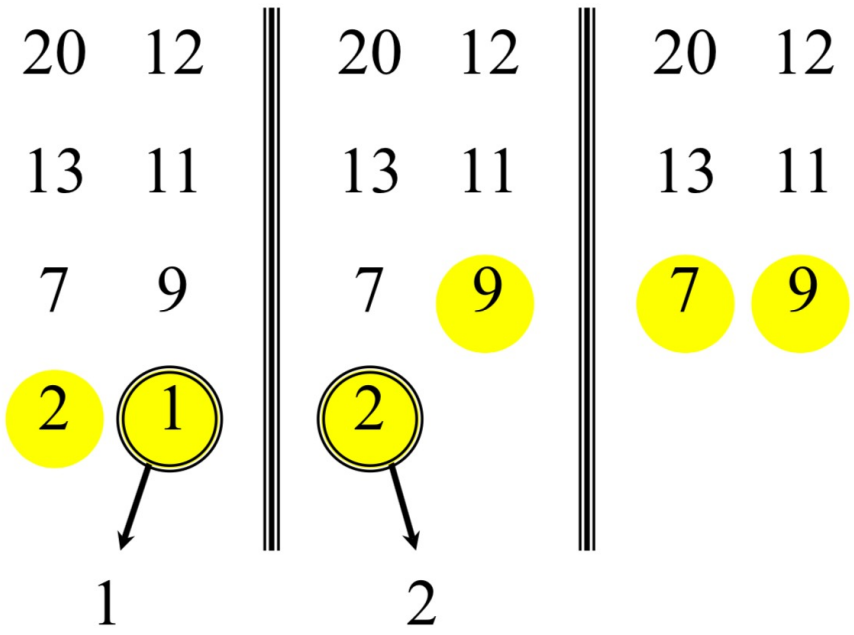
Merge Sort – Merging Two Sorted Arrays

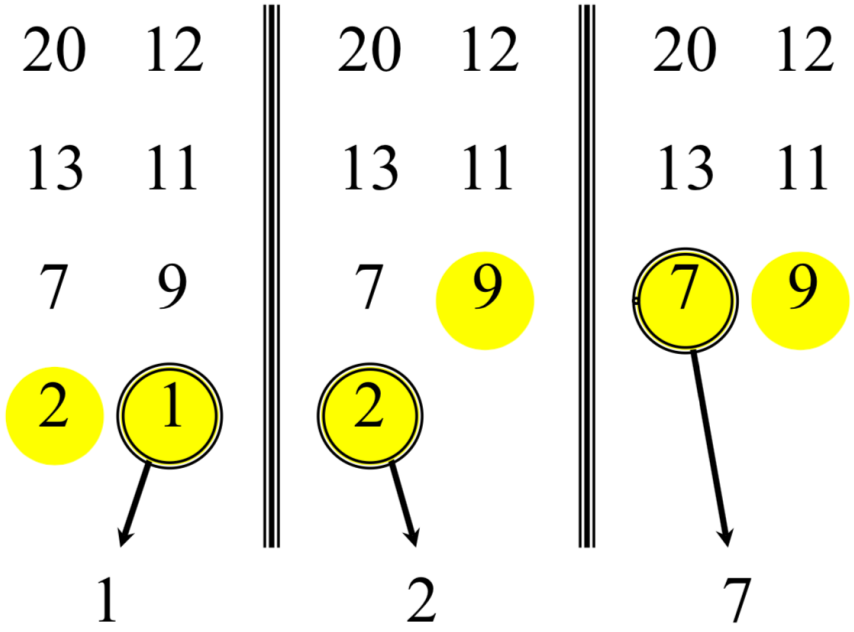


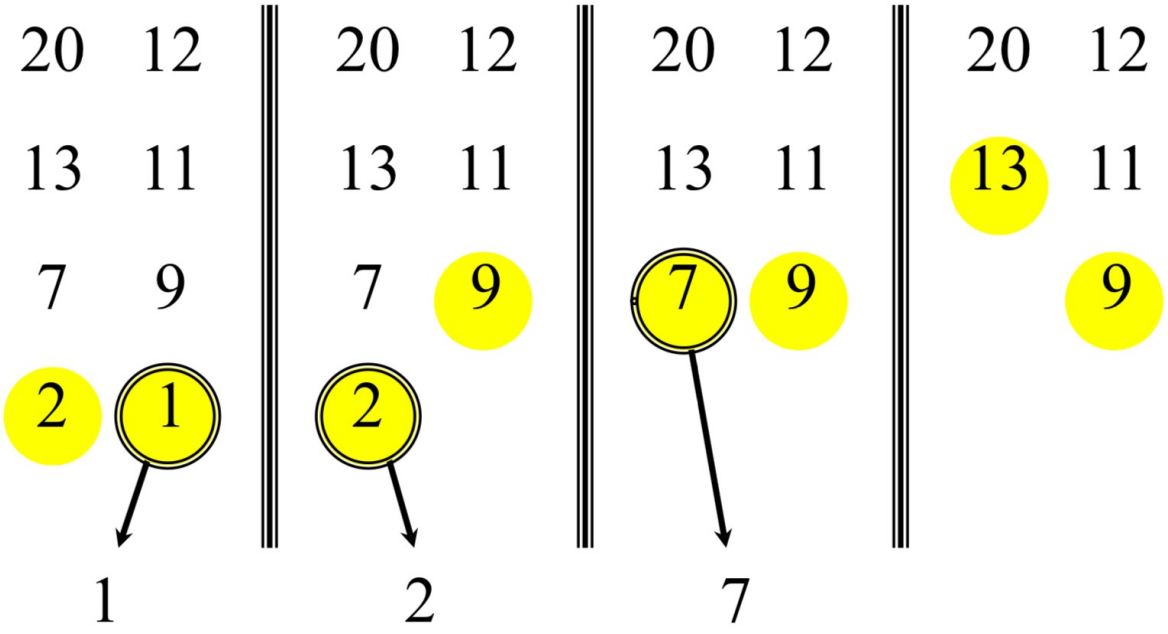

20 12
13 11
7 9
2 1
1

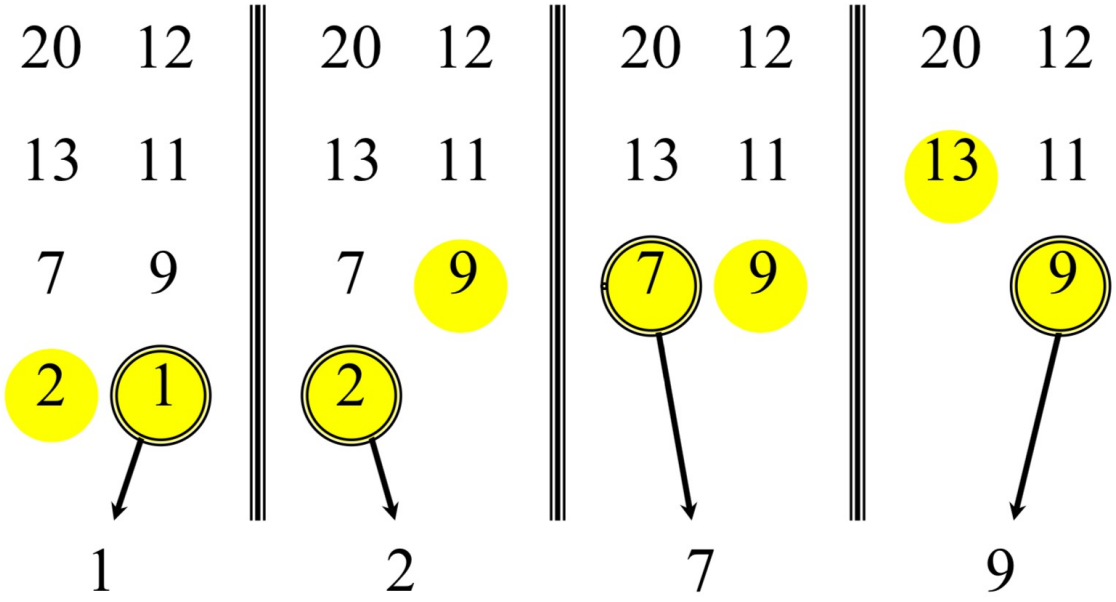

20 12
13 11
7 9
2 1
1
20 12
13 11
7 9
2

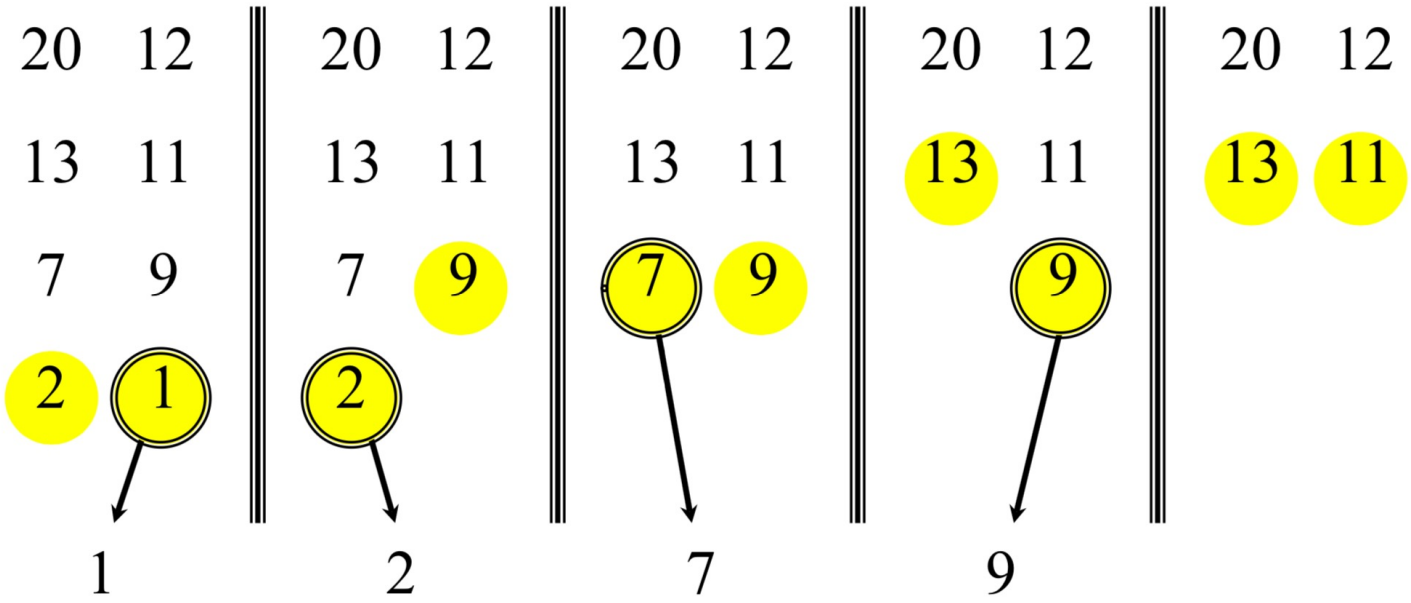

20 12
13 11
7 9
2 1
1
20 12
13 11
7 9
2
2

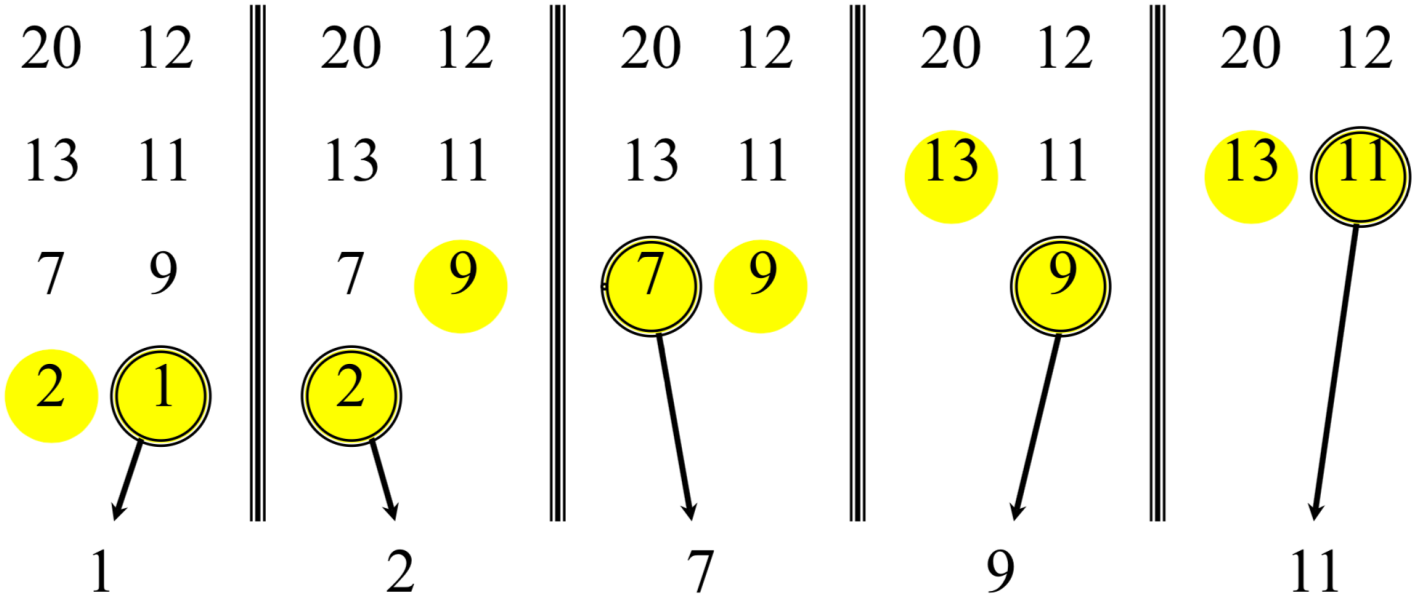

20 12
13 11
7 9
2 1
1
20 12
13 11
7 9
2
2
20 12
13 11
7 9

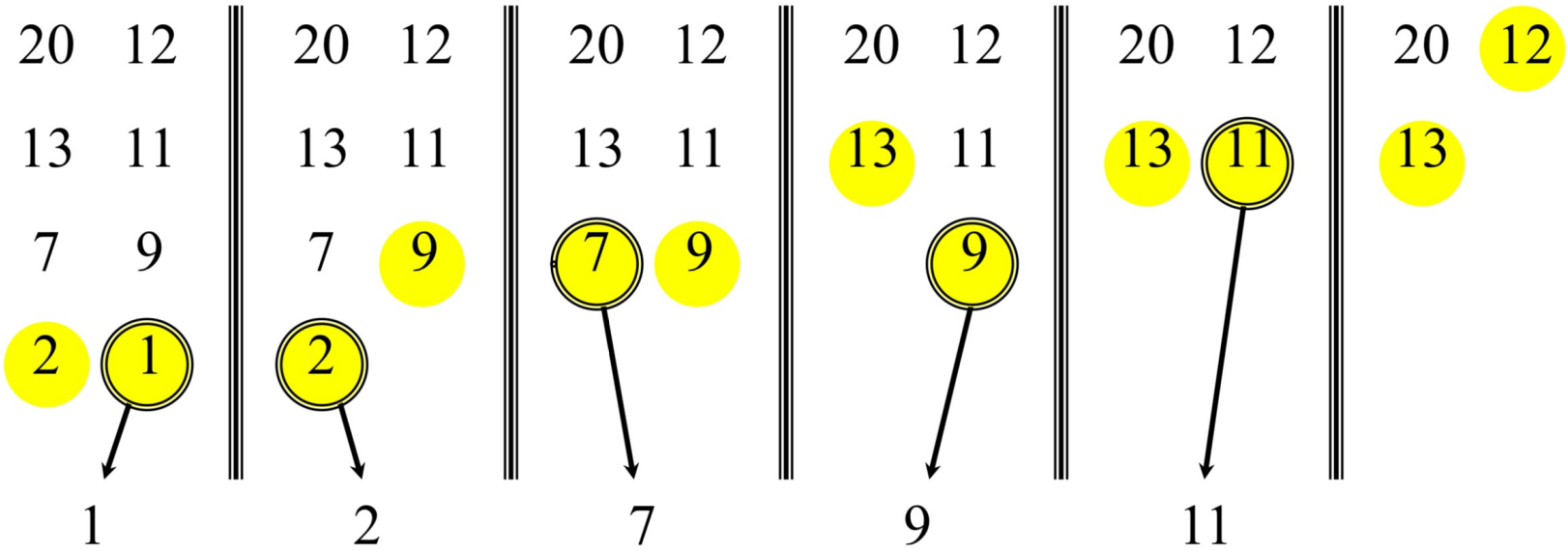

20 12
13 11
7 9
2 1
1
20 12
13 11
7 9
2
2
20 12
13 11
7 9
7
7

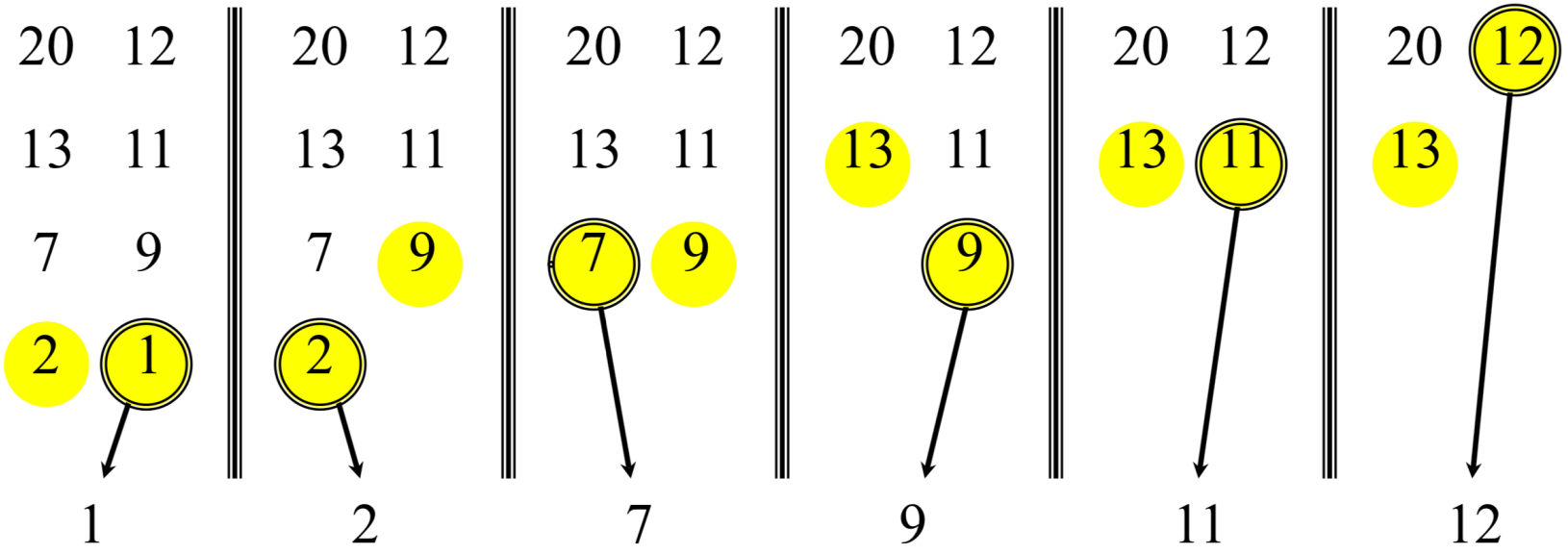

20 12
13 11
7 9
2 1
20 12
13 11
7 9
2
20 12
13 11
7 9
20 12
13 11
9
1
2
7

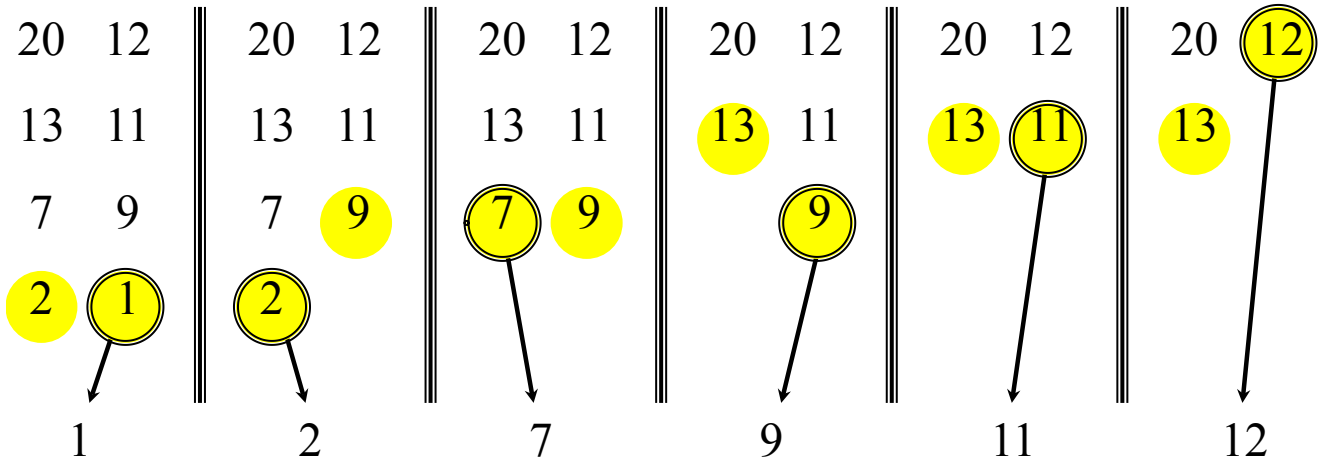

20 12
13 11
7 9
2 1
1
20 12
13 11
7 9
2
2
20 12
13 11
7 9
7
20 12
13 11
13
9
9


20 12
13 11
7 9
2 1
1
20 12
13 11
7 9
2
2
20 12
13 11
7 9
7
20 12
13 11
9
9
20 12
13 11


20 12
13 11
7 9
2 1
1
20 12
13 11
7 9
2
2
20 12
13 11
7 9
7
20 12
13 11
9
9
20 12
13 11
11
11


20 12
13 11
7 9
2 1
1
20 12
13 11
7 9
2
2
20 12
13 11
7 9
7
7
20 12
13 11
9
13
9
20 12
13 11
13 11
9
11
20 12
13
13

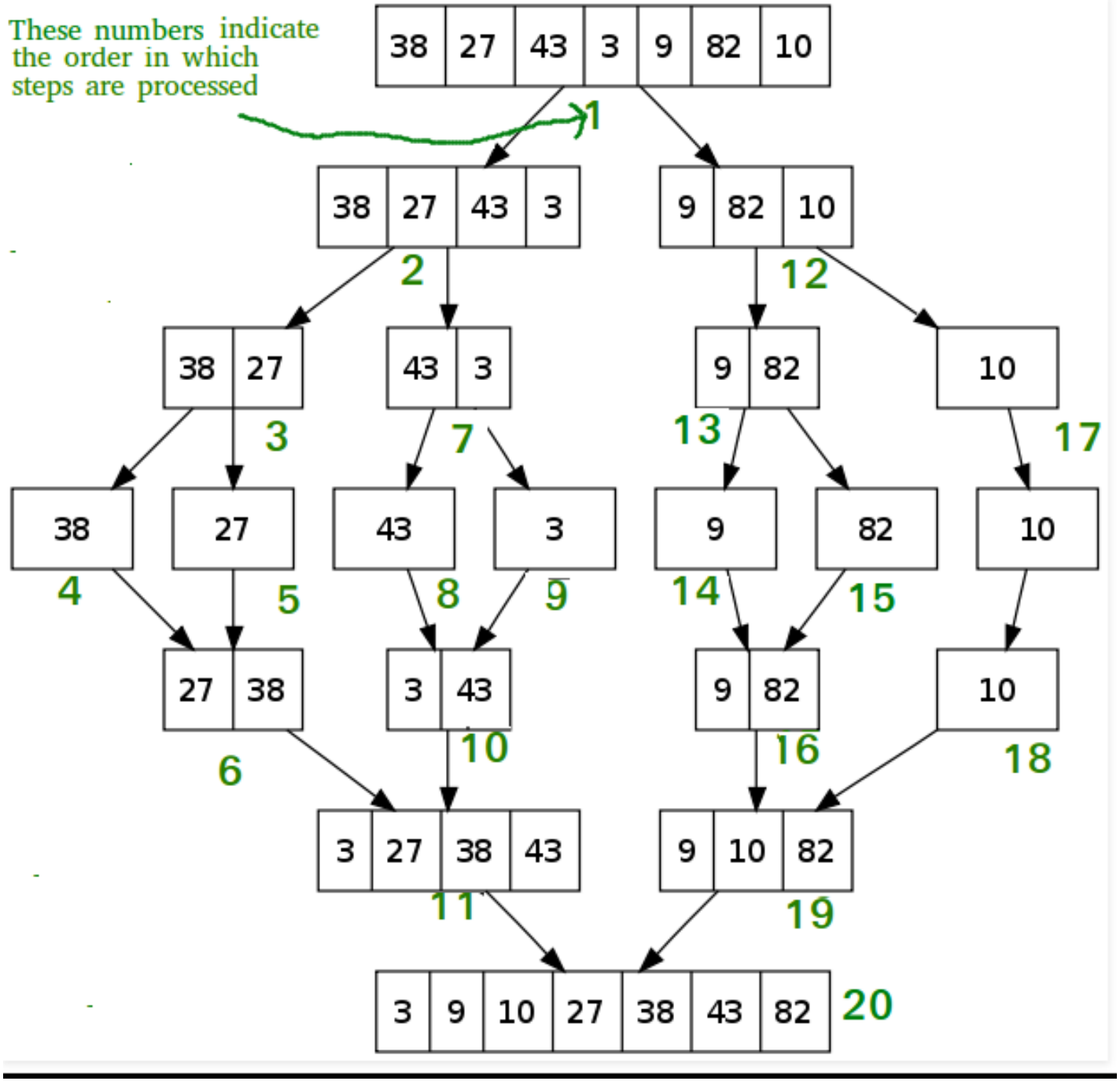

20 12
13 11
7 9
2 1
1
20 12
13 11
7 9
2
2
20 12
13 11
7 9
7
7
20 12
13 11
9
13
9
20 12
13 11
13 11
11
20 12
13
12
12



T(n) = Θ(n) to merge a total of n elements (linear time)

Merge Sort – Pseudocode

```
MERGE-SORT(A, p, r)
1   if p < r
2       q = ⌊(p + r)/2⌋
3       MERGE-SORT(A, p, q)
4       MERGE-SORT(A, q + 1, r)
5       MERGE(A, p, q, r)
```


These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
10
4
5
8
9
14
15
27 38
3 43
9 82
10
6
10
16
18
3 27 38 43
9 10 82
11
19
3 9 10 27 38 43 82
20

Merge Sort – Pseudocode

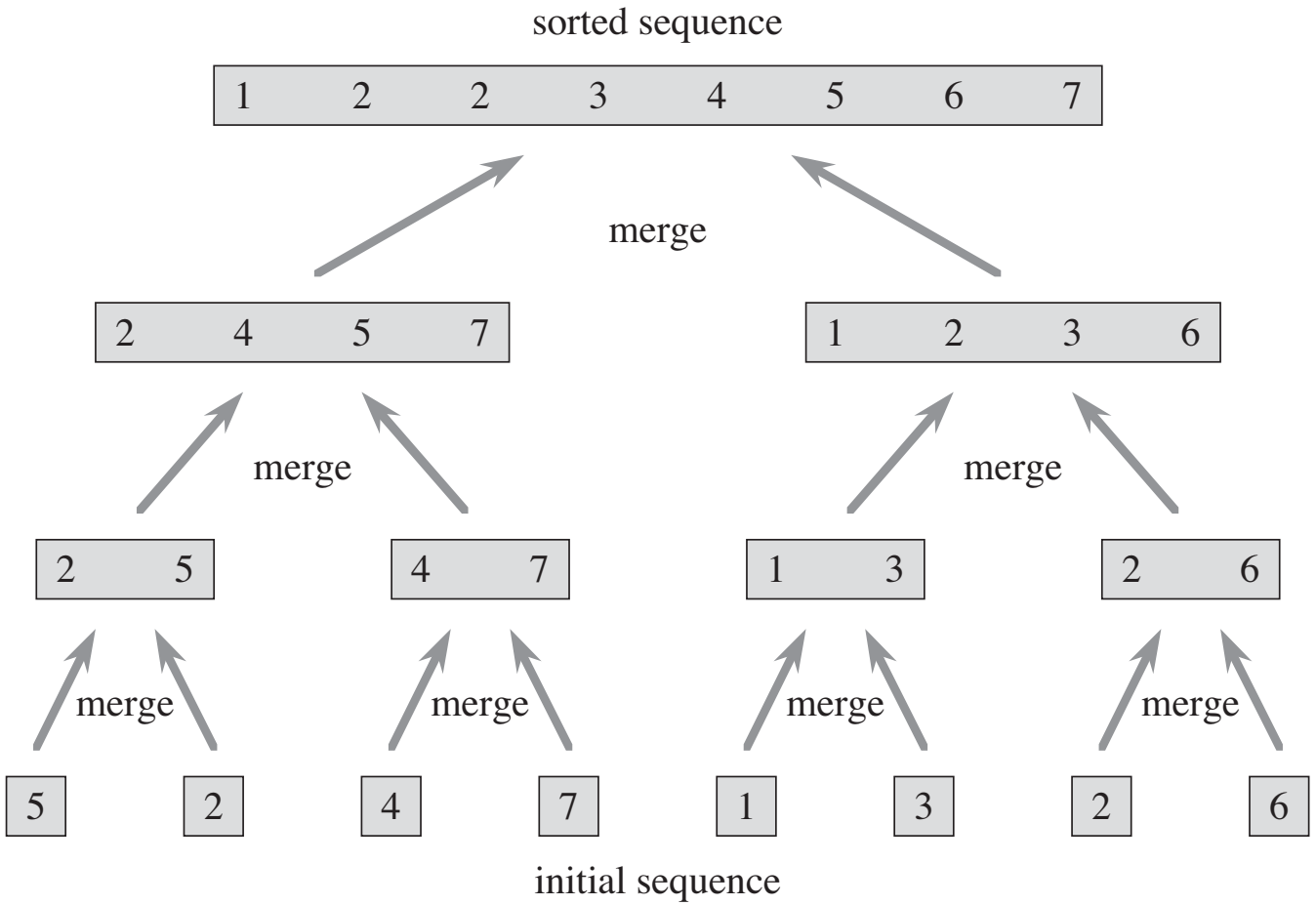
```
MERGE(A, p, q, r)
 1  n1 = q − p + 1
 2  n2 = r − q
 3  let L[1 .. n1 + 1] and R[1 .. n2 + 1] be new arrays
 4  for i = 1 to n1
 5      L[i] = A[p + i − 1]
 6  for j = 1 to n2
 7      R[j] = A[q + j]
 8  L[n1 + 1] = ∞
 9  R[n2 + 1] = ∞
10  i = 1
11  j = 1
12  for k = p to r
13      if L[i] ≤ R[j]
14          A[k] = L[i]
15          i = i + 1
16      else A[k] = R[j]
17          j = j + 1
```

```
merge ( array a, array b )
    array c

    while ( a and b have elements )
        if ( a[0] > b[0] )
            add b[0] to the end of c
            remove b[0] from b
        else
            add a[0] to the end of c
            remove a[0] from a

    // At this point either a or b is empty

    while ( a has elements )
        add a[0] to the end of c
        remove a[0] from a

    while ( b has elements )
        add b[0] to the end of c
        remove b[0] from b

    return c
```

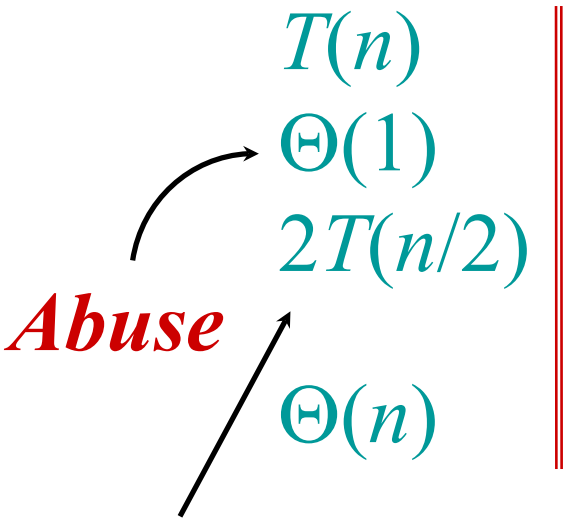
Example

- Illustrate the operation of merge sort on the array:

A = {5 , 2, 4, 7, 1, 3, 2, 6}



Merge Sort - Analysis



MERGE-SORT $A[1\,..\,n]$

1. If $n = 1$, done.
2. Recursively sort $A[\,1\,..\lceil n/2 \rceil\,]$ and $A[\,\lceil n/2 \rceil + 1\,..\,n\,]$.
3. ***"Merge"*** the 2 sorted lists

Sloppiness: Should be $T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor)$, but it turns out not to matter asymptotically.

Merge Sort - Analysis (Recurrence)

$$T(n) = \begin{cases} \Theta(1) \text{ if } n = 1; \\ 2T(n/2) + \Theta(n) \text{ if } n > 1. \end{cases}$$

- We shall usually omit stating the base case when $T(n) = \Theta(1)$ for sufficiently small n, but only when it has no effect on the asymptotic solution to the recurrence.

Recurrence Equation or Recurrence

When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation or recurrence, which describes the **overall running time on a problem of size n in terms of the running time on smaller inputs**

Merge Sort – Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

Merge Sort – Recursion Tree

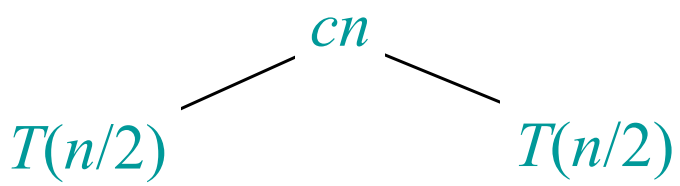
Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

$T(n)$

The total running time progressively expands to form a recursion tree

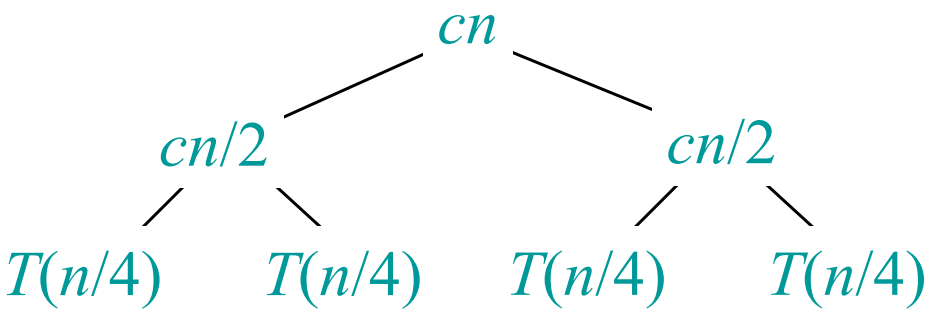
Merge Sort – Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



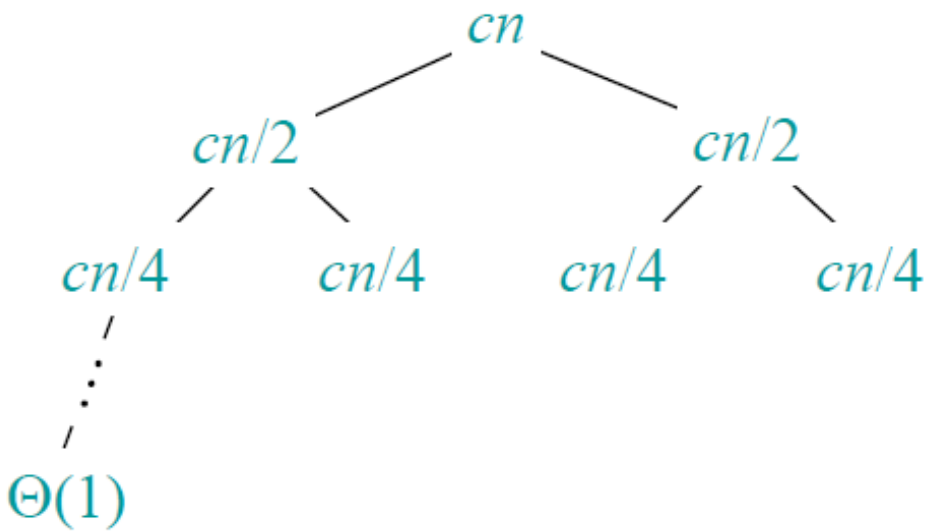
Merge Sort – Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



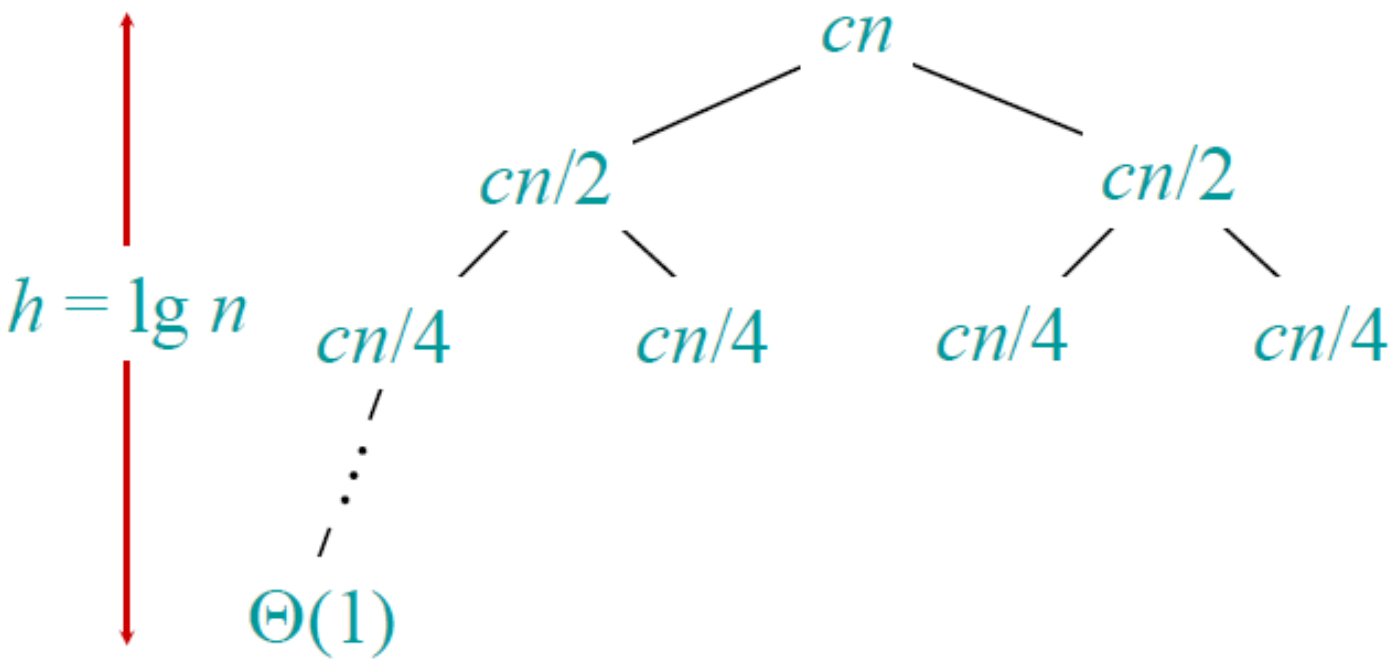
Merge Sort – Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



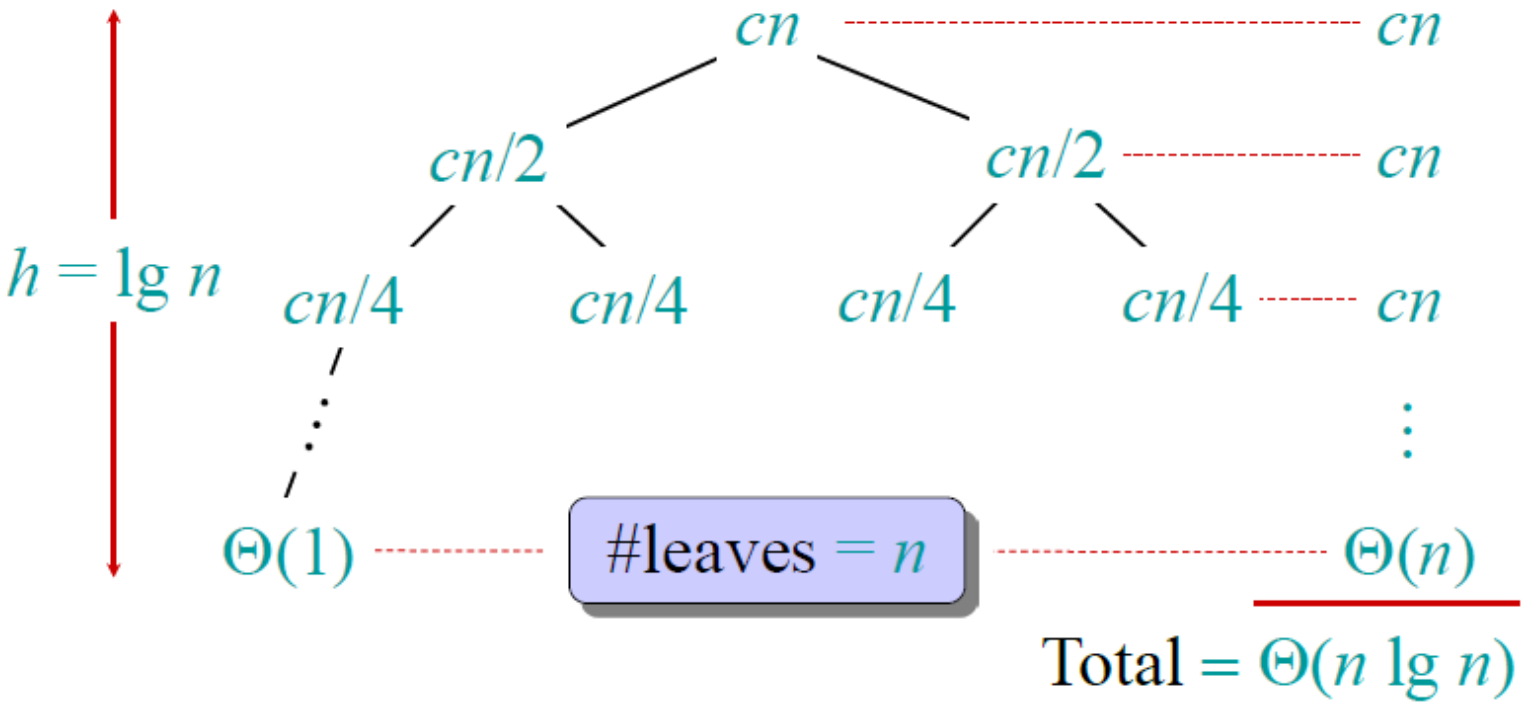
Merge Sort – Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.

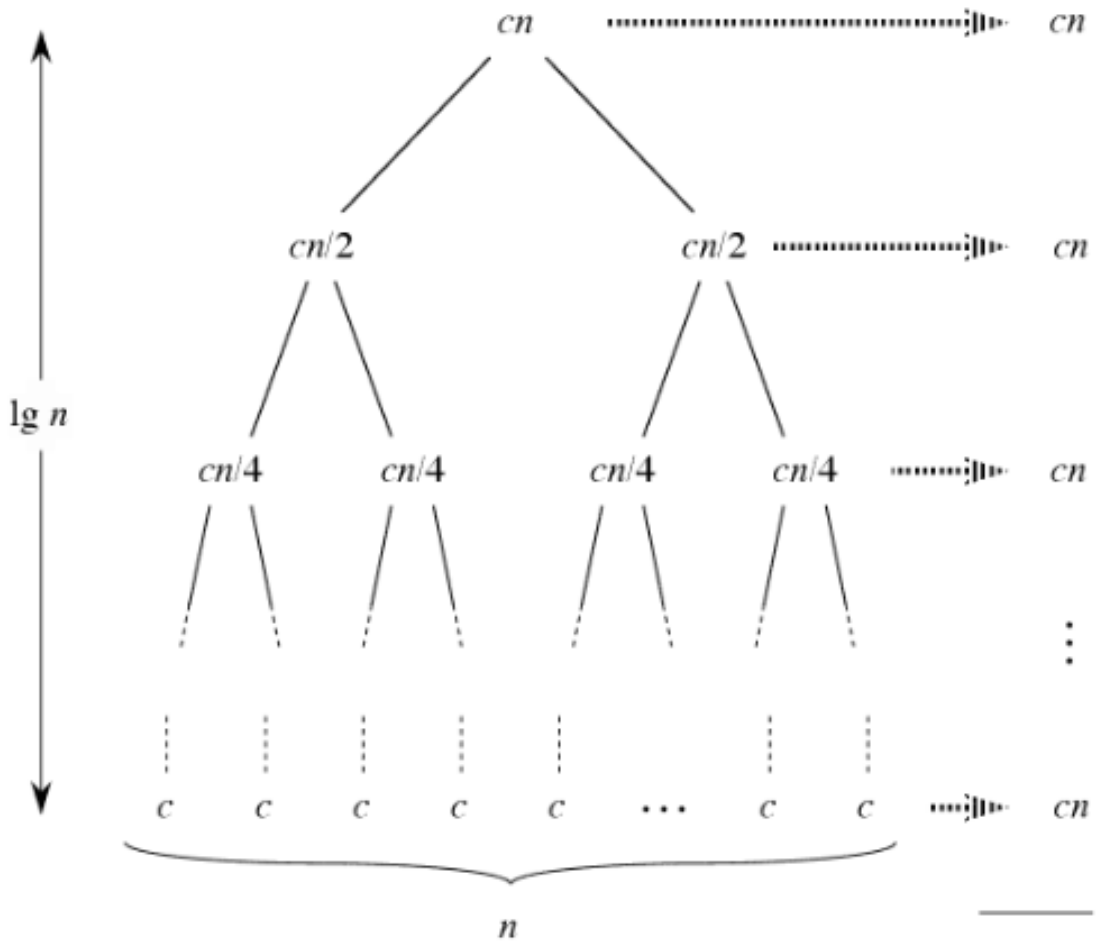


Merge Sort – Recursion Tree

Solve $T(n) = 2T(n/2) + cn$, where $c > 0$ is constant.



Recursion tree



Analyzing Run Time

- What is the level of the leaves?
- At a given level, j, exactly how **many distinct sub-problems are there**, as a function of the level j?
- For each of those distinct sub-problems at level j, what is the input size? (So, what is the size of the the array, which is passed to a sub-problem residing at level j of this recursion tree.)

Merge Sort

- $\Theta(n \lg n)$ grows more slowly than $\Theta(n^2)$.
- Therefore, merge sort asymptotically beats insertion sort in the worst case.
- In practice, merge sort beats insertion sort for $n > 30$ or so.

Divide and Conquer – Other Problems

- Quicksort
- Powering a number
- Fibonacci Numbers
- Matrix Multiplication (Strassen's Algorithm)

Quick Sort

- One of the most prevalent algorithms based on divide and conquer paradigm
- It is fast (as fast as Merge sort)
- It has the advantage of sorting in place (It does not require additional memory)

Quick Sort

- Partition an array around a pivot element
- For now let's select the first element as the pivot element
- Re-arrange the array such that:
  - All the elements at the left of the pivot are less than the pivot
  - All the elements at the right of the pivot are greater than the pivot

Quick Sort

Input:

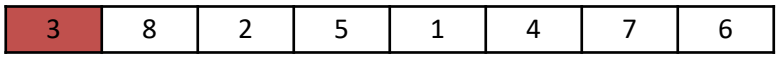
3	8	2	5	1	4	7	6

Output:

1	2	3	4	5	6	7	8

Quick Sort

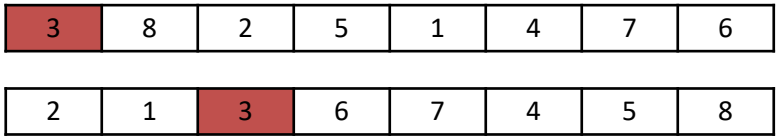
Input:



- Key idea: partition the array around the pivot element
- Re-arrange array so that
  - Left of pivot => less than pivot
  - Right of pivot => greater than pivot
- Puts pivot in its rightful position

Quick Sort

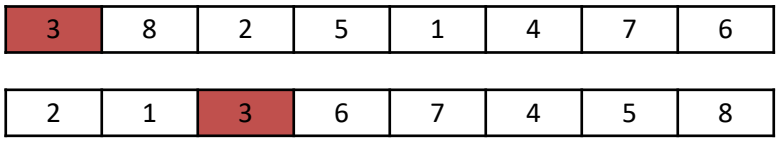
Input:



- Key idea: partition the array around the pivot element
- Re-arrange array so that
  - Left of pivot => less than pivot
  - Right of pivot => greater than pivot
- Puts pivot in its rightful position

Quick Sort

Input:



- Partitioning:
  - It is linear (O (n))
  - It reduces the problem size
    - Enables divide and conquer

Quick Sort

Quicksort an n-element array:

1. ***Divide:*** Partition the array into two subarrays around a ***pivot*** x such that elements in lower subarray $\leq x \leq$ elements in upper subarray.

2. ***Conquer:*** Recursively sort the two subarrays.
3. ***Combine:*** Trivial.

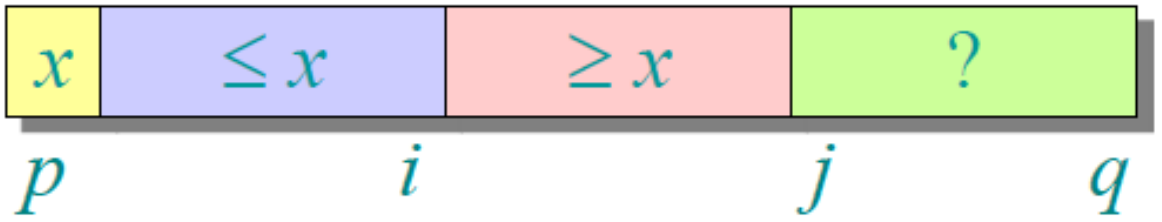
Key: *Linear-time partitioning subroutine.*

Partitioning Subroutine

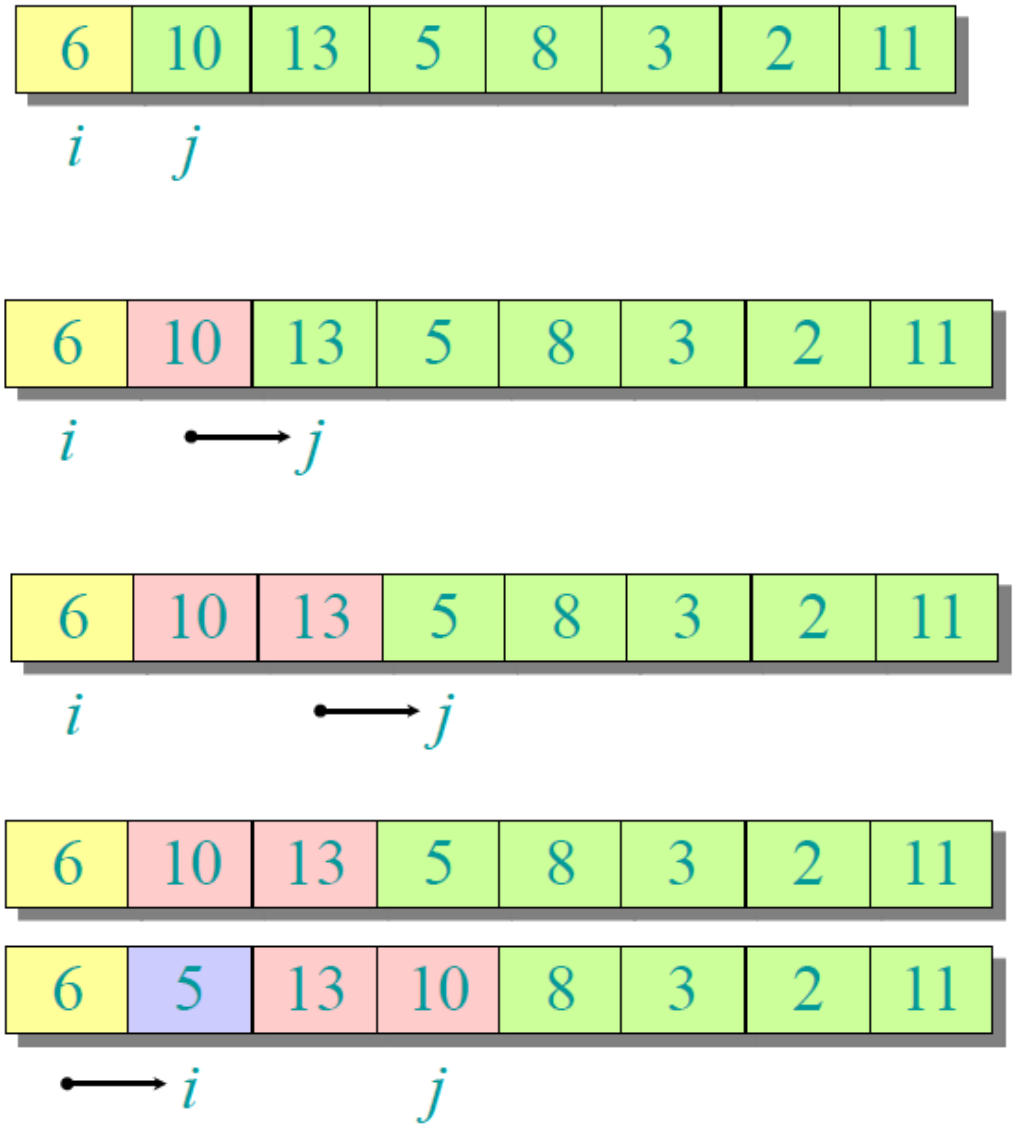
```
PARTITION(A, p, q)   ▷ A[p . . q]
    x ← A[p]         ▷ pivot = A[p]
    i ← p
    for j ← p + 1 to q
        do if A[j] ≤ x
                then  i ← i + 1
                      exchange A[i] ↔ A[j]
    exchange A[p] ↔ A[i]
    return i
```

Running time $= O(n)$ for n elements.

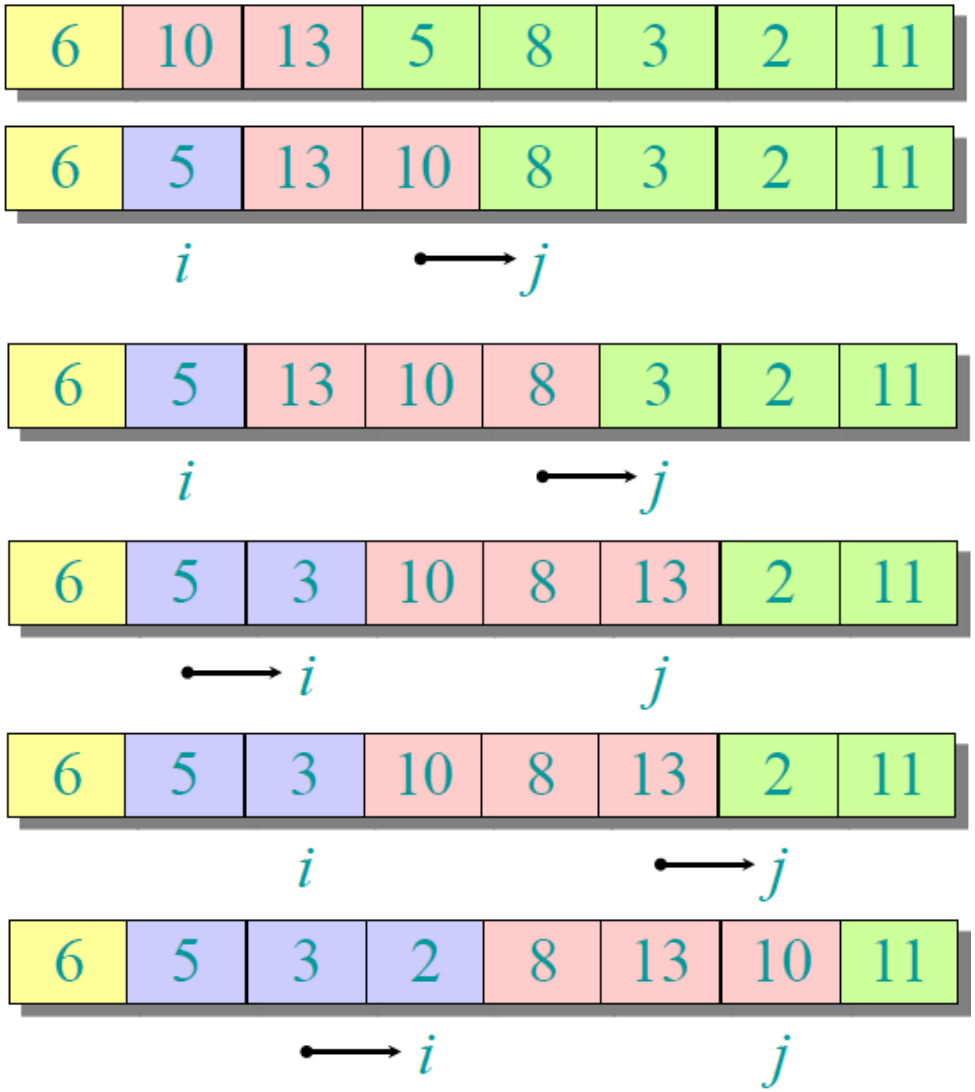
Invariant:



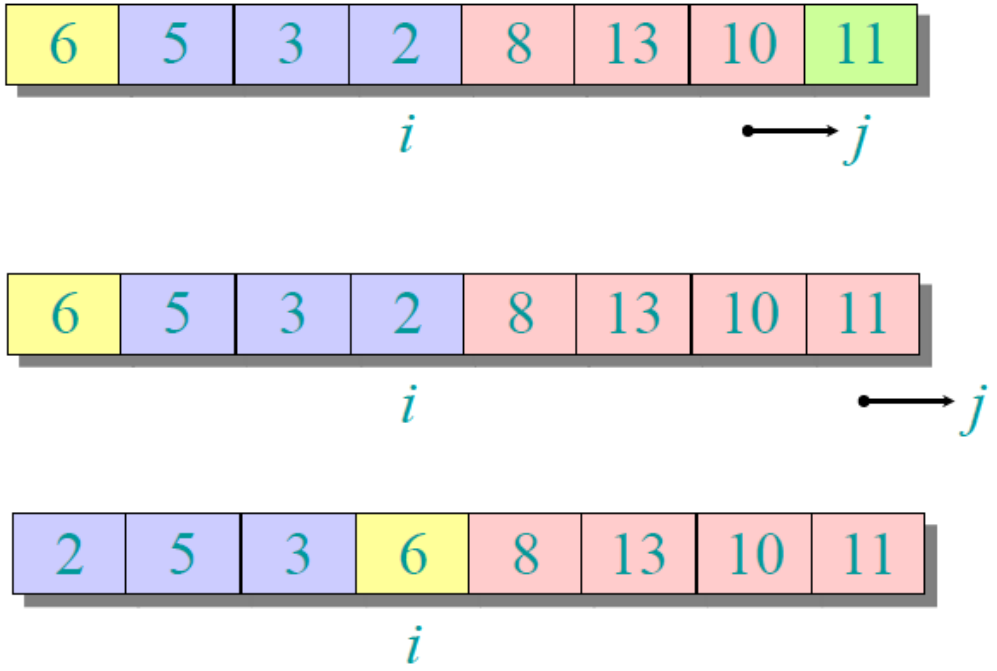
Example of Partitioning



Example of Partitioning



Example of Partitioning



Pseudo Code for Quick Sort

```
QUICKSORT(A, p, r)
    if p < r
        then q ← PARTITION(A, p, r)
             QUICKSORT(A, p, q–1)
             QUICKSORT(A, q+1, r)
```

Initial call: QUICKSORT(*A*, 1, *n*)

Quick Sort

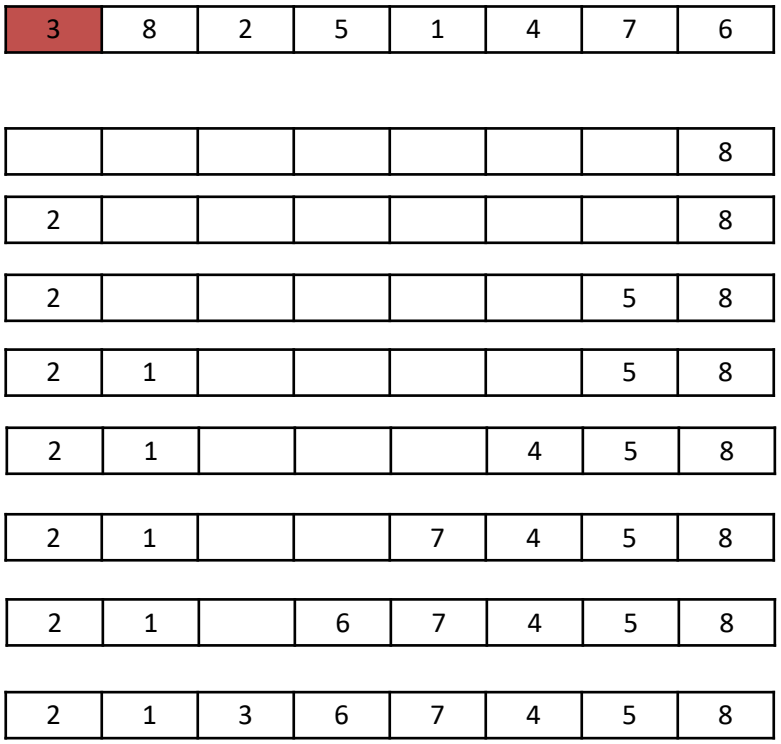
High level description:

Quicksort(array, length n)

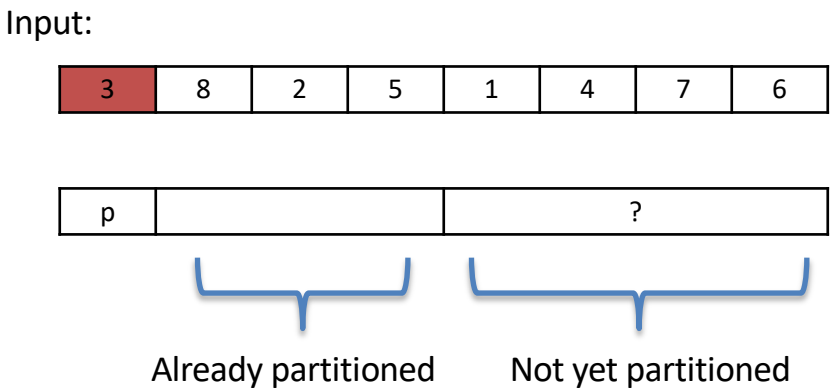
- If n ==1 return
- p = choose pivot (array, n)
- Partition array around p
- Recursively sort 1s part
- Recursively sort 2nd part

Quick sort – partitioning around a pivot – additional array

Input:

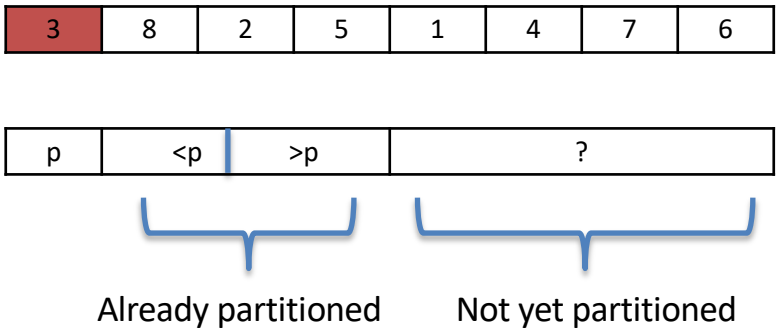


Quick sort – partitioning around a pivot – in-place



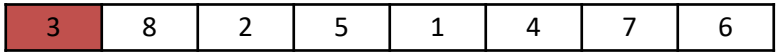
Quick sort – partitioning around a pivot – in-place

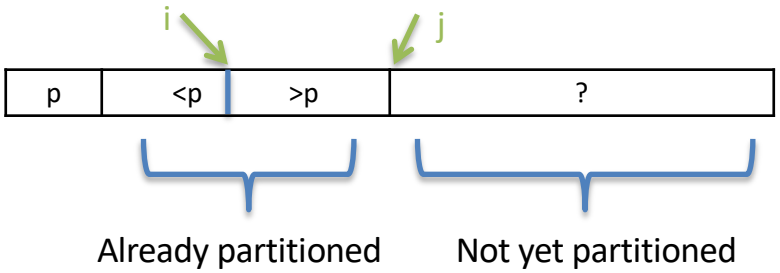
Input:



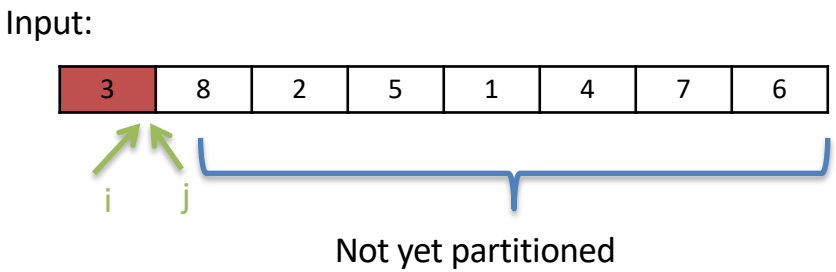
Quick sort – partitioning around a pivot – in-place

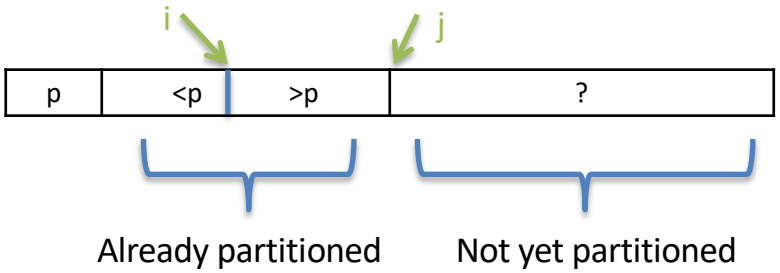
Input:



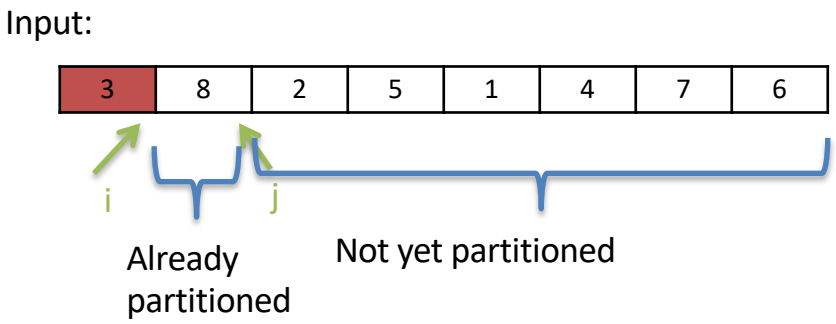


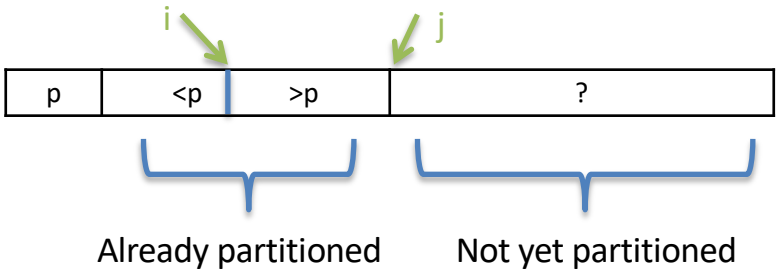
Quick sort – partitioning around a pivot – in-place



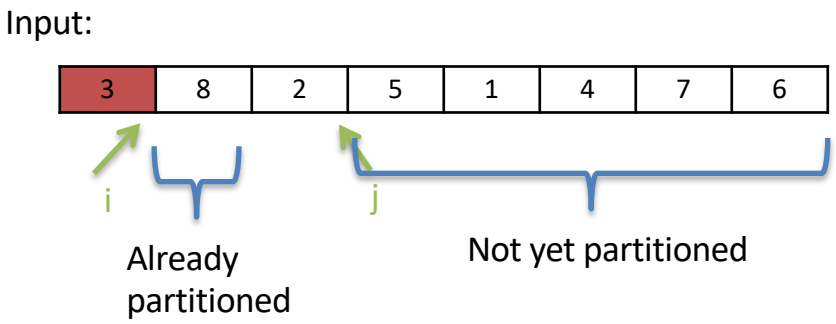


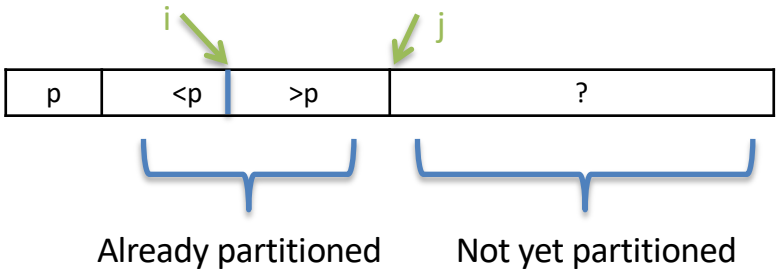
Quick sort – partitioning around a pivot – in-place



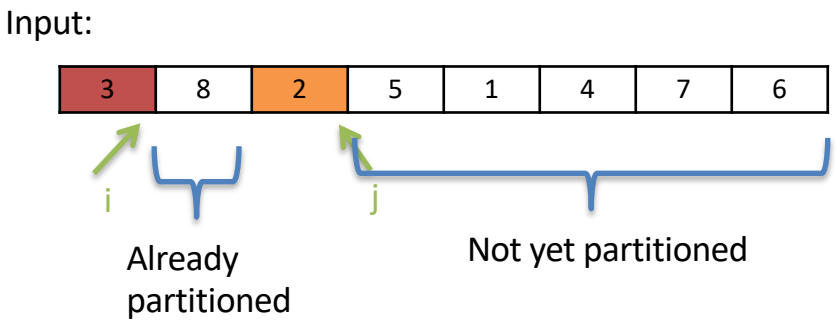


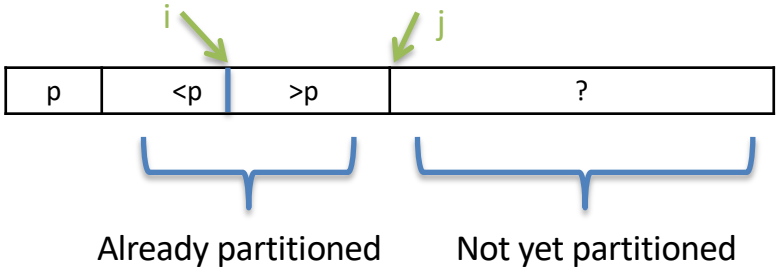
Quick sort – partitioning around a pivot – in-place



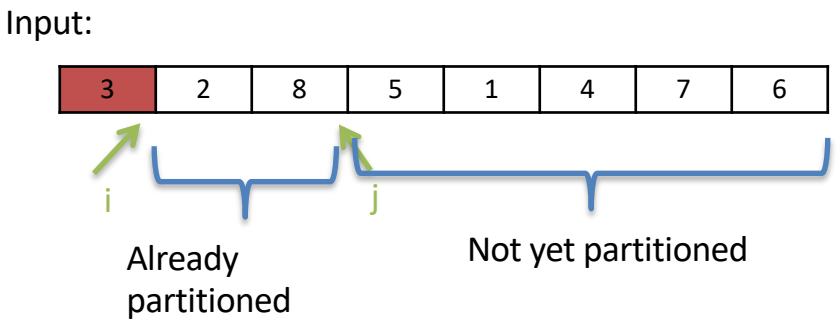


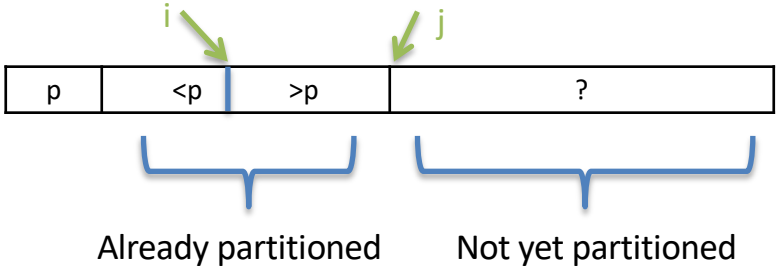
Quick sort – partitioning around a pivot – in-place



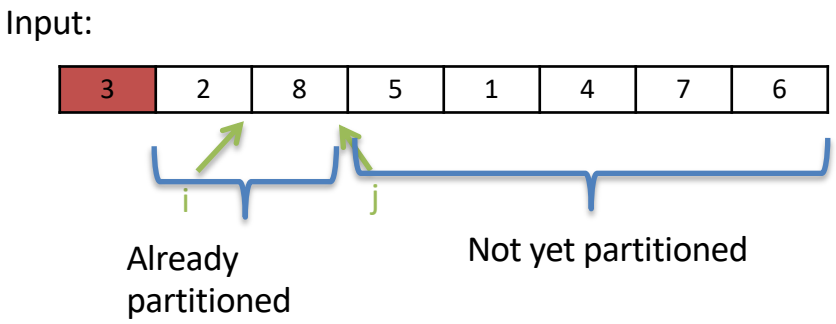


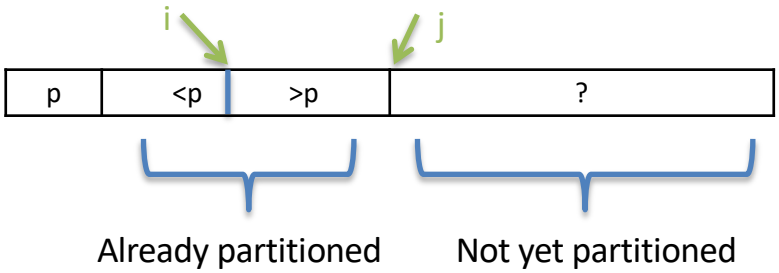
Quick sort – partitioning around a pivot – in-place



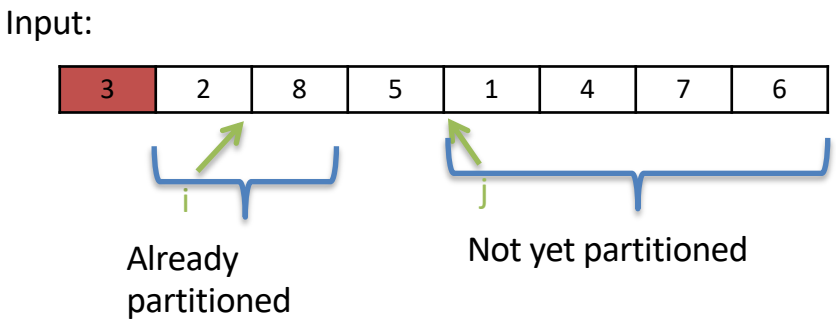


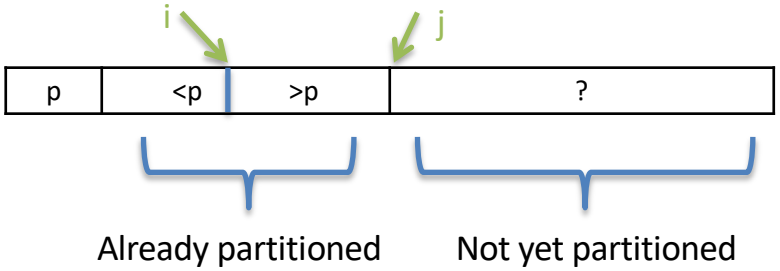
Quick sort – partitioning around a pivot – in-place



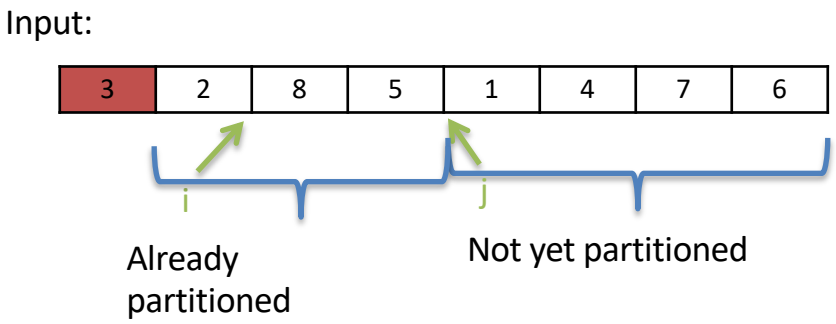


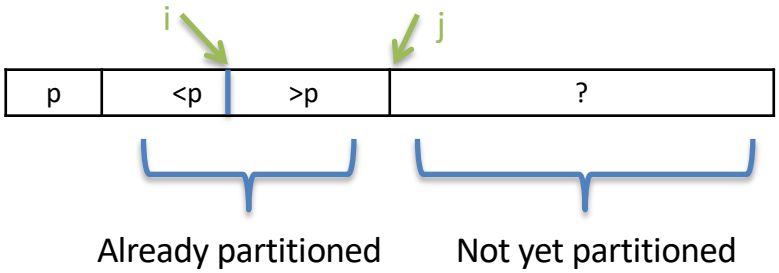
Quick sort – partitioning around a pivot – in-place



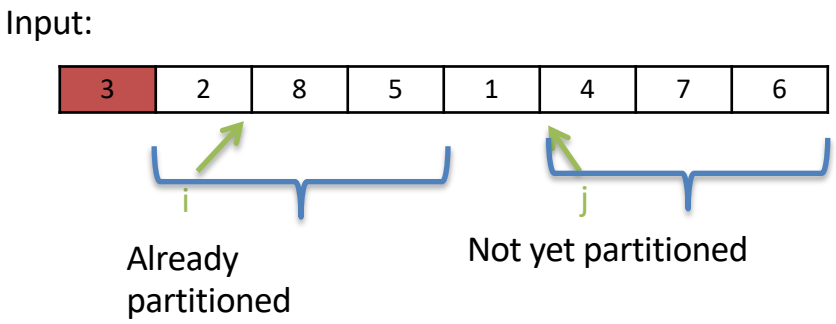


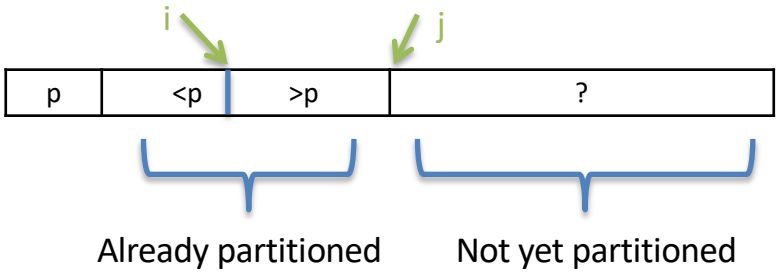
Quick sort – partitioning around a pivot – in-place



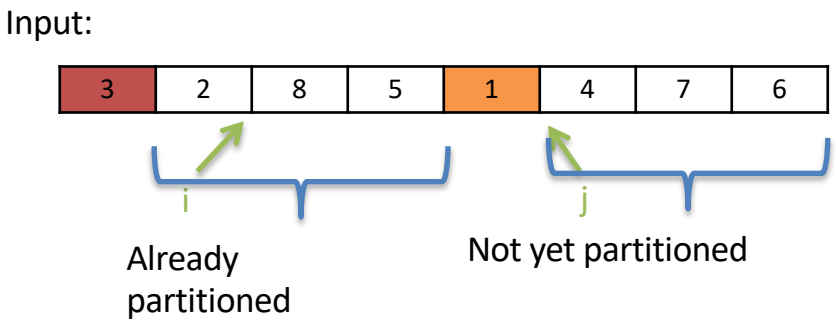


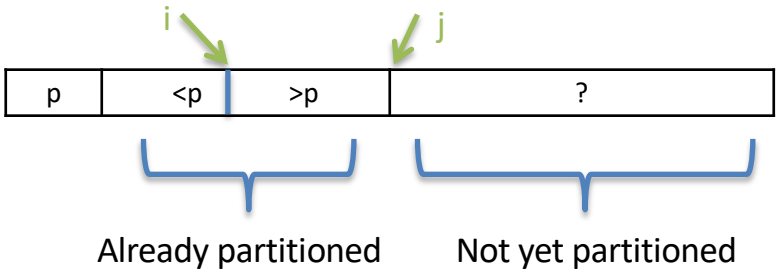
Quick sort – partitioning around a pivot – in-place



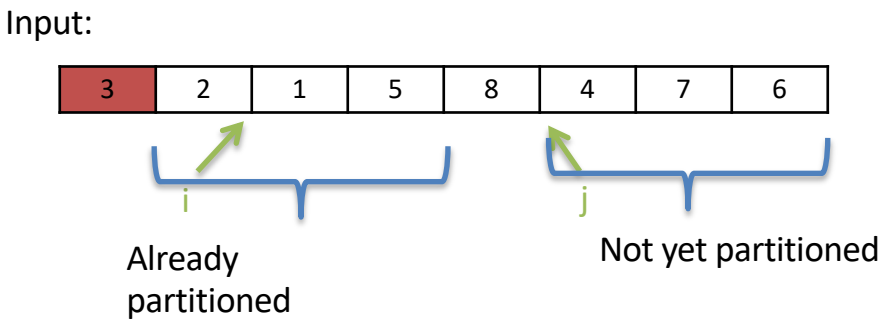


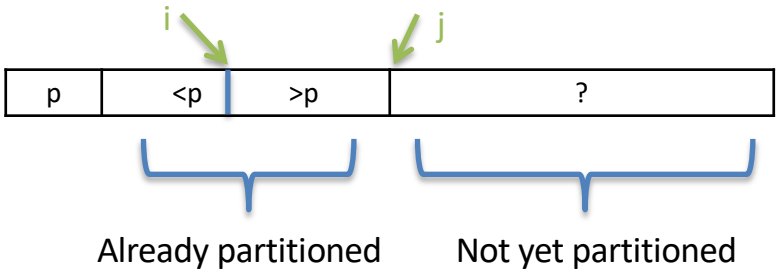
Quick sort – partitioning around a pivot – in-place



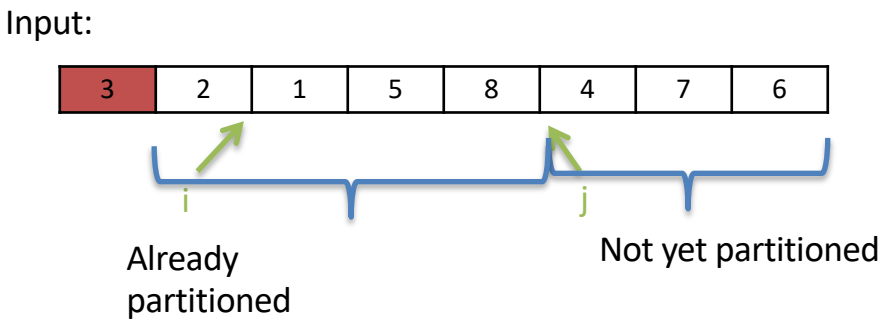


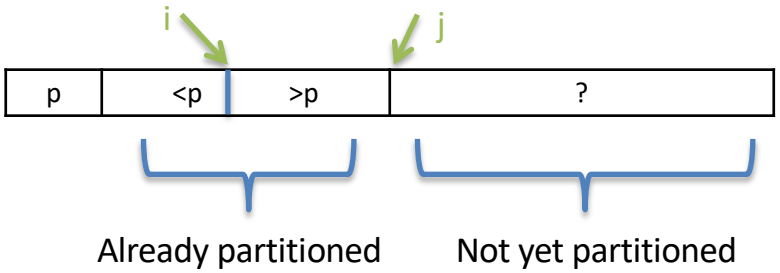
Quick sort – partitioning around a pivot – in-place



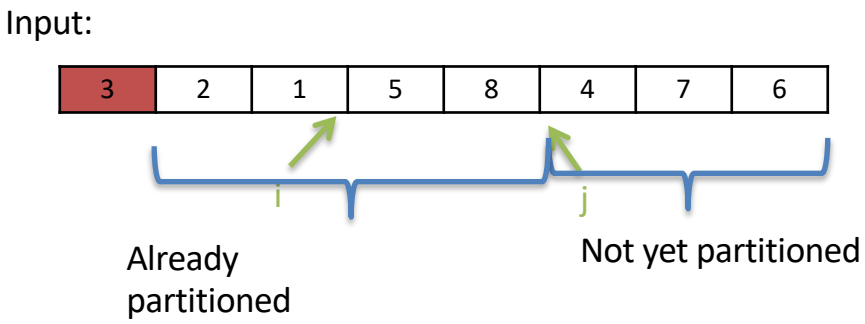


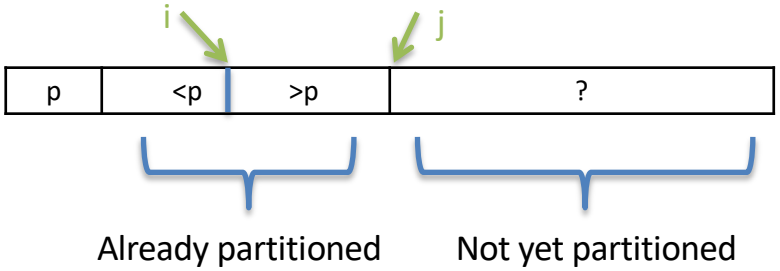
Quick sort – partitioning around a pivot – in-place



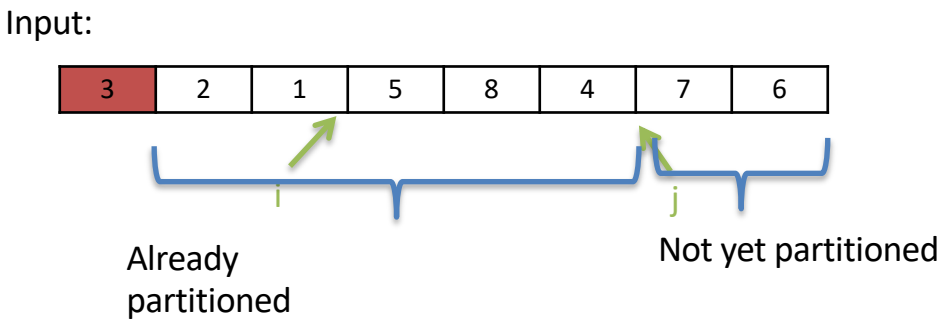


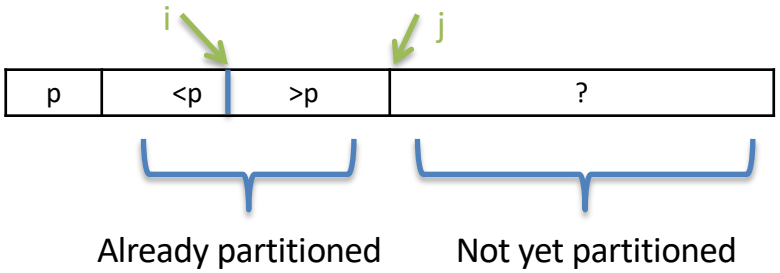
Quick sort – partitioning around a pivot – in-place



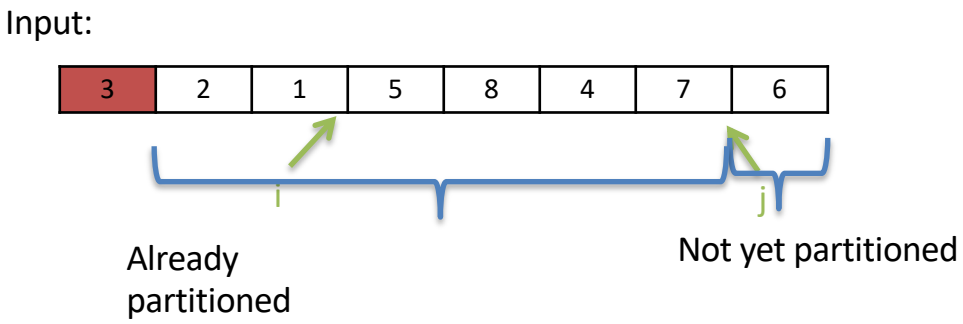


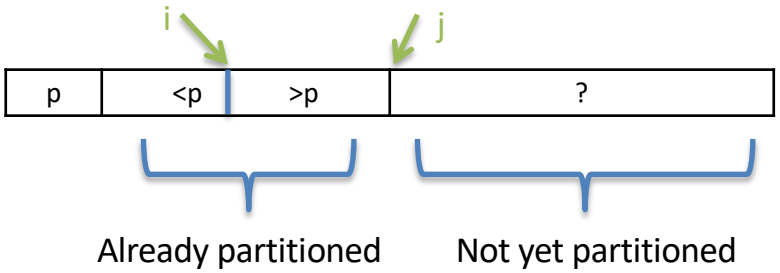
Quick sort – partitioning around a pivot – in-place



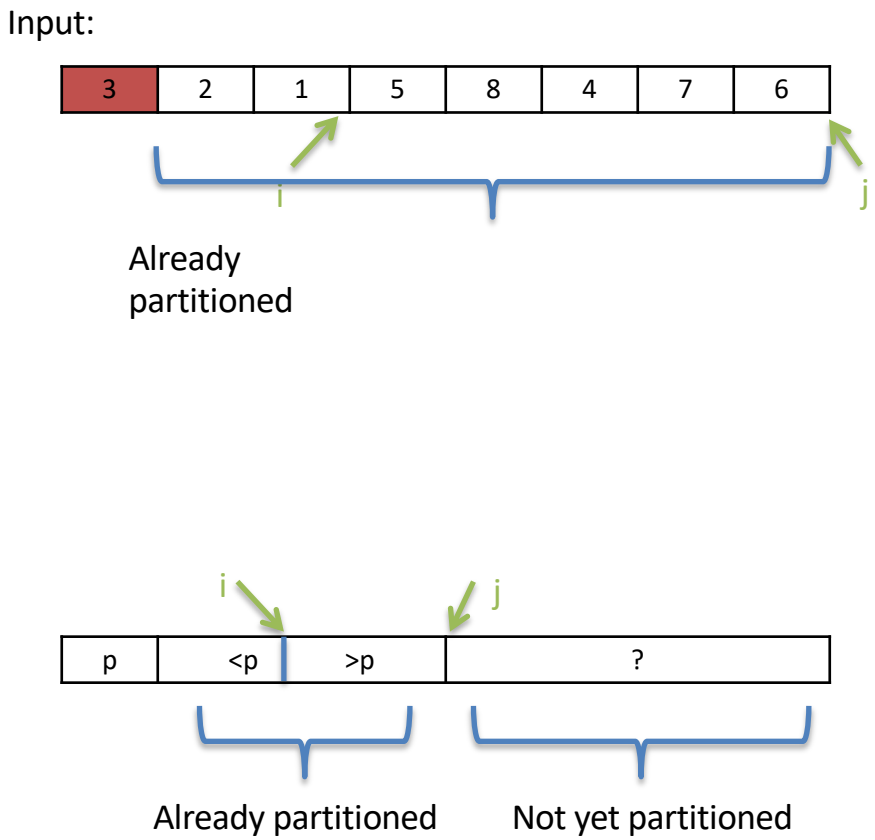


Quick sort – partitioning around a pivot – in-place



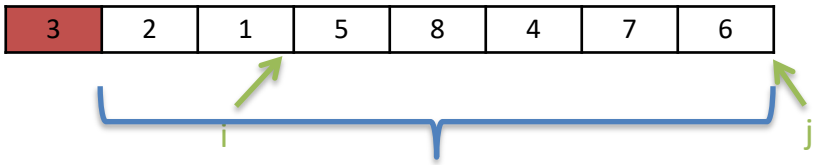


Quick sort – partitioning around a pivot – in-place



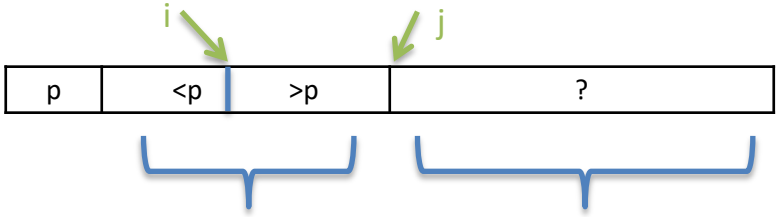
Quick sort – partitioning around a pivot – in-place

Input:



Already partitioned

now take care of the pivot position

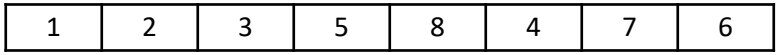


Already partitioned

Not yet partitioned

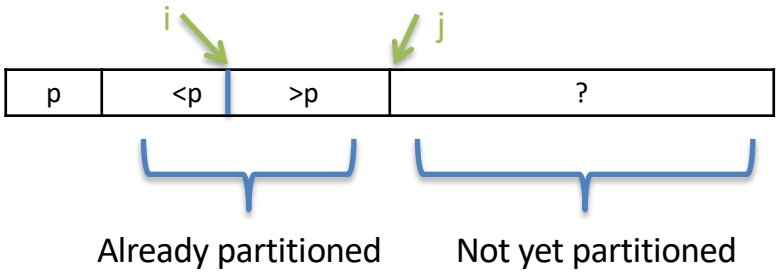
Quick sort – partitioning around a pivot – in-place

Input:



Already
partitioned

now take care of the pivot position



Analysis of Quick Sort

- Assume all input elements are distinct.
- In practice, there are better partitioning algorithms for when duplicate input elements may exist.
- Let $T(n)$ = worst-case running time on an array of n elements.

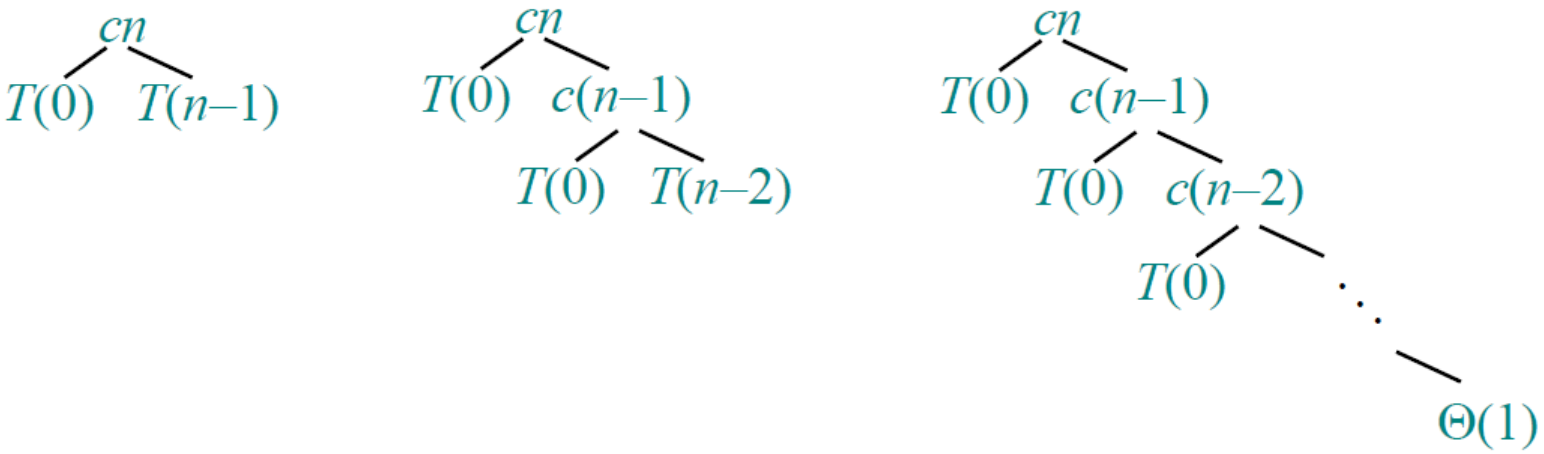
Worst Case of Quick Sort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

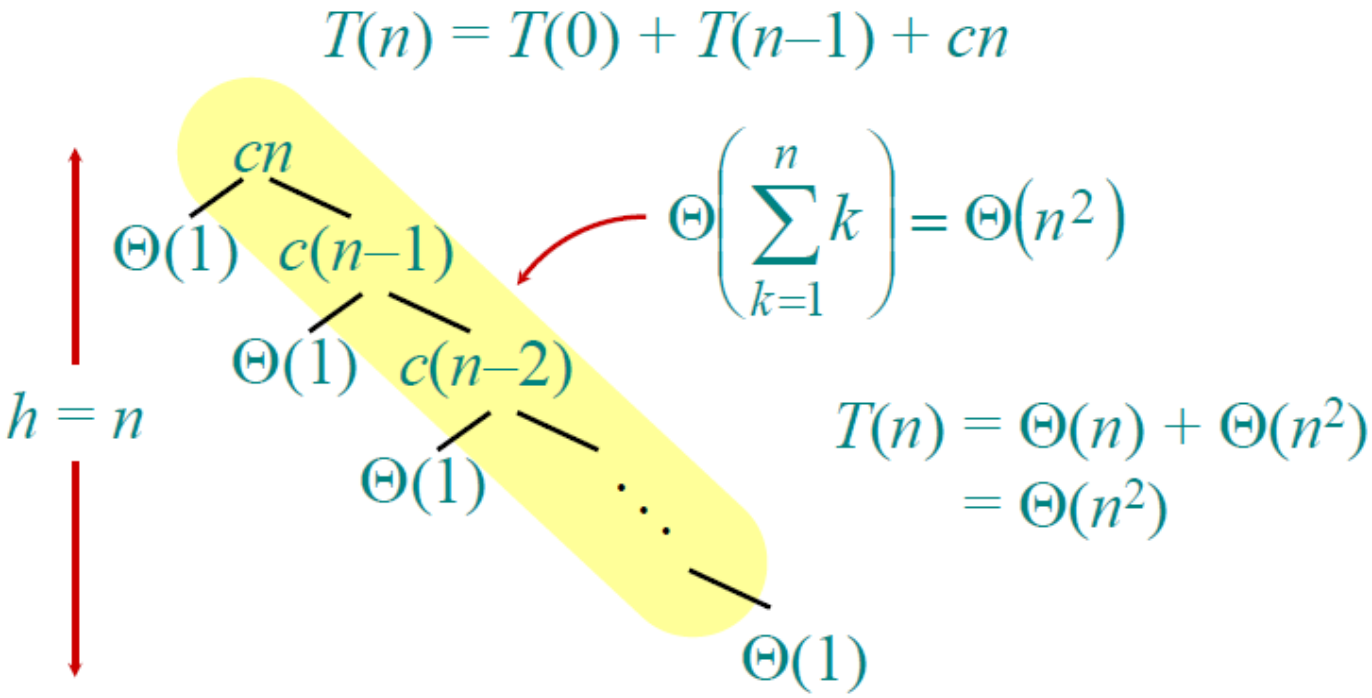
$$
\begin{aligned}
T(n) &= T(0) + T(n-1) + \Theta(n) \\
&= \Theta(1) + T(n-1) + \Theta(n) \\
&= T(n-1) + \Theta(n) \\
&= \Theta(n^2) \qquad \textit{(arithmetic series)}
\end{aligned}
$$

Worst Case Recursion Tree

$$T(n) = T(0) + T(n-1) + cn$$



Worst Case Recursion Tree



Best-case analysis

If we're lucky, PARTITION splits the array evenly:

$$T(n) = 2T(n/2) + \Theta(n)$$
$$= \Theta(n \lg n) \qquad \text{(same as merge sort)}$$

In the Average Case, PARTITION produces a mix of "good" and "bad" splits. The running time of quicksort, when levels alternate between good and bad splits is like the running time for good splits alone: still O(n lg n), but with a slightly larger constant hidden by the O-notation

Exercise

- Illustrate the operation of merge sort on the array:

A = {3 , 41, 52, 26, 38, 57, 9, 49}

Exercise

- Illustrate the operation *Partition* on the array :

A = { 13, 19, 9, 5, 12, 8, 7, 4, 11, 2, 6, 21}

Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$.

Naive algorithm: $\Theta(n)$.

Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$.

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

Powering a number

Problem: Compute a^n, where $n \in \mathbb{N}$.

Naive algorithm: $\Theta(n)$.

Divide-and-conquer algorithm:

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & \text{if } n \text{ is even;} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & \text{if } n \text{ is odd.} \end{cases}$$

$$T(n) = T(n/2) + \Theta(1) \Rightarrow T(n) = \Theta(\lg n) .$$