Sorting II: Quicksort

Instructor: Meng-Fen Chiang

COMPCSI220: WEEK 9

THE UNIVERSITY OF AUCKLAND
Te Whare Wananga o Tamaki Makaurau
NEW ZEALAND

Slides adapted from Mark Wilson, Georgy Gimel’farb, Simone Linz, Tanya Gvozdeva, and Kaiqi Zhao

Quicksort Algorithm

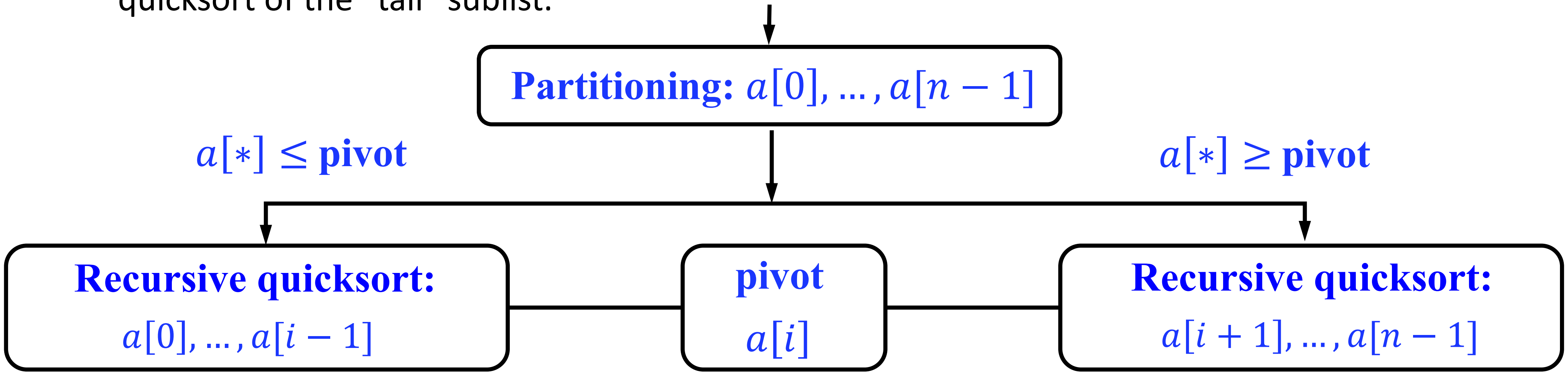
Proposed in 1959/60 by
Sir Charles Antony Richard (Tony) **Hoare**

Born: 11.01.1934 (Colombo, Sri Lanka)
Fellow of the Royal Society (1982)
Fellow of the Royal Academy of Engineering (2005)

- Like mergesort, the divide-and-conquer paradigm.
- Unlike mergesort, subarrays for sorting and merging are formed dynamically, depending on the input, rather than are predetermined.
- Almost all the work: in the division into subproblems.
- Very fast on “random” data, but unsuitable for mission-critical applications due to the very bad worst-case behaviour.

Recursive Quicksort

- If the size, n, of the list, is 0 or 1, return the list. Otherwise:
  1. Choose one of the items in the list as a pivot.
  2. Partition the remaining items into two disjoint sublists, such that all items greater than the pivot follow it, and all elements less than the pivot precede it.
  3. Return the result of quicksort of the "head" sublist, followed by the pivot, followed by the result of quicksort of the "tail" sublist.



Partitioning: $a[0], \dots, a[n-1]$

$a[*] \leq$ **pivot**

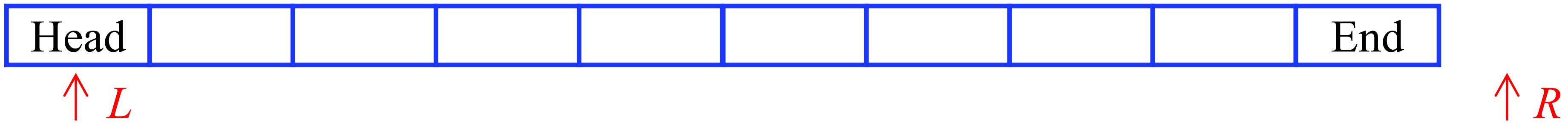
$a[*] \geq$ **pivot**

Recursive quicksort: $a[0], \dots, a[i-1]$

pivot $a[i]$

Recursive quicksort: $a[i+1], \dots, a[n-1]$

Partitioning Algorithm



1. Initialisation:
    - Start pointers L and R at the head of the list and at the end plus one, respectively.
    - Swap the pivot element, p, to the head of the list.
2. Iteration: **while** TRUE, **do**:
    - Decrease R **until** it meets an element less than or equal to p.
    - Increase L **until** it meets an element greater than or equal to p.
    - If $L < R$: Swap the elements pointed by L and R, else swap the pivot element with the element pointed to by R and return R.

Example: Partitioning a List

Data to sort; pivot $p = a[7] = 31$

										Description
25	8	2	91	15	50	20	*31*	70	65	Initial list $L = 0;\ R = 10$
31	8	2	91	15	50	20	25	70	65	Move pivot to head
31	8	2	91	15	50	20	25	70	65	Decrease R from 10 to 7
31	8	2	91	15	50	20	25	70	65	Increase L from 0 to 3
31	8	2	*25*	15	50	20	*91*	70	65	Swap $a[R]$ and $a[L]$
31	8	2	25	15	50	20	91	70	65	Decrease R from 7 to 6
31	8	2	25	15	50	20	91	70	65	Increase L from 3 to 5
31	8	2	25	15	*20*	*50*	91	70	65	Swap $a[R]$ and $a[L]$
31	8	2	25	15	20	50	91	70	65	Stop once $L = R$
20	8	2	25	15	*31*	50	91	70	65	Swap $a[R]$ with pivot

Head (left) sublist $\leq p \leq$ Tail (right) sublist

Proof: Correctness of Partitioning

- After each swap of elements $a[L]$ and $a[R]$
  - each element to the left of index L, as well as $a[L]$ ≤ the pivot p;
  - each element to the right of index R, as well as $a[R]$ ≥ the pivot p.

- After the final swap of p with $a[R]$, which does not exceed p, all elements smaller than p are to its left, and all larger are to its right.
  - Quicksort is easier to program for array, than other types of lists.
  - Constant-time pivot selection is only for arrays, but not linked lists.

    Let the element at $(L+R)/2$ be the pivot, what is the running time for linked list?
  - Partition needs a doubly-linked list to scan forward and backward.

Pseudocode for Quicksort

Algorithm 1 Quicksort - basic

```
function QUICKSORT(list $a[0..n-1]$, integer $l$, integer $r$) ↴
                                                  sorts the sublist $a[l..r]$
    if $l < r$ then
        $i \leftarrow \text{pivot}(a, l, r)$              ▷ return position of pivot
        $j \leftarrow \text{PARTITION}(a, l, r, i)$       ▷ return final position of pivot
        $\text{QUICKSORT}(a, l, j-1)$                     ▷ sort left sublist
        $\text{QUICKSORT}(a, j+1, r)$                     ▷ sort right sublist
```

Proof: Correctness of Quicksort

- By math induction on the size n of the list.
- **Basis**: If n = 1, the algorithm is correct.
- **Hypothesis**: It is correct on lists of size smaller than n.
- **Inductive step**: After positioning, the pivot p at position i; i=1,…,n−1, splits a list of size n into the head sublist of size i and the tail sublist of size n−1−i.
    - Elements of the head sublist are not greater than p.
    - Elements of the tail sublist are not smaller than p.
    - By the induction hypothesis, both the head and tail sublists are sorted correctly.
    - Therefore, the whole list of size n is sorted correctly.

Time Complexity Analysis

- The choice of a pivot is most critical:
  - The wrong choice may lead to the worst-case quadratic time complexity.
  - A good choice equalizes both sublists in size and leads to linearithmic (“$n \log n$”) time complexity.

- The worst-case choice: the pivot happens to be the largest (or smallest) item
  - Then one subarray is always empty.
  - The second subarray contains $n-1$ elements, i.e. all the elements other than the pivot.
  - Quicksort is recursively called only on this second subarray.

Time Complexity Analysis (Contd.)

- The worst-case time complexity of quicksort is $\Theta(n^2)$.
- **Proof**: The partitioning step on n elements: n−1 comparisons (stop once $L \geq R$).
  - At each next step for n≥1, the number of comparisons is one less, so that

$$T(n) = T(n-1) + (n-1);\ T(1) \ = \ 0$$

  - "Telescoping" $T(n) - T(n-1) = n - 1$:

$$\begin{aligned} &T(n) + T(n-1) + T(n-2) + \cdots + T(3) + T(2) \\ &\quad -T(n-1) - T(n-2) - \cdots - T(3) - T(2) - T(1) \\ &\quad = (n-1) + (n-2) + \cdots + 2 + 1 - 0 \end{aligned}$$

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

  - This yields that $T(n) \in \Theta(n^2)$.

Time Complexity Analysis (Contd.)

- The number of comparisons satisfies a recurrence like $C_n = C_p + C_{n-p-1} + n - 1$.

- We can prove that if elements are distinct and all input permutations are equally likely, then quicksort has average running time in $O(n \log n)$.
  - Average number of comparison $\bar{C}_n = \left(\frac{1}{n}\sum_{p=0}^{n-1}(C_p + C_{n-p-1})\right) + n - 1$
  - The recurrence for this average running time is basically

$$T(n) = \frac{2}{n} \sum_{0 \leq p \leq n-1} T(p) + n - 1$$

  - This needs some tricks to estimate the closed-form formula.

- The recurrence for this average running time $T(n) = \frac{2}{n} \sum_{0 \le p \le n-1} T(p) + n - 1$
  - This implies

$$nT(n) = 2 \sum_{0 \le p \le n-1} T(p) + n(n-1) = 2T(n-1) + 2 \sum_{0 \le p \le n-2} T(p) + n(n-1)$$

  - And

$$(n-1)T(n-1) = 2 \sum_{0 \le p \le n-2} T(p) + (n-1)(n-2)$$

  - Thus: $nT(n) = 2T(n-1) + (n-1)T(n-1) - (n-1)(n-2) + n(n-1)$

$$= (n+1)T(n-1) + 2(n-1)$$

$$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \boxed{\frac{2(n-1)}{n(n+1)}} \longrightarrow \frac{4}{n+1} - \frac{2}{n}$$

“Telescoping” $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{4}{n+1} - \frac{2}{n}$ to get the explicit form:

$$\frac{T(n)}{n+1} + \frac{T(n-1)}{n} + \frac{T(n-2)}{n-1} + \cdots + \frac{T(2)}{3} + \frac{T(1)}{2} - \frac{T(n-1)}{n} - \frac{T(n-2)}{n-1} - \cdots - \frac{T(2)}{3} - \frac{T(1)}{2} - \frac{T(0)}{1}$$

$$= \left(\frac{4}{n+1} - \frac{2}{n}\right) + \left(\frac{4}{n} - \frac{2}{n-1}\right) + \left(\frac{4}{n-1} - \frac{2}{n-2}\right) + \cdots + \left(\frac{4}{2} - \frac{2}{1}\right), \text{ or}$$

$$\frac{T(n)}{n+1} = T(0) + 4\left(\frac{1}{n+1} + \cdots + \frac{1}{2}\right) - 2\left(\frac{1}{n} + \cdots + 1\right) = \frac{4}{n+1} + 2\left(\frac{1}{n} + \cdots + \frac{1}{2}\right) - 2$$

$$= \frac{4}{n+1} + 2(H_n - 1) - 2 = 2H_n - 4 + \frac{4}{n+1}$$

Then, the closed-formed formula is
$T(n) = 2(n+1)H_n - 4(n+1) + 4$

This gives $T(n) \in \Theta(n \log n)$

$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$ is the n^{th} harmonic number and $H_n \in \Theta(\log n)$.

Choice of Pivot

- Passive pivot choice − a fixed position in each sublist
- $\Omega(n^2)$ running time for frequent in practice nearly sorted lists under the naïve selection of the first or last position.
- A more reasonable choice: the middle element of each sublist.
- Random inputs resulting in $\Omega(n^2)$ time are rather unlikely.
- But still: vulnerability to an “algorithm complexity attack” with specially designed “worst-case" inputs.

Active Pivot Strategy

- The best active pivot – the exact median of the list, dividing it into (almost) equal sized sublists, – is computationally inefficient.
- The median-of-three strategy to approximate the true median
- The pivot $p = \ median \ \{a[i_{beg}], a[i_{mid}], a[i_{end}]\}$ where i_{beg}, i_{end} and $i_{mid} = \left\lfloor \frac{i_{beg}+i_{end}}{2} \right\rfloor$

  refer to the first, last and middle elements, respectively, of a sublist, $a[i_{beg}, \ldots, i_{end}]$

An example: $a \ = \ (45, 25, 15, 31, 75, 80, 60, 20, 19)$

$$\text{median}\{45, 75, 19\} \rightarrow \quad 19 \leq 45 \leq 75 \rightarrow \quad 45$$

$$a \ = ((19, 25, 15, 31, 20), 45, (80, 60, 75))$$

Active Pivot Strategy (Contd.)

- Bad performance is still possible with the median-of-three strategy, but becomes much less likely, than for a passive strategy.

- Random choice of the pivot
  - The expected running time is $\Theta(n \log n)$ for any given input.
  - No adversary can force the bad behaviour by choosing nasty inputs.
  - A small extra overhead for generating a “random” pivot position.
  - Bad cases: only by bad luck, independent of the input.
  - An alternative: to first randomly shuffle the input in linear, $\Theta(n)$, time and use then the naïve pivot selection.

SUMMARY

- Quicksort Illustration
  - Partitioning Algorithm
  - Correctness of Partitioning
  - Correctness of Quicksort
  - Choice of Pivot

- Time Complexity Analysis