Data structures and Algorithms

SORTING ALGORITHMS

(Part II)

Nguyễn Ngọc Thảo
nnthao@fit.hcmus.edu.vn

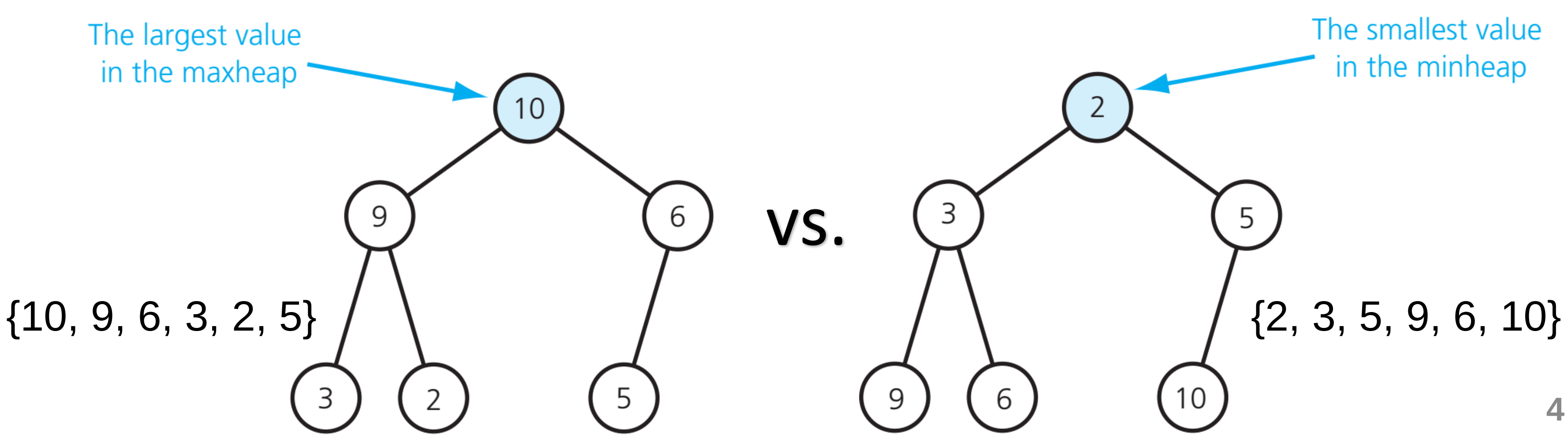
Outline

- Heap sort
- Merge sort
- Quick sort

Heap sort

Heap structures

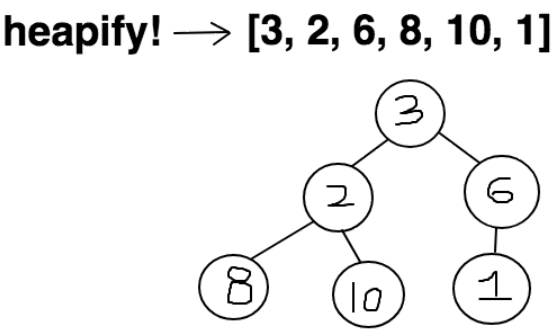
- A max heap is a sequence of n elements, $(h_1, h_2, .., h_n)$, such that $h_i \geq h_{2i}$ and $h_i \geq h_{2i+1}$ for all $i = 1, 2, .., \left\lfloor \frac{n}{2} \right\rfloor$
- The sequence of elements $h_{\left\lfloor \frac{n}{2} \right\rfloor + 1}, .., h_n$ is a natural heap.
- The element h_1 of a heap is the largest value.
- We also have min heap with opposite characteristics.



Heap construction

- The heap is extended to the left where in each step a new element is included and properly positioned by a sift.

```
left = ⌊n/2⌋;
while (left > 0) {
   sift(a, left, n);
   left--;
}
```



- “Sift down”: The element on top of the subheap is swapped with its larger comparands.
  - This procedure stops when the element on top of the subheap is larger than or equal to both its comparands.

Heap construction: Implementation

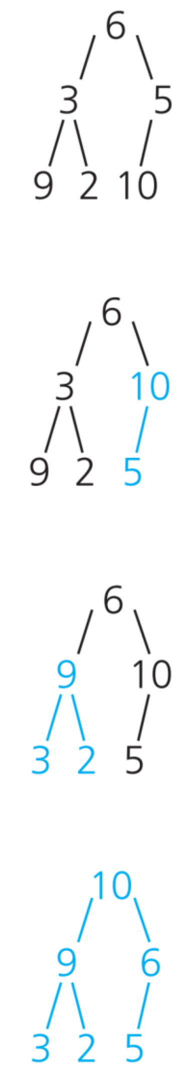
```
void heapRebuild(int start, int arr[], int n){
   int leftChild = 2 * start + 1; // A left child must exist
   if (leftChild >= n) return;
   int largerChild = leftChild;   // Make assumption about larger child
   int rightChild = 2 * start + 2;    // A right child might not exist
   if (rightChild < n){           // Whether a right child exists
      // A right child exists; check whether it is larger
      if (arr[rightChild] > arr[largerChild])
         largerChild = rightChild;      // Assumption was wrong
   }
   // If arr[start] is smaller than the larger child, swap values
   if (arr[start] < arr[largerChild]){
      swap(arr[largerChild], arr[start]);
      heapRebuild(largerChild, arr, n); // Recursion at that child
   }
}
```

An example of heap construction

Array

	0	1	2	3	4	5
Original array	6	3	5	9	2	10
After `heapRebuild(2)`	6	3	10	9	2	5
After `heapRebuild(1)`	6	9	10	3	2	5
After `heapRebuild(0)`	10	9	6	3	2	5

Tree representation of the array



Example: Build max/min heap from an array of integers

Create a max heap from the following array.

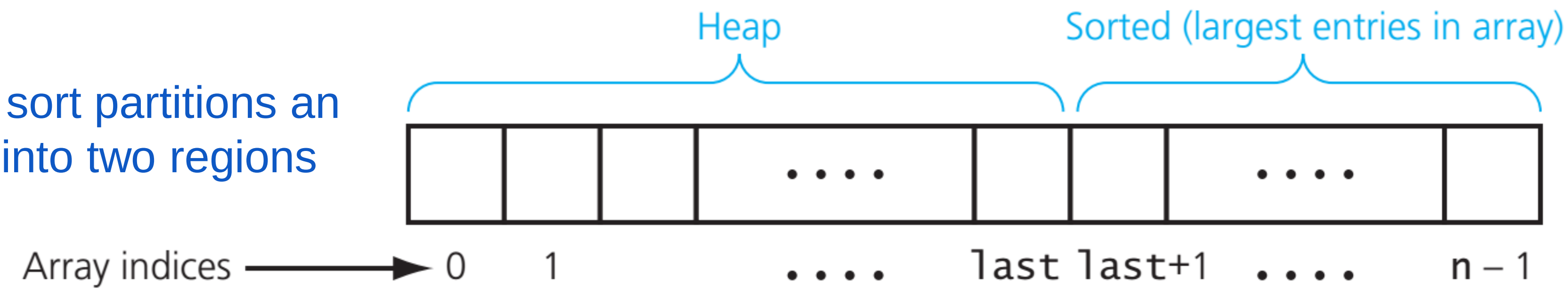
5	1	2	8	6	10	3	9	4	7
0	1	2	3	4	5	6	7	8	9

Similarly, create a min heap from the given array.

Heap sort (J.W.J. Williams, 1964)

- Improve selection sort by retaining from each scan more information than just the recognition of the single least item.
- Construct a max heap from the given array and repeatedly **move** the largest element in the heap to the end of the array
- Elements are moved from the heap in descending order and placed into sequentially decreasing positions in the array.

Heap sort partitions an array into two regions



Heap sort: Algorithm

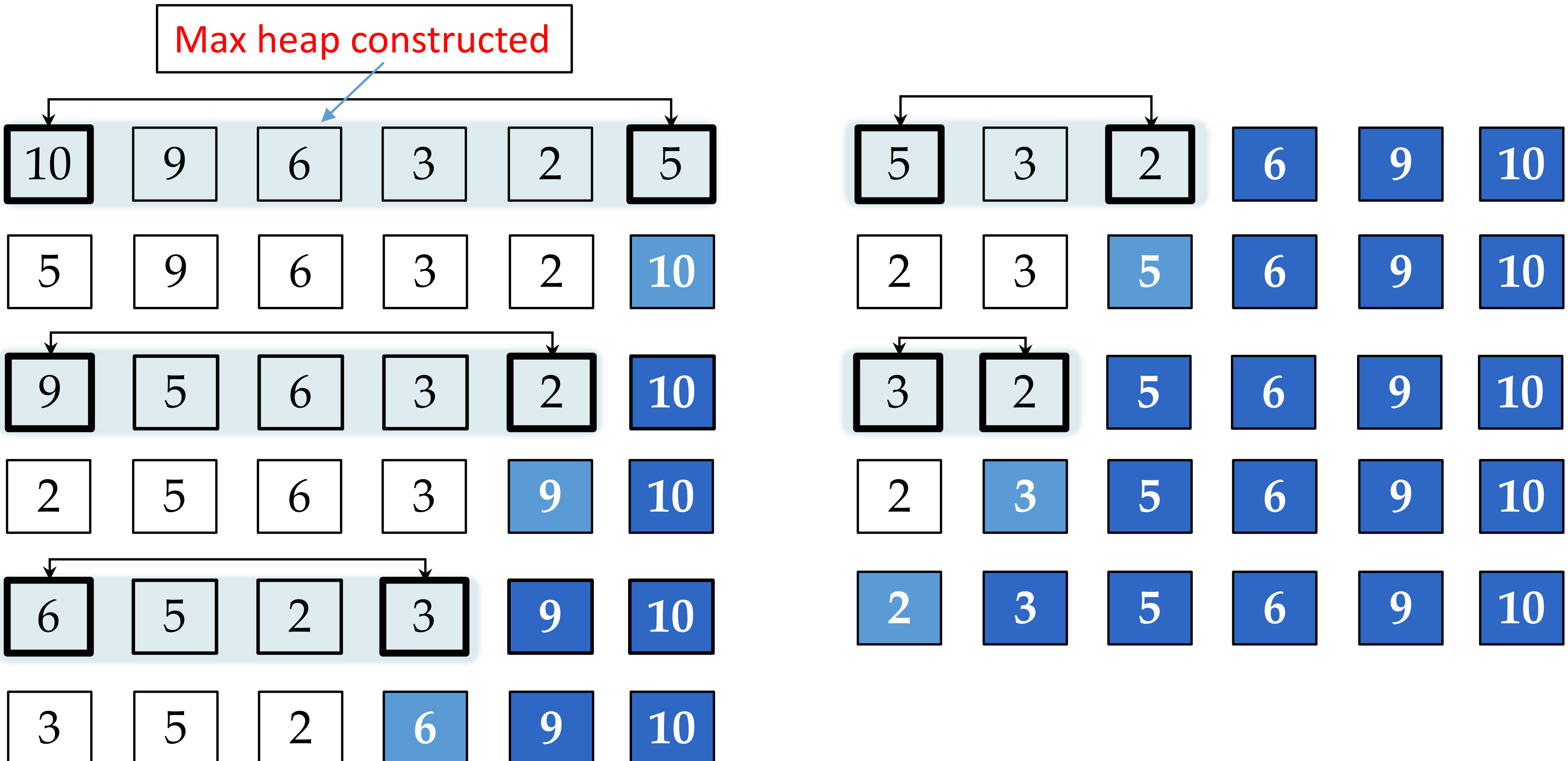
- Consider the array of n elements, $a[1..n]$.
- Phase 1. Heap construction. Construct a heap for the array
- Phase 2. Maximum deletion. Apply maximum key deletion $n-1$ times to the remaining heap
    - Swap the first element and the last element of the heap
    - Decrease $heapSize$ by 1, $heapSize = n-1;$
    - While $heapSize > 1$
        - Rebuild the heap at the first position, $a[1..heapSize]$
        - Swap the first element and the last element of the heap
        - Decrease $heapSize$ by 1, $heapSize = heapSize - 1;$

Heap sort: Imeplementation

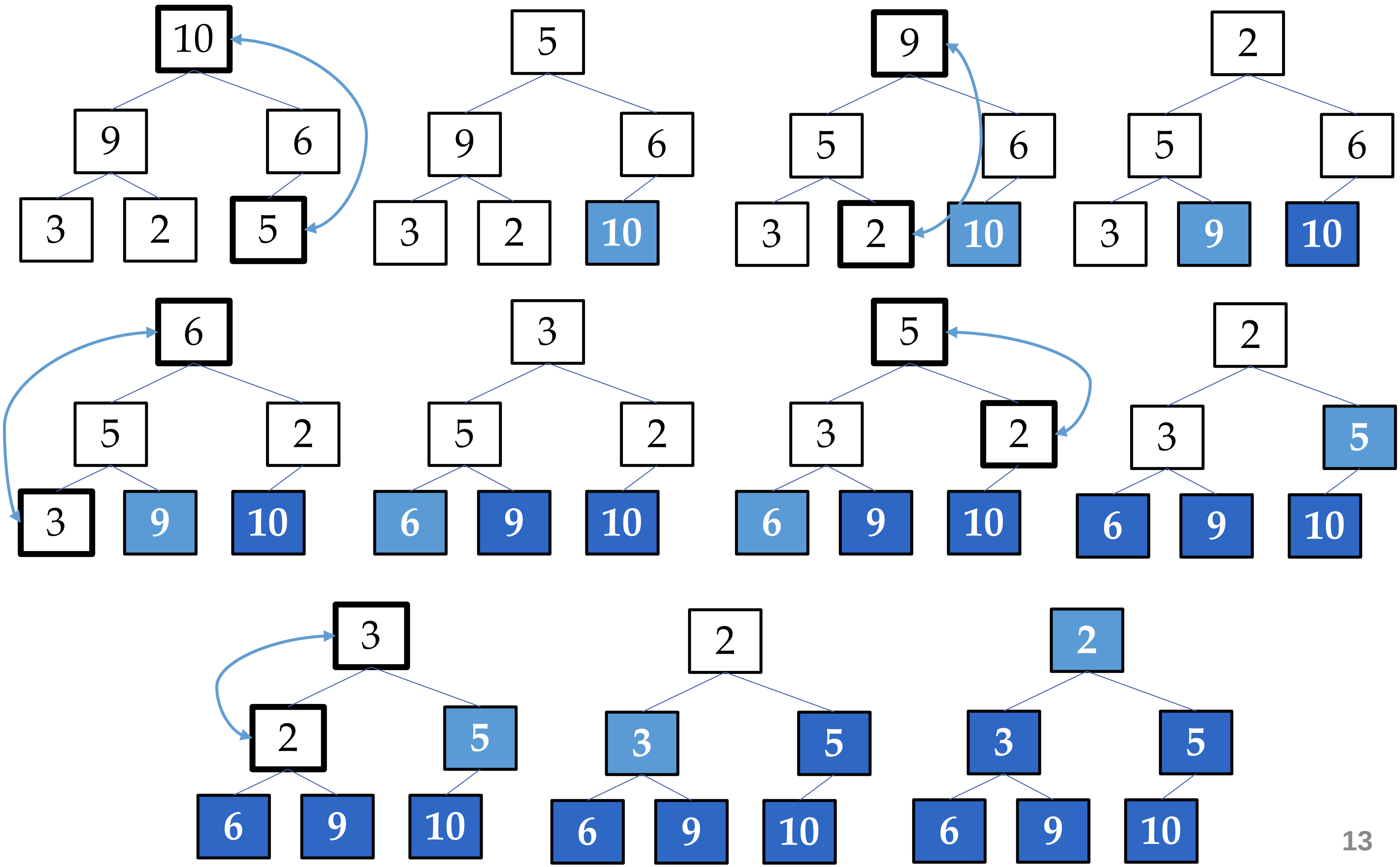
```
void heapSort(int arr[], int n){
    // Build initial heap
    for (int index = (n - 1) / 2; index >= 0; index--)
        heapRebuild(index, arr, n);
    swap(arr[0], arr[n - 1]); // swap the largest element to the end
    int heapSize = n - 1;     // Heap region size decreases by 1
    while (heapSize > 1) {
        heapRebuild(0, arr, heapSize);
        heapSize--;
        swap(arr[0], arr[heapSize]);
    }
}
```

Example: Heap sort on an array of integers

Sort the following array of integers, **{6, 3, 5, 9, 2, 10}**



Heap sort: An analysis



Heap sort: An analysis

- The heap sort algorithm includes two stages.
- **Heap construction** takes $O(n \log_2 n)$ time.
  - There are $\left\lfloor \frac{n}{2} \right\rfloor$ sifts, each of which runs in $O(\log_2 n)$ time.
- **Sorting** takes $O(n \log_2 n)$ time.
  - It executes $n - 1$ steps where each step runs in $O(\log_2 n)$ time.
- Thus, heap sort is $\mathbf{O}(\boldsymbol{n} \mathbf{log_2} \boldsymbol{n})$ in all cases.
- It is not recommended for small numbers of elements.

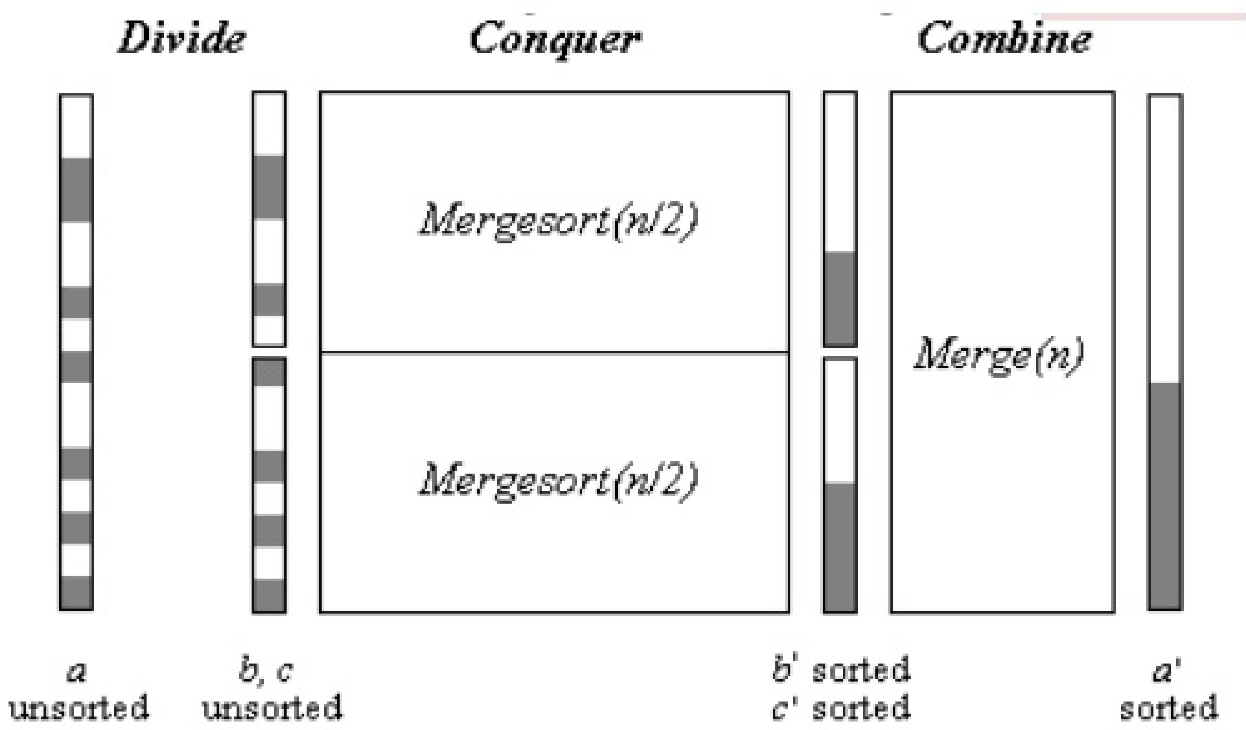
Checkpoint 05b: Heap sort on an array

Trace the **heap sort** as it sorts the following array into **ascending order**, **{20, 80, 40, 25, 60, 30}**.

Merge sort

Merge sort (John von Neumann, 1945)

- Recursively divide the array into halves, sort each half, and then merge the sorted halves into one sorted array.



Merge sort: Divide-and-conquer

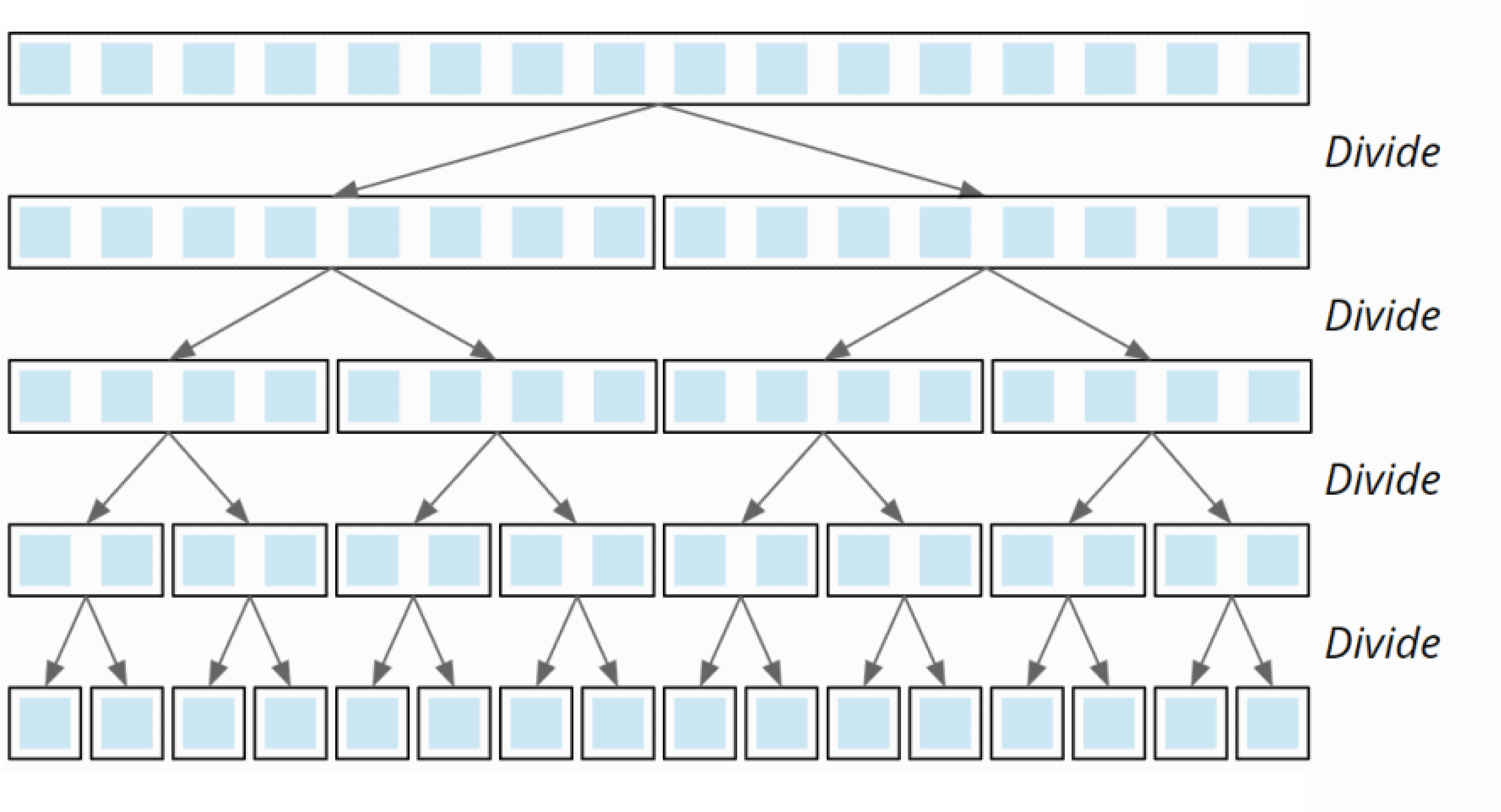


Image credit: happycore.eu

Merge sort: Divide-and-conquer

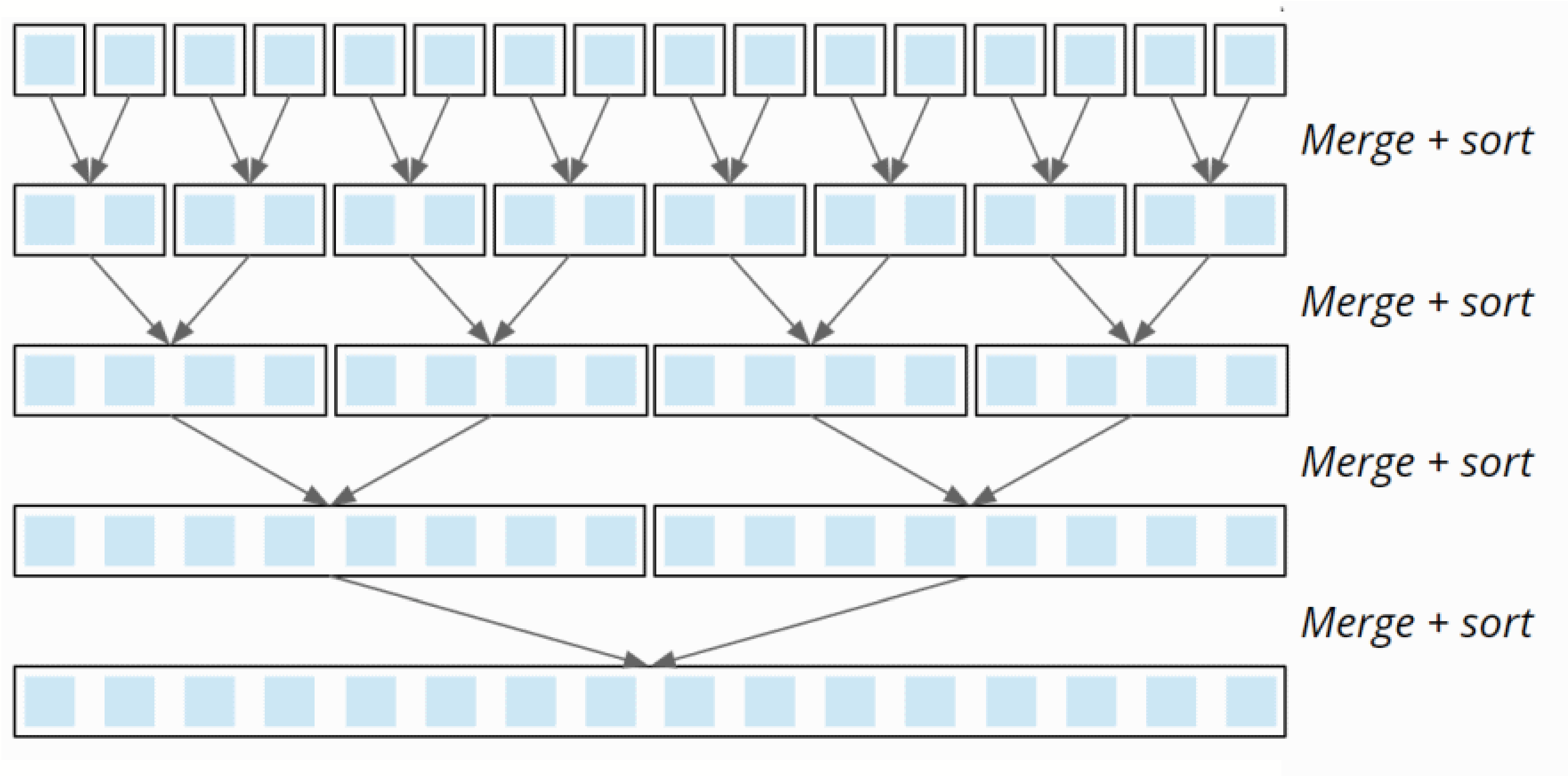
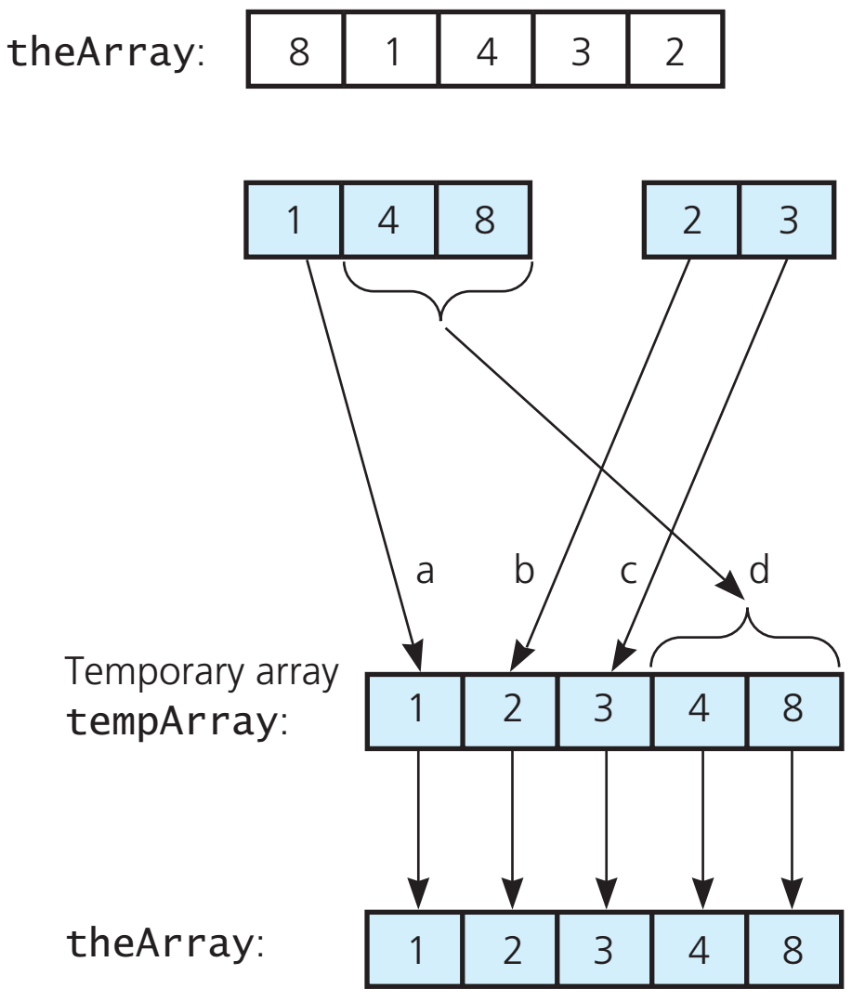


Image credit: happycore.eu

Merge sort: Algorithm



Divide the array in half

Sort the halves

Merge the halves:

a. 1 < 2, so move 1 from left half to `tempArray`
b. 4 > 2, so move 2 from right half to `tempArray`
c. 4 > 3, so move 3 from right half to `tempArray`
d. Right half is finished, so move rest of left half to `tempArray`

Copy temporary array back into original array

Merge sort: Implementation

- The recursive calls continue dividing the array into pieces until each piece contains only one element
- Obviously, an array of one element is sorted.

```
void mergeSort(int arr[], int first, int last){
   if (first < last) {
      int mid = (first + last) / 2;      // Index of midpoint
      mergeSort(arr, first, mid);        // Sort left half
      mergeSort(arr, mid + 1, last);     // Sort right half
      merge(arr, first, mid, last);   // Merge the two halves
   }
}
```

Merge sort: Implementation

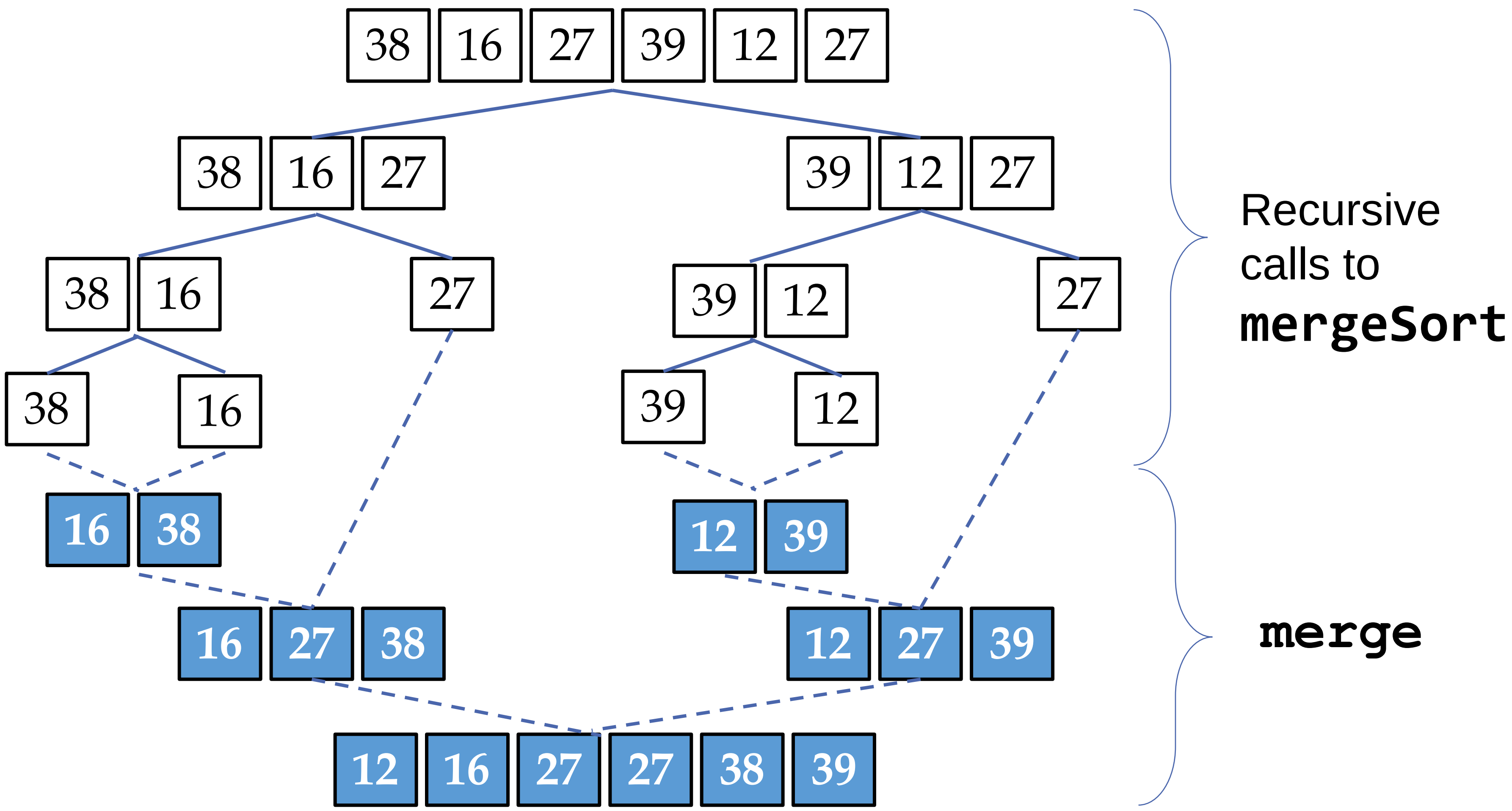
- The algorithm then merges these small pieces into larger sorted pieces until one sorted array results.

```
void merge(int arr[], int first, int mid, int last){
   // Initialize the local indices to indicate the subarrays
   int first1 = first, last1 = mid;           // The first subarray
   int first2 = mid + 1, last2 = last;       // The second subarray
   // Copy the smaller element into the temp array
   int tempArr[MAX_SIZE];                    // Temporary array
   int index = first1;              // Next available location in tempArr
   while ((first1 <= last1) && (first2 <= last2)) {
      // At this point, tempArr[first..index-1] is in order
      ... ... ...
```

Merge sort: Implementation

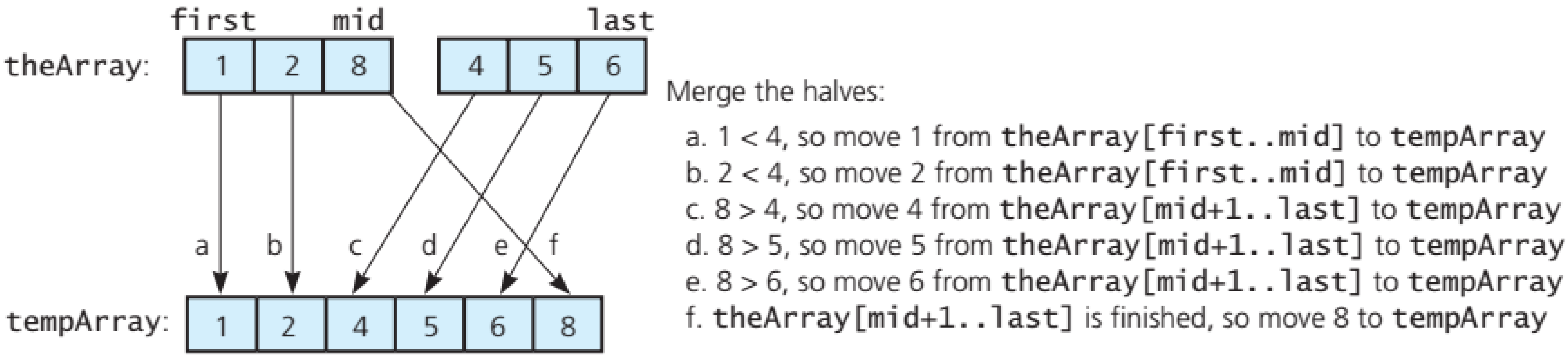
```
void merge(int arr[], int first, int mid, int last){
      ... ... ...
      if (arr[first1] <= arr[first2])
        tempArr[index++] = arr[first1++];
      else
        tempArr[index++] = arr[first2++];
   }
   while (first1 <= last1) // Finish the first subarray, if necessary
      tempArr[index++] = arr[first1++];
   while (first2 <= last2) // Finish the second subarray, if necessary
      tempArr[index++] = arr[first2++];
   // Copy the result back into the original array
   for (index = first; index <= last; ++index)
      arr[index] = tempArr[index];
}
```

Example: Merge sort on an array of integers



Merge sort: An analysis

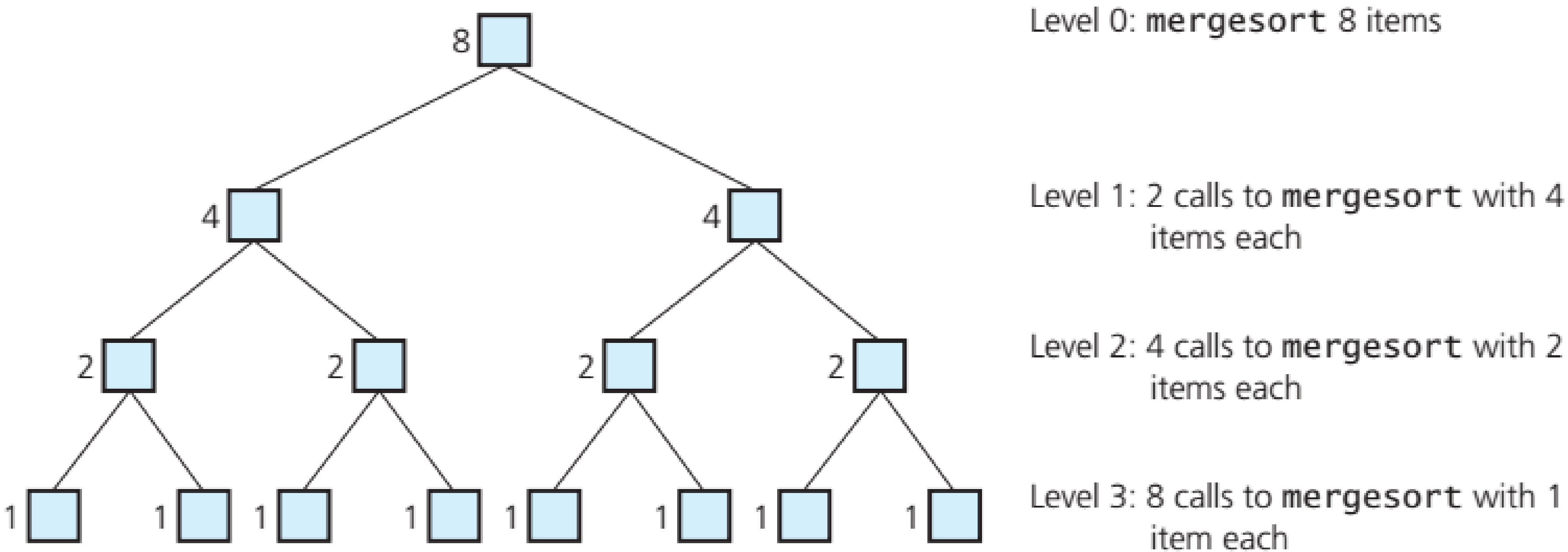
- At most $n - 1$ comparisons to merge the two segments whose total number of elements is n



- n moves from the original array to the temporary array, and
- n moves for copying the data back
- Thus, each merge requires $3n - 1$ major operations

Merge sort: An analysis

- Each call to **`mergeSort`** recursively calls itself twice.



- The levels of recursive calls: $k = \log_2 n$ (for $n = 2^k$) or $k = 1 + \lfloor \log_2 n \rfloor$ (for $n \neq 2^k$)

Merge sort: An analysis

- At level 0, the original call to **`mergeSort`** call **`merge`** once: $3n - 1$ operations
- At level 1, two calls to **`mergeSort`**, and hence to **`merge`**, occur: $2 \times \left(3\frac{n}{2} - 1\right) = 3n - 2$ operations
- …
- At level m, 2^m calls to **`merge`** occur: $2^m\left(3\frac{n}{2^m} - 1\right) = 3n - 2^m$
- Each level of the recursion requires $\mathrm{O}(n)$ operations, and there are either $\log_2 n$ or $1 + \lfloor \log_2 n \rfloor$ levels.
- Thus, merge sort is $\mathbf{O}(\boldsymbol{n} \log_2 \boldsymbol{n})$ in all cases.

Merge sort: An analysis

- Same performance regardless of the initial order of elements
- The `merge` step requires an auxiliary array, which requires a non-constant amount of memory.
- The extra storage and copying of entries are disadvantages.

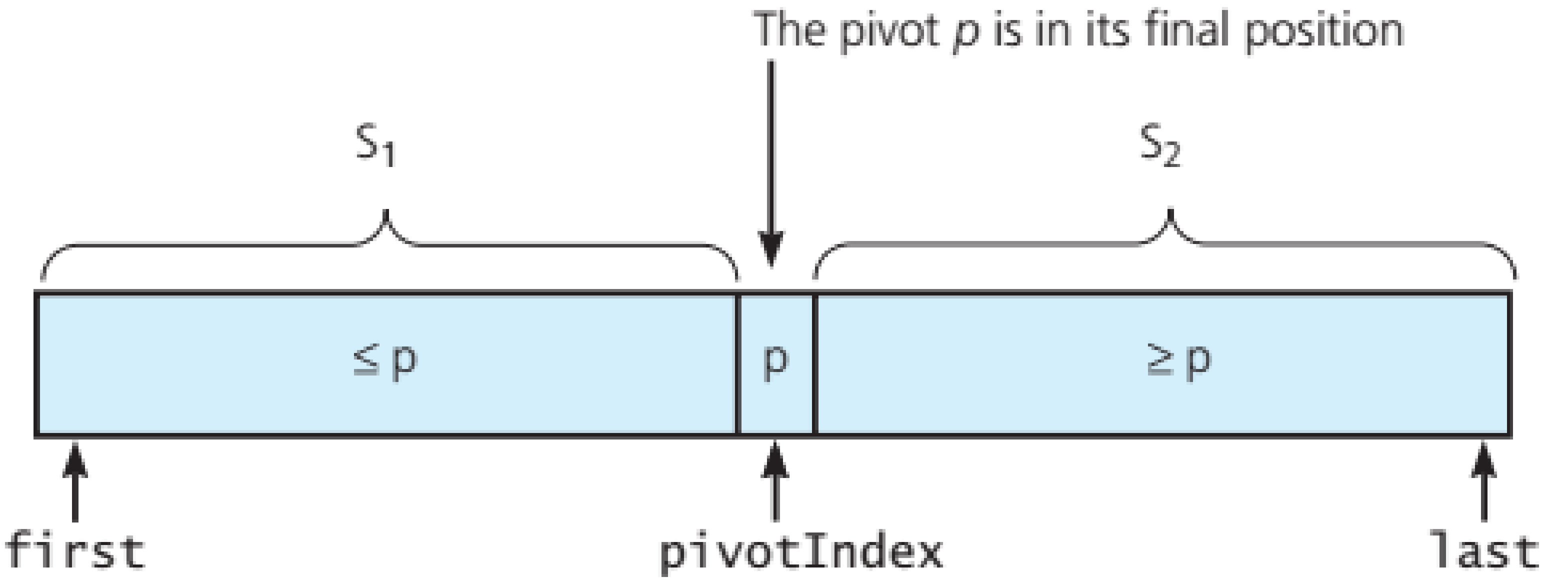
Checkpoint 06: Merge sort on an array

Trace the **merge sort** as it sorts the following array into **ascending order**, **{20, 80, 40, 25, 60, 30}**.

Quick sort

Quick sort (C. A. R. Hoare, 1962)

- Partition the initial array segment into two regions as follows



- Recursively partition on smaller segments, i.e. S_1 and S_2, until the array contains only one element

Quick sort (V1): Algorithm

- Consider an initial array, $a[first..last]$
- Step 1. Pick the pivot $p = a[k]$, where $k = \lfloor (first + last)/2 \rfloor$
- Step 2. Identity pairs of elements that are not in their correct positions and swap them
    - Set the increment variables, $i = first$ and $j = last$
    - While $a[i] < p$ do increase i by 1. While $a[j] > p$ do decrease j by 1.
    - If $i \leq j$ then swap $a[i]$ with $a[j]$, increase i by 1 and decrease j by 1.
    - Go to **Step 3**
- Step 3. Check whether the two smaller subarrays overlap
    - If $i < j$ then go to **Step 2**
    - Otherwise, recursively go to **Step 1** with $a[first..j]$ and $a[i..last]$

Quick sort (V1): Implementation

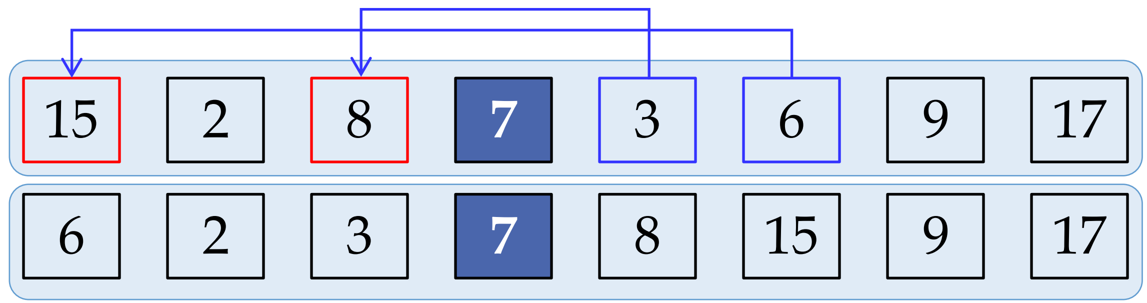
```
void quickSort(int arr[], int first, int last) {
   int pivot = arr[(first + last) / 2];
   int i = first, j = last;
   do {
       while (arr[i] < pivot) i++;
       while (arr[j] > pivot) j--;
       if (i <= j) {
          swap(arr[i], arr[j]);
          i++; j--;
       }
   } while (i <= j);
   if (first < j) quickSort(arr, first, j);
   if (i < last) quickSort(arr, i, last);
}
```

Example: Quick sort (V1) on an array of integers

- Partition the original array: $first = 0, last = 7, pivot = a[3]$

$i = 0, j = 5;\ i = 2, j = 4$

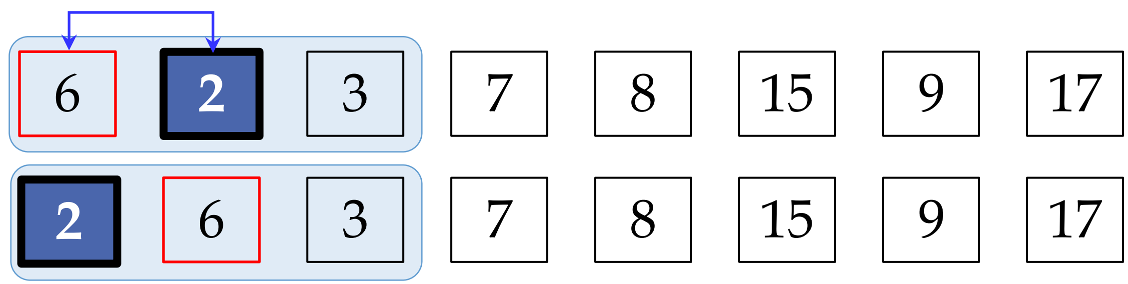
$i = 3, j = 3$



- Partition the subarray $a[0..2]$ with $pivot = a[1]$

$i = 0, j = 1$

$i = 1, j = 0$



Example: Quick sort (V1) on an array of integers

- Partition the subarray $a[1..2]$ with $pivot = a[2]$

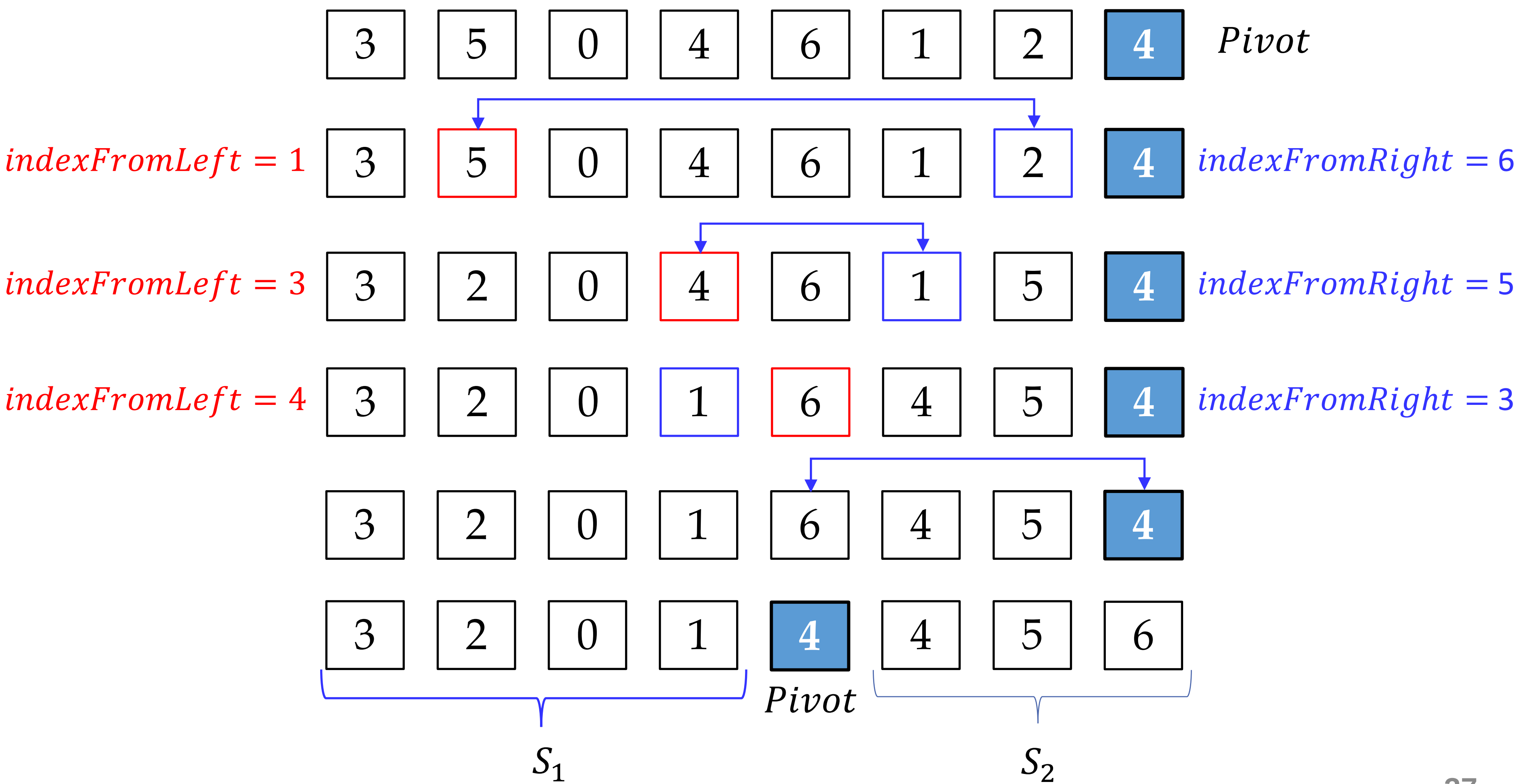
$i = 1, j = 2$	2	6	3	7	8	15	9	17
$i = 2, j = 1$	2	3	6	7	8	15	9	17

- Continue with the other subarrays

Quick sort (V2): Algorithm

- The chosen pivot is swapped with the last element $a[last]$ to get it out of the way during partition.
  - Various strategies exist for making the choice of pivot.
- The algorithm still maintains two searches
  - A forward search starting at the first entry looks for the first entry that is greater than or equal to the pivot, and
  - A backward search starting at the next-to-last entry looks for the first entry that is less than or equal to the pivot.
- Place the pivot between the two subarrays, S_1 and S_2, by swapping $a[indexFromLeft]$ and $a[last]$

Example: A partitioning using Quick sort (V2)



Entries equal to the pivot

- Both of S_1 and S_2 can contain entries equal to the pivot.
  - Both forward and backward searches stop when they encounter an entry that equals the pivot, and a swap occurs.
  - Such an entry has a chance of landing in each of the subarrays.
- *Why not always place any entries that equal the pivot into the same subarray?*
  - Such a strategy would tend to make one subarray larger than the other, and thus diminish the performance of quick sort.

Median-of-three pivot: Idea

- The ideal pivot should be the median value in the array.
- Take as pivot the median of three entries: the first entry, the middle entry and the last entry

5	8	6	4	**9**	3	7	1	**2**

- Sort only those entries and use the middle value as the pivot

2	8	6	4	**5**	3	7	1	**9**
				$Pivot$				

- Position the pivot before partitioning

							$Pivot$	
2	8	6	4	1	3	7	**5**	9
	$\uparrow$					$\uparrow$		
	$indexFromLeft$					$indexFromRight$		

Median-of-three pivot: Implementation

```
int sortFirstMiddleLast(int arr[], int first, int last){
   int mid = first + (last - first) / 2;
   if (arr[first] > arr[mid])
      swap(arr[first], arr[mid]);
   if (arr[mid] > arr[last])
      swap(arr[mid], arr[last]);
   if (arr[first] > arr[mid])
      swap(arr[first], arr[mid]);
   return mid;
}
```

Quick sort (V2): Implementation

- Use insertion sort instead on arrays of fewer than ten entries

```
void quickSort(int arr[], int first, int last){
   if (last - first + 1 < MIN_SIZE)
      insertionSort(arr + first, last – first + 1);
   else {
      // Create the partition: S1 | Pivot | S2
      int pivotIndex = partition(arr, first, last);
      // Sort subarrays S1 and S2
      quickSort(arr, first, pivotIndex - 1);
      quickSort(arr, pivotIndex + 1, last);
   }
}
```

Quick sort (V2): Implementation

```
int partition(int arr[], int first, int last){
    // Choose pivot using median-of-three selection
    int pivotIndex = sortFirstMiddleLast(arr, first, last);
    // Reposition pivot so it is last in the array
    swap(arr[pivotIndex], arr[last-1]);
    pivotIndex = last-1;
    int pivot = arr[pivotIndex];
    // Determine the regions S1 and S2
    ......
    // Place pivot in proper position between S1 and S2
    swap(arr[pivotIndex], arr[indexFromLeft]);
    pivotIndex = indexFromLeft;  // and mark its new location
    return pivotIndex;
}
```

Quick sort (V2): Implementation

```
int partition(int arr[], int first, int last){
    ......
    int indexFromLeft = first+1, indexFromRight = last - 2;
    bool done = false;
    while (!done) {
        // Locate first entry on left that is >= pivot
        while (arr[indexFromLeft] < pivot) indexFromLeft++;
        // Locate first entry on right that is <= pivot
        while (arr[indexFromRight] > pivot) indexFromRight--;
        // Swap the two found entries
        if (indexFromLeft < indexFromRight){
            swap(arr[indexFromLeft], arr[indexFromRight]);
            indexFromLeft++; indexFromRight--;
        }
        else done = true;
    }
    ......
}
```

Quick sort: An analysis

Best case	Worst case	Average case
$\mathbf{O}(\boldsymbol{n}\log_2 \boldsymbol{n})$	$\boldsymbol{O}(\boldsymbol{n}^2)$	$\mathbf{O}(\boldsymbol{n}\log_2 \boldsymbol{n})$

- Quick sort is at least as well as any known comparison algorithm on data of random order.
- Quick sort vs. Merge sort
  - Quick sort can be faster in practice and does not require the additional memory that merge sort needs for merging
  - The efficiency of a merge sort is somewhere between the possibilities for a quick sort.

Checkpoint 07: Quick sort on an array

Trace the **quick sort**’s partitioning algorithm as it partitions the following array,

{24, 97, 40, 67, 88, 85, 15, 66, 53, 44, 26, 48, 16, 52, 45, 23, 90, 18, 49, 80}

Acknowledgements

This part of the lecture is adapted from the following materials.

[1] Pr. Nguyen Thanh Phuong (2020) "*Lecture notes of CS163 – Data structures*" University of Science - Vietnam National University HCMC.

[2] Pr. Van Chi Nam (2019) "*Lecture notes of CSC14004 – Data structures and algorithms*" University of Science - Vietnam National University HCMC.

[3] Frank M. Carrano, Robert Veroff, Paul Helman (2014) "*Data Abstraction and Problem Solving with C++: Walls and Mirrors*" Sixth Edition, Addion-Wesley. **Chapter 10.**

[4] Anany Levitin (2012) "*Introduction to the Design and Analysis of Algorithms*" Third Edition, Pearson.

Exercises

01. Sorting algorithms on an array

- Consider the following array of integers, {26, 48, 12, 92, 28, 6, 33}.
- Apply each of the following sorting algorithms to arrange the elements in the given array in ascending order.
  - Heap sort
  - Quick sort (with median-of-three pivot)
  - Merge sort

02. Which algorithm is best?

- For each of the following situations, name the best sorting algorithm from among those we studied. There may be more than one answer.
  a) You need a very fast sort on average, and you can only use a constant amount of extra space.
  b) The array is in perfect sorted order.
  c) You have a large data set, but you know all the values are between 0 and 999.
  d) Copying your data is very fast, but comparisons are relatively slow

03. Parsimony in sorting algorithms

- A sorting algorithm is parsimonious if it never compares the same pair of input value twice (Assuming that all input values are distinct).
- Which of the following sorting algorithms is parsimonious?
  - Heap sort
  - Quick sort
  - Merge sort
- Give an example or counter-example for each of the above algorithms.