Algorithm Quicksort: Analysis of Complexity

Lecturer: Georgy Gimel'farb

COMPSCI 220 Algorithms and Data Structures

1. Algorithm quicksort
2. Correctness of quicksort
3. Quadratic worst-case time complexity
4. Linearithmic average-case time complexity
5. Choosing a better pivot
6. Partitioning algorithm

Algorithm QuickSort

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

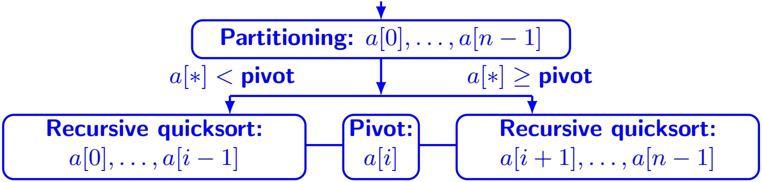
Born: 11.01.1934 (Colombo, Sri Lanka)
Fellow of the Royal Society (1982)
Fellow of the Royal Academy of Engineering (2005)

- 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.

Basic 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. Next, **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. Finally, return the result of quicksort of the "head" sublist, followed by the pivot, followed by the result of quicksort of the "tail" sublist.



Lemma 2.13 (Textbook): Quicksort is correct.

Proof: 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; $i = 1, \ldots, n-1$, 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.

Any implementation specifies what to do with items equal to the pivot.

Analysing Quicksort: The Worst Case $T(n) \in \Omega(n^2)$

The choice of a pivot is most critical:

- The wrong choice may lead to the worst-case quadratic time complexity.
- A good choice equalises 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 group.

However, quicksort is fast on the "randomly scattered" pivots.

Analysing Quicksort: The Worst Case $T(n) \in \Omega(n^2)$

Lemma 2.14 (Textbook): The worst-case time complexity of quicksort is $\Omega(n^2)$.

Proof. The partitioning step: at least, $n-1$ comparisons.

- At each next step for $n \geq 1$, the number of comparisons is one less, so that $T(n) = T(n-1) + (n-1)$; $T(1) = 0$.
- "Telescoping" $T(n) - T(n-1) = n - 1$:

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

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

Analysing Quicksort: The Average Case $T(n) \in \Theta(n \log n)$

For any pivot position i; $i \in \{0, \ldots, n-1\}$:

- Time for partitioning an array : cn
- The head and tail subarrays contain i and $n-1-i$ items, respectively: $T(n) = cn + T(i) + T(n-1-i)$

Average running time for sorting (a more complex recurrence):

$$
\begin{aligned}
T(n) &= \frac{1}{n} \sum_{i=0}^{n-1} \left(T(i) + T(n-1-i) + cn \right) \\
&= \frac{2}{n} \left(T(0) + T(1) + \ldots + T(n-2) + T(n-1) \right) + cn, \text{ or} \\
nT(n) &= 2 \left(T(0) + T(1) + \ldots + T(n-2) + T(n-1) \right) + cn^2 \\
(n-1)T(n-1) &= 2 \left(T(0) + T(1) + \ldots + T(n-2) \right) + c(n-1)^2
\end{aligned}
$$

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

Thus, $nT(n) \approx (n+1)T(n-1) + 2cn$, or $\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}$

Analysing Quicksort: The Average Case $T(n) \in \Theta(n \log n)$

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

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

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

$$= \frac{2c}{n+1} + \frac{2c}{n} + \ldots + \frac{2c}{3} + \frac{2c}{2},$$ or

---

$$\frac{T(n)}{n+1} = \frac{T(0)}{1} + 2c\left(\frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n} + \frac{1}{n+1}\right) \approx 2c(H_{n+1} - 1) \approx c' \log n$$

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

Therefore, $T(n) \approx c'(n+1)\log n \in \Theta(n \log n)$.

Quicksort is our first example of dramatically different worst-case and average-case performances!

Implementations of Quicksort

Choices to be made for implementing the basic quicksort algorithm:

- How to implement the list?
- How to choose the pivot?
- How to partition the list around the 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 $p = \text{median}\left\{a[i_{\text{beg}}], a[i_{\text{mid}}], a[i_{\text{end}}]\right\}$ where i_{beg}; i_{end}, and $i_{\text{mid}} = \left\lfloor \frac{i_{\text{beg}}+i_{\text{end}}}{2} \right\rfloor$ refer to the first, last, and middle[a] elements, respectively, of a sublist, $a[i_{\text{beg}}], a[i_{\text{beg}}+1, \ldots, a[i_{\text{end}}]$.

[a] $\lfloor z \rfloor$ is an integer floor of the real value z.

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

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

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

Active Pivot Strategy

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.

Partitioning Algorithm

Head									End

$\uparrow L$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ $\uparrow R$

1. **Initialisation:**
   1. Start pointers L and R at the head of the list and at the end plus one, respectively.
   2. Swap the pivot element, p, to the head of the list.
2. **Iteration:** **while** $L < R$, **do**:
   1. Decrement R
      until it meets an element less than or equal to p.
   2. Increment L
      until it meets an element greater than or equal to p.
   3. Swap the elements pointed by L and R.
3. Once $L = R$, swap the pivot element with the element pointed to by L.

Example 2.17 (Textbook): 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	Stop R
31	8	2	91	15	50	20	25	70	65	Stop L
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	Stop R
31	8	2	25	15	50	20	91	70	65	Stop L
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 $R = L$
20	8	2	25	15	*31*	50	91	70	65	Swap $a[L]$ with pivot

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

Correctness of Partitioning

Lemma 2.18 (Textbook): The Partitioning Is Correct

Proof. After each swap of elements $a[L]$ and $a[R]$,

- each element to the left of index L, as well as $a[L]$, is less than or equal to the pivot p;
- each element to the right of index R, as well as $a[R]$, is greater than or equal to the pivot p.

After the final swap of p with $a[L]$, 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.
  - What time will the median-of-three take for a linked list?
- Partition needs a doubly-linked list to scan forward and backward.

Pseudocode for Array-Based Quicksort

algorithm `quickSort` — sorts the subarray $a[l..r]$

Input: array $a[0..n-1]$; array indices l, r

begin

  if $l < r$ **then**

    $i \leftarrow \texttt{pivot}(a, l, r)$ — return position of pivot

    $j \leftarrow \texttt{partition}(a, l, r, i)$ — return final position of pivot

    $\texttt{quickSort}(a, l, j-1)$ — sort left subarray

    $\texttt{quickSort}(a, j+1, r)$ — sort right subarray

  end if

  return a

end