The University of Auckland

Algorithm MergeSort

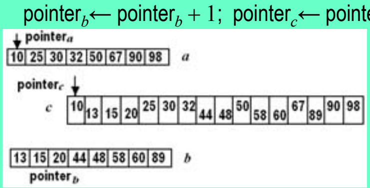
- **John von Neumann** (*1945!*): a recursive divide-and-conquer approach
- Three basic steps:
  - If the number of items is 0 or 1, return
  - Recursively sort the first and the second halves separately
  - Merge two presorted halves into a sorted array
- Linear time merging $O(n)$ yields MergeSort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

Lecture 7 COMPSCI.220.FS.T - 2004 1

The University of Auckland

$O(n)$ Merge of Sorted Arrays

if a[pointer$_a$] < b[pointer$_b$] then c[pointer$_c$] ← a[pointer$_a$];
pointer$_a$ ← pointer$_a + 1$; pointer$_c$ ← pointer$_c + 1$
else c[pointer$_c$] ← b[pointer$_b$];
pointer$_b$ ← pointer$_b + 1$; pointer$_c$ ← pointer$_c + 1$



Lecture 7 COMPSCI.220.FS.T - 2004 2

The University of Auckland

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

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

Lecture 7 COMPSCI.220.FS.T - 2004 3

The University of Auckland

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

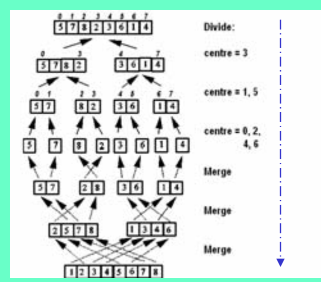
Lecture 7 COMPSCI.220.FS.T - 2004 4

The University of Auckland

How MergeSort works

$2n$ comparisons for random data

n comparisons for sorted/reverse data



Lecture 7 COMPSCI.220.FS.T - 2004 5

The University of Auckland

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 work for copying to the temporary array and back
- Useful only for external sorting
- For internal sorting: QuickSort and HeapSort are much better

Lecture 7 COMPSCI.220.FS.T - 2004 6

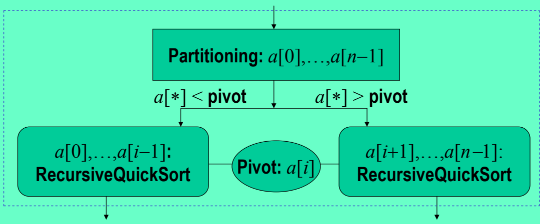
Algorithm QuickSort

- C. A. R. Hoare (1961): the divide-and-conquer approach
- Four basic steps:
  - If $n = 0$ or 1, return
  - Pick a **pivot** item
  - Partition the remaining items into the left and right groups with the items that are smaller or greater than the pivot, respectively
  - Return the **QuickSort** result for the left group, followed by the pivot, followed by the **QuickSort** result for the right group

Lecture 7 COMPSCI.220.FS.T - 2004 7

Recursive QuickSort

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



Lecture 7 COMPSCI.220.FS.T - 2004 8

Analysis of QuickSort: the worst case $\mathrm{O}(n^2)$

- If the pivot happens to be the largest (or smallest) item, then one group is always empty and the second group contains all the items but the pivot
- Time for partitioning an array: $c \cdot n$
- Running time for sorting: $\mathrm{T}(n) = \mathrm{T}(n-1) + c \cdot n$
- "Telescoping" (recall the basic recurrences):

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

Lecture 7 COMPSCI.220.FS.T - 2004 9

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

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

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

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

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

Lecture 7 COMPSCI.220.FS.T - 2004 10

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

$$n\mathrm{T}(n) - (n-1)\mathrm{T}(n-1) \rightarrow n\mathrm{T}(n) = (n+1)\mathrm{T}(n-1) + 2cn$$

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

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

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

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

Lecture 7 COMPSCI.220.FS.T - 2004 11

Analysis of QuickSort: the choice of the pivot

- Never use the first a[low] or the last a[high] item!
- A reasonable choice is the middle item:

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

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

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

a[low], a[middle], a[high]

Lecture 7 COMPSCI.220.FS.T - 2004 12

The University of Auckland

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	←Indices		
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);		
25	8	2	91	15	50	20	31	65	70	$p=a[i]$; **swap**(i, a[high−1])		
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++

Lecture 7 COMPSCI.220.FS.T - 2004 13

The University of Auckland

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	65		3	7	$a[i] \ge p > a[j]$;
			31				91					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])		

Lecture 7 COMPSCI.220.FS.T - 2004 14

The University of Auckland

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 $\le n-1$

Lecture 7 COMPSCI.220.FS.T - 2004 15

The University of Auckland

Recursive QuickSelect

- If high = low = $k-1$: return $a[k-1]$
- Pick a median-of-three pivot and split the remaining elements into two disjoint groups just as in QuickSort:

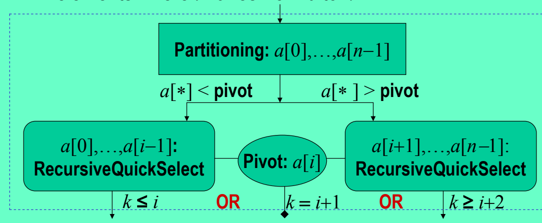
  a[low], …, $a[i-1]$ $< a[i] =$ pivot $< a[i+1]$, …, a[high]
- Recursive calls:
  - $k \le i$: **RecursiveQuickSelect**(a, low, $i-1$, k)
  - $k = i + 1$: **return** $a[i]$
  - $k \ge i + 2$: **RecursiveQuickSelect**(a, $i + 1$, high, k)

Lecture 7 COMPSCI.220.FS.T - 2004 16

The University of Auckland

Recursive QuickSelect

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



Lecture 7 COMPSCI.220.FS.T - 2004 17

The University of Auckland

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

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

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

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

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

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

Lecture 7 COMPSCI.220.FS.T - 2004 18