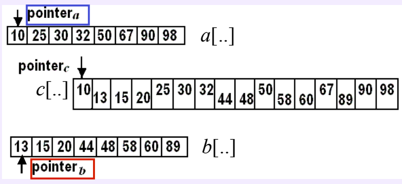
Algorithm MergeSort

- **Professor John von Neumann** (*1945!*): a recursive divide-and-conquer approach
- **Three basic steps**:
  - If the number of items is 0 or 1, return
  - Otherwise, partition the array into two halves and recursively sort the first and the second halves separately
  - Finally, merge the two sorted halves into a sorted array
- Linear time merging $O(n)$ yields MergeSort time complexity $O(n \log n)$

$O(n)$ Merge of Sorted Arrays

if $a[\text{pointer}_a] < b[\text{pointer}_b]$ then $c[\text{pointer}_c] \leftarrow a[\text{pointer}_a]$; $\text{pointer}_a \leftarrow \text{pointer}_a + 1$; $\text{pointer}_c \leftarrow \text{pointer}_c + 1$
else $c[\text{pointer}_c] \leftarrow b[\text{pointer}_b]$; $\text{pointer}_b \leftarrow \text{pointer}_b + 1$; $\text{pointer}_c \leftarrow \text{pointer}_c + 1$

$10 < 13 \rightarrow 10$
$25 \geq 13 \rightarrow 13$
$25 \geq 15 \rightarrow 15$
$25 \geq 20 \rightarrow 20$
$25 < 44 \rightarrow 25$
$30 < 44 \rightarrow 30$
…



Structure of MergeSort

begin MergeSort (an integer array $a[]$ of size n)
1. Allocate a temporary array*) $tmp[]$ of size n
2. **RecursiveMergeSort**(a, tmp, 0, $n-1$)

end MergeSort

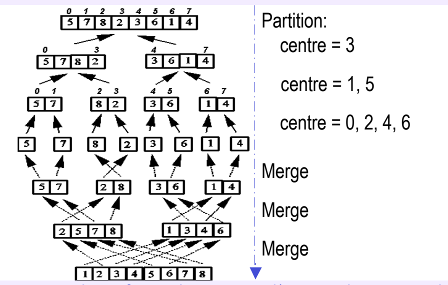
*) To merge each successive pair of the ordered subarrays a[left],…, a[centre] and a[centre+1],…, a[right] and copy the merged array back to a[left],…, a[right]

Recursive MergeSort

begin RecursiveMergeSort (an integer array $a[]$ of size n; a temporary array tmp of size n; range: left, right)
- if left < right then
- centre ← $\lfloor (\text{left} + \text{right}) / 2 \rfloor$
- RecursiveMergeSort(a, tmp, left, centre);
- RecursiveMergeSort(a, tmp, centre + 1, right);
- Merge(a, tmp, left, centre + 1, right);
- end if

end RecursiveMergeSort

How MergeSort works



$2n$ or n comparisons for random or sorted/reverse data, respectively

Analysis of MergeSort

+ $O(n \log n)$ best-, average-, and worst-case complexity because the merging is always linear

—Extra $O(n)$ temporary array for merging data

—Extra copying to the temporary array and back

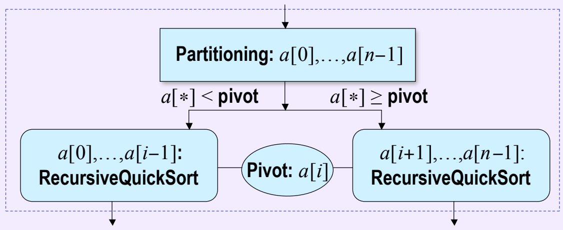
- Useful only for external sorting
- For internal sorting: QuickSort and HeapSort are much better

Algorithm QuickSort

- Sir C.A.R. Hoare (1961): the divide-and-conquer approach
- Four basic steps:
  - If $n = 0$ or 1, return
  - Otherwise, choose one of the items as a **pivot**
  - Partition the remaining items into two disjoint subarrays by placing the items greater than the pivot to its right and all the others to its left
  - Return the result of **QuickSort** of the left subarray, followed by the pivot, followed by the result of **QuickSort** of the right subarray

Recursive QuickSort

- $T(n) = c \cdot n$ (pivot positioning) $+ T(i) + T(n-1-i)$



Analysis of QuickSort: the worst case $O(n^2)$

- If the pivot happens to be the largest (or smallest) item, then one subarray is always empty whereas the second subarray contains all the items except the pivot
- Time for partitioning an array: cn
- Running time for sorting: $T(n) = T(n-1) + cn$
  - "Telescoping" (recall the basic recurrences):

$$T(n) = c\frac{n(n+1)}{2}$$

Analysis of QuickSort: the average case $O(n \log n)$

- The left and right subarrays contain i and $n-1-i$ items, respectively; $i = 0, \ldots, n-1$
- Time for partitioning an array: cn
- Average running time for sorting:

$$T(n) = \tfrac{2}{n}\big(T(0) + \ldots + T(n-2) + T(n-1)\big) + cn, \text{ or}$$

$$nT(n) = 2\big(T(0) + \ldots + T(n-2) + T(n-1)\big) + cn^2$$

$$(n-1)T(n-1) = 2\big(T(0) + \ldots + T(n-2)\big) \qquad + c(n-1)^2$$

Analysis of QuickSort: the average case $O(n \log n)$

$$nT(n) - (n-1)T(n-1) \rightarrow nT(n) = (n+1)T(n-1) + 2cn$$

"Telescoping": $$\frac{T(n)}{n+1} \cong \frac{T(n-1)}{n} + \frac{2c}{n+1}$$

Explicit form: $$\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2c\left(\frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n+1}\right)$$

$$\approx 2cH_{n+1} \approx C \log n$$

where $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} \approx \ln n + 0.577$ is the n^{th} harmonic number

Analysis of QuickSort: the choice of the pivot

- **Never use** the first a[low] or the last a[high] item! (Why?)
- A reasonable choice → the middle item:

$$a[\text{middle}] = \left\lfloor \frac{\text{low} + \text{high}}{2} \right\rfloor$$

where $\lfloor z \rfloor$ is an integer "floor" of the real value z

- **Good choice → the median of three:**

median $\{a[\text{low}], a[\text{middle}], a[\text{high}]\}$

  - Example: median$\{45, 19, 75\}$ → $[19 \leq \mathbf{45} \leq 75] = 45$

Pivot positioning in QuickSort: low=0 , middle=4, high=9

Data to be sorted										Description		
0	1	2	3	4	5	6	7	8	9	←Index		
25	8	2	91	70	50	20	31	15	65	Initial array a		
25	8	2	91	65	50	20	31	15	70	i=**MedianOfThree**(a,low,high]; $p=a[i]$; **swap**(i, a[high–1])		
25	8	2	91	15	50	20	31	65	70			
25	8	2	91	15	50	20	31	65	70	i	j	Condition
	8						31			1	7	$a[i] < p > a[j]$; i++
		2					31			2	7	$a[i] < p > a[j]$; i++

Pivot positioning in QuickSort: low=0 , middle=4, high=9

25	8	2	91	15	50	20	31	65	70	i	j	Condition
			91 31				31 91	65		3	7	$a[i] \geq p > a[j]$; swap; i++; j––
				15		20		65		4	6	$a[i] < p > a[j]$; i++
					50	20		65		5	6	$a[i] < p > a[j]$; i++
						20		65		6	6	$a[i] < p > a[j]$; i++
								65		7	6	$i > j$; **break**
25	8	2	31	15	50	20	65	91	70	**swap**($a[i]$, $p = a$[high–1])		

Data selection: QuickSelect

- Goal: find the ***k*-th smallest item** of an array a of size n
- If k is fixed (e.g., the median), then selection should be faster than sorting
- Linear average-case time $O(n)$ by a small change of QuickSort
- Basic Recursive QuickSelect: to find the k-th smallest item in a subarray:

  (a[low], a[low + 1], …, a[high])

  such that $0 \le$ low $\le k - 1 \le$ high $\le n - 1$

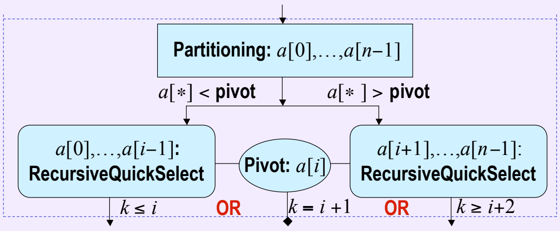
Recursive QuickSelect

- If high = low = $k - 1$: return $a[k - 1]$; otherwise pick a median-of-three pivot and split the remaining items into two disjoint subarrays just as in QuickSort:

  a[low], …, $a[i-1]$ < $a[i]$ = pivot ≤ $a[i+1]$, …, a[high]
- Recursive calls:
  - $k \le i$: **RecursiveQuickSelect**(a, low, $i - 1$, k)
  - $k = i + 1$: **return** $a[i]$
  - $k \ge i + 2$: **RecursiveQuickSelect**(a, $i + 1$, high, k)

Recursive QuickSelect

Average running time $T(n) = cn$ (partitioning of an array)
+ average time for selecting among i or $(n - 1 - i)$ items where i varies from 0 to $n-1$



QuickSelect: low=0, high=$n-1$

- $T(n) = c \cdot n$ (splitting the array) + { $T(i)$ OR $T(n-1-i)$ }
- Average running time:

$$T(n) = \frac{1}{n}\left(T(0) + \ldots + T(n-2) + T(n-1)\right) + cn$$

or $nT(n) = T(0) + \ldots + T(n-2) + T(n-1) + cn^2$

$$(n-1)T(n-1) = T(0) + \ldots + T(n-2) \qquad + c(n-1)^2$$

$nT(n) - (n-1)\, T(n-1) \rightarrow T(n) - T(n-1) \cong 2c$
or $T(n)$ is $O(n)$