University of Applied Sciences and Arts
of Southern Switzerland

SUPSI

Algorithms and Data Structures
Computer Engineering
Sorting

Fabio Landoni
fabio.landoni@supsi.ch

Outline

- Sorting problem
- Insertion-Sort
- Selection-Sort
- Bubble-Sort
- Counting Sort
- Binary Heap
- Heap-Sort
- Divide-and-Conquer
- Merge-Sort
- Quick-Sort
- Radix-Sort

Sorting problem

Book ref.: CLRS 1.1, 2 - Introduction

Sorting problem

- INPUT: a sequence of n numbers $(a_1, a_2, \cdots, a_n)$, called *input sequence*.
- OUTPUT: a permutation (reordering) $(i_1, i_2, \cdots, i_n)$ of $(1, 2, \cdots, n)$, called *solution*, so that $a_{i_1} \leq a_{i_2} \leq \cdots \leq a_{i_n}$. In that case $(a_{i_1}, a_{i_2}, \cdots, a_{i_n})$ is called *output sequence*.

Note: for a given input sequence, the sorting problem can have more than one solution. However, all the solutions are mapped into the same unique output sequence.

Sorting problem: example

- INPUT: input sequence $(50, 0, -4, -11, 0, 59, 3)$, where:

$$a_1 = 50, a_2 = 0, a_3 = -4, a_4 = -11,$$
$$a_5 = 0, a_6 = 59, a_7 = 3$$

- OUTPUT:
  - Solutions: permutations

$$\pi_a = (4, 3, 2, 5, 7, 1, 6)$$
$$\pi_b = (4, 3, 5, 2, 7, 1, 6)$$

  - Output sequence:

$$(-11, -4, 0, 0, 3, 50, 59)$$

A solution to the sorting problem: ordering in place

- Represent the input sequence $(a_1, a_2, \cdots, a_n)$ with an array $A[1..n]$ of length n so that $A[i] = a_i, \forall i \in [1, n]$.

input sequence: $(50, 0, -4, -11, 0, 59, 3)$ with $n = 7$

	1	2	3	4	5	6	7
A	50	0	-4	-11	0	59	3

- Without using an auxiliary array, reorder the elements of array A by exchanging them so that in the end $A[1] \leq A[2] \leq \cdots \leq A[n]$.

output sequence: $(-11, -4, 0, 0, 3, 50, 59)$

	1	2	3	4	5	6	7
A	-11	-4	0	0	3	50	59

Vocabulary

An algorithm is said to be:

- **Stable:** if items with the same value appear in the output array in the same order as they do in the input array.
- **In place:** if it rearranges the values within the array A, with at most a constant number of them stored outside the array at any time.

Insertion sort

Book ref.: CLRS 2.1

Insertion-Sort

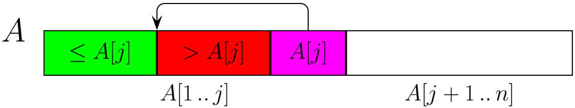
Algorithm 1 Insertion-Sort

Input A: input array to sort, n: length of A

```
procedure INSERTION-SORT(A, n)
    for j = 2 to n do
        k = A[j]
        i = j − 1
        while  i ≥ 1 AND A[i] > k do
            A[i + 1] = A[i]
            i = i − 1
        end while
        A[i + 1] = k
    end for
end procedure
```

Insertion-Sort

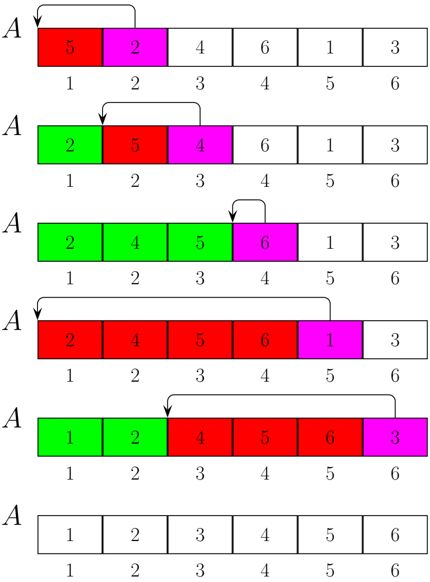
Insertion-Sort algorithm scans array A from left to right by looping over index j from 2 to n. At each iteration element $A[j]$ is placed in the "right place" in subarray $A[1 \cdots j]$ by moving all elements larger than $A[j]$ one position to the right and leaving unchanged all other elements.



Note:

- At the end of each `for` iteration the subarray $A[1 \cdots j]$ contains elements in sorted order.
- Insertion sort runs in $\Theta(n^2)$ worst-case time.

Insertion-Sort: example



Selection-sort

Selection-Sort

Algorithm 2 Selection-Sort

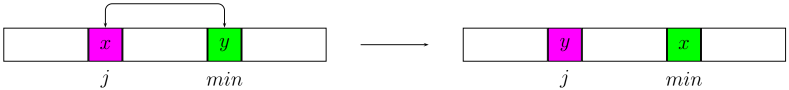
Input A: input array to sort, n: length of A

```
procedure SELECTION-SORT(A, n)
    for j = 1 to n − 1 do
        min = j
        for i = j + 1 to n do
            if A[i] < A[min] then
                min = i
            end if
        end for
        temp = A[j]
        A[j] = A[min]
        A[min] = temp
    end for
end procedure
```

Selection-Sort

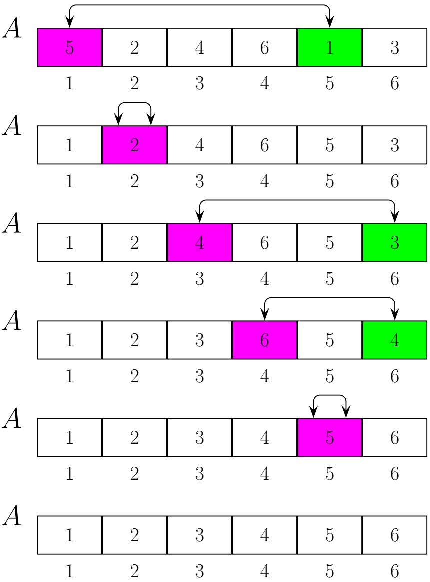
Selection-Sort algorithm scans array A from left to right by looping over index j from 1 to $n-1$. For each index j following operations are performed:

1. find index min in subarray $A[j \cdots n]$ so that $A[min] \leq A[j], A[j+1], \cdots, A[n]$
2. swap $A[j]$ with $A[min]$



Note: the running time of the algorithm is $\Theta(n^2)$ for all cases.

Selection-Sort: example



Bubble-Sort

Book ref.: CLRS 2.3 - Problems

Bubble-Sort

Algorithm 3 Bubble-Sort

Input A: input array to sort, n: length of A

```
procedure BUBBLE-SORT(A, n)
    for i = 1 to n − 1 do
        for j = n downto i + 1 do
            if A[j] < A[j − 1] then
                temp = A[j]
                A[j] = A[j − 1]
                A[j − 1] = temp
            end if
        end for
    end for
end procedure
```

Bubble-Sort

Bubble-Sort algorithm scans array A from left to right by looping over index i from 1 to $n-1$. For each index i it goes through the elements from last to $i+1$-th and swaps $A[j]$ with $A[j-1]$ if they are out of order.

Note:

- At the end of each `for` iteration on index i the first i-th smallest elements are in the correct place.
- Note: the running time of the algorithm is $\Theta(n^2)$ for all cases.

Bubble-Sort: example

3	1	5	2	4

First round

3	1	5	2	4

3	1	2	5	4

1	3	2	5	4

Second round

1	3	2	4	5

1	2	3	4	5

Counting sort

Book ref.: CLRS 8.2

Counting sort

Assumption

Each of the n input elements:

- it is an integer
- it is in the range 0 to k.

Note: when $k = O(n)$ (i.e., the maximum value of the data to be processed is of the same order of magnitude as the number of data), counting sort runs in $\Theta(n)$ time.

Counting Sort

Algorithm 4 Counting Sort

Input A: input array to sort, n: length of A, B: array of length n that holds the sorted output, k: maximum value contained in A

```
procedure COUNTING-SORT(A, n, B, k)
    C[0..k] = array of length k + 1 so that C[i] = 0, ∀i ∈ [0..k]
    for j = 1 to n do
        C[A[j]] = C[A[j]] + 1
    end for                        ▷ C[i] = number of elements equal to i
    for i = 1 to k do
        C[i] = C[i] + C[i − 1]
    end for                        ▷ C[i] = number of elements ≤ i
    for j = n downto 1 do
        B[C[A[j]]] = A[j]
        C[A[j]] = C[A[j]] − 1
    end for
end procedure
```

Counting sort

For each element x of the set to be ordered, it determines how many elements are less than x. It uses this information to assign to x its final position in the ordered array. Scheme must be slightly modified to handle the situation in which several elements have the same value: we do not want to put them all in the same position (see line 11 of the algorithm).

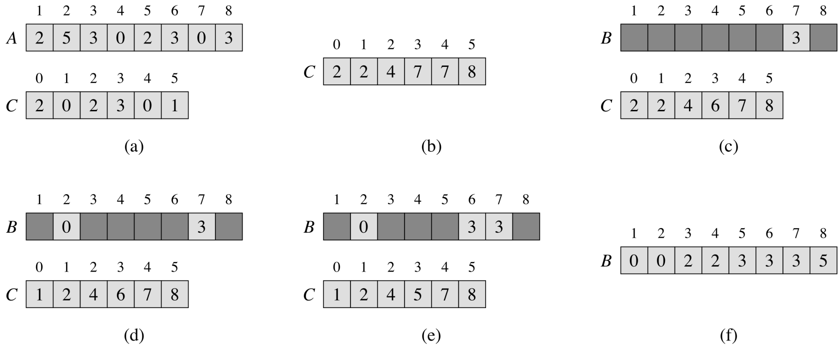
Features:

- stable
- not in-place.

Note:

- No comparison between input elements occur anywhere in the code.
- Worst-case running time: $\Theta(k + n)$

Counting Sort: example

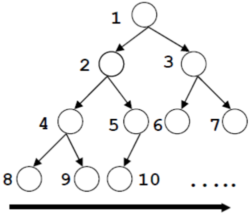


Heap and Heap sort

Book ref.: CLRS 6

Binary Heap

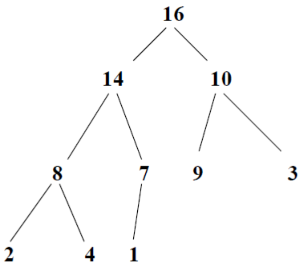
- **Nearly complete binary tree:** binary tree completely filled on all levels except possibly the lowest, which is filled from the left up to a point.



- **Binary heap:** data structure that can be viewed as a nearly complete binary tree, where the values in the nodes satisfy a heap property.
  Applications:
  - sorting (heap sort)
  - priority queue

Heap property

- **Max-heap** property: the value of a node is at most the value of its parent (i.e., the value of a child node is less than or equal to that of the parent node).



- **Min-heap** property: the value of a node is greater than or equal to that of the parent node.

Heap sort algorithm is based on max-heap property.

From now on when we talk about heap property we mean max-heap property (unless otherwise specified).

Heap implementation with an array

Attributes of an array A that represents a heap:

- `A.length`: number of elements in the array
- `A.heap-size`: how many elements in the heap are stored within A.

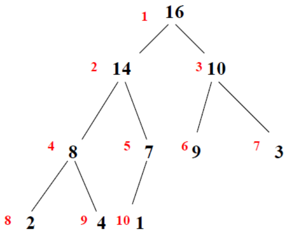
Only the elements in $A[1..A.\text{heap-size}]$, where $0 \leq A.\text{heap-size} \leq A.\mathit{length}$, are valid elements of the heap.

Heap implementation with an array

Coding of a nearly complete binary tree with an array A:

- The root of the tree is $A[1]$.
- Given the index i of a node:
  - the indices of its parent is: $\left\lfloor \frac{i}{2} \right\rfloor$
  - the indices of its left child is: $2i$
  - the indices of its right is: $2i + 1$

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Maintaining the heap property

- Following various operations on a heap it can happen that a node violates the heap property.
- `HEAPIFY` procedure takes an heap A and the index i of a node that potentially violates the property and restores the partial sorting property on the entire heap.
- Assumption for `HEAPIFY` procedure: child sub-trees of node i are heap roots that respect the heap property.

Maintaining the heap property

Algorithm 5 Heapify

Input A: input array, i: index of a node that potentially violates the heap-property

Assumption: child sub-trees of node i are heaps

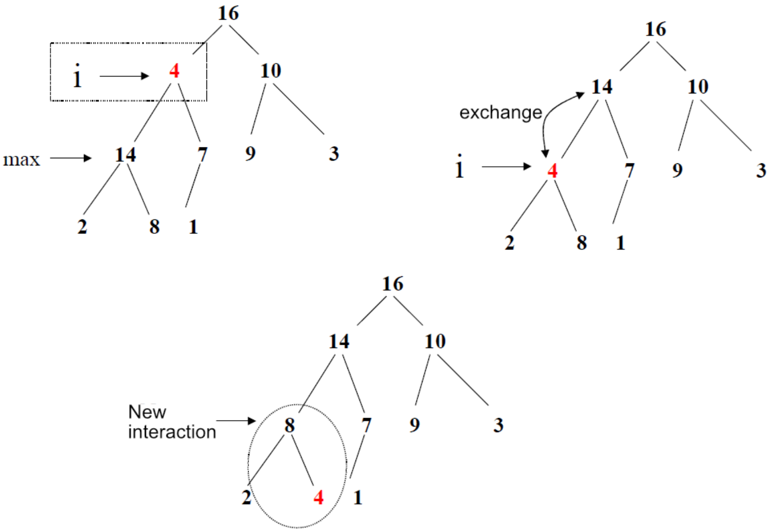
```
procedure HEAPIFY(A, i)
    l = LEFT(i), r = RIGHT(i)
    if l ≤ A.heap-size and A[l] > A[i] then
        largest = l
    else
        largest = i
    end if
    if r ≤ A.heap-size and A[r] > A[largest] then largest = r end if
    if largest ≠ i then
        exchange A[i] with A[largest]
        HEAPIFY(A, largest)
    end if
end procedure
```

Maintaining the heap property

The idea is to "float down" the value at $A[i]$ (i.e., the node that violates the property of partial ordering) until the heap property is restored.

At each step, the largest of the elements $A[i], A[LEFT(i)]$ and $A[RIGHT(i)]$ is determined and its index is stored in *largest*. If $A[i]$ is largest, then the subtree rooted at i is already an heap. Otherwise, one of the two children has the largest element, and $A[i]$ is swapped with $A[largest]$, which causes node i and its children to satisfy the heap property. The node indexed by *largest*, however, now has the original value $A[i]$, and thus the subtree rooted at *largest* might violate the heap property. Consequently we call the procedure recursively on that subtree.

Maintaining the heap property: example



Building a heap

- To convert an array $A[1...n]$ into an heap we can use the HEAPIFY procedure in a bottom-up manner (that is starting from the lowest levels of the tree).
- The elements in subarray $A[(\lfloor n/2 \rfloor + 1)..n]$ are all leaves of the tree. Therefore, each of them are already a heap of one element.
- We apply the HEAPIFY procedure starting from the parent elements to the nodes $A[(\lfloor n/2 \rfloor + 1)..n]$ and we go through the remaining nodes.
- By proceeding in a bottom-up way, we ensure that the subtrees of a note subject to the HEAPIFY procedure are heaps.

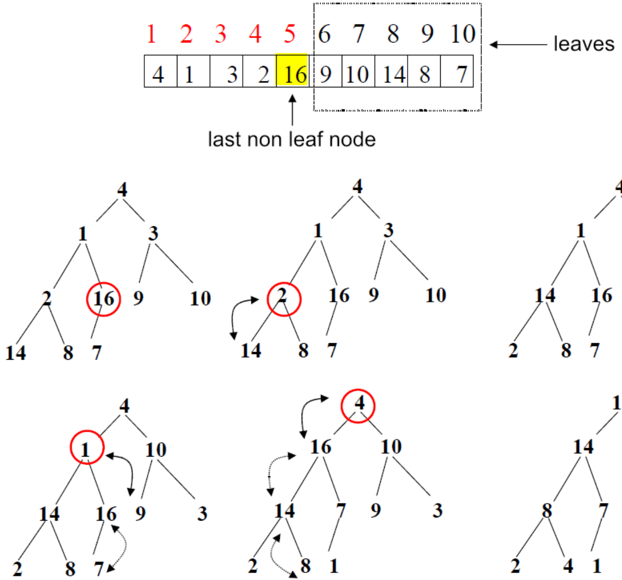
Building a heap

Algorithm 6 Build-Heap

Input A: input array, n: length of A

```
procedure BUILD-HEAP(A, n)
    A.heap-size = n
    for i = ⌊n/2⌋ downto 1 do
        HEAPIFY(A, i)
    end for
end procedure
```

Building a heap: example



Heap-Sort

Algorithm 7 Heap-Sort

Input A: input array, n: length of A

```
procedure HEAP-SORT(A, n)
    BUILD-HEAP(A, n)
    for i = n downto 2 do
        exchange A[1] with A[i]
        A.heap-size = A.heap-size − 1
        HEAPIFY(A, 1)
    end for
end procedure
```

The algorithm assumes that all heaps are max-heaps, resulting in A sorted in ascending order. (Using min-heaps you would get A sorted in descending order.)

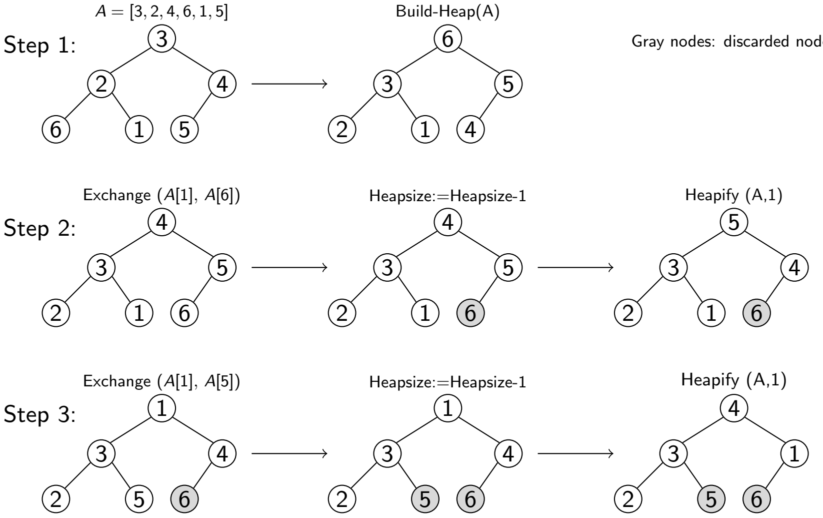
Heap-Sort

Heap sort starts by building an heap on the array $A[1..n]$. Since the maximum element is stored at the root $A[1]$, we can put it into its correct final position by exchanging it with $A[n]$. Discarding node n (by decrementing A.heap-size) we have that the subarray $A[1..n-1]$ can become an heap with $n-1$ nodes by simply applying the `HEAPIFY(A,1)`. Repeat this process until heap size is 2.

Heap-Sort takes time $O(n \cdot \log_2 n)$ since:

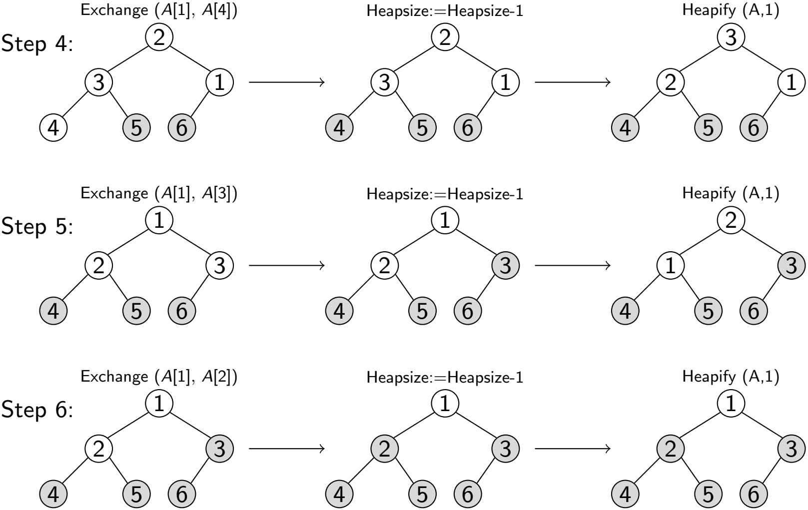
- `BUILD-HEAP` takes $O(n \cdot \log_2 n)$
- `HEAPIFY` takes $O(\log_2 n)$
- `HEAPIFY` is called $n-1$ times

Heap-Sort: example 1/2



Gray nodes: discarded nodes

Heap-Sort: example 2/2



Divide-and-Conquer

Book ref.: CLRS 2.3.1

Divide-and-Conquer

Divide-and-conquer is a paradigm used to solve a problem recursively, applying the following three steps at each level of recursion:

1. **Divide:** divide the problem into a number of subproblem that are smaller instances of the same problem.
2. **Conquer:** conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
3. **Combine:** combine the solutions to the subproblems into the solution for the original problem.

Merge sort and quicksort are examples of algorithms based on this technique.

Merge sort

Book ref.: CLRS 2.3

Divide-and-Conquer in merge sort

1. **Divide:** divide the n-element sequence to be sorted into two subsequences of $n = 2$ elements each.
2. **Conquer:** sort the two subsequences recursively using merge sort.
3. **Combine:** merge the two sorted subsequences to produce the sorted answer.

Merge sort - merge procedure

Key operation of merge sort algorithm is the merging of two sorted sequences in the *combine* step.

Algorithm 8 Merge

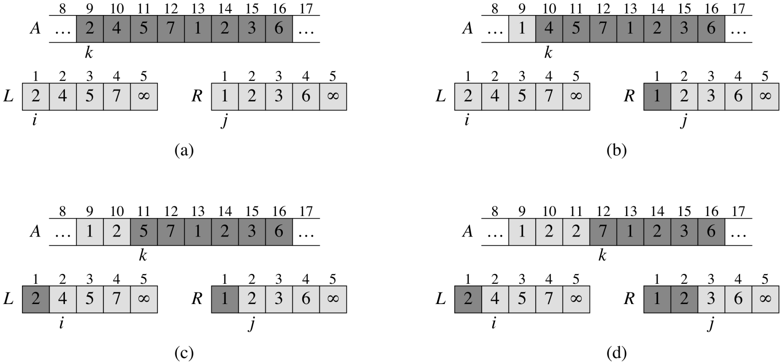
Input A: input array, p, q, r: indices of the array so that $p \leq q < r$
Assumption: Subarrays $A[p..q]$ and $A[q+1..r]$ are in sorted order.

procedure $\textsc{Merge}(A, p, q, r)$
  $n_1 = q - p + 1, n_2 = r - q$, $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ new arrays
  for $i = 1$ **to** n_1 **do** $L[i] = A[p + i - 1]$ **end for**
  for $j = 1$ **to** n_2 **do** $R[j] = A[q + j]$ **end for**
  $L[n_1 + 1] = \infty$, $R[n_2 + 1] = \infty$, $i = 1, j = 1$
  for $k = p$ **to** r **do**
    if $L[i] \leq R[j]$ **then**
      $A[k] = L[i]$, $i = i + 1$
    else
      $A[k] = R[j]$, $j = j + 1$
    end if
  end for
end procedure

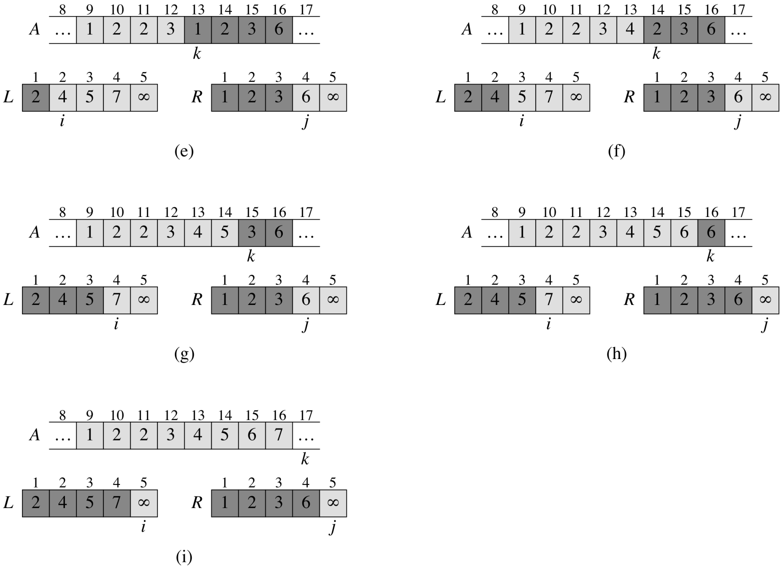
Merge sort - merge procedure

- `MERGE` procedure assumes that the two arrays to merge are already sorted.
- At each iteration:
  - choose the smaller item between the two first elements of the arrays that have not been copied
  - copy this element to the array that will contain the result.
- At the start of each iteration of the last for loop, the subarray $A[p..k-1]$ contains $k-p$ smallest elements of $L[1..n_1+1]$ and $R[1..n_2+1]$, in sorted order. Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into A.
- `MERGE` procedure takes time $\Theta(n)$ where $n = r - p + 1$ (total number of elements being merged)

Merge procedure: example 1/2



Merge procedure: example 2/2



Merge sort

Algorithm 9 Merge sort

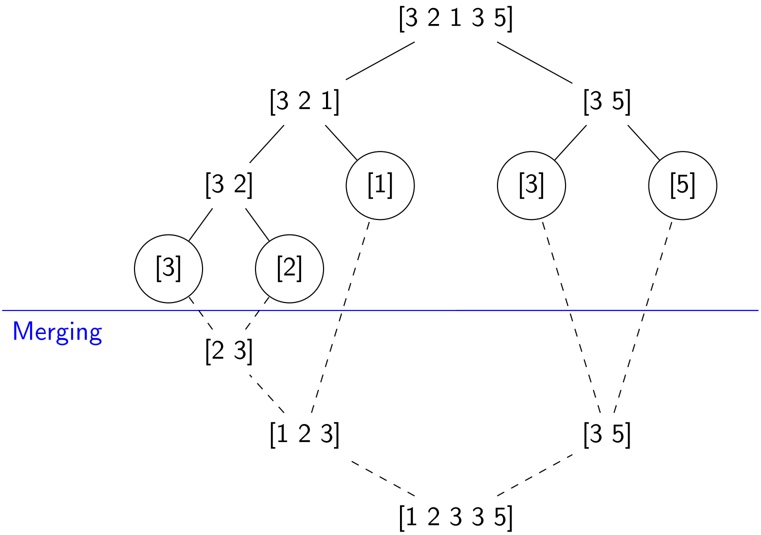
Input A: input array, p, r: indices of the array (the algorithm sorts elements in subarray $A[p..r]$)

```
procedure MERGE SORT(A, p, r)
    if p < r then
        q = ⌊(p + r)/2⌋
        MERGE SORT(A, p, q)
        MERGE SORT(A, q + 1, r)
        MERGE(A, p, q, r)
    end if
end procedure
```

Merge sort

- `MERGE SORT` computes an index q that partitions $A[p..r]$ into the subarrays $A[p..q]$ ($\lceil n/2 \rceil$ elements) and $A[q+1..r]$ ($\lfloor n/2 \rfloor$ elements).
- By calling $\texttt{MERGE}(A, p, q, r)$ the procedure performs an ordered merge of the ordered subarrays $A[p..q]$ and $A[q+1..r]$.
- By calling $\texttt{MERGE SORT}(A, p, r)$ the procedure sorts elements in the subarray $A[p..r]$. To sort an entire array A, the initial call is $\texttt{MERGE SORT}(A, 1, n)$ where $n = A.length$.
- Note that: if $p \geq r$ it follows that $A[p..r]$ has at most one element and is therefore already sorted.
- `MERGE SORT` procedure takes time $\Theta(n \cdot \log_2 n)$ where $n =$ number of elements to be sorted.

Merge sort: example



Quicksort

Book ref.: CLRS 7

Divide-and-Conquer in quicksort

1. **Divide:** Partition (rearrange) array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$ by computing q such that $A[i] \leq A[q] \leq A[j]$ $\forall i \in [p, q-1]$ and $\forall j \in [q+1..r]$
2. **Conquer:** sort subarrays $A[p..q-1]$ and $A[q+1..r]$ by recursive calls to quick sort.
3. **Combine:** No work is needed. $A[p..r]$ is sorted since the two subarrays are already sorted.

Quicksort

Algorithm 10 Quicksort

Input A: input array, p, r: indices of the array (the algorithm sorts elements in subarray $A[p..r]$)

```
procedure QUICKSORT(A, p, r)
    if p < r then
        q = PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
    end if
end procedure
```

Quicksort

- By calling $\texttt{QUICKSORT}(A, p, r)$ the procedure sorts elements in the subarray $A[p..r]$. To sort an entire array A, the initial call is $\texttt{QUICKSORT}(A, 1, n)$ where $n = A.length$.
- After line 3 it holds that: $A[q] < A[i], \forall i \in [q + 1, r]$
- On average QUICKSORT expected running time is $\Theta(n \cdot \log_2 n)$
- Worst-case running time of QUICKSORT is $\Theta(n^2)$, when the input array is already sorted.
- QUICKSORT perform in place sort.

Quicksort - Partitioning

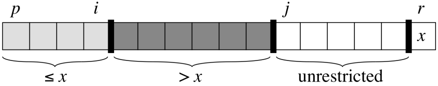
Algorithm 11 Partition

Input A: input array, p, r: indices of the array (the algorithm rearrange the subarray $A[p..r]$)

```
procedure Partition(A, p, r)
    x = A[r]
    i = p − 1
    for j = p to r − 1 do
        if A[j] ≤ x then
            i = i + 1
            exchange A[i] with A[j]
        end if
    end for
    exchange A[i + 1] with A[r]
    return i + 1
end procedure
```

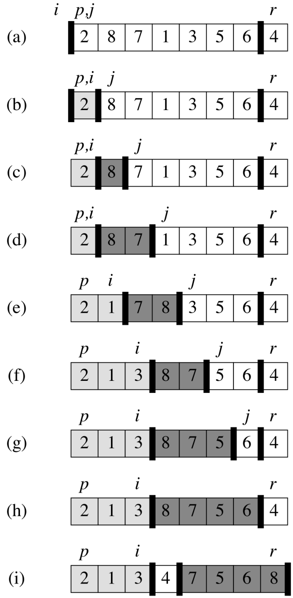
Quicksort - Partitioning

- PARTITION rearranges the subarray $A[p..r]$ in place.
- PARTITION selects an element ($x = A[r]$) as a pivot around which to partition the subarray $A[p..r]$
- at the beginning of each loop iteration, $\forall k \in [p, r]$:
  1. if $p \leq k \leq i$, then $A[k] \leq x$
  2. if $i + 1 \leq k \leq j - 1$, then $A[k] > x$

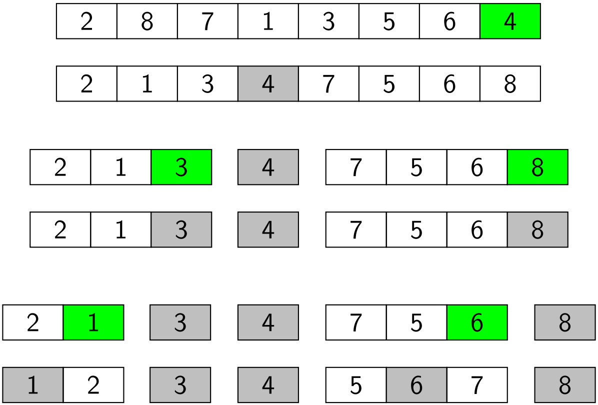


- PARTITION ends by swapping the pivot element (x) with the leftmost element grater than x (i.e., moving the pivot into its correct place) and return the pivot's new index.
- Running time of PARTITION is $\Theta(n)$ where n number of elements to be sorted.

Partition: example



Quicksort: example



Radix sort

Book ref.: CLRS 8.3

Radix sort

Algorithm 12 Radix-sort

Input A: array to sort, d number of digits of each elements (digit 1 is the lowest-order digit and digit d is the highest-order digit)

```
procedure RADIX-SORT(A, d)
    for i = 1 to d do
        use a stable sort to sort array A on digit i
    end for
end procedure
```

Radix sort

- Radix sort sorts on the least significant digit first. Then it sorts on the second-least significant digit. The process continues until the numbers have been sorted on all d digits.
- In order for radix sort to work correctly the algorithm used to sort the digits must be stable.
- Given n $d-$digit numbers so that each digit is in the range $[0, k-1]$. Radix-sorts running time is $\Theta(d(n+k))$ if the stable sort it uses takes $\Theta(n+k)$ time.

Radix sort: example

329	720	720	329
457	355	329	355
657	436	436	436
839	457	839	457
436	657	355	657
720	329	457	720
355	839	657	839

Sources

Sources

- Book “Introduction to Algorithms” (Third Edition), by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein, published by MIT Press and McGraw-Hill.
- Course “Algorithms and data structures”, Prof. Roberto Montemanni, 2018 - 2019
- Course “Algorithms and data structures”, Prof. Carlo Spinedi, 2011