Algorithm Analysis

Brute-Force, Divide and Conquer

Overview

- Brute-force. Why?
  - Normally **straightforward**.
  - Not too slow for some applications.
  - **Benchmark** of the more well-designed algorithms.
  - Brute-force attack -> check every possible password.
- Brute-force algorithms.
  - Maximum/minimum searching
  - Sorting
  - Closest pairing
  - Convex hull
  - Exhaustive search
    - Assignment problem
  - Back tracking
    - Assignment problem
    - Subset sum problem

Maximum & Minimum Searching

- Max and min of a collection (array/sequence).
- $A[pos]$ is the maximum item in $A[0], A[1], \cdots, A[n-1]$.
- Time complexity?
  - $O(n)$: all the n items have been traversed.

$$
\begin{array}{l}
\text{max}(A, n)\ \{ \\
\quad pos = 0 \\
\quad \text{for } (i = 1; i < n;\ i++)\ \{ \\
\quad\quad \text{if } (A[i] > A[pos])\ \{ \\
\quad\quad\quad pos = i \\
\quad\quad \} \\
\quad \} \\
\quad \text{return } pos \\
\}
\end{array}
$$

- Search an item.
- Running time:
  - **Best**: element is at position 0, so one comparison.
  - **Worst**: element last pos., or not in the collection.
  - **Average**: depends on the probability.
    - If normally distributed, $O\left(\frac{n}{2}\right) = O(n)$.
  - Overall: $O(n)$.

$$
\begin{array}{l}
\text{search}(A, n, val)\ \{ \\
\text{for}\ \ (i = 0; i < n;\ i++)\ \{ \\
\quad\quad \text{if } (val == A[i])\ \{ \\
\quad\quad\quad \text{return } i \\
\quad\quad \} \\
\quad \} \\
\}
\end{array}
$$

Brute-Force Sorting Methods

- **Selection sort**: searching max in the rest items.
- Descending or ascending?
- Time complexity:
  - $O(n) + O(n-1) + \cdots + O(1) = O(n^2)$.

```
selectionSort(array, size)
  repeat (size - 1) times
  set the first unsorted element as the minimum
  for each of the unsorted elements
    if element < currentMinimum
      set element as new minimum
  swap minimum with first unsorted position
end selectionSort
```

```
void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;

      for (int i = step + 1; i < size; i++) {
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])  min_idx = i;
    }

    // put min at the correct position
    if (min_idx != step)
        swap(&array[min_idx], &array[step]);
  }
}
```

https://www.youtube.com/watch?v=xWBP4lzkoyM

https://en.wikipedia.org/wiki/Insertion_sort; https://media.geeksforgeeks.org/wp-content/uploads/insertionsort.png

Additional remarks about selection sort

```
void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;

      for (int i = step + 1; i < size; i++) {
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])  min_idx = i;
    }

    // put min at the correct position
    if (min_idx != step) // missing stmt in old version
      swap(&array[min_idx], &array[step]);
  }
}
```

- Min number of swap involving any particular item is 0
- with a sorted list, no swap is required.
- For example the particular item is 5
- 5, 7, 10, 13, 15
- min_idx = step = 0;
- The i loop will not change min_idx which remains as step=0
- Since min_idx is still the same as step, no swap will occur

Additional remarks about selection sort

```
void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;

      for (int i = step + 1; i < size; i++) {
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])  min_idx = i;
    }

    // put min at the correct position
    if (min_idx != step)
        swap(&array[min_idx], &array[step]);
  }
}
```

- Max number of swap involving any particular item is 1
- For example the particular item is 5
- 15, 7, 10, 13, 5
- For step=0
  - min_idx=step=0 which is 15
  - The i loop will move from step+1 which is 7,10,13,5 and found 5 as the min (min_idx=4)
  - Since min_idx is now 4 and <> step=0, we swap 15 and 5 (1 swap)
  - The result is 5, 7, 10, 13, 15
- For step=1
  - min_idx=step=1 which is 7
  - The i loop will move from step+1 which is 10,13,5
  - The particular item 5 will never be touched again thus no more swap will occur

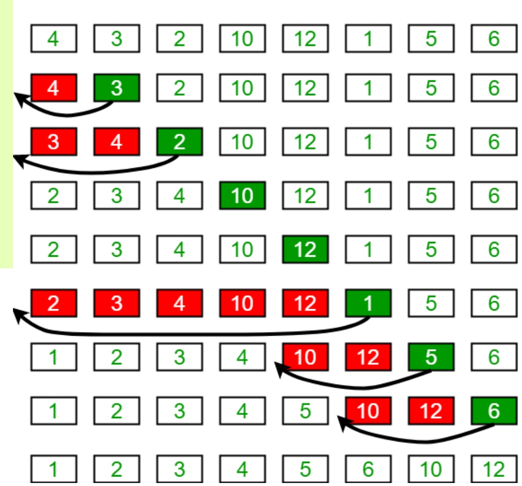
Brute-Force Sorting Methods

- **Insertion sort**
  - An array of len m.
  - Sort the first $m-1$ items.
  - Insert the last into the sorted first $m-1$ items.
  - $T(n) = T(n-1) + O(n)$ -> $O(n \times n) = O(n^2)$.

```
insertionSort(array)
  mark first element as sorted
  for each unsorted element X
    'extract' the element X
    for j <- lastSortedIndex down to 0
      if current element j > X
        move sorted element to the right by 1
    break loop and insert X here
end insertionSort
```

```
void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;

    // Compare key with elements on left until an element smaller
    // shift bigger to the right
    while (key < array[j] && j >= 0) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
```

Insertion Sort Execution Example

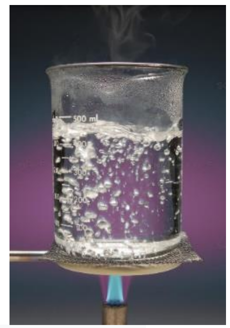


https://www.youtube.com/watch?v=OGzPmgsI-pQ

https://en.wikipedia.org/wiki/Insertion_sort; https://media.geeksforgeeks.org/wp-content/uploads/insertionsort.png

Brute-Force Sorting Method

- **Bubble sort.**
  - Large val flows to the right.
  - When no value flows -> sorted.
  - $O(n) + O(n-1) + \cdots + O(1) = O(n^2)$.
- Which one do you like more?



i = 0	j	0	1	2	3	4	5	6	7
	0	5	3	1	9	8	2	4	7
	1	3	5	1	9	8	2	4	7
	2	3	1	5	9	8	2	4	7
	3	3	1	5	9	8	2	4	7
	4	3	1	5	8	9	2	4	7
	5	3	1	5	8	2	9	4	7
	6	3	1	5	8	2	4	9	7
i = 1	0	3	1	5	8	2	4	7	9
	1	1	3	5	8	2	4	7	
	2	1	3	5	8	2	4	7	
	3	1	3	5	8	2	4	7	
	4	1	3	5	2	8	4	7	
	5	1	3	5	2	4	8	7	
i = 2	0	1	3	5	2	4	7	8	
	1	1	3	5	2	4	7		
	2	1	3	5	2	4	7		
	3	1	3	2	5	4	7		
	4	1	3	2	4	5	7		
i = 3	0	1	3	2	4	5	7		
	1	1	3	2	4	5			
	2	1	2	3	4	5			
	3	1	2	3	4	5			
i = 4	0	1	2	3	4	5			
	1	1	2	3	4				
	2	1	2	3	4				
i = 5	0	1	2	3	4				
	1	1	2	3					
i = 6	0	1	2	3					
		1	2						

```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        swapped := false
        for i := 1 to n-1 inclusive do
            /* if this pair is out of order */
            if A[i-1] > A[i] then
                /* swap them and remember something changed */
                swap(A[i-1], A[i])
                swapped := true
            end if
        end for
    until not swapped
end procedure
```

https://en.wikipedia.org/wiki/Bubble_sort; https://www.geeksforgeeks.org/bubble-sort/

String Matching

- Long seq $y[1], y[2], \cdots, y[n]$ and short sequence $x[1], x[2], \cdots, x[m]$ with $m < n$.
- Question: is x part of y?
- Iterate every item in y in $O(n)$.
- Check if there is a match in $O(m)$.
- Total: $O(mn)$.

```
for ( j = 0; j <= n - m; j++ ) {
   for ( i = 0; i < m && x[i] == y[i + j]; i++ );
   if ( i >= m ) return j;
}
```

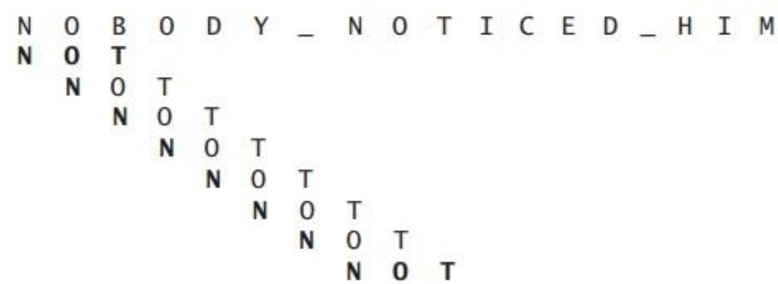
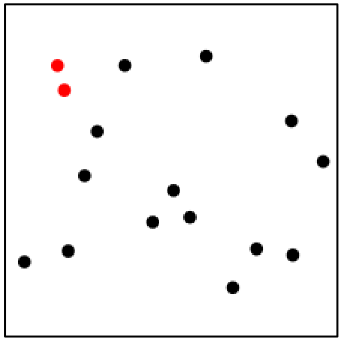


FIGURE 3.3 Example of brute-force string matching. The pattern's characters that are compared with their text counterparts are in bold type.

https://www.brainkart.com/media/extra/1aEIeYW.jpg

Closest Pair

- In a 2D-plane, n points $P[1], P[2], \cdots, P[n]$.
- How many combinations of two points (or pair)?
- Compute the **distances** of **all** the pairs -> $O(n^2)$.
  - Each distance computation -> $O(1)$.
- Record the smallest distance.



$$\| pq \| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2}$$

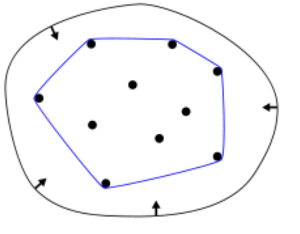
```
minDist = infinity
for i = 1 to length(P) - 1 do
    for j = i + 1 to length(P) do
        let p = P[i], q = P[j]
        if dist(p, q) < minDist  then
            minDist = dist(p, q)
            closestPair = (p, q)
return closestPair
```

https://en.wikipedia.org/wiki/Closest_pair_of_points_problem; CAN STOP HERE ON WED.

Convex Hull

- Convex hull: the smallest convex polygon containing all the points P.
- Fact 1: a convex hull is a subset of points in P.
  - Contradiction: inside or outside, neither is possible.
- Fact 2: enclosed by a series of lines.
- Fact 3: all the points in one side of each line. (Hint: contradiction)
- Fact 4: each line can be determined by a pair of 2 points.
- Idea: enumerate all the point pairs -> find all the possible/feasible lines -> convex hull.
- Complexity: $O(n^2)$ pairs, each pair check $O(n)$ distances -> $O(n^3)$.

for each point P_i
    for each point P_j where $P_j \neq P_i$
        Compute the line segment for P_i and P_j
        for every other point P_k where $P_k \neq P_i$ and $P_k \neq P_j$
            If each P_k is on one side of the line segment, label P_i and P_j
            in the convex hull



https://en.wikipedia.org/wiki/Convex_hull

Exhaustive Search

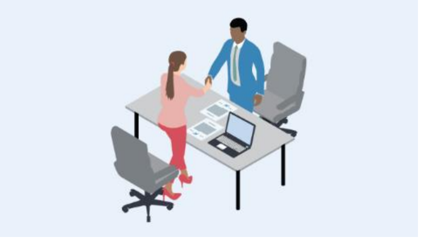
- Enumerate **all combinations** to find the optimal -> **exhaustive**.
- Accordingly, what is another brute-force algorithm for convex hull?
  - For every subset of points in P, enumerate all the possible connection paths of the points.
  - Runs for ever for large point set.
- **Job assignment problem.**
  - n **appliants**, n **jobs**, one applicant per job.
  - $c[i][j]$ **cost** of assigning applicant i to job j.
  - **Best assignment**: **min** total **cost**.
- $n = 4$ in the example.

$c[\cdot][\cdot]$	Job 1	Job 2	Job 3	Job 4
Applicant 1	9	2	7	8
Applicant 2	6	4	3	7
Applicant 3	5	8	1	8
Applicant 4	7	6	9	4

- Let $< a_1, a_2, \cdots, a_n >$ be an assignment w/ applicant i assigned to job a_i.
- Total cost: $C = c[1][a_1] + c[2][a_2] + \cdots + c[n][a_n] = \sum_{i=1}^{n} c[i][a_i]$.
  - i.e., $< 1,2,3,4 >$ -> $C = c[1][1] + c[2][2] + c[3][3] + c[4][4] = 9 + 4 + 1 + 4 = 18.$
- How to find the min one?

Job Assignment Problem

- **Enumerate** all the assignments.
- Combinatorial: $\boldsymbol{n}!$ -> $4! = 4 * 3 * 2 * 1 = 24$.
  - Cannot sustain when n is large.
- The best one: $< 2, 1, 3, 4 >$ ->
- $C = c[1][2] + c[2][1] + c[3][3] + c[4][4] = 2 + 6 + 1 + 4 = 13$.
- The worst one: $< 1, 4, 2, 3 >$ ->
- $C = 9 + 7 + 8 + 9 = 33$.
- $\frac{33}{13} = 2.5 \times$ -> big performance difference.
- Smarter way?
  - Applicant 1, find the min-cost job.
  - Applicant 2, the min-cost job among the rest.
  - …
  - We have: $< 2, 3, 1, 4 >$ → $C = c[1][2] + c[2][3] + c[3][1] + c[4][4] = 2 + 3 + 5 + 4 = 14$.
  - Cost diff only $14 - 13 = 1$. Not bad. You will know more in the following weeks.



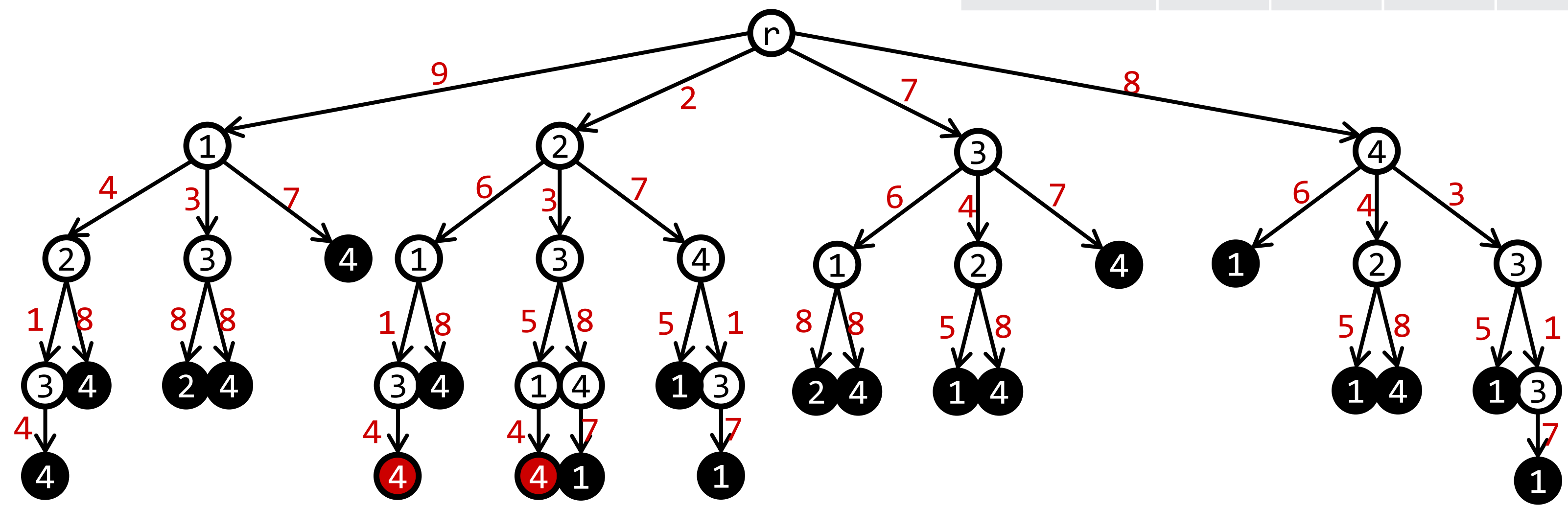
$c[\cdot][\cdot]$	Job 1	Job 2	Job 3	Job 4
Applicant 1	9	2	7	8
Applicant 2	6	4	3	7
Applicant 3	5	8	1	8
Applicant 4	7	6	9	4

https://www.experis.com.sg/blog/2019/04/5-things-not-to-say-in-a-job-interview

Backtracking – Job Assignment Example

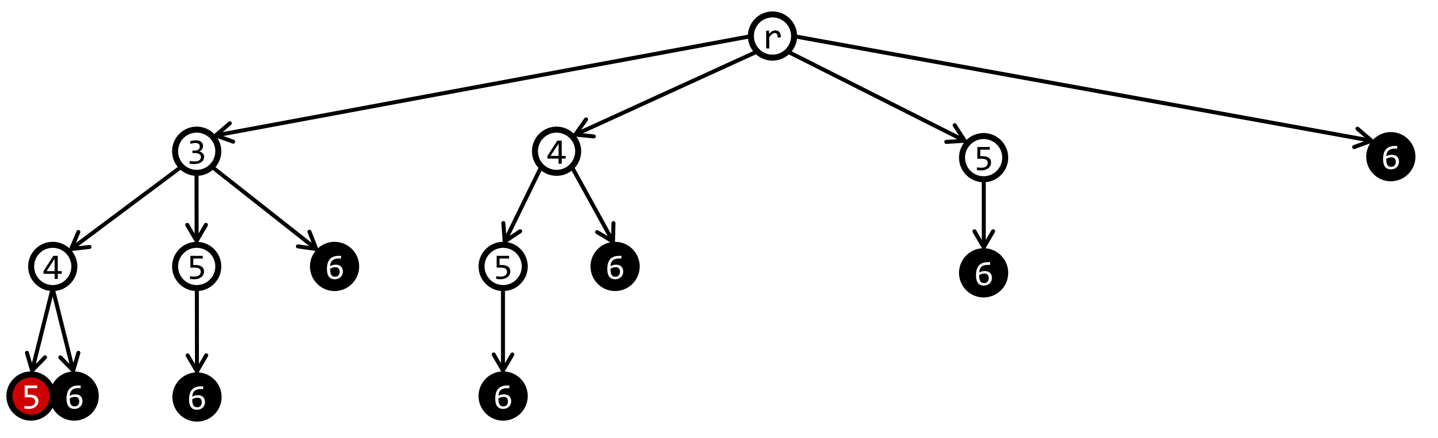
- Represent all solutions in a **tree**.
- Traverse in the tree in **depth-first** order.
  - Branch & Bound, breadth-first (self-learning).
- **Prune** if meaningless to traverse deeper.
- i.e., all the assignments with cost ≤ 14.
  - Returns ≥ 1 valid solutions.

$c[\cdot][\cdot]$	Job 1	Job 2	Job 3	Job 4
Applicant 1	9	2	7	8
Applicant 2	6	4	3	7
Applicant 3	5	8	1	8
Applicant 4	7	6	9	4



Backtracking – Subset Sum Problem

- Given a set of values, find a subset of the values such that the sum of the values in the subset equals to a specified value.
- Example: $n = 4$ values $\{3, 4, 5, 6\}$. Any subset with sum of 12?
- Similar to the job assignment one.
- Note the difference of subset -> **order insensitive**.
- 2^n subsets, and summation of each subset $O(n)$ -> $O(2^n n)$.



Basic Idea

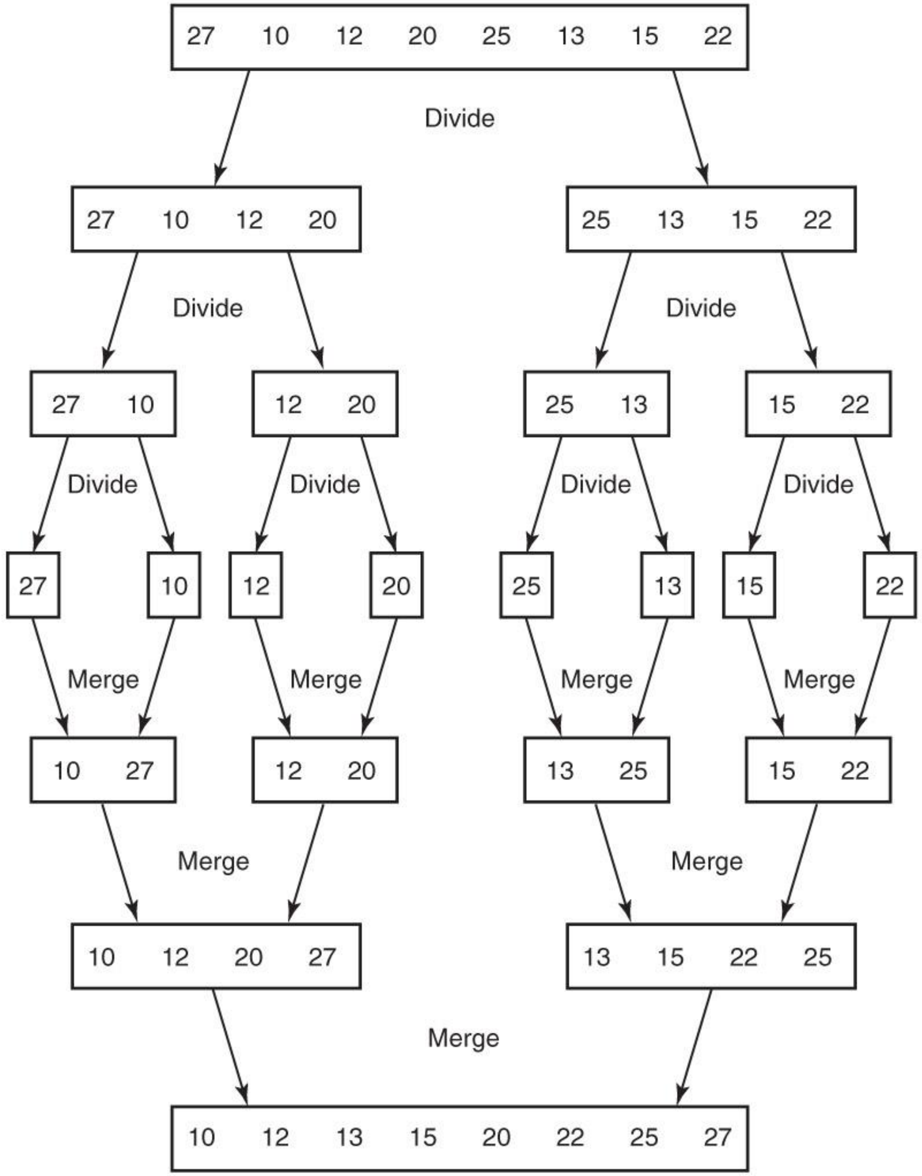
- **Algorithmic paradigm**: generic framework underlies the design of a **class** of algorithms.
  - Backtracking
  - Brute-force search
  - **Divide and conquer**
  - Dynamic programming
  - Greedy algorithm
- Divide and conquer: **multi-branched recursion.**
  - **Divide** a big problem into small subproblems.
    - Normally of the same type.
  - **Solve** the subproblems one by one.
  - **Combine** the solutions to the subproblems.
  - Do this **recursively**.
- Outcome: a solution to the original big problem.

Merge Sort

Method: Divide & conquer

1. Divide the unsorted list into two nearly equal size sub-lists.
2. Sort each sub-list recursively by applying merge sort.
3. Merge the two sub-lists back into one sorted list.

Merge Sort: Example Lecture 5 slide 9



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

Master Method Case 2: O(nlg(n))

$$T(n) = 2T\left(\frac{n}{2}\right) + n \qquad T(n) = O(nlgn)$$

$$T(n) \le 2\left(c\frac{n}{2}lg\frac{n}{2}\right) + n = cn(lgn - lg2) + n = cnlgn - cn + n = cnlgn - (c-1)n$$

$$T(n) \le cnlgn - (c-1)n \le cnlgn \quad via\ substitution\ method$$

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

n: n, split into $T\left(\frac{n}{2}\right)$ and $T\left(\frac{n}{2}\right)$

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

n: $\frac{n}{2}$ $\frac{n}{2}$

$T\left(\frac{n}{2^2}\right)$ $T\left(\frac{n}{2^2}\right)$ $T\left(\frac{n}{2^2}\right)$ $T\left(\frac{n}{2^2}\right)$

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

n: $\frac{n}{2^2}$ $\frac{n}{2^2}$ $\frac{n}{2^2}$ $\frac{n}{2^2}$

$n \times lgn$

master method

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

$a = 2,\ b = 2,\ f(n) = n$

$g(n) = n^{\log_2 2}$

Since $f(n) = g(n)$
$= n$

case 2

$n \times lgn$

Merge algorithm

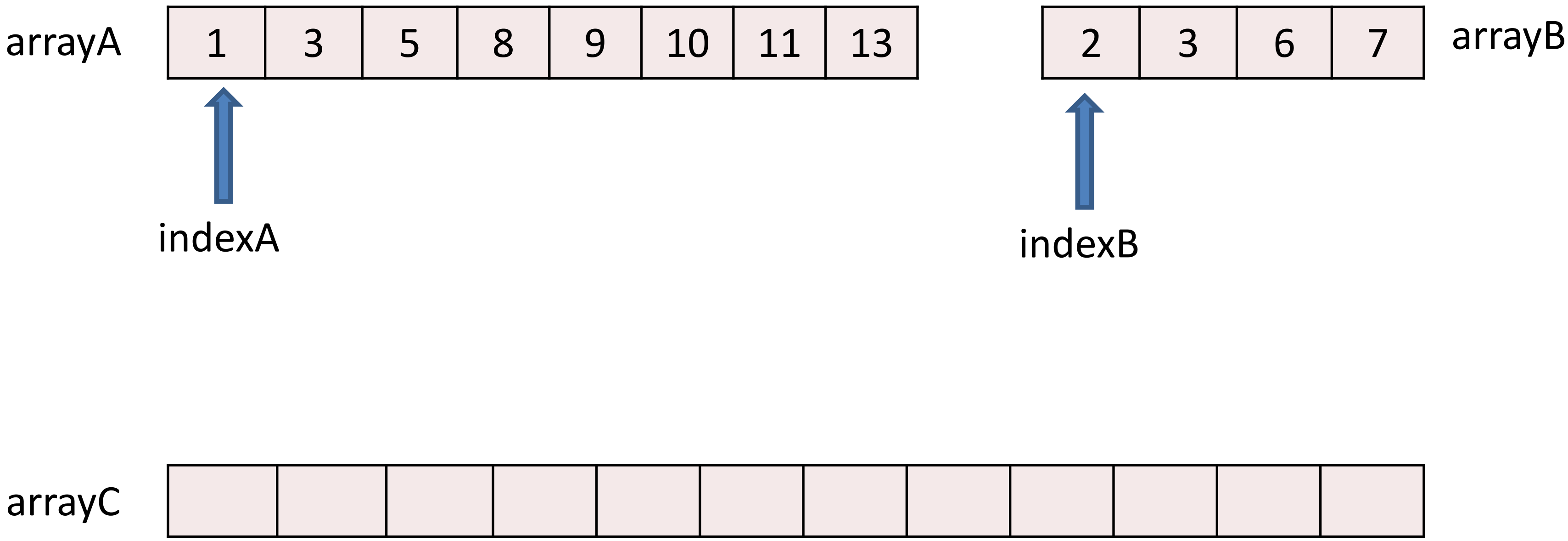
```
/* l is for left index and r is right index of the
sub-array of arr to be sorted */
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}
```

Merge algorithm

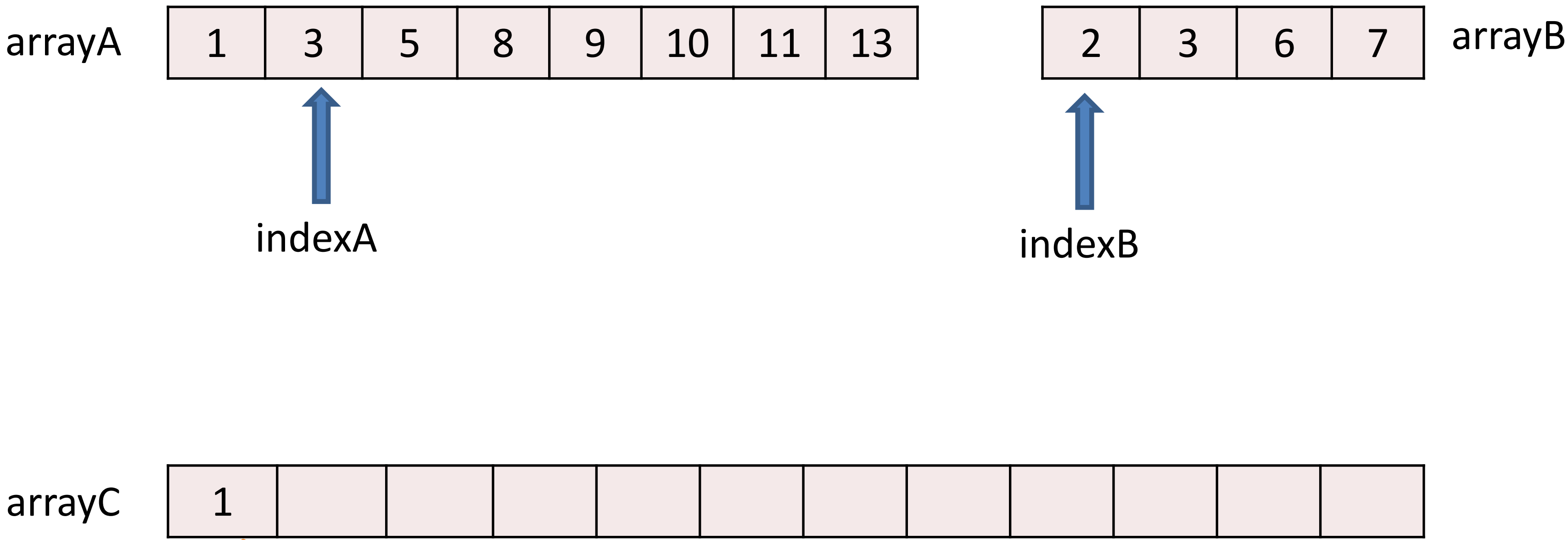
```
// Merges two subarrays of arr[].
// First subarray is arr[l..m]
// Second subarray is arr[m+1..r]
void merge(int arr[], int l, int m, int r)
{
   int i, j, k;
   int n1 = m - l + 1;
   int n2 = r - m;

   /* create temp arrays */
   int L[n1], R[n2];

   /* Copy data to temp arrays L[] and R[] */
   for (i = 0; i < n1; i++)
      L[i] = arr[l + i];
   for (j = 0; j < n2; j++)
      R[j] = arr[m + 1 + j];
```

```
/* Merge the temp arrays back into arr[l..r]*/
   i = 0; // Initial index of first subarray
   j = 0; // Initial index of second subarray
   k = l; // Initial index of merged subarray
   while (i < n1 && j < n2) {
      if (L[i] <= R[j]) {
         arr[k] = L[i];  i++;
      }
      else {
         arr[k] = R[j];  j++;
      }
      k++;
   }

   /* Copy the remaining elements of L[], if there
   are any */
   while (i < n1) {  arr[k] = L[i];  i++;   k++; }

   /* Copy the remaining elements of R[], if there
   are any */
   while (j < n2) {  arr[k] = R[j];  j++;  k++;  }
}
```

Merge Algorithm Demo

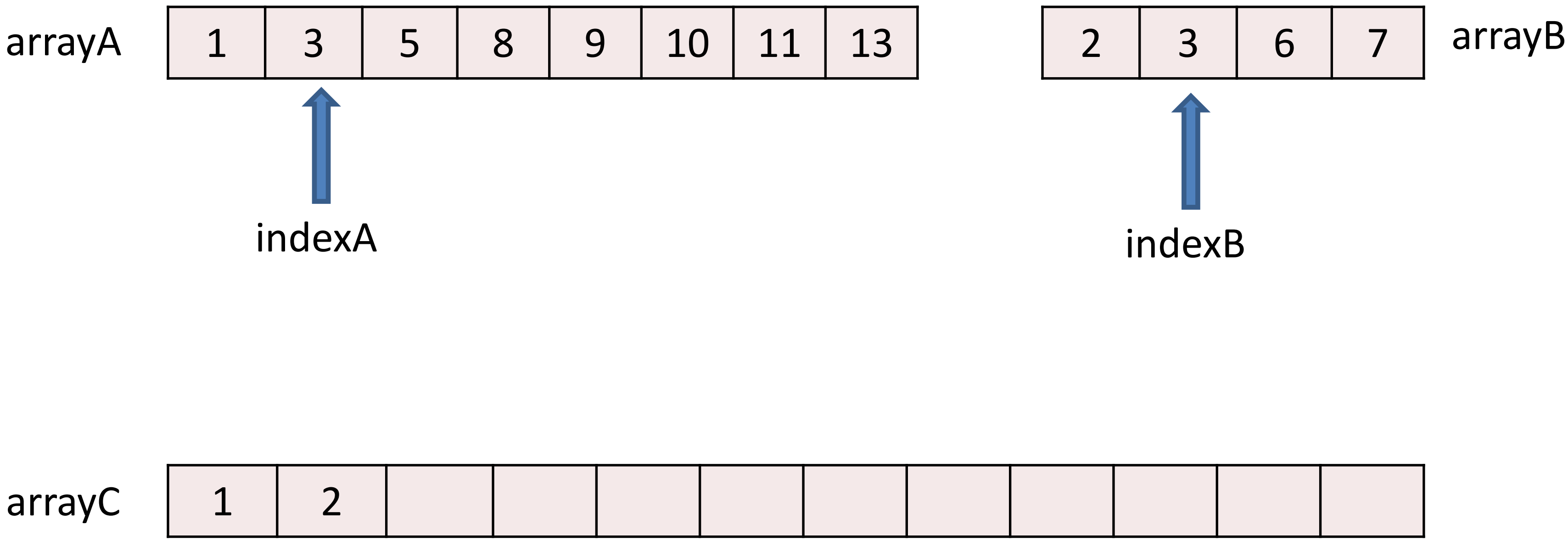


Merge Algorithm Demo



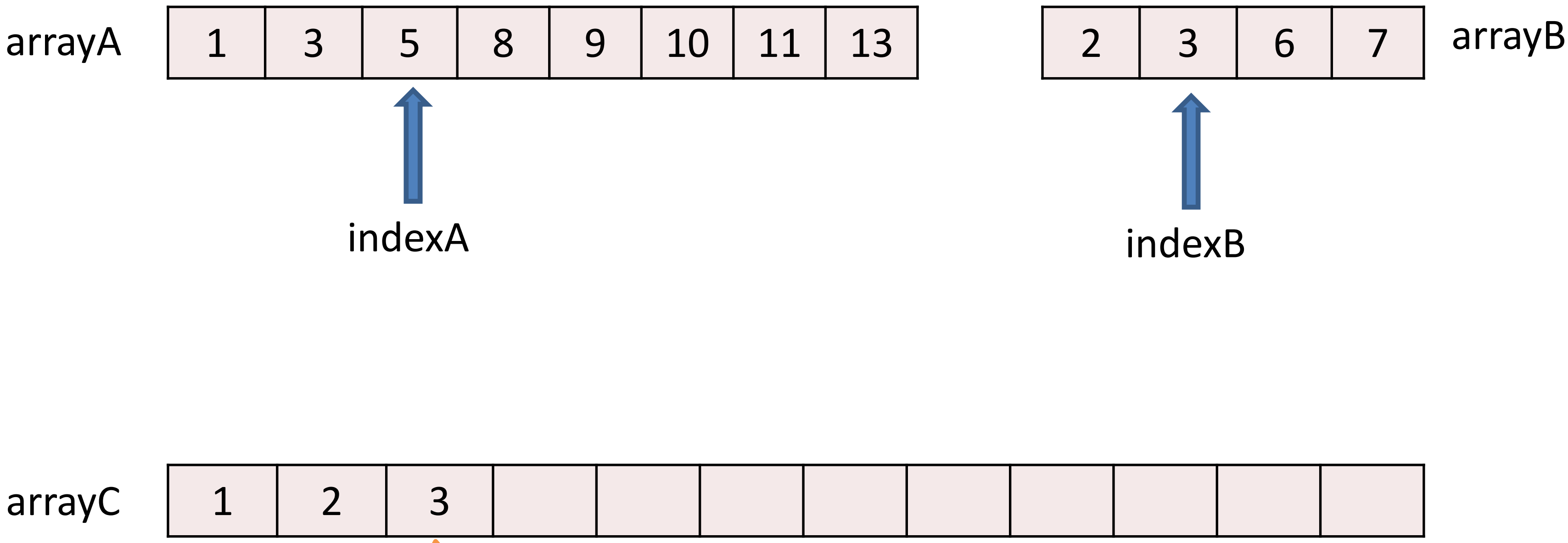
Compare 1 and 2.
1 is smaller, so 1 is copied to arrayC, and pointer indexA moves right.

Merge Algorithm Demo



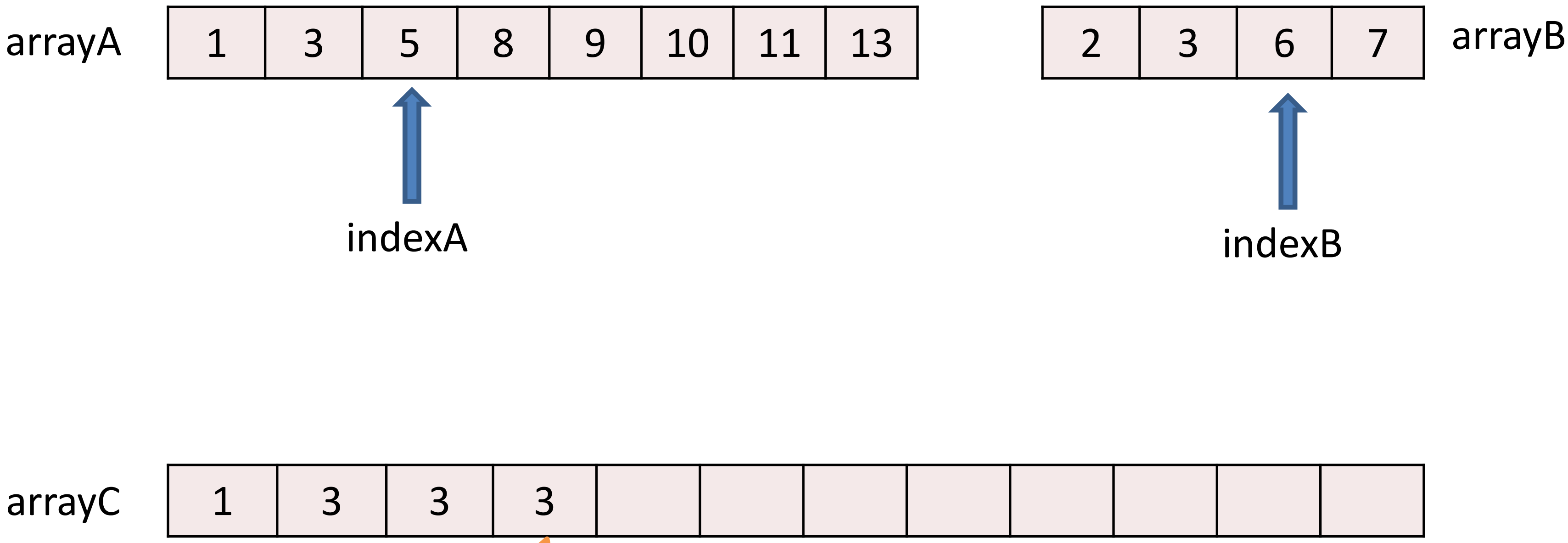
Compare 3 and 2.
2 is smaller, so 2 is copied to arrayC, and
pointer indexB moves right.

Merge Algorithm Demo



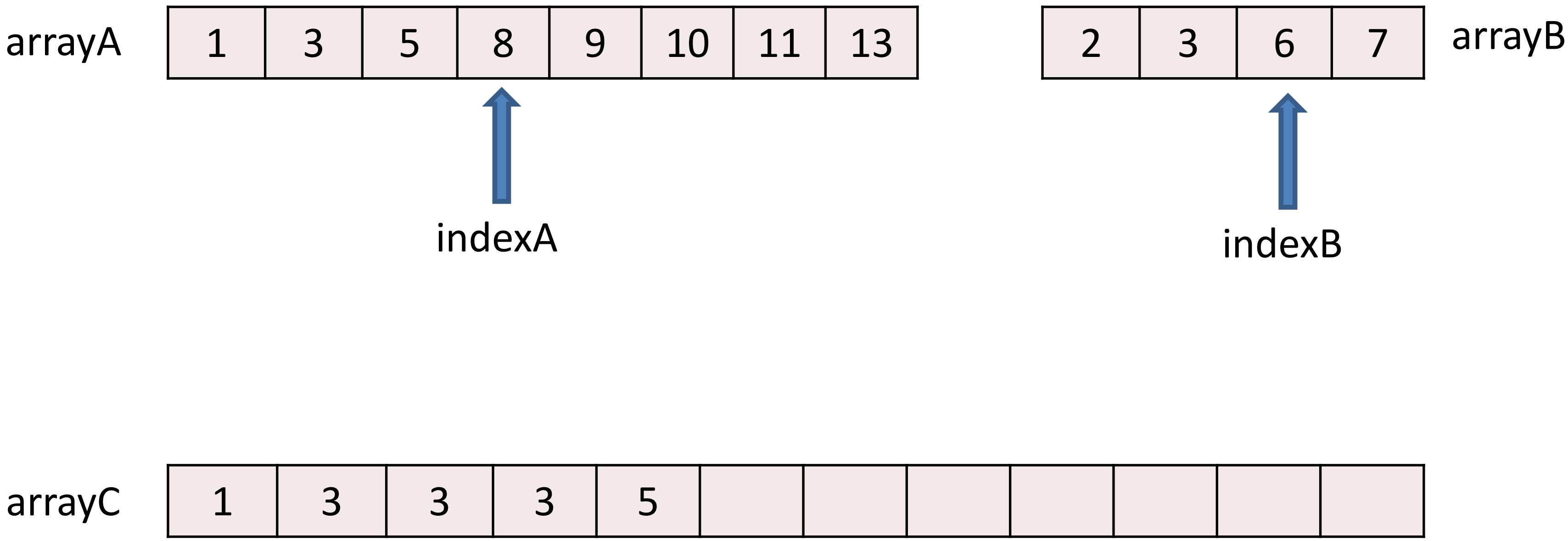
Compare 3 and 3.
3 in arrayA is not smaller than 3 in arrayB,
so the 3 in arrayA is copied to arrayC, and
pointer indexA moves right.

Merge Algorithm Demo



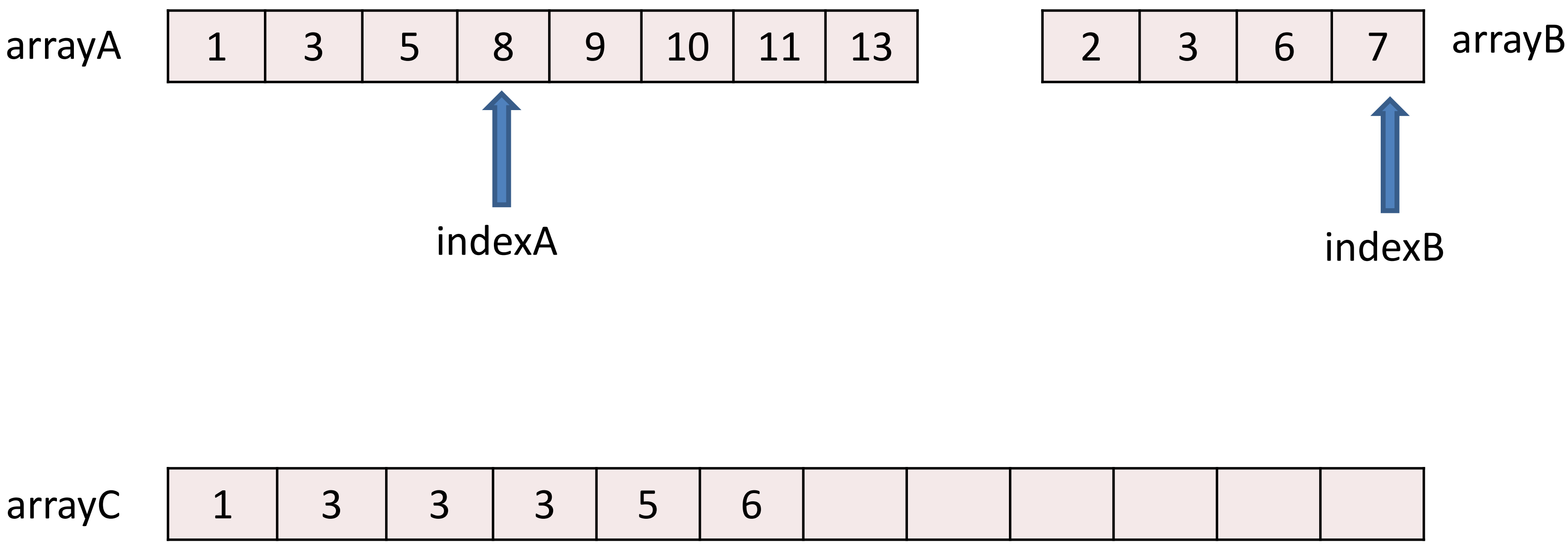
Compare 3 and 5.
3 is smaller than 5, so the 3 is copied to arrayC, and pointer indexB moves right.

Merge Algorithm Demo



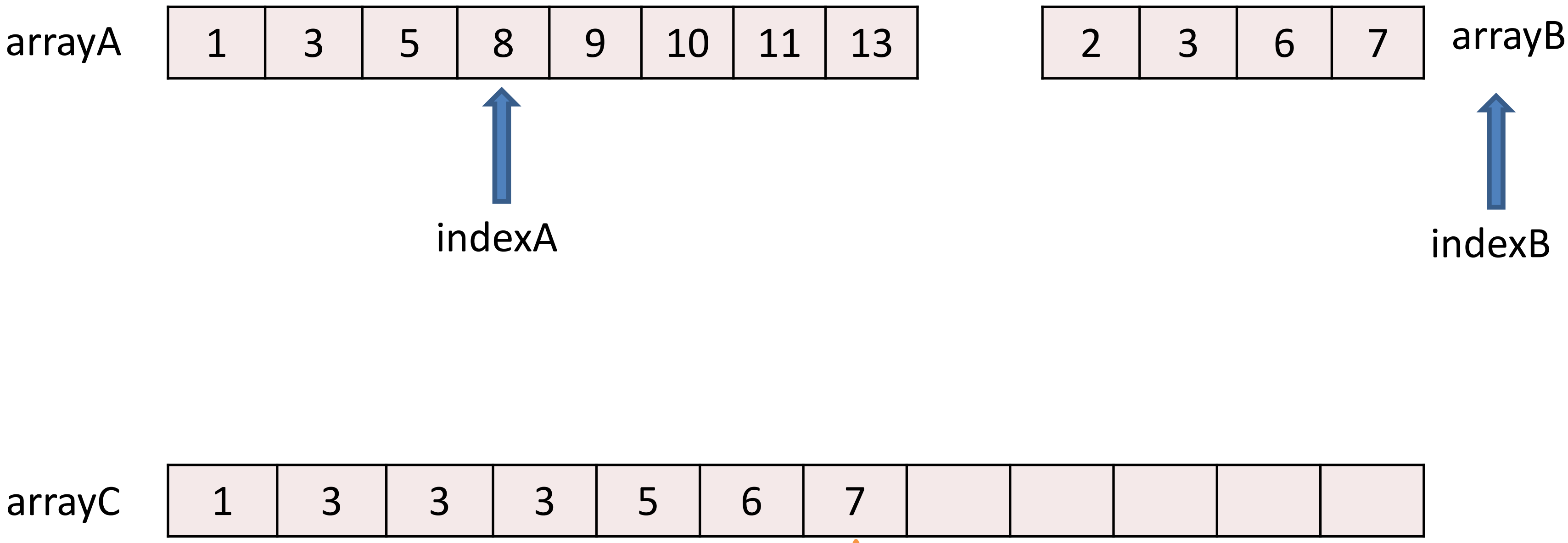
Compare 5 and 6.
5 is smaller than 6, so 5 is copied to arrayC, and pointer indexA moves right.

Merge Algorithm Demo



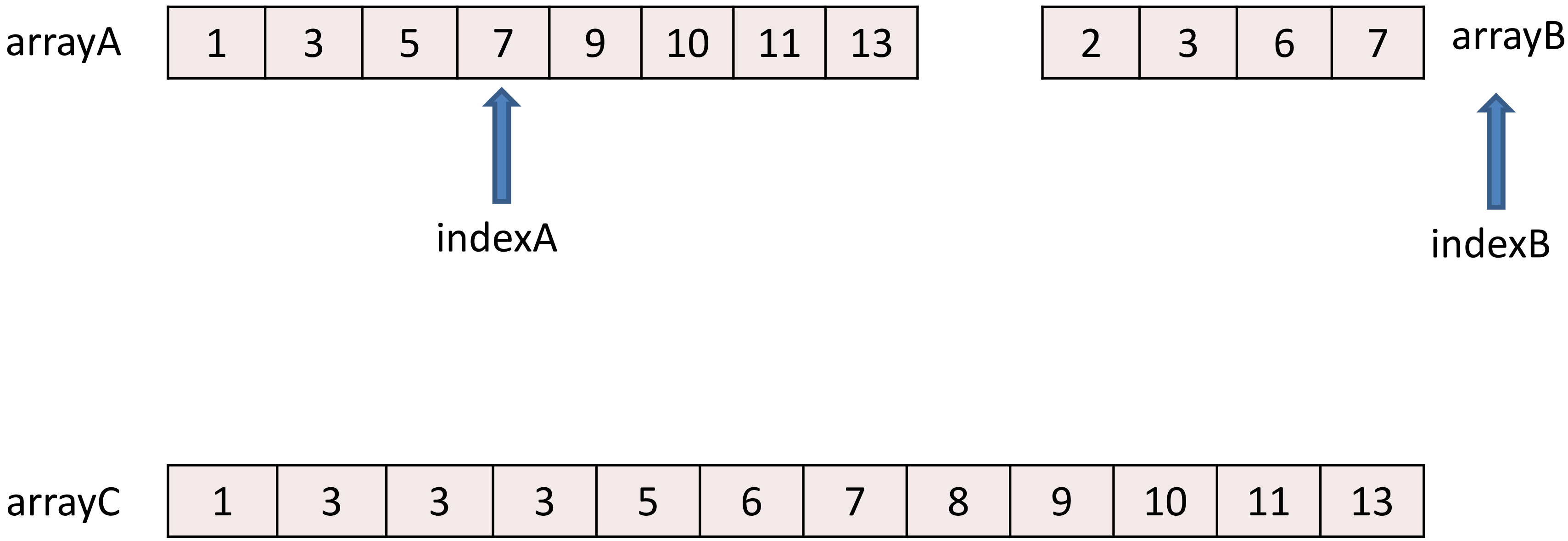
Compare 8 and 6.
6 is smaller than 8, so 6 is copied to arrayC, and pointer indexB moves right.

Merge Algorithm Demo



Compare 8 and 7.
7 in arrayB is copied to arrayC, and
pointer indexB moves right.

Merge Algorithm Demo



The remaining elements of arrayA are copied into arrayC.

Merge Sort algorithm


```
def mergeSort(array):
    size = len(array)

    if size is 1:
        return array

    midIndex = size/2
    firstHalf = array[0:midIndex]
    secondHalf = array[midIndex:size]

    firstHalf = mergeSort(firstHalf)
    secondHalf = mergeSort(secondHalf)
    array = merge(firstHalf, secondHalf)

    return array

print mergeSort([27,10,[illegible],20,25,13,15,22])
```

Base case:
When the `array` has one element, it is already sorted.
So return the `array` for merging.

Divide the `array` into two almost equal halves: `firstHalf` and `secondHalf`.

Recursively MergeSort divide `firstHalf` and divide `secondHalf`.

Merge the sorted `firstHalf` and `secondHalf`.

Return the sorted `array`.

Merge Sort – Complexity

- Time complexity
  - $merge$ is $\mathrm{O}(n)$.
  - $merge$ is called $\mathrm{O}(\log n)$ times recursively.
  - $mergeSort$ is $\mathrm{O}(n \log n)$.
- Space complexity
  - $merge$ uses an additional $arrayC$.
  - If $arrayC$ was local inside $merge$, much more storage would be used because of recursive calls.
  - Consider using a global $arrayC$ in the implementation.

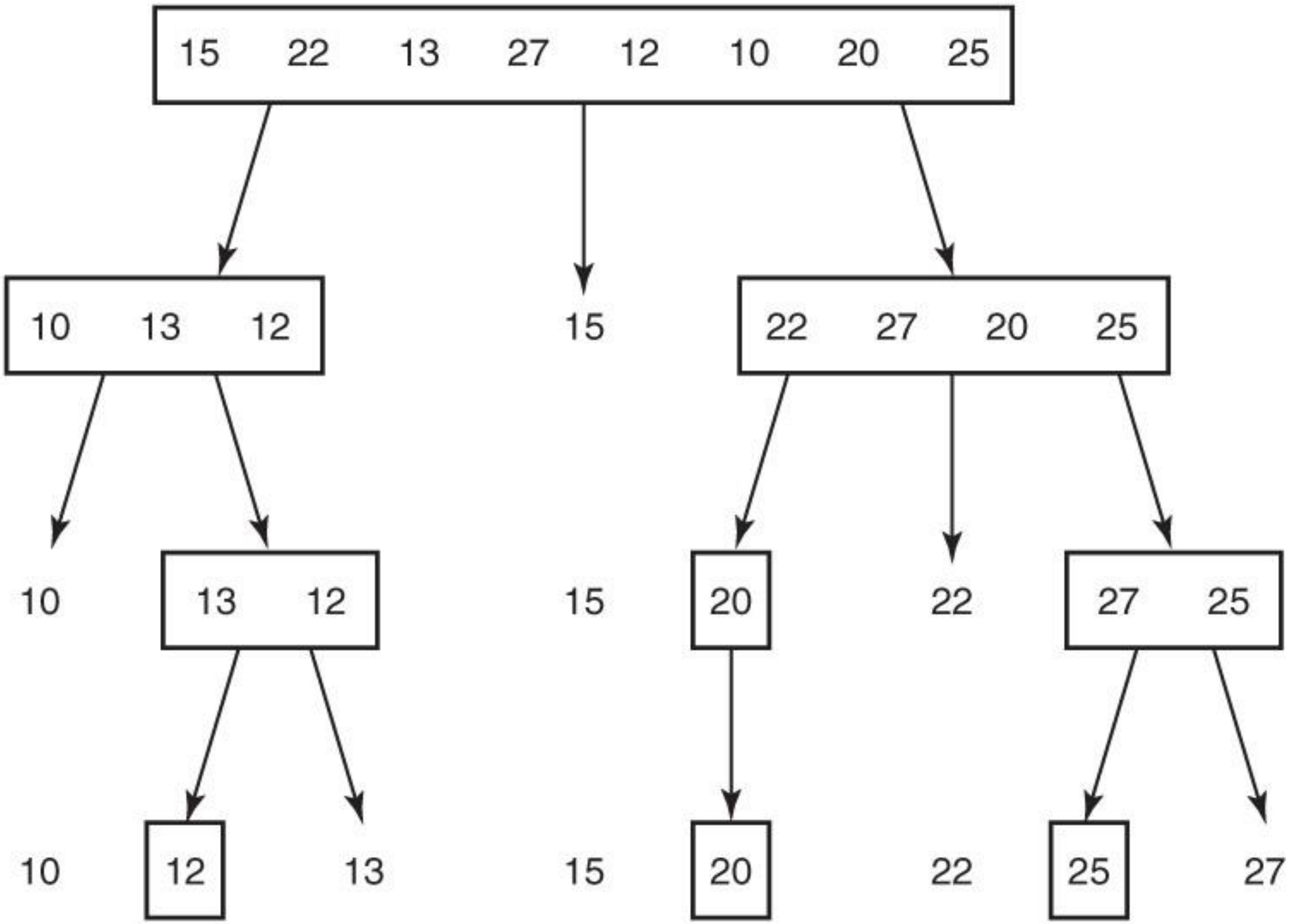
Quick Sort

Method: Divide-and-conquer.

- Pick an element ($pivot$) from the list.
  - $pivot$ is arbitrarily chosen.
  - Normally, the first element is selected.
- Partition the list into two halves such that:
  - All the elements in the first half are smaller than the pivot.
  - All the elements in the second half are greater than or equal to the pivot.

- Quick-sort the 1st half.
- Quick-sort the 2nd half.

Quick Sort: Example



Quick Sort algorithm

```
int partition(int arr[], int low, int high)
{
    int i = low;
    int j = high;
    int pivot = arr[low];
    while (i < j)
    {
        while (pivot >= arr[i])  i++;
        while (pivot < arr[j])  j--;
        if (i < j)  swap(arr[i], arr[j]);
    }
    swap(arr[low], arr[j-1]);
    return j;
}
```

```
void quickSort(int arr[], int low, int high)
{
    if (low < high)
    {
        int pivot = partition(arr, low, high);
        quickSort(arr, low, pivot - 1);
        quickSort(arr, pivot + 1, high);
    }
}

int main()
{
    int arr[] = {4, 2, 8, 3, 1, 5, 7,11,6};
    int size = sizeof(arr) / sizeof(int);
    cout<<"Before Sorting"<<endl;
    quickSort(arr, 0, size - 1);
    cout<<"After Sorting"<<endl;
    return 0;
}
```

Quicksort Animation

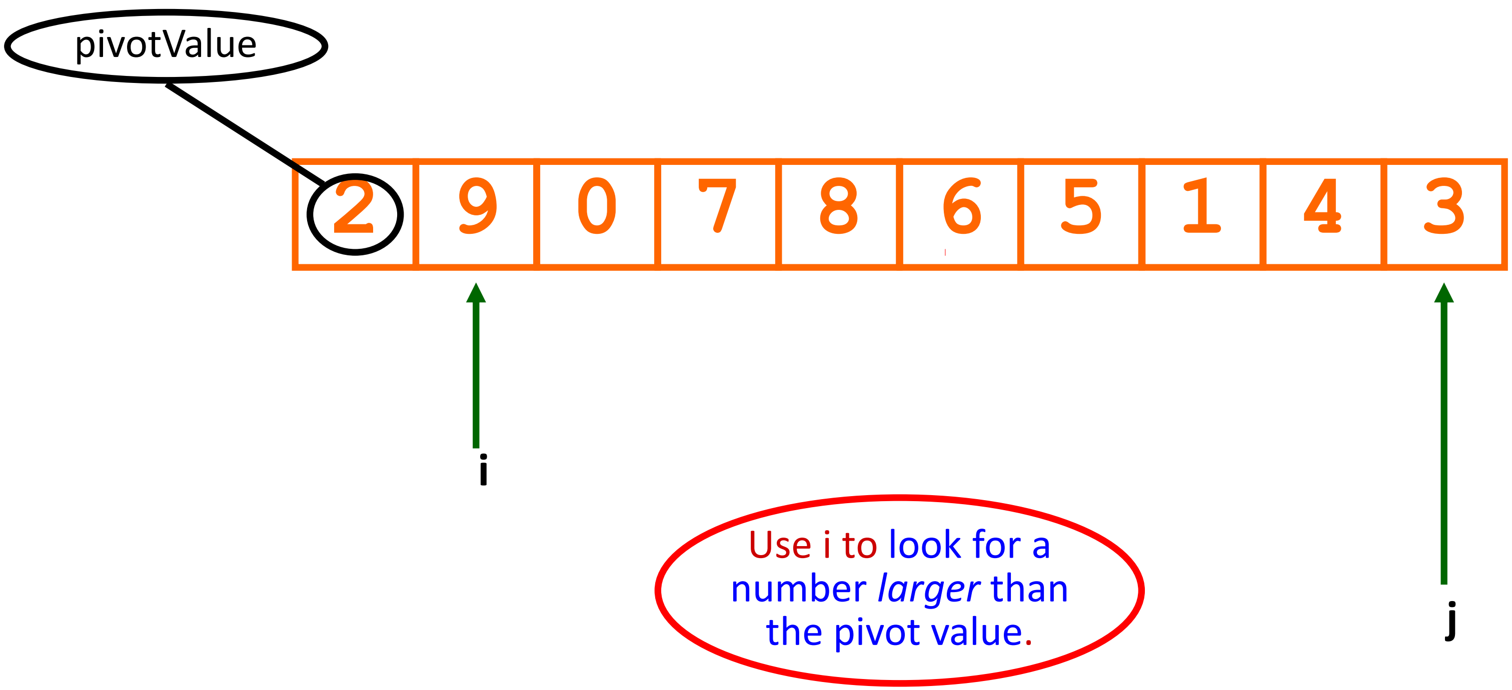
Split()

pivotValue

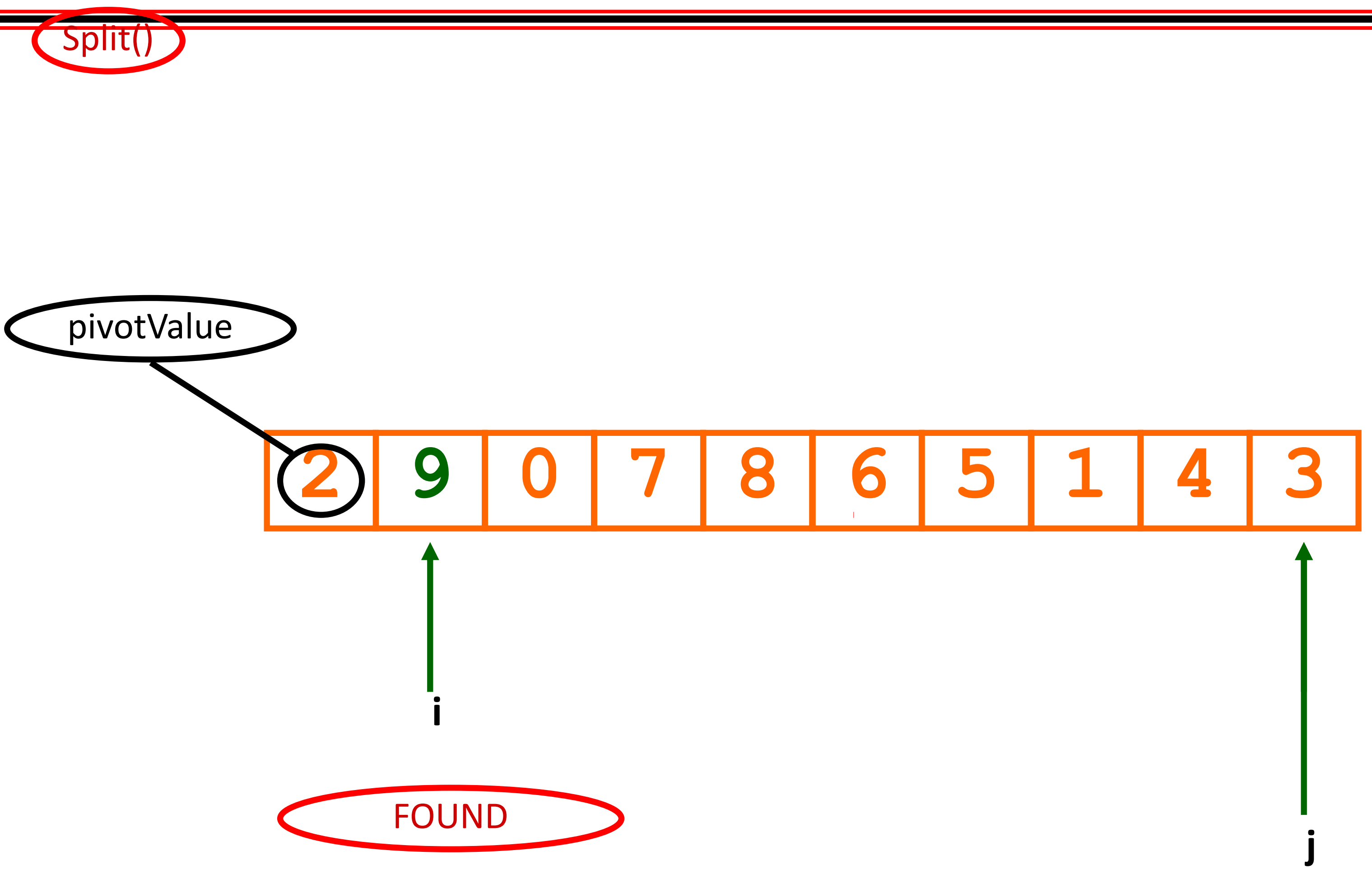
2	9	0	7	8	6	5	1	4	3

Quicksort Animation

Split()

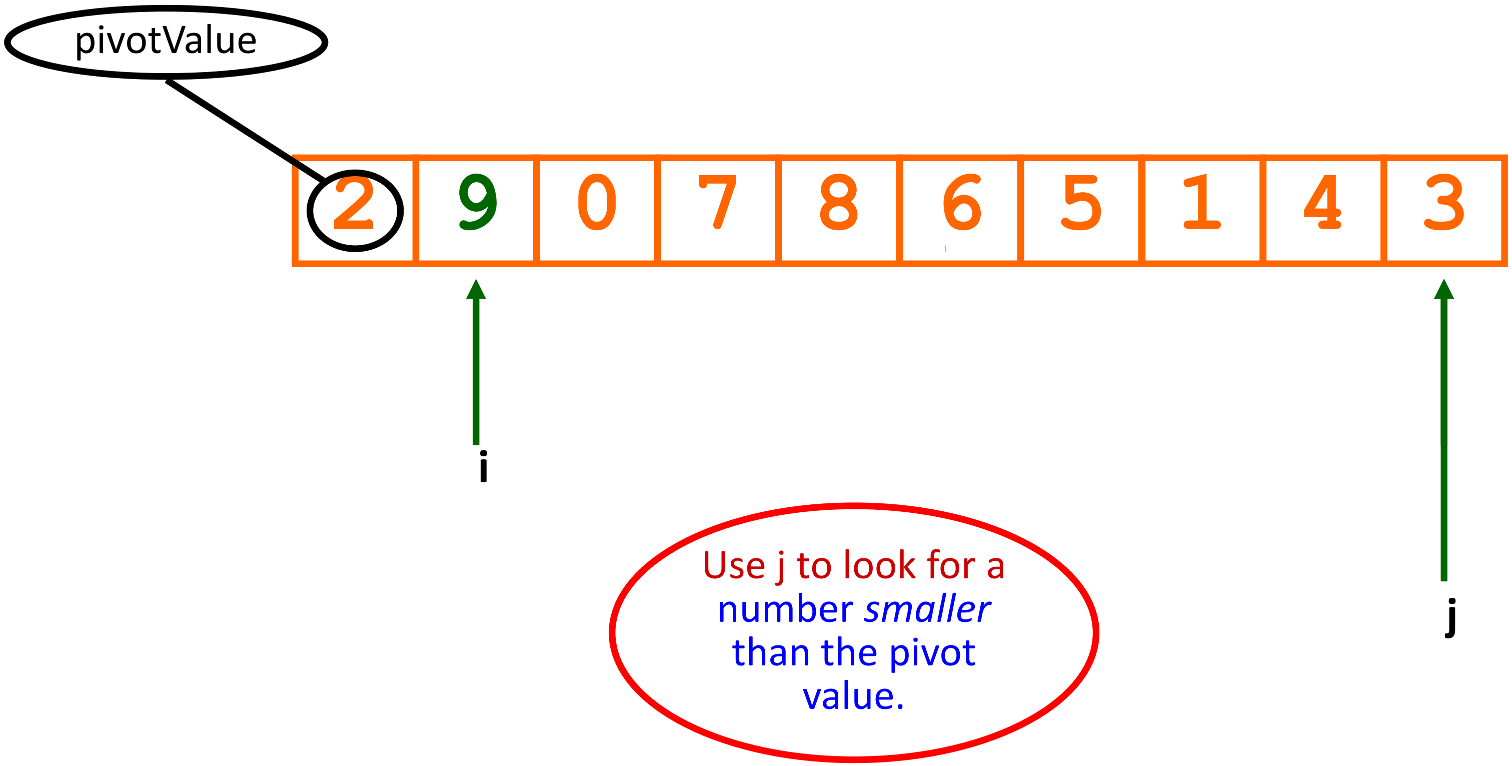


Quicksort Animation



Quicksort Animation

Split()



Quicksort Animation

Split()

pivotValue

2	9	0	7	8	6	5	1	4	3

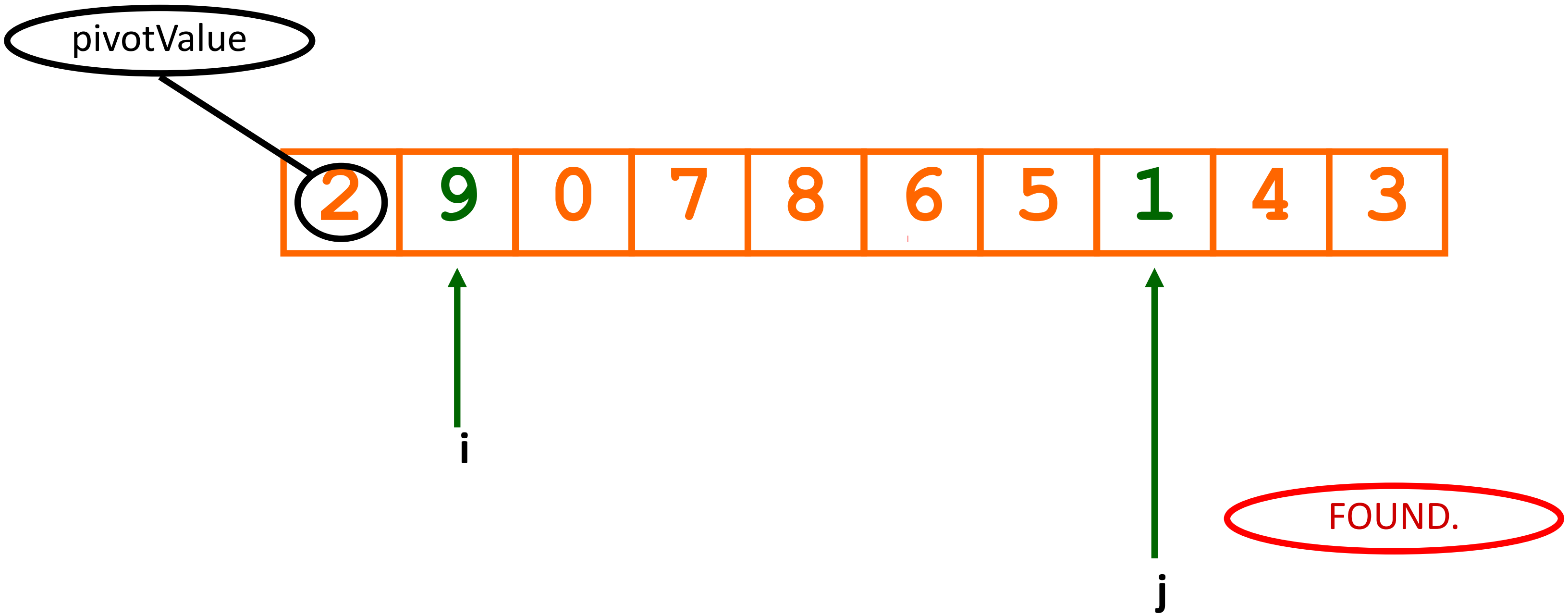
i

j

Use j to look for a number *smaller* than the pivot value.

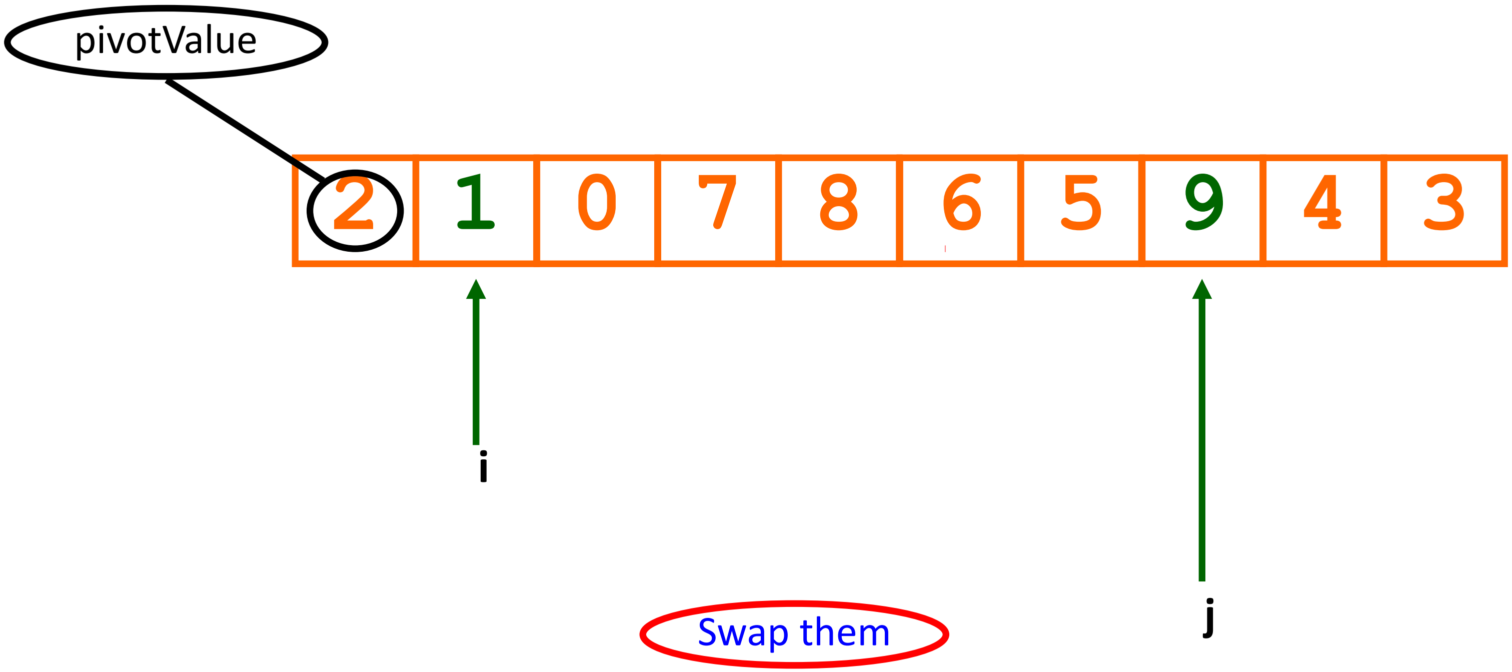
Quicksort Animation

Split()



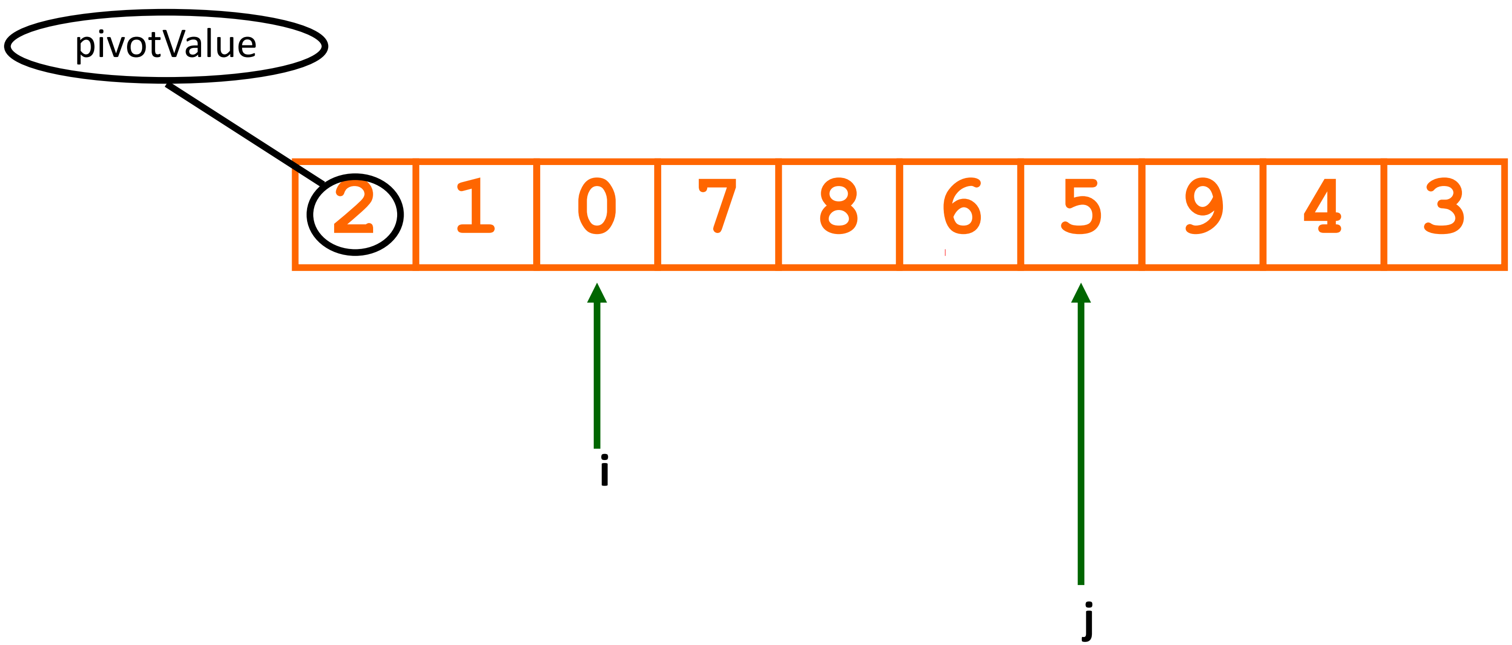
Quicksort Animation

Split()



Quicksort Animation

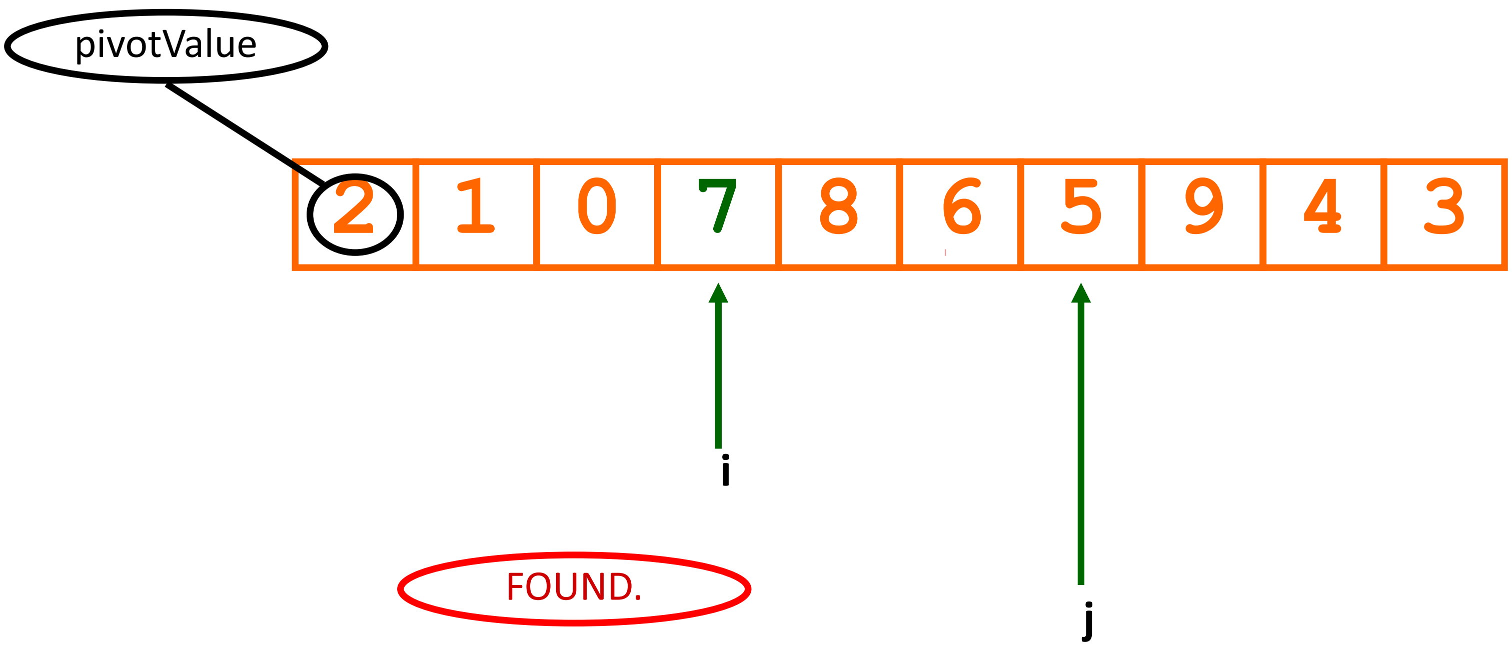
Split()



Use i to look for a number *larger* than the pivot value.

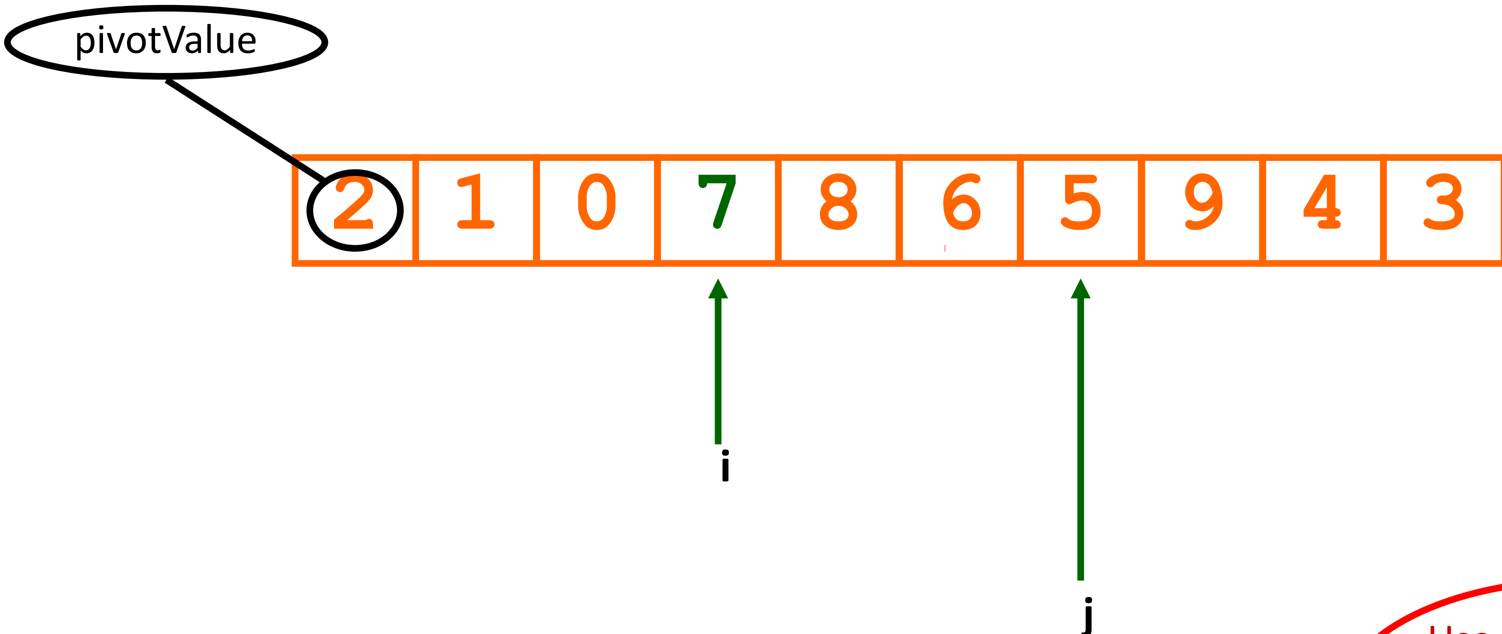
Quicksort Animation

Split()



Quicksort Animation

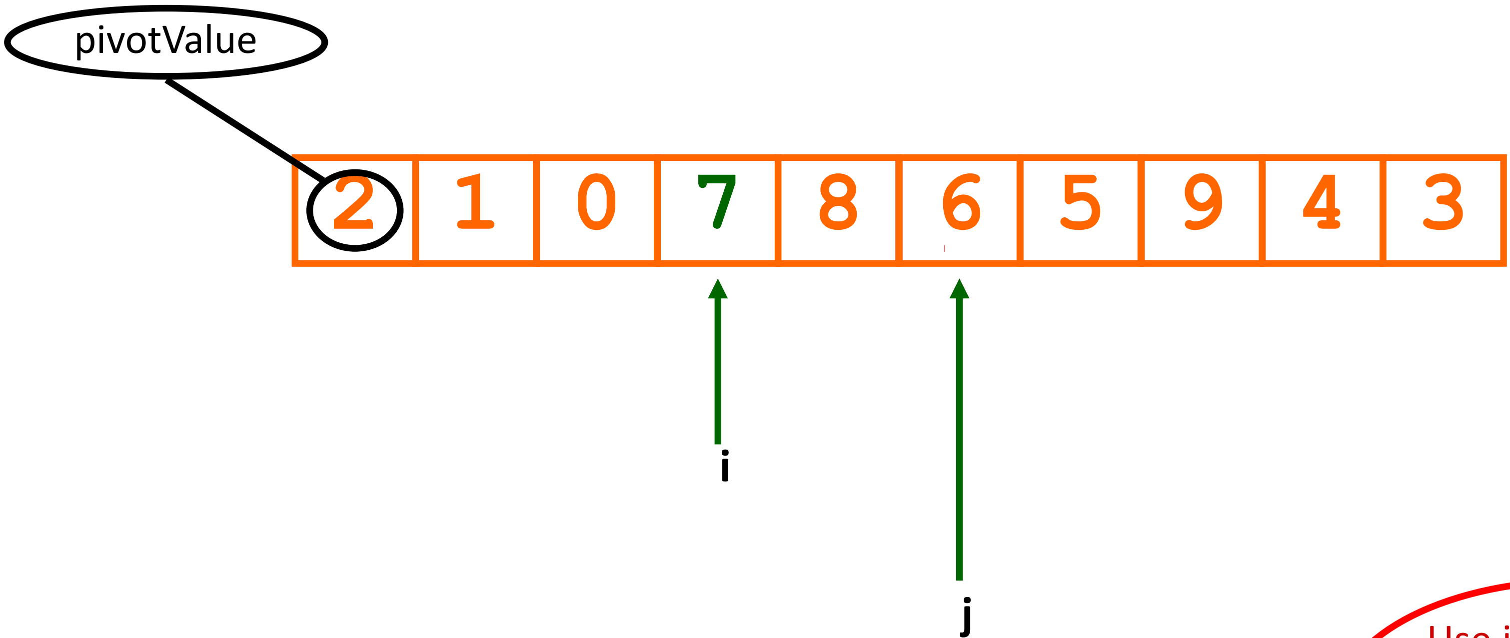
Split()



Use j to look for a number *smaller* than the pivot value.

Quicksort Animation

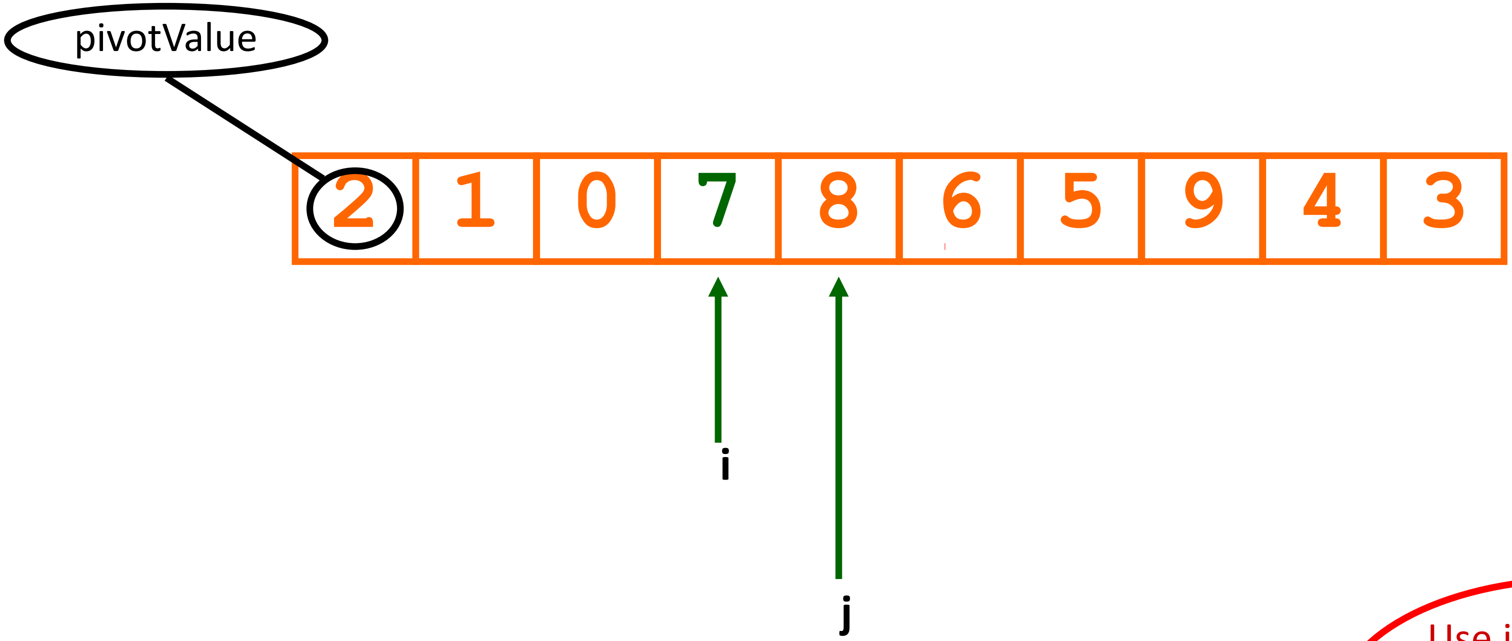
Split()



Use j to look for a number *smaller* than the pivot value.

Quicksort Animation

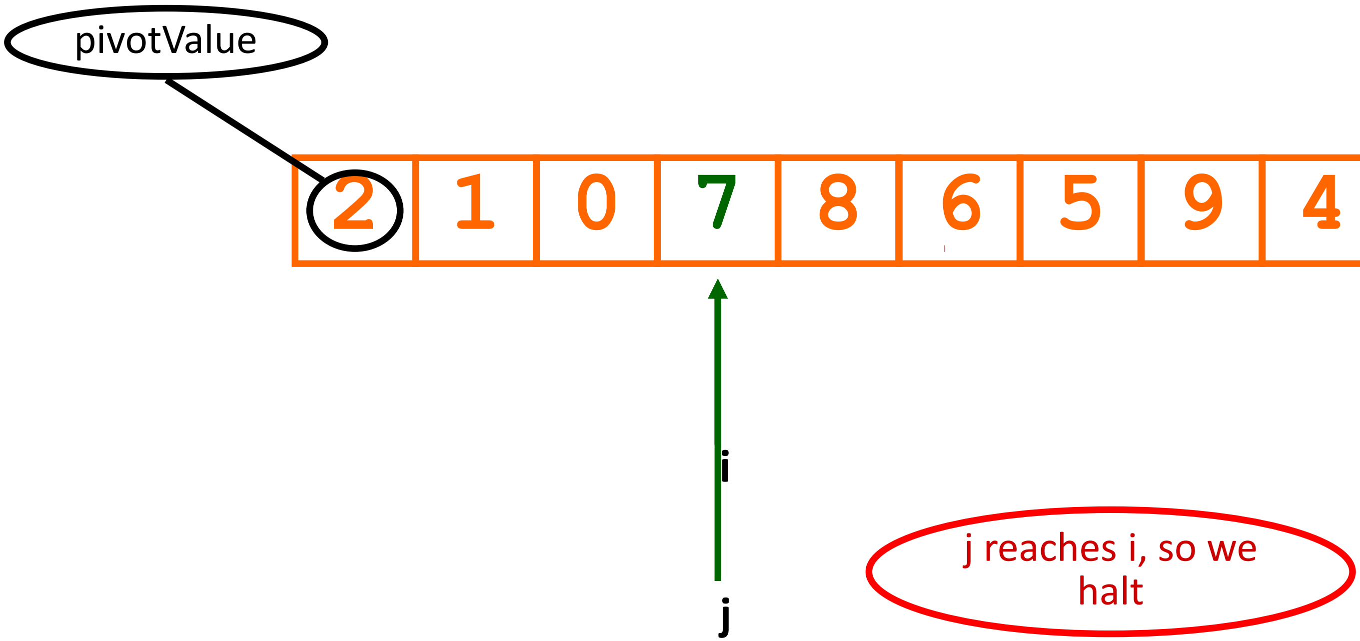
Split()



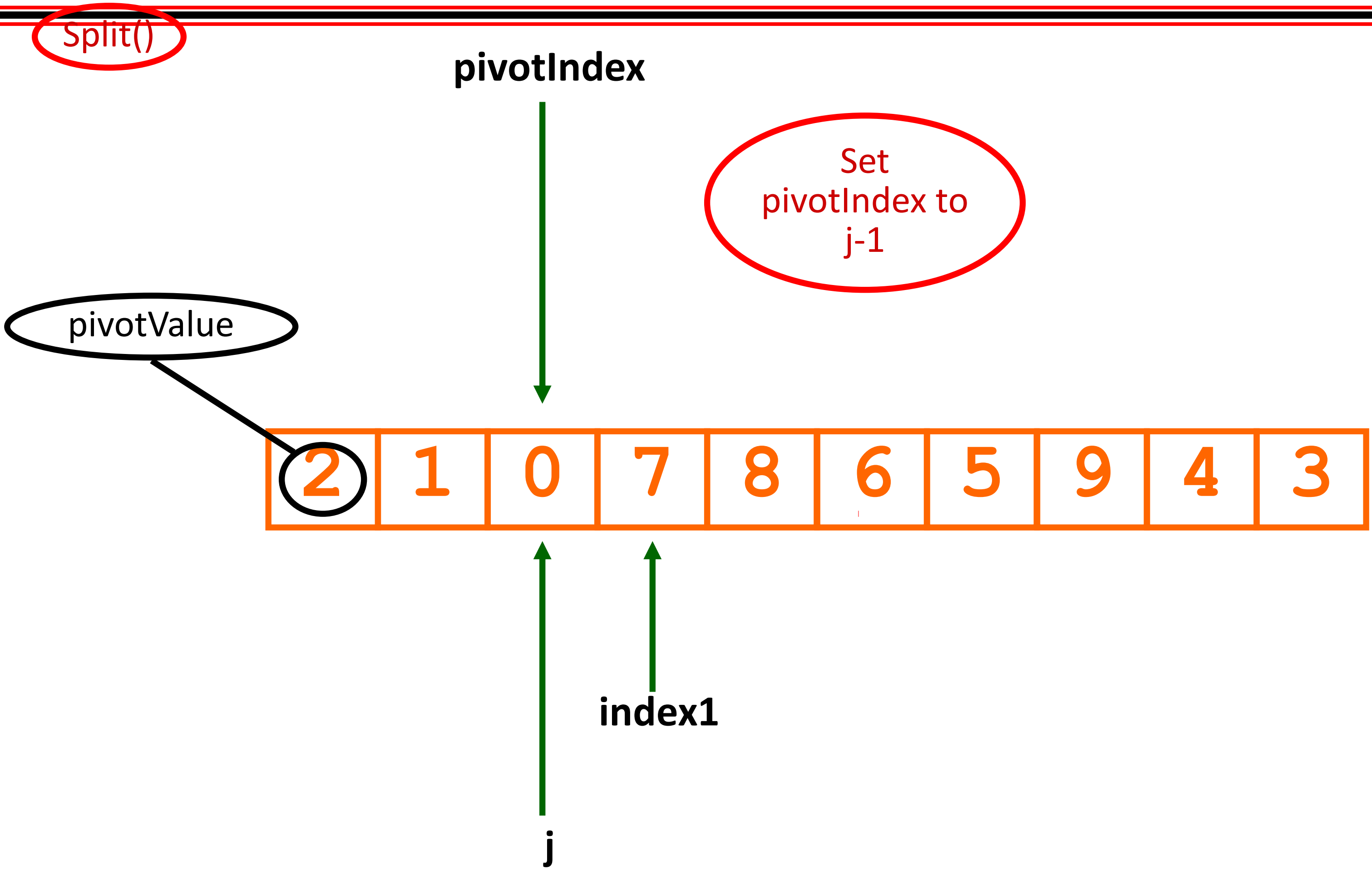
Use j to look for a number *smaller* than the pivot value.

Quicksort Animation

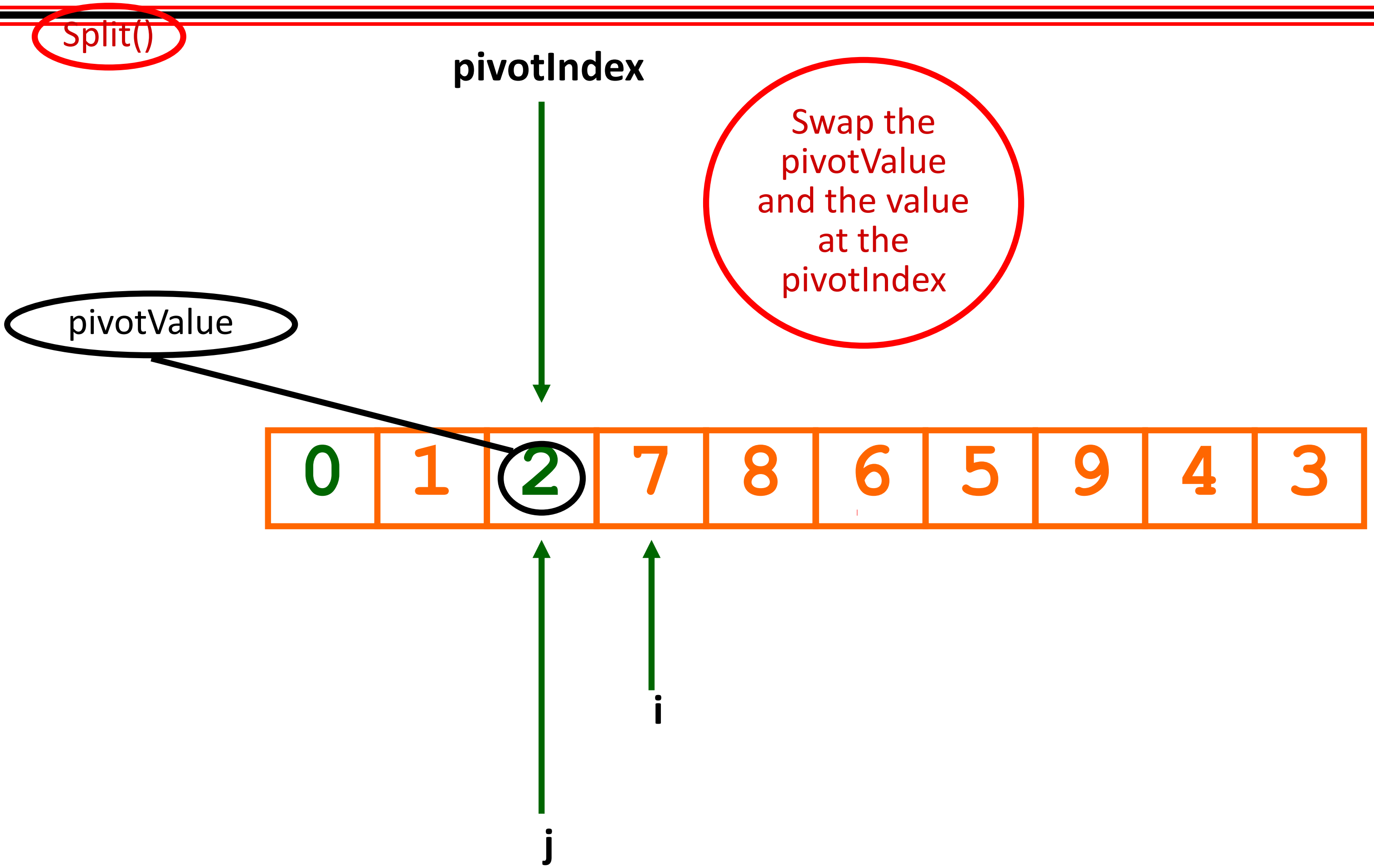
Split()



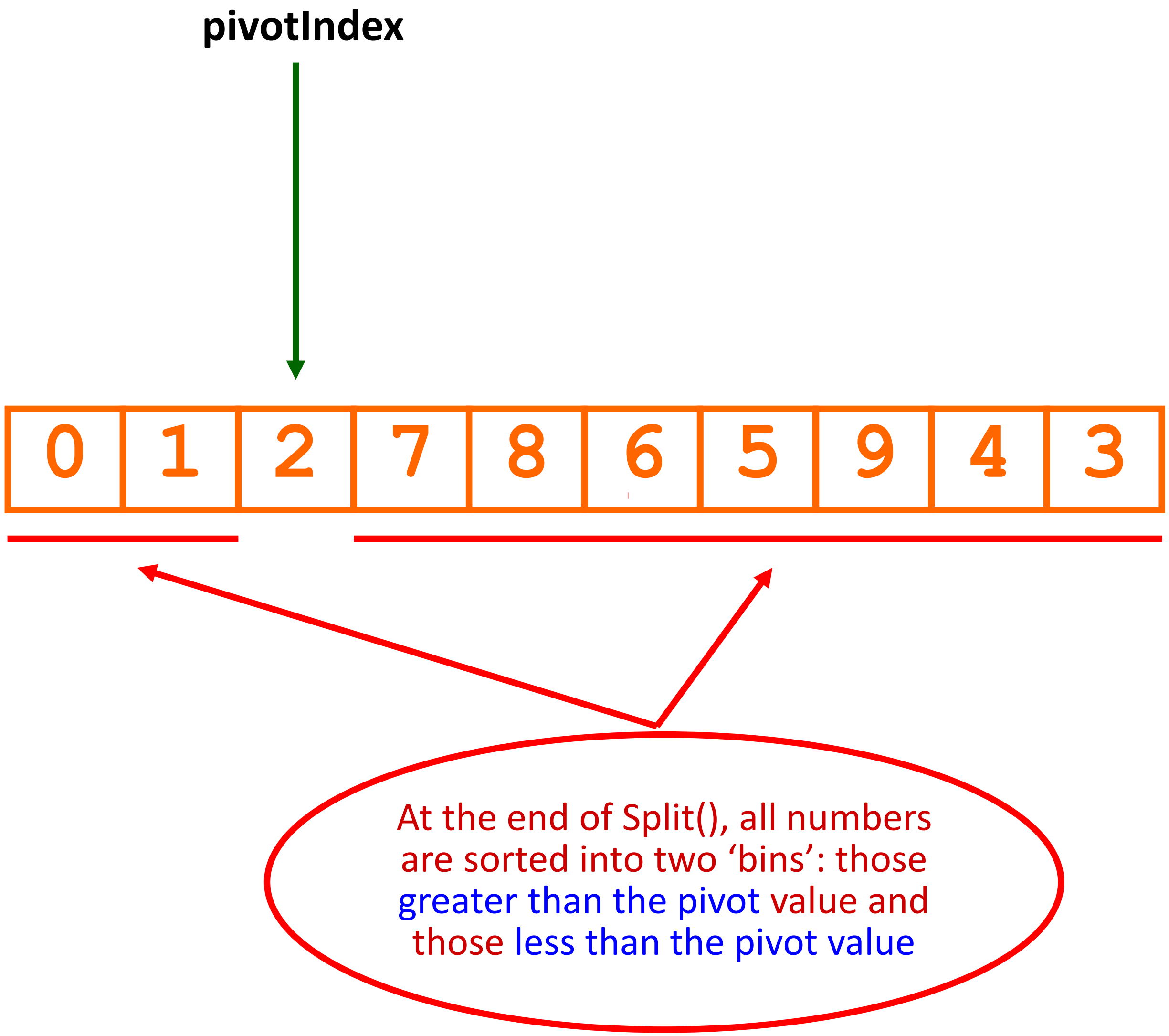
Quicksort Animation



Quicksort Animation



Quicksort Animation



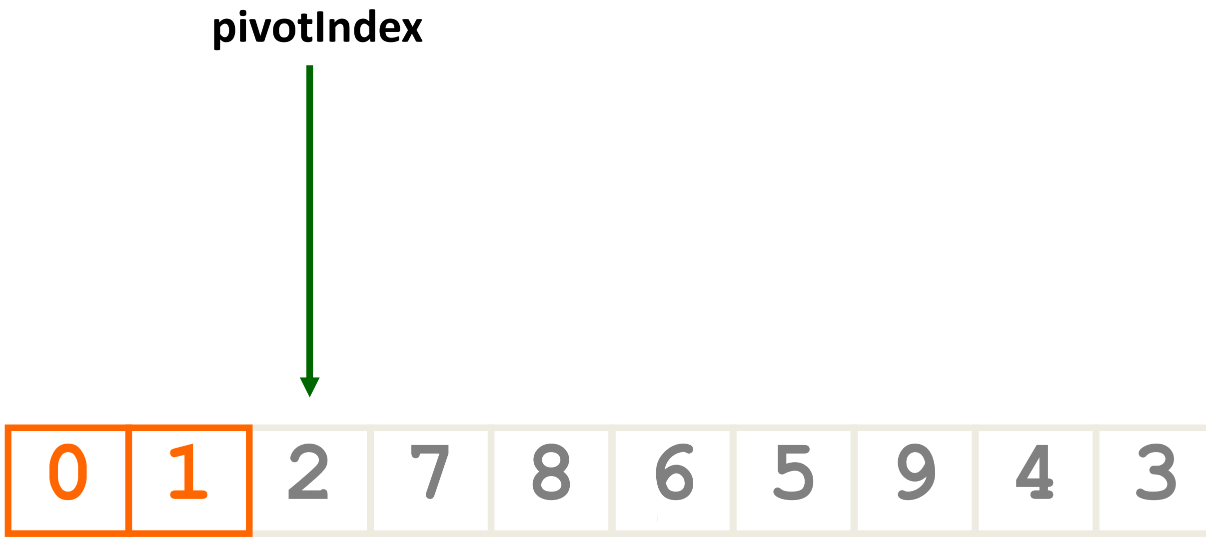
Quicksort Animation

pivotIndex

0	1	2	7	8	6	5	9	4	3

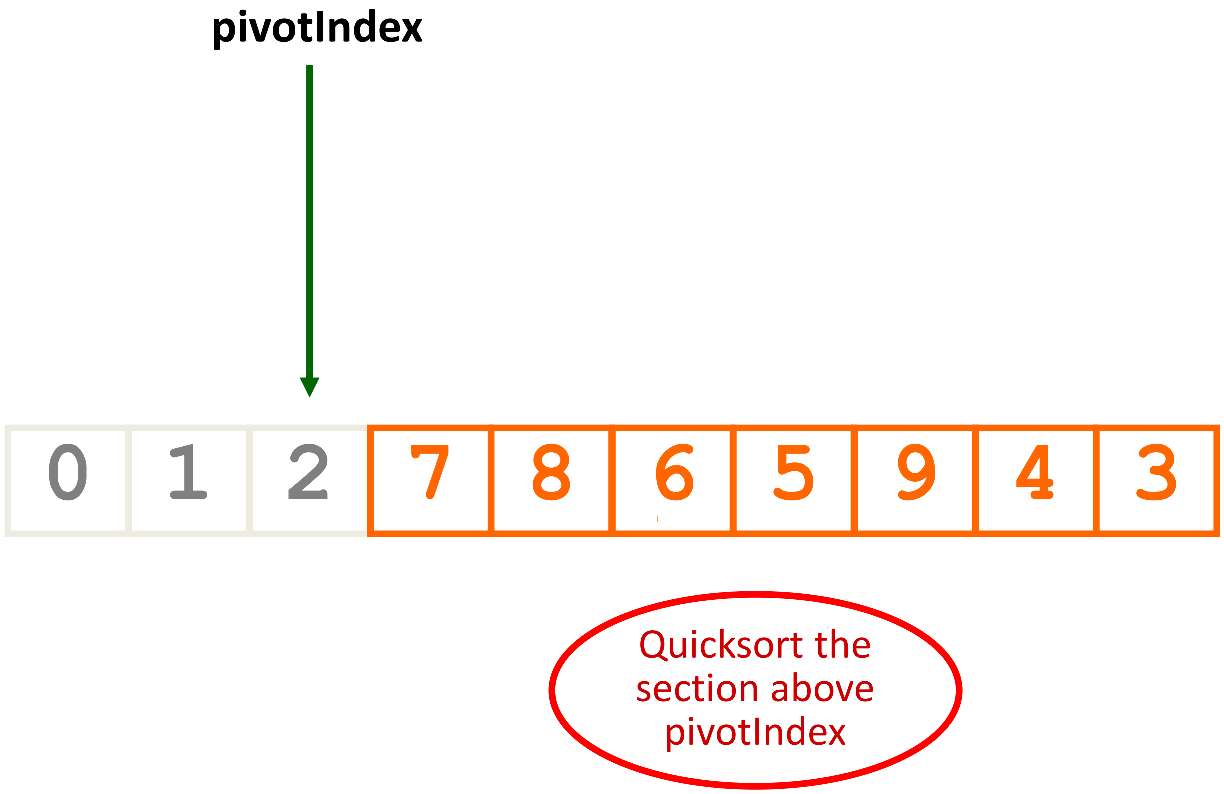
Quicksort the section below pivotIndex

Quicksort Animation



Less than three elements and already in order

Quicksort Animation



Quicksort Animation

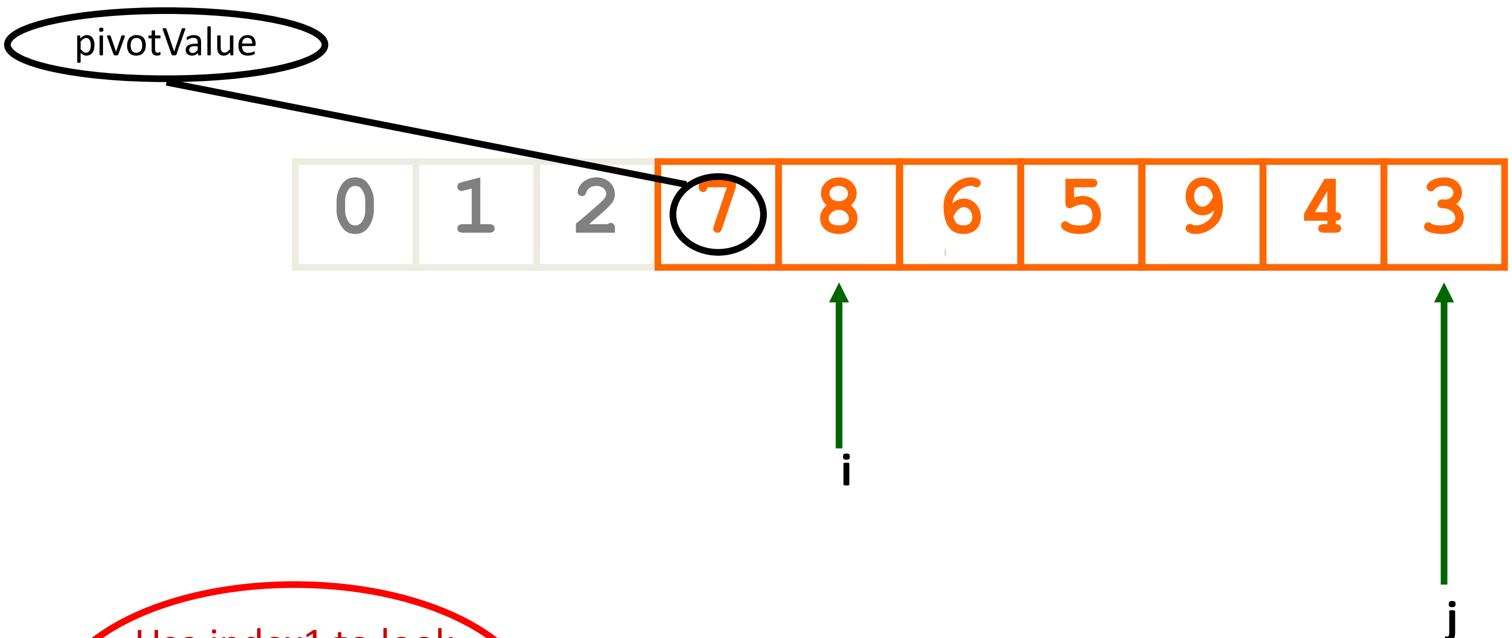
Split()

pivotValue

0	1	2	7	8	6	5	9	4	3

Quicksort Animation

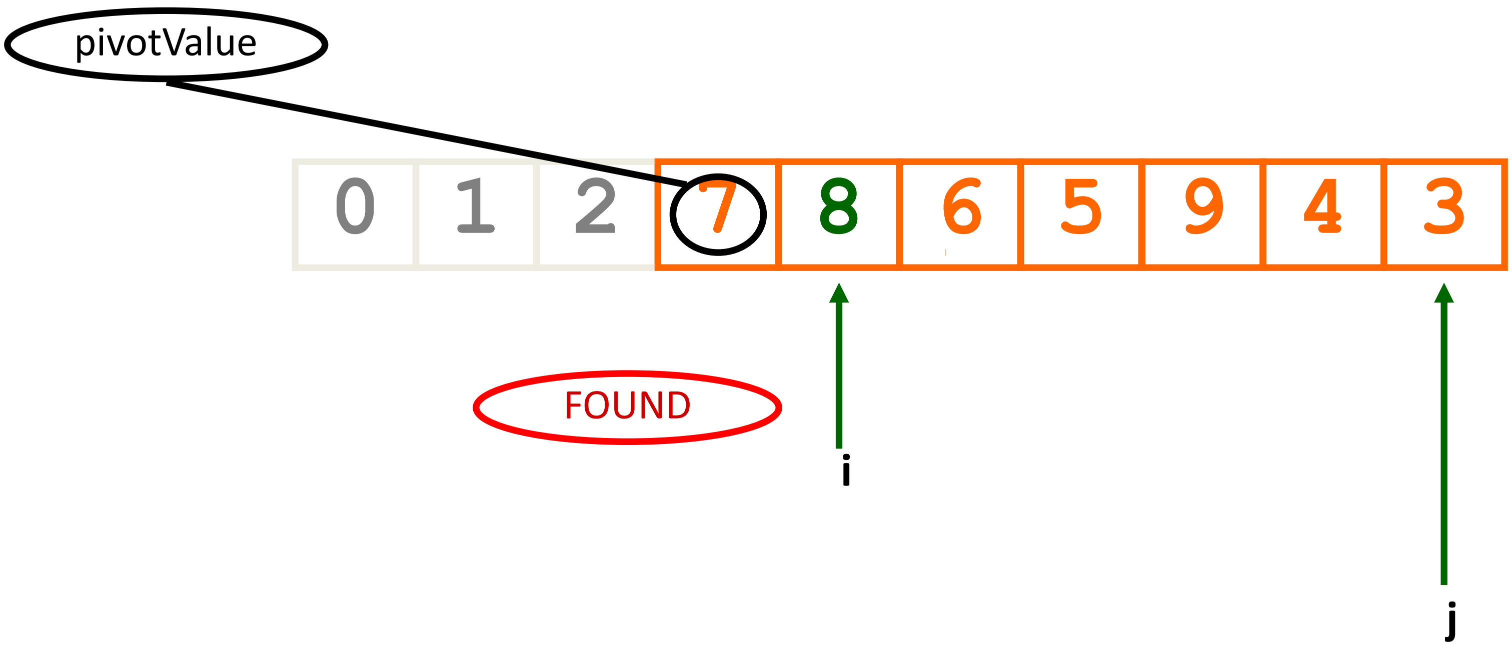
Split()



Use index1 to look for a number *larger* than the pivot value.

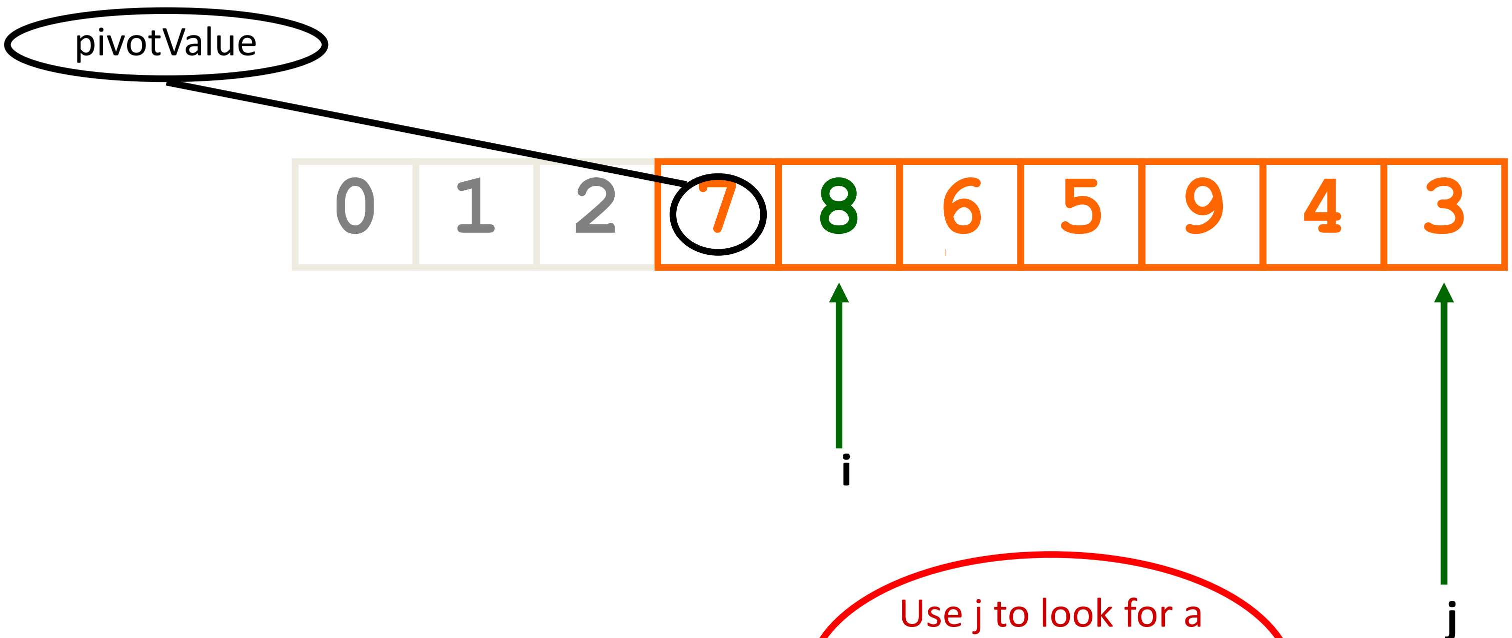
Quicksort Animation

Split()



Quicksort Animation

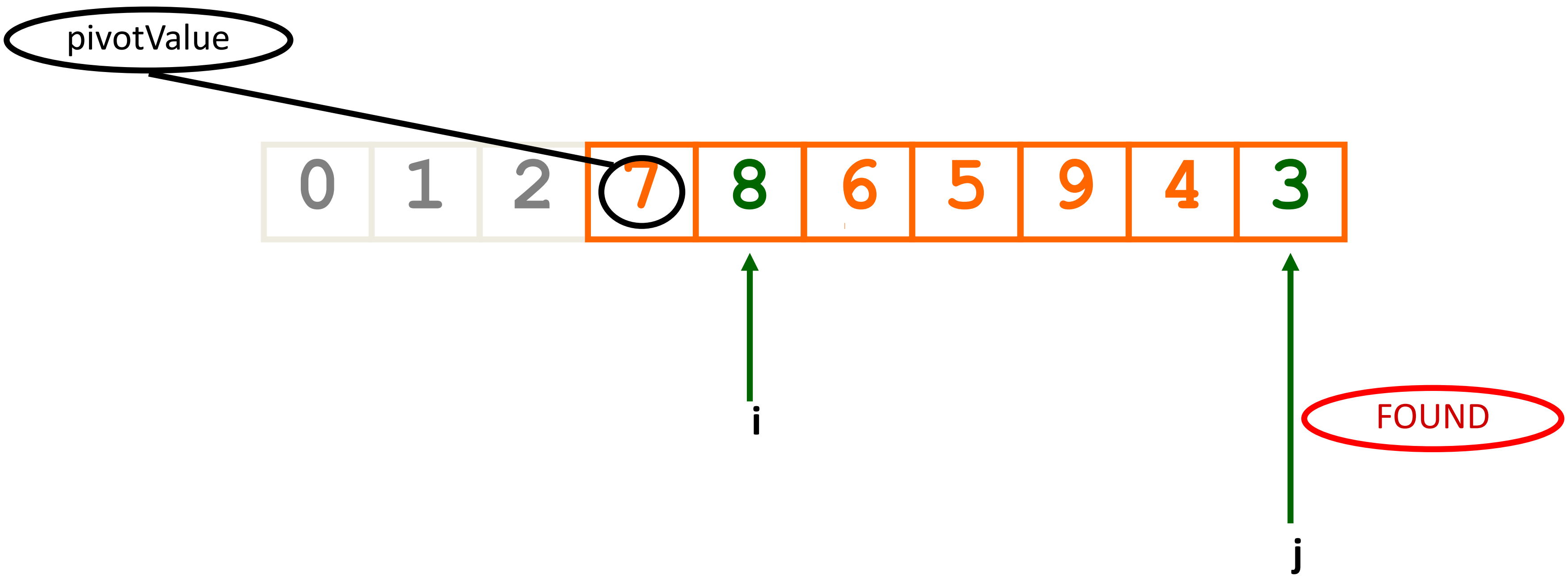
Split()



Use j to look for a number *smaller* than the pivot value.

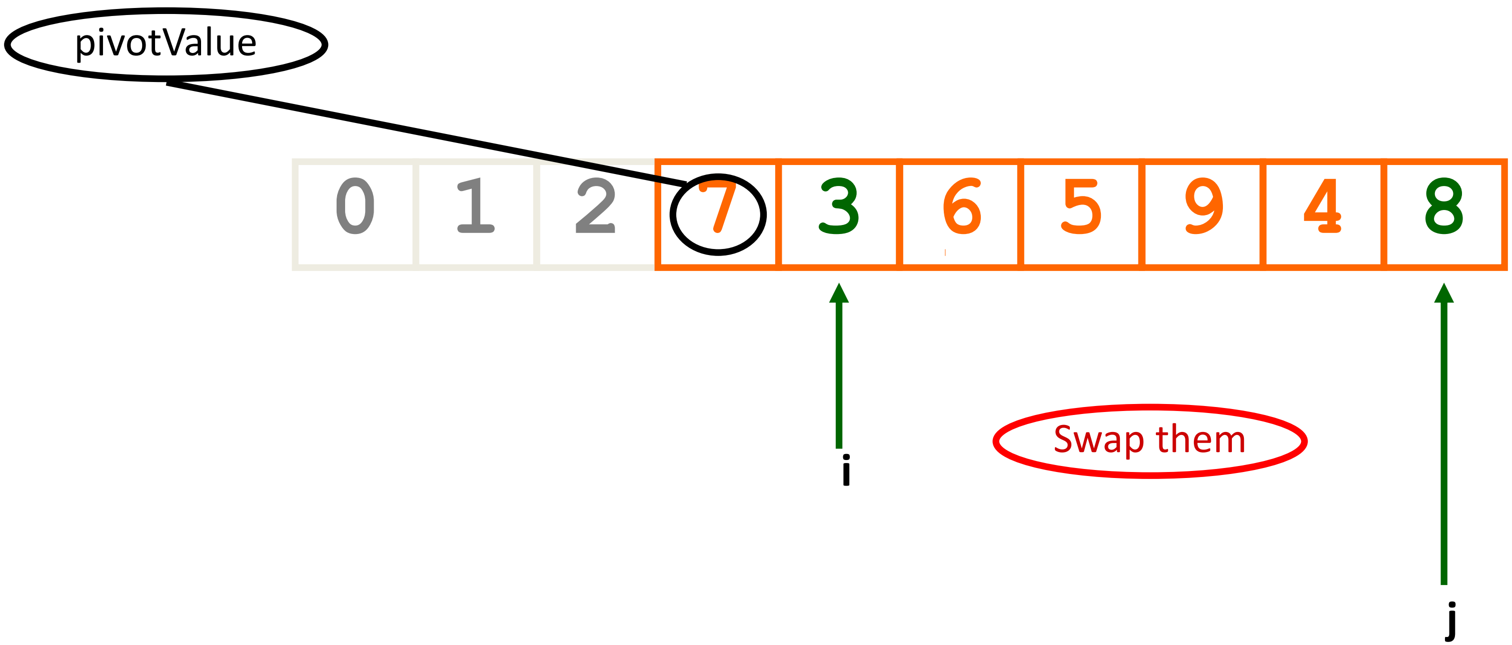
Quicksort Animation

Split()



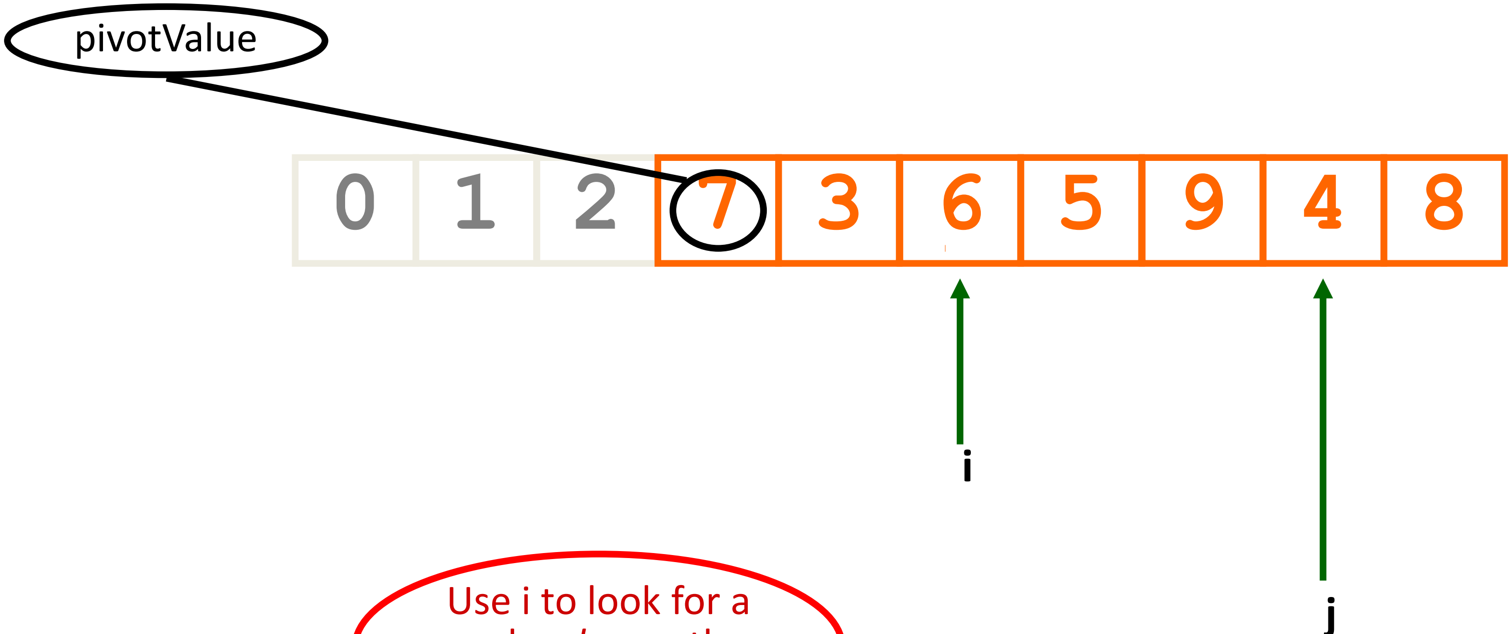
Quicksort Animation

Split()



Quicksort Animation

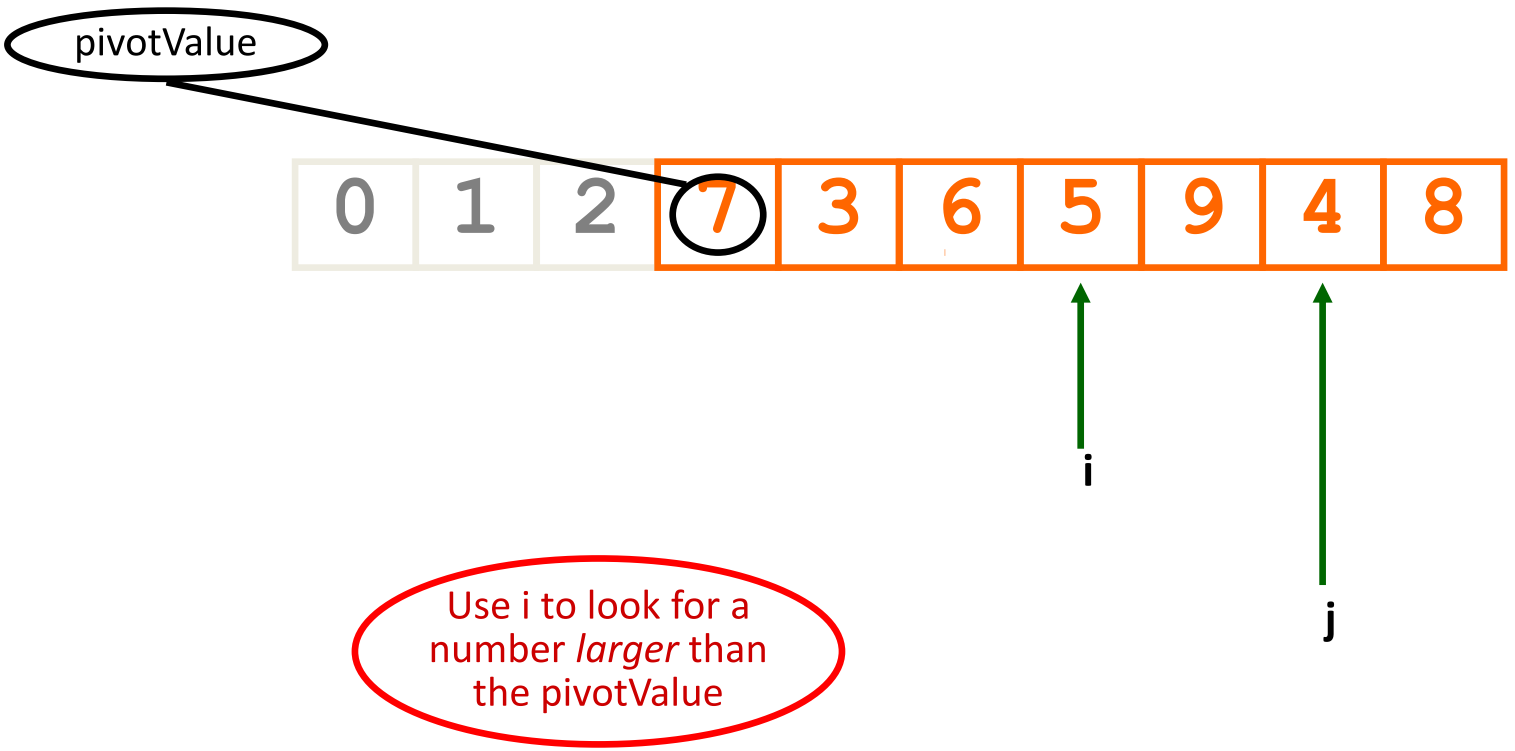
Split()



Use i to look for a number *larger* than the pivotValue

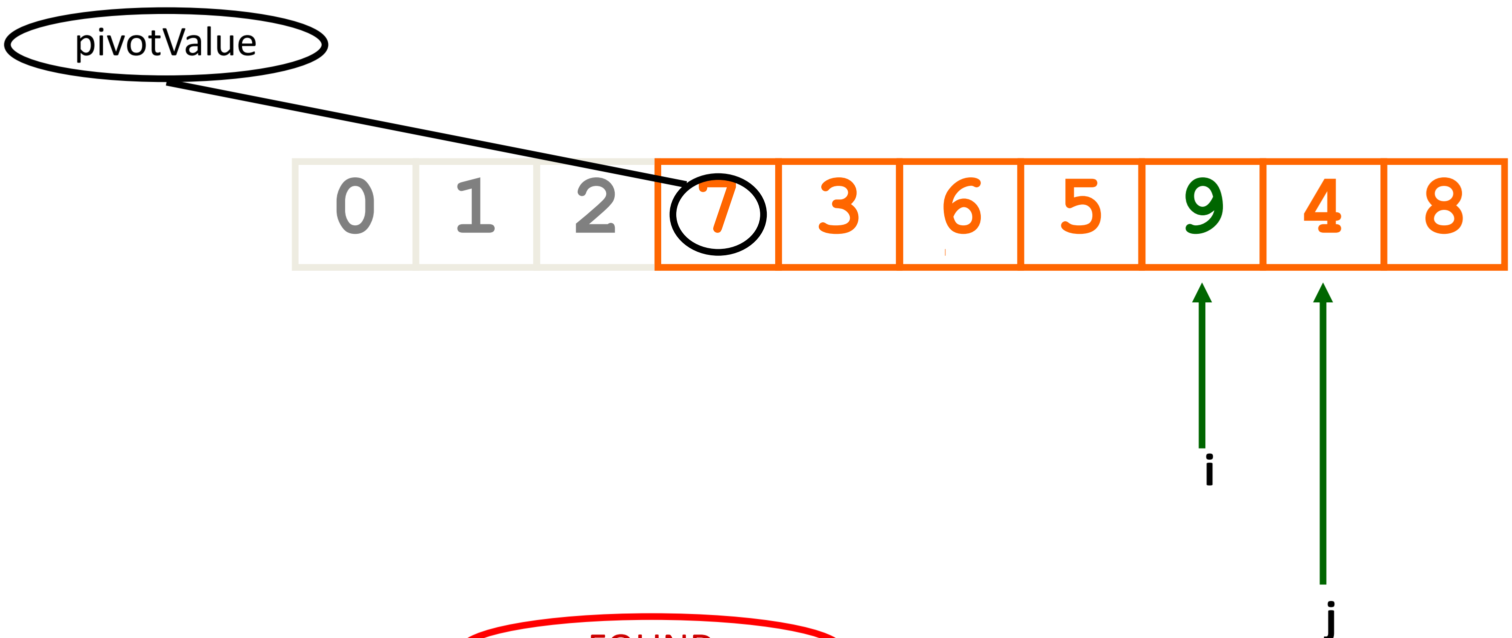
Quicksort Animation

Split()



Quicksort Animation

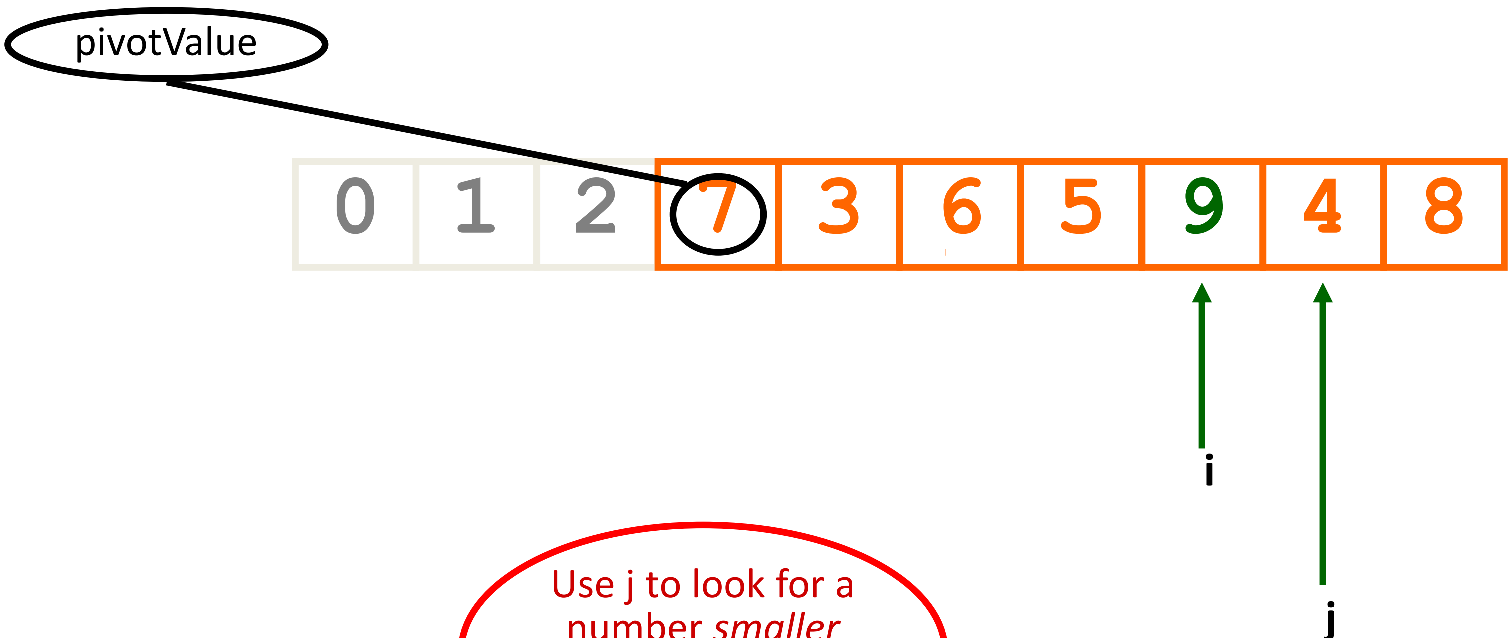
Split()



FOUND

Quicksort Animation

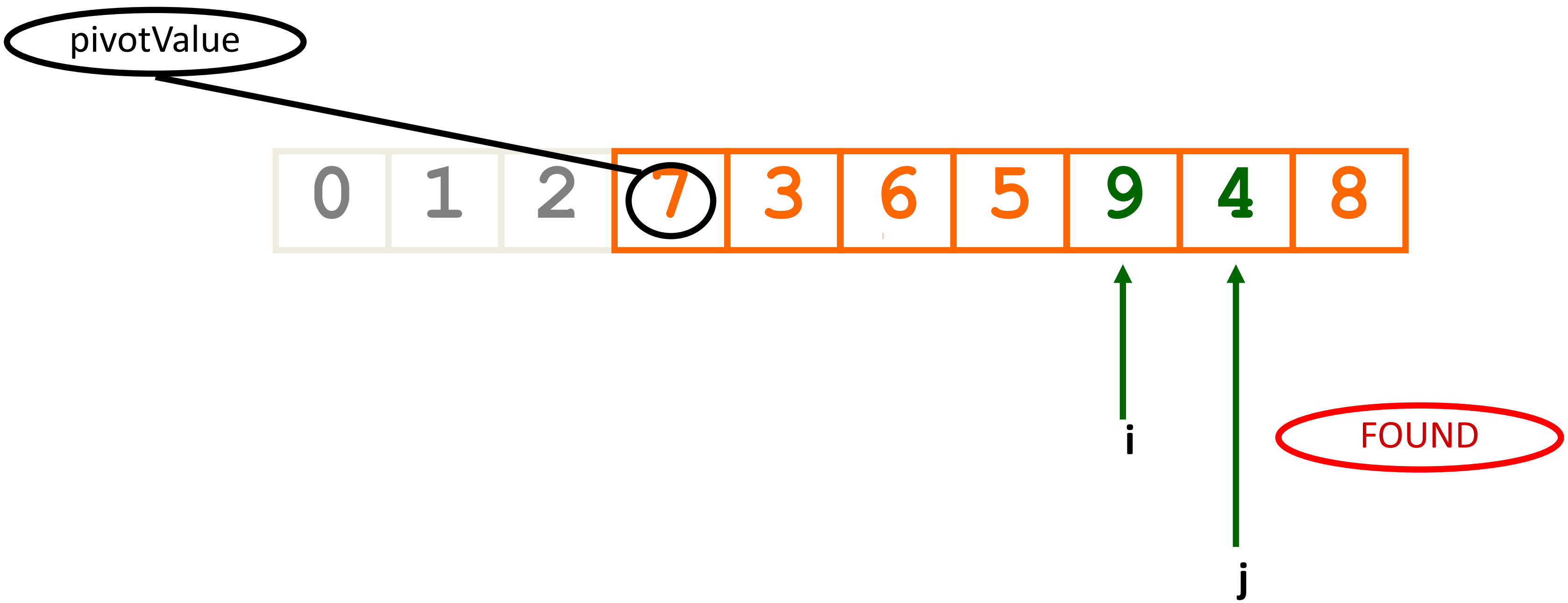
Split()



Use j to look for a number *smaller* than the pivot value

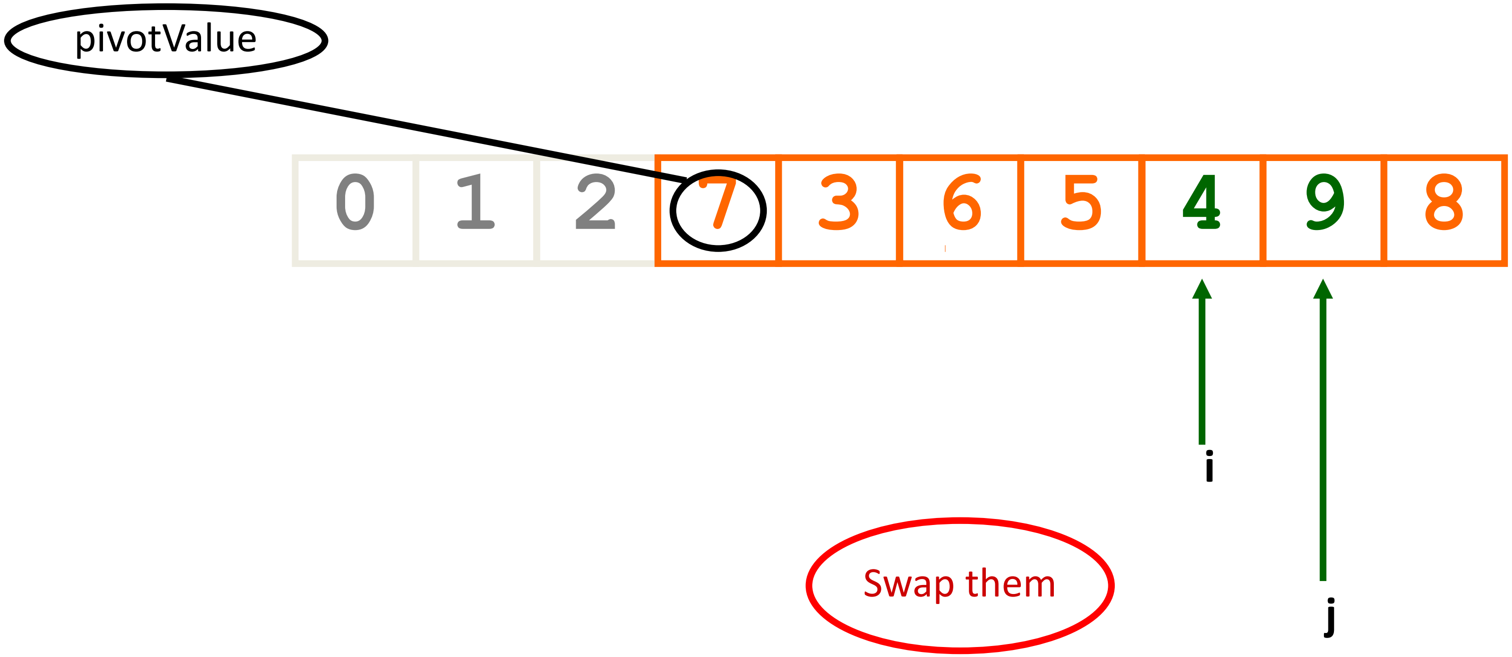
Quicksort Animation

Split()



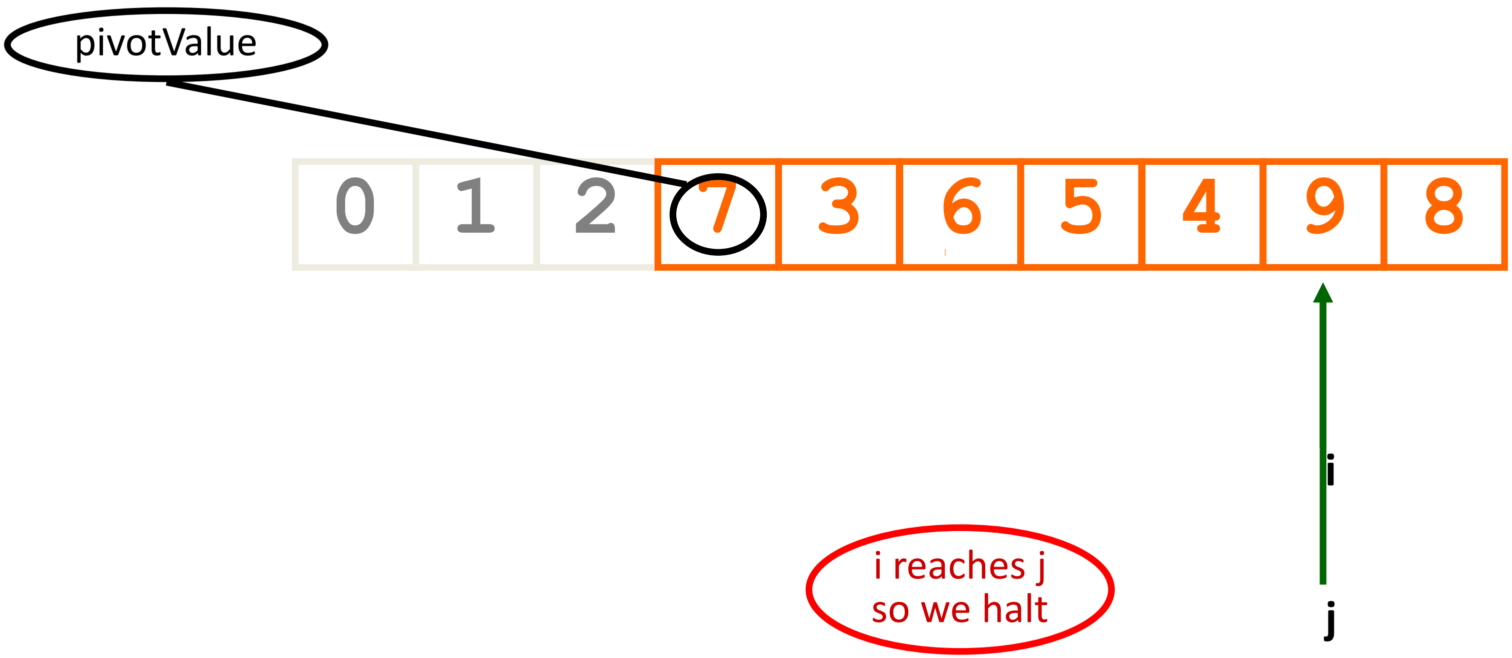
Quicksort Animation

Split()



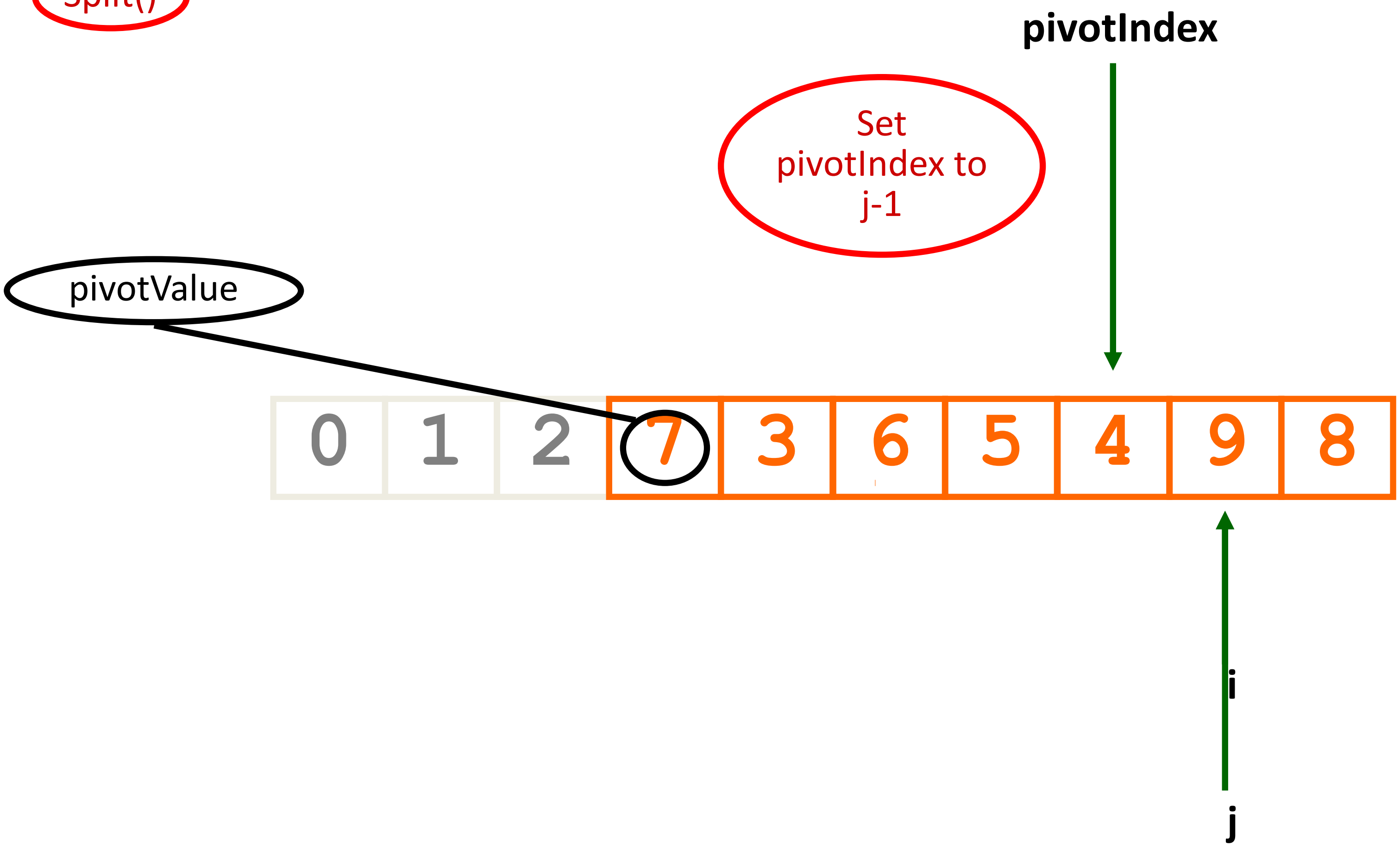
Quicksort Animation

Split()



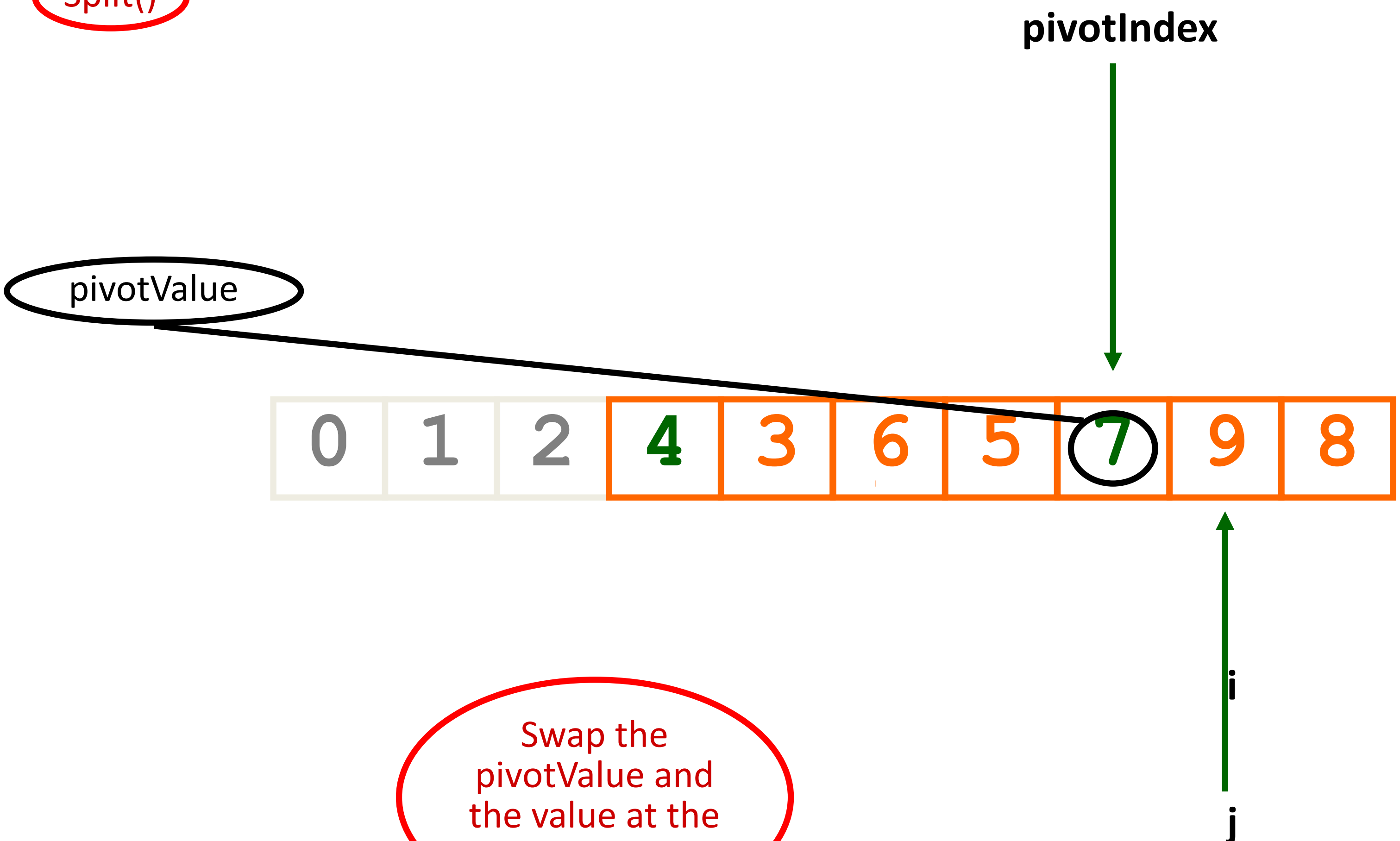
Quicksort Animation

Split()



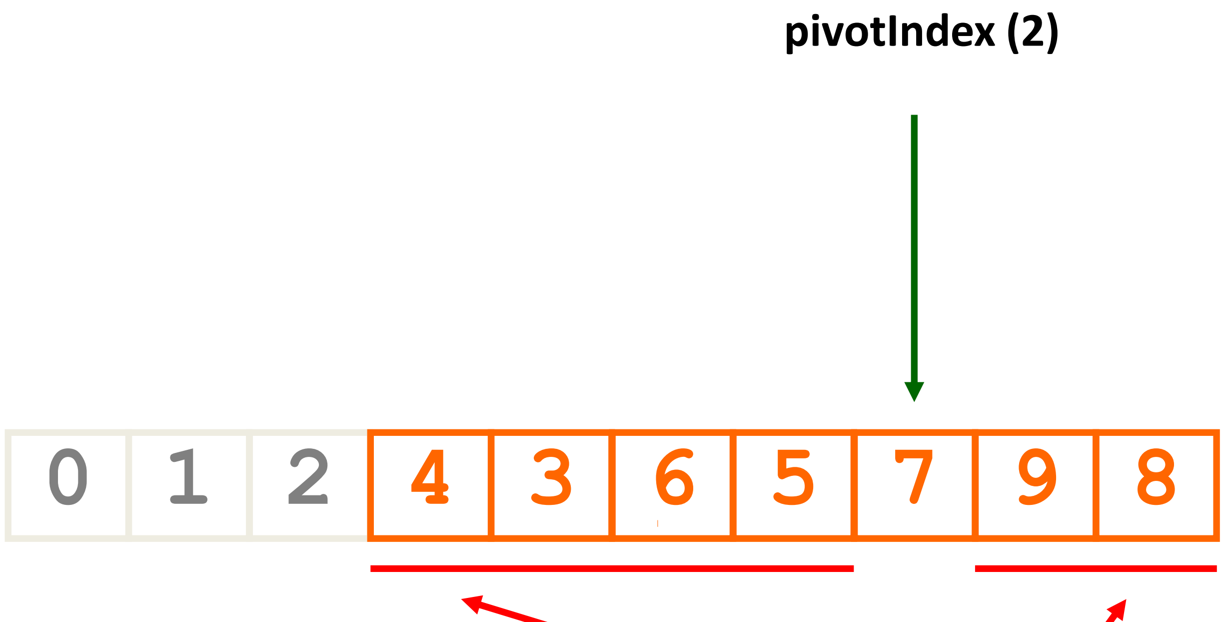
Quicksort Animation

Split()



Swap the pivotValue and the value at the pivotIndex

Quicksort Animation



At the end of Split(), all numbers are sorted into two 'bins': those greater than the pivot value and those less than the pivot value

Quicksort Animation

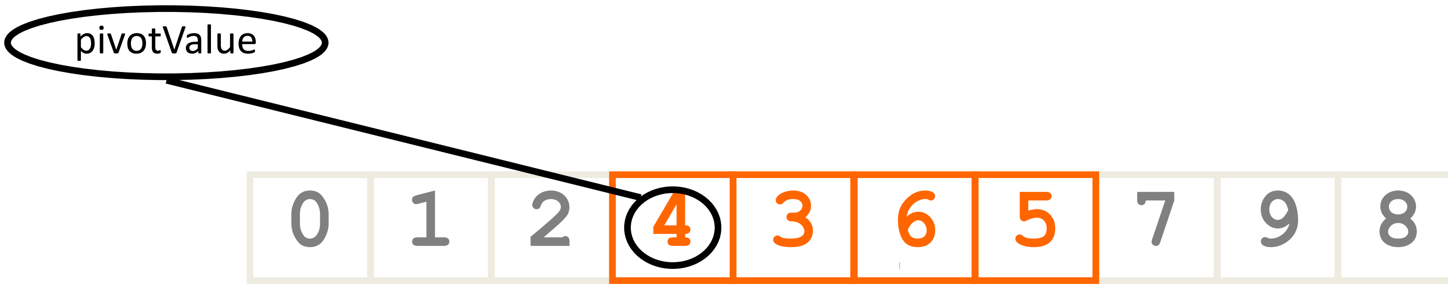
pivotIndex (2)

0	1	2	4	3	6	5	7	9	8

Quicksort the section below the pivotIndex

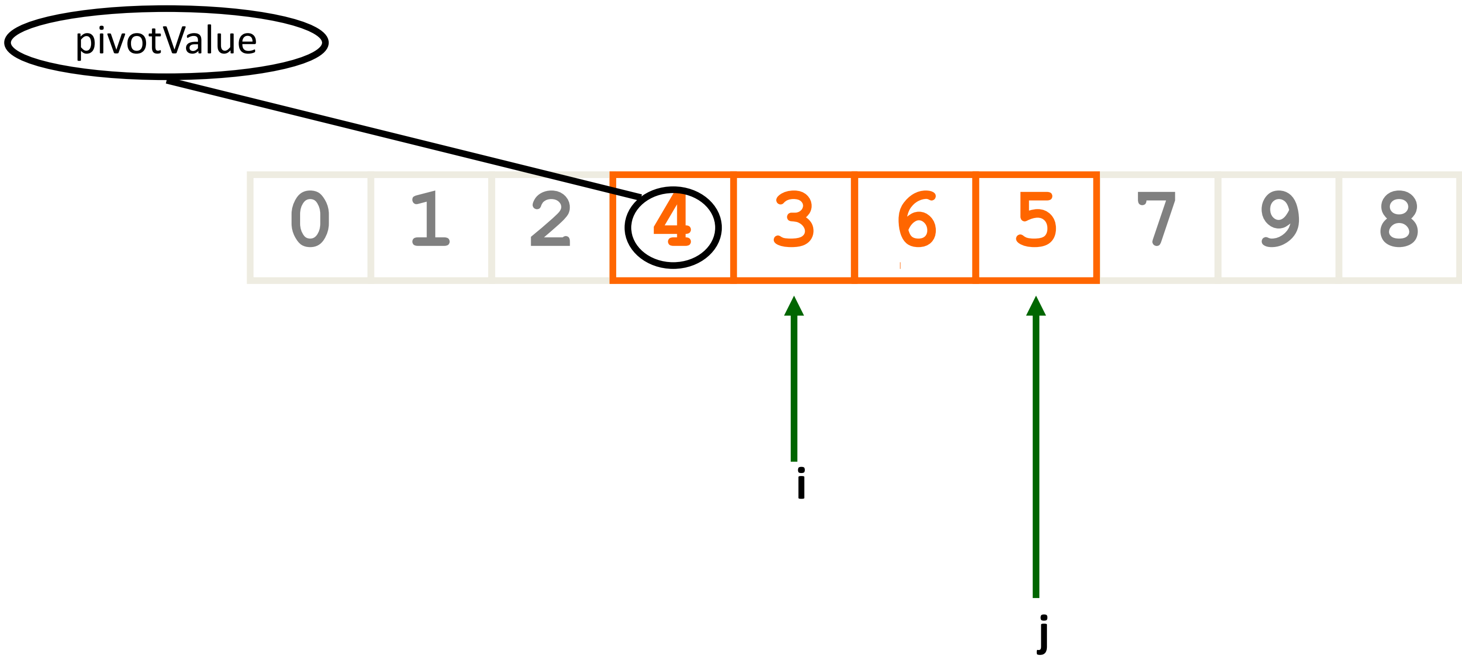
Quicksort Animation

Split()



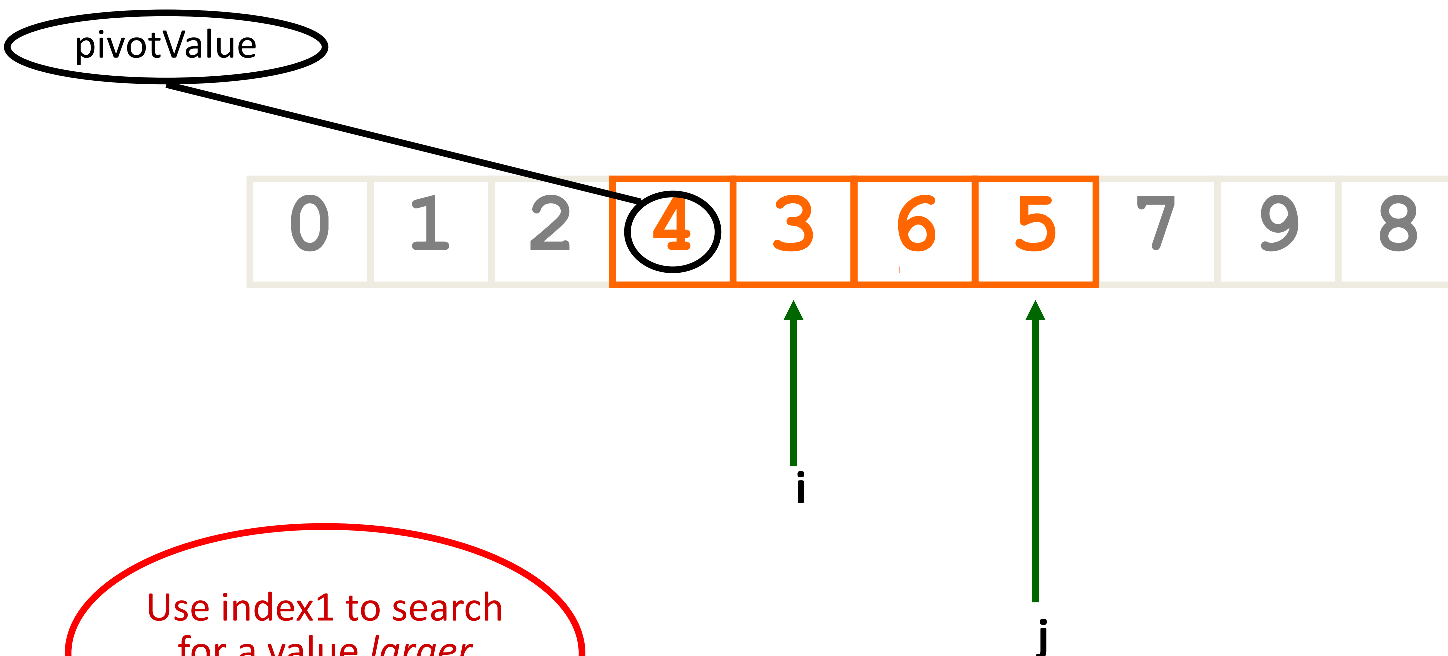
Quicksort Animation

Split()



Quicksort Animation

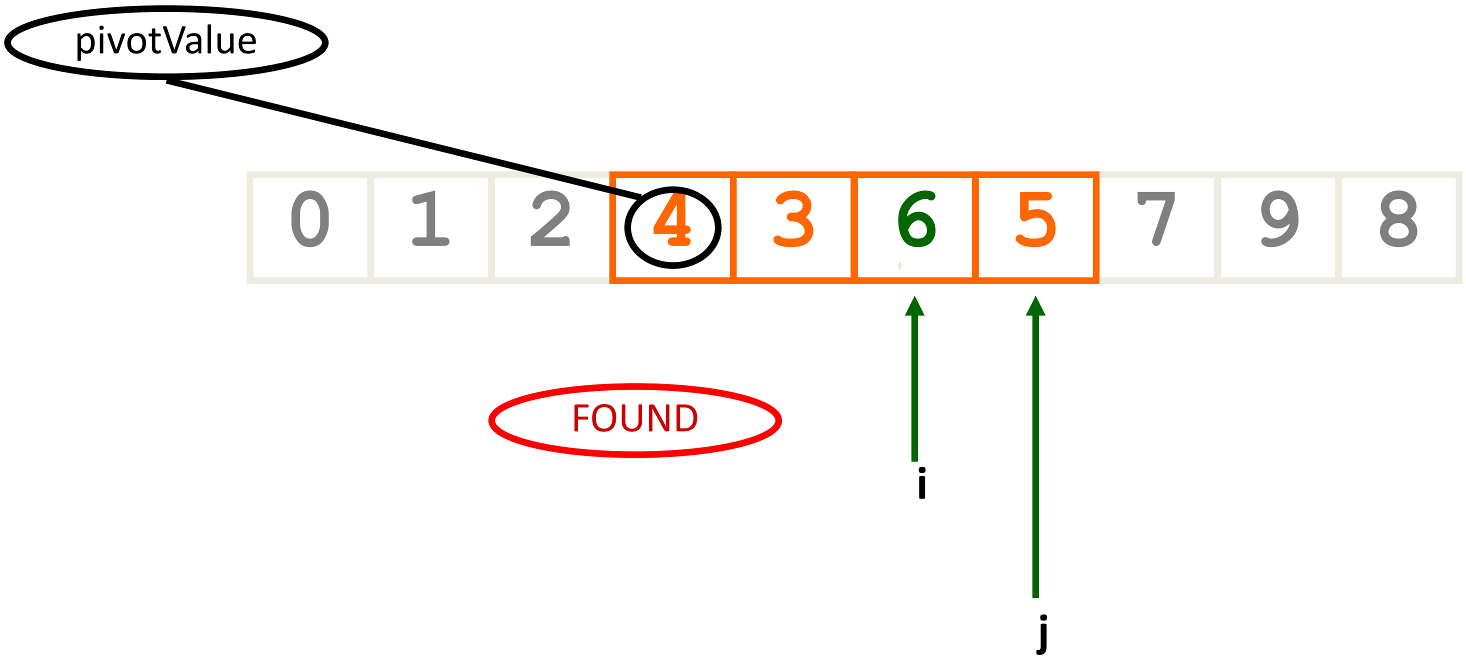
Split()



Use index1 to search for a value *larger* than the pivotValue

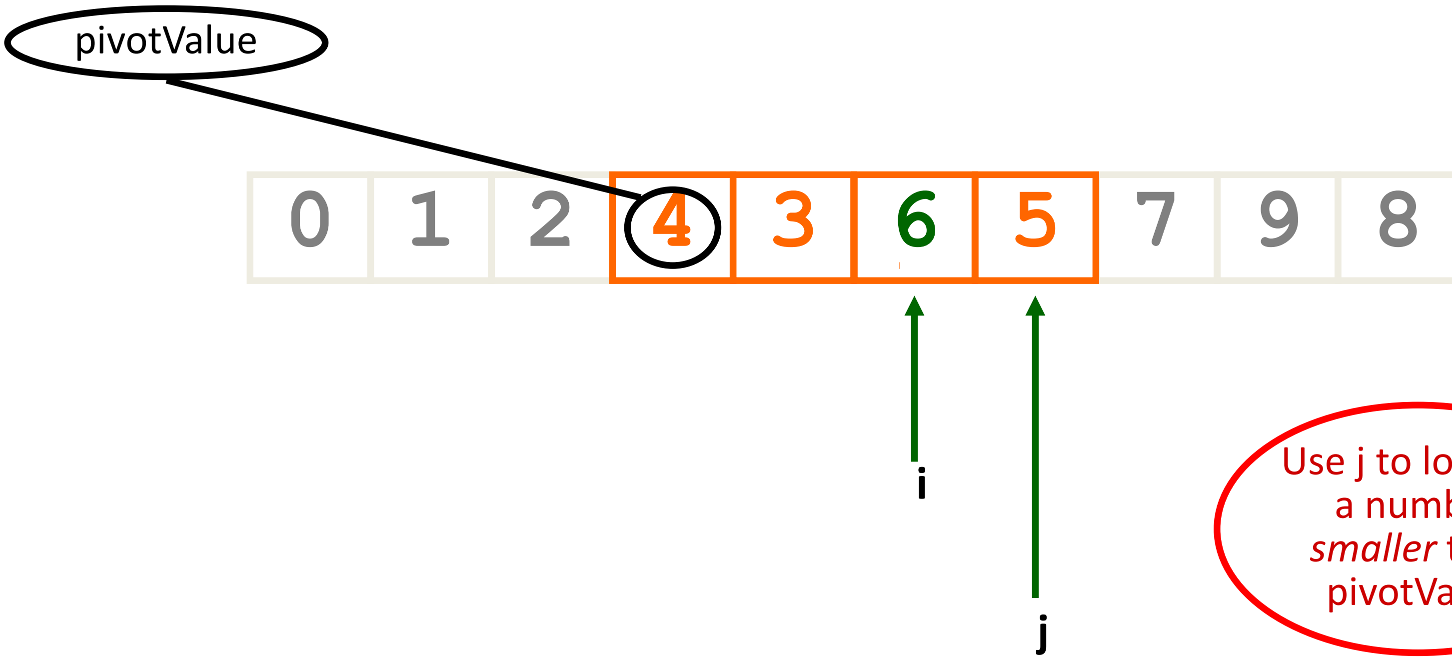
Quicksort Animation

Split()



Quicksort Animation

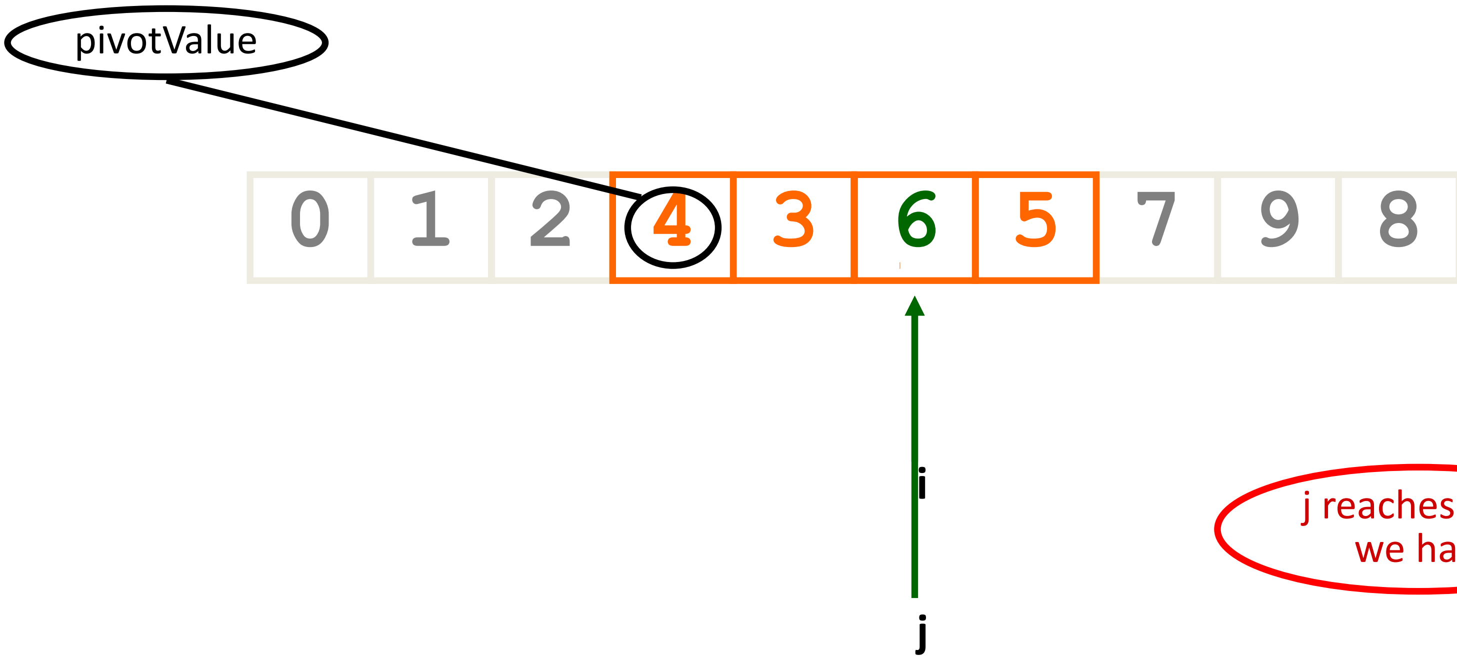
Split()



Use j to look for a number *smaller* than pivotValue

Quicksort Animation

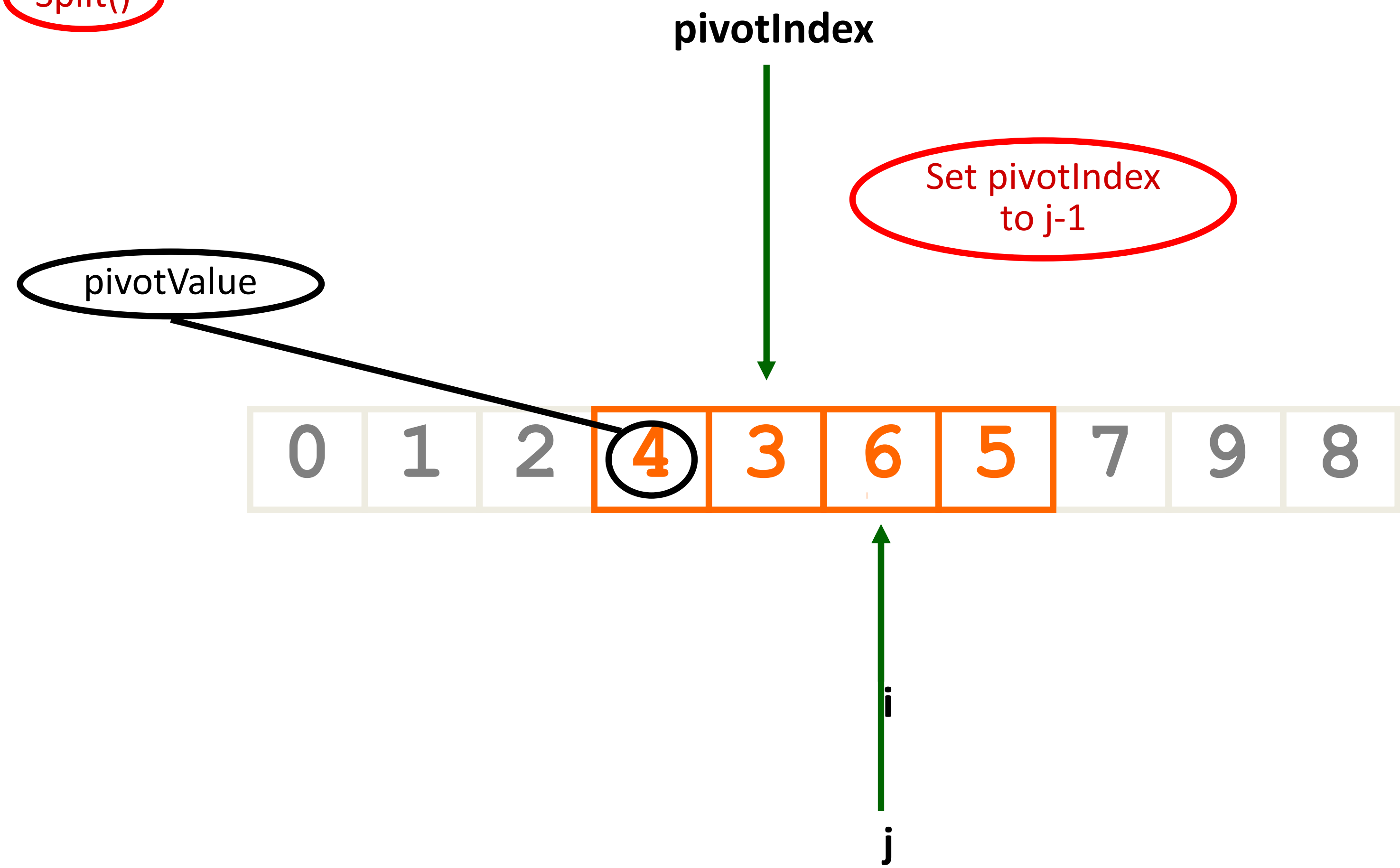
Split()



j reaches i, so we halt

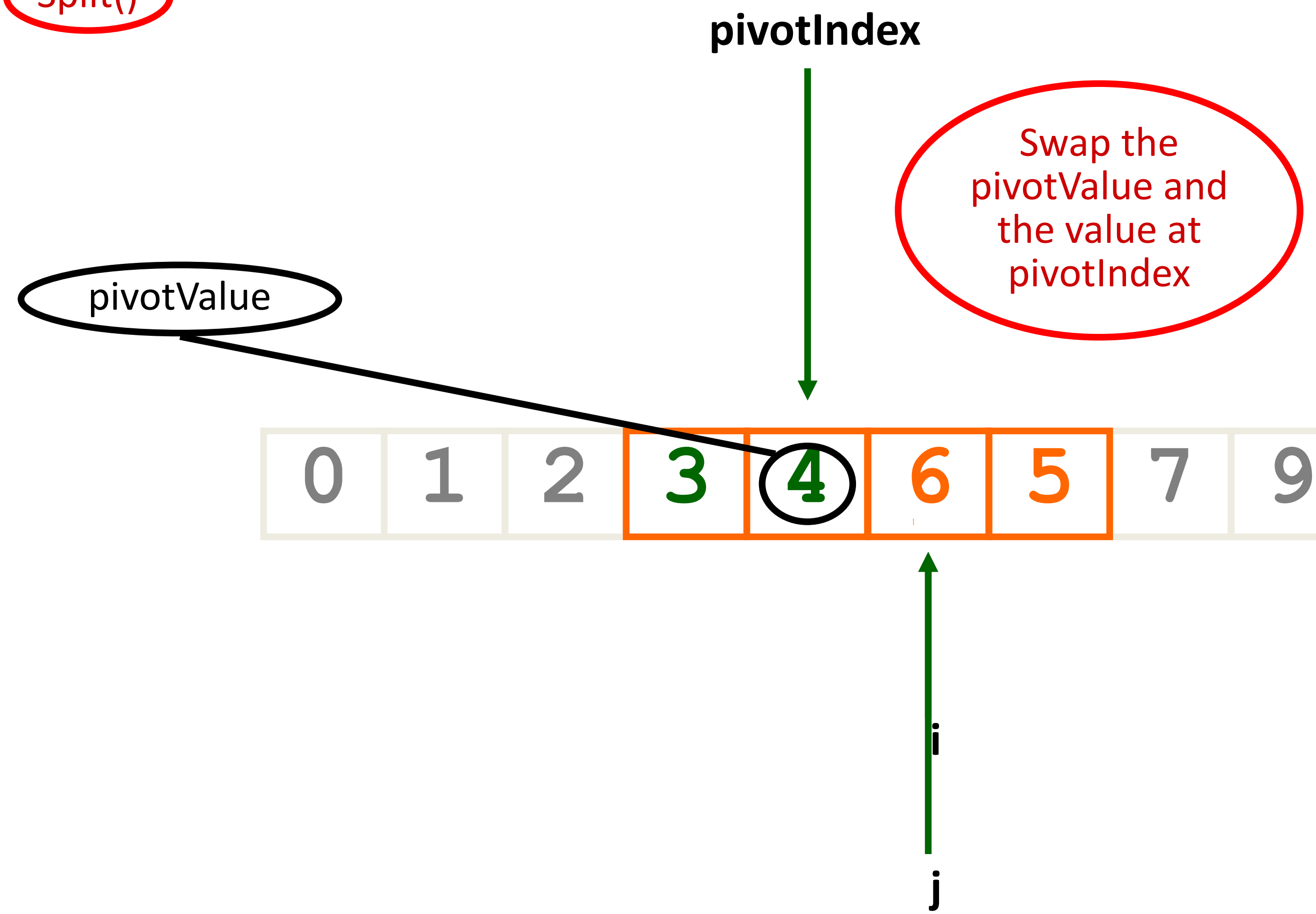
Quicksort Animation

Split()

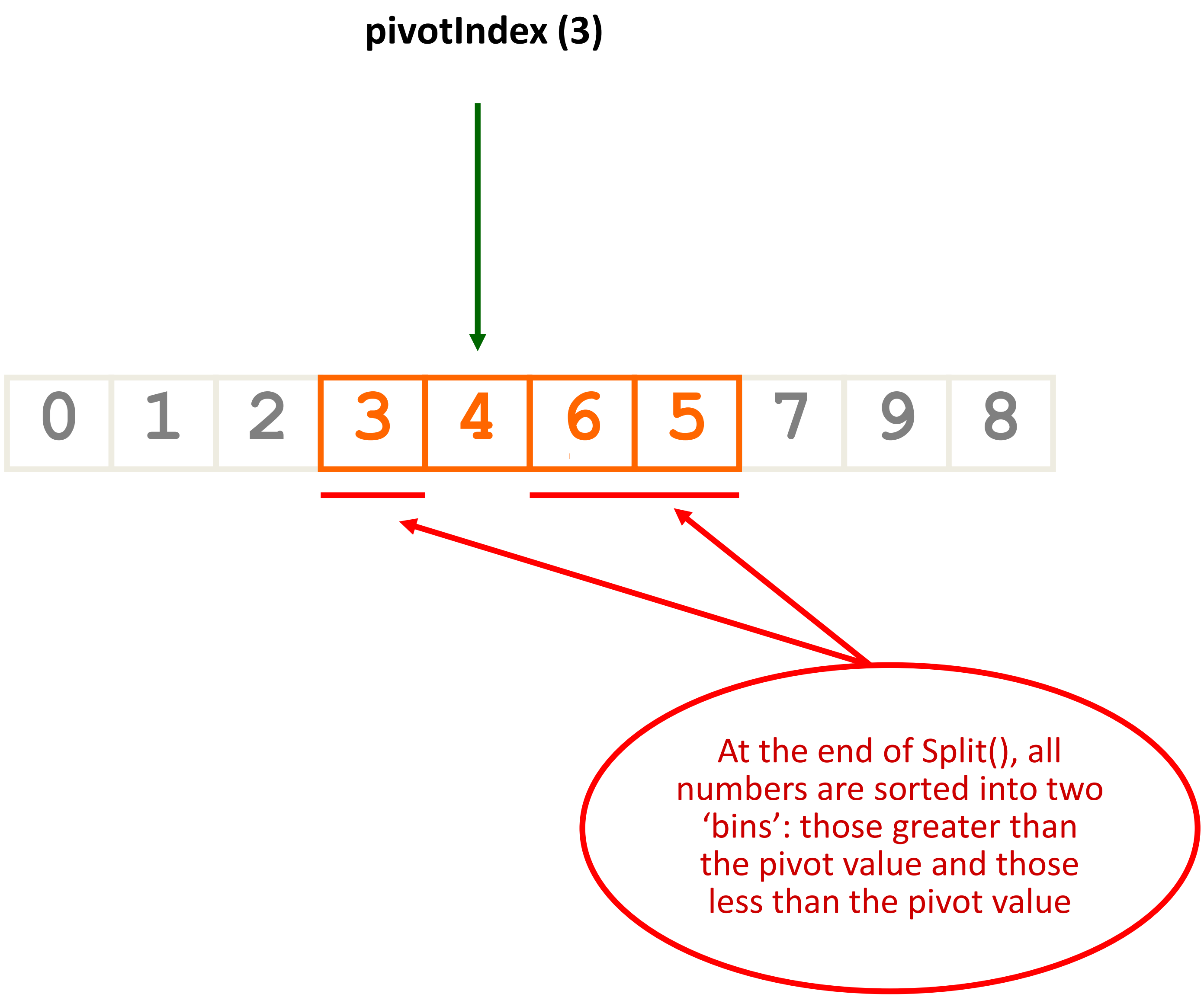


Quicksort Animation

Split()



Quicksort Animation



Quicksort Animation

pivotIndex (3)

0	1	2	3	4	6	5	7	9	8

Only 1 value
below pivotIndex,
so do nothing

Quicksort Animation

pivotIndex (3)

0	1	2	3	4	6	5	7	9	8

Only two values above pivotIndex, but out of order

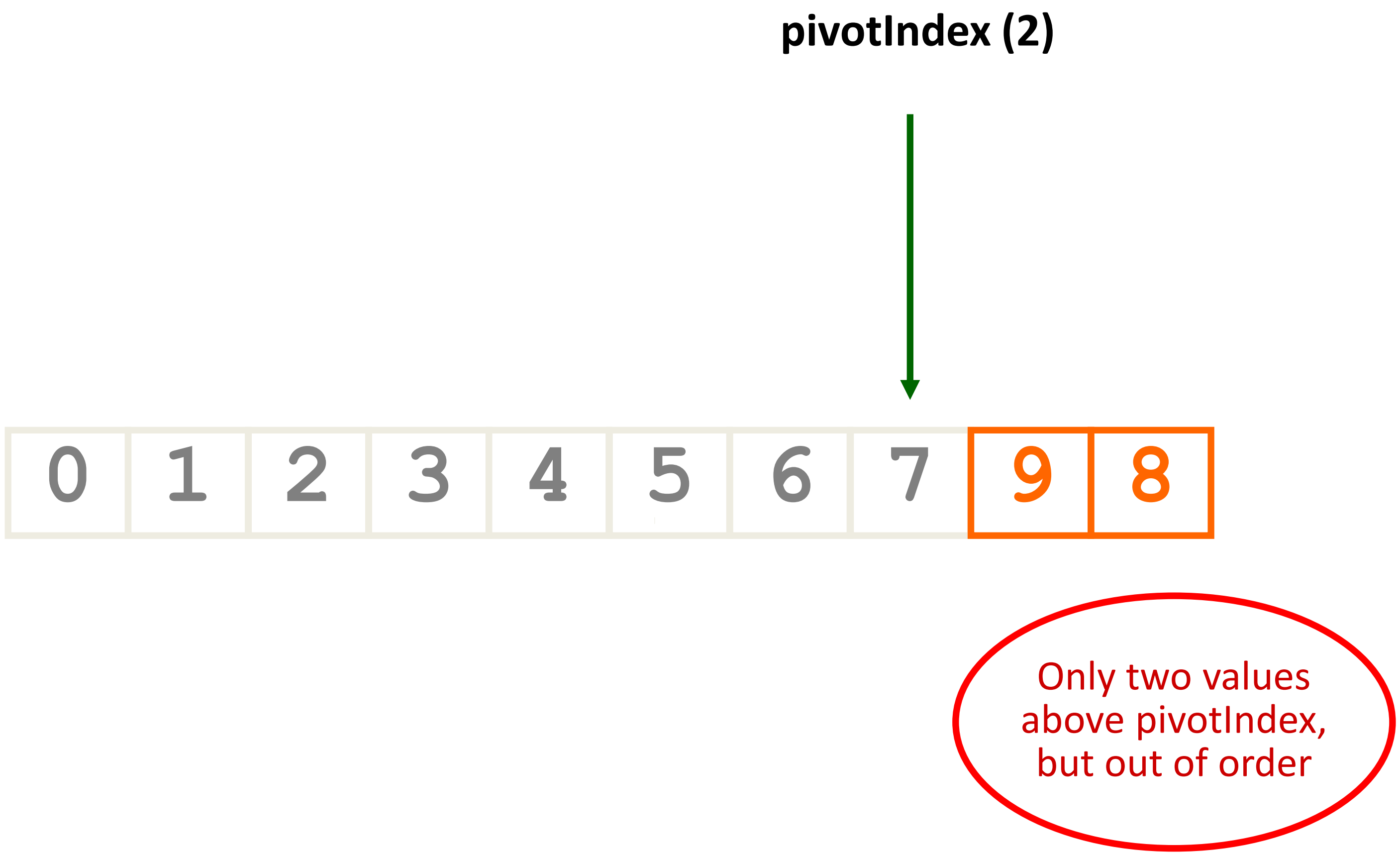
Quicksort Animation

pivotIndex (3)

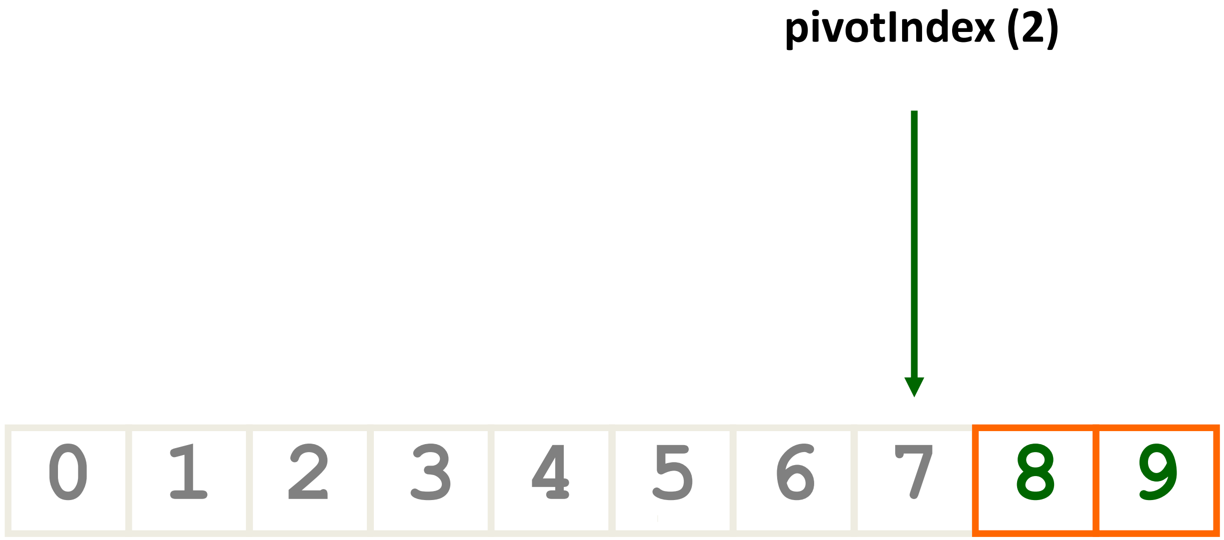
0	1	2	3	4	5	6	7	9	8

So swap them

Quicksort Animation



Quicksort Animation



So swap them

Quicksort Animation

pivotIndex

0	1	2	3	4	5	6	7	8	9

Done!!

Quick Sort - Complexity

Time complexity

- T(n) = T(k) + T(n-k-1) + θ(n)
  - K = number of elements smaller than pivot
- On average, each partition halves the size of the array to be sorted
- On average, each partition swaps half the elements.
- On average, algorithm is $O(n \log n)$.
  - T(n) = 2T(n/2) + θ(n)
- Worst case, algorithm is $O(n^2)$.

Quick Sort – Choice of Pivot

- In this version of quicksort, the leftmost element of the partition is used as the pivot element.
- Unfortunately, this causes worst-case behavior on already sorted arrays because the size of sub-array is only reduced by 1.

- This problem is easily solved by choosing:
  1. a random index for the pivot, or
  2. the middle index of the partition for the pivot, or
  3. the median of the first, middle and last elements of the partition for the pivot.