COMP 250: Quicksort

Carlos G. Oliver

February 13, 2018
ALGORITHMS
BY COMPLEXITY

MORE COMPLEX →

LEFTPAD  QUICKSORT  GIT  MERGE  SELF-DRIVING  CAR  GOOGLE  SEARCH  BACKEND

SPRAWLING EXCEL SPREADSHEET BUILT UP OVER 20 YEARS BY A CHURCH GROUP IN NEBRASKA TO COORDINATE THEIR SCHEDULING

¹https://xkcd.com/1667/
Quicksort

- Invented by Sir Tony Hoare in 1959 while on exchange in the Soviet Union working on machine translation.
- Tony needed a good way to sort Russian words alphabetically to efficiently look them up in a “dictionary”, so he invented QuickSort.

“I think Quicksort is the only really interesting algorithm that I ever developed.” – Sir Tony Hoare

Fun fact: Tony Hoare invented the “null pointer”, and he is sorry.

Main idea in QuickSort

- Idea: why don’t we start by finding the final position of a given element in the **sorted** array?
- When is an element in its final position of a sorted array?
- For any element \( x \) in a **sorted** array, all elements before it (to the left) are smaller, and all elements to the right are larger.

| <= \( x \) | \( x \) | >= \( x \) |

- Goal: find the position in the array such that the above condition holds → linear time.
- Split the array at \( x \) and sort the resulting subarrays.
- QuickSort is an *in-place* sorting algorithm. i.e. does not require extra memory for execution (like the temporary array in MergeSort).
Algorithm 1: QuickSort(A)

Result: Sorted array A.

1 if len(A) \leq 1 then
2 | return A
3 else
4 | x = A.removeFirst()
5 | list1 = A.getElementsLessThan(x)
6 | list2 = A.getElementsNotLessThan(x)
7 | list1 = QuickSort(list1)
8 | list2 = QuickSort(list2)
9 | return concatenate(list1, x, list2)
Quicksort illustration
QuickSort “in place” pseudocode

- If we’re clever about how we get arrange the elements around the pivot we can do without temporary arrays.
- This is the job of the partition function

---

**Algorithm 2: QuickSort(A, p, q)**

**Result:** Sorted array A between indices p and q

1. if \( p \geq q \) then
2. \( x \leftarrow \text{Partition}(A, p, q) \)
3. \( \text{QuickSort}(A, p, x - 1) \)
4. \( \text{QuickSort}(A, x + 1, q) \)
5. return
Partition

- The job of partition is for some “pivot” value (usually we take the first or last element), arrange all elements such that elements less than the pivot are to the left of the pivot, and conversely for the right.
- partition places the pivot in the correct position and returns that position.
- This procedure runs in linear time. i.e. $\Theta(n)$
Algorithm 3: Partition(A, p, q)

Result: Index left and rearranges elements of A such that

1. x ← A[q]
2. left ← p
3. right ← q − 1
4. while left ≤ right do
   5. while left ≤ right ∧ A[left] < x do
      6. left ← left + 1
   7. while left ≤ right ∧ A[right] ≥ x do
      8. right ← right − 1
   9. if left < right then
      10. swap(A[left], A[right])
11. swap(A[p], A[left])
12. return left
Partition (Version 1) Loop invariant intuition

- At each step we ensure that elements to the right of right and left of left are on the correct side of the pivot.
Partition example

Carlos G. Oliver
COMP 250: Quicksort
February 13, 2018 11 / 21
Back to the big picture
Algorithm 4: Partition(A, p, q)

Result: Index left and rearranges elements of A such that

1 \( x ← A[q] \)
2 \( left ← p \)
3 for right \( ← p + 1 \) to q do
4     if A[right] ≤ x then
5         left \( ← left + 1 \)
6         swap(A[left-1], A[right])
7 swap(A[p], A[left])
8 return left
Partition (Version 2) Loop Invariant

\[ p \quad \text{left} \quad \text{right} \quad q \]

\[ \begin{array}{c}
\text{\langle x} \\
\text{\rangle x} \\
\text{?} \\
x
\end{array} \]
Demo of partition (version 2)

1. swap(2,3)
2. swap(2,4)
3. swap(2,7)
4. swap(7,5)
5. swap(2,3)

Carlos G. Oliver
COMP 250: Quicksort
February 13, 2018
QuickSort Time Complexity: Worst case

- Worst case of QuickSort occurs on sorted (or inversely sorted) arrays.
- All elements will fall to the same side of the pivot so the splits decrease recursive calls by 1 only.
- \( T_{\text{worst}}(n) \in \Theta(n^2) \)

![Diagram of QuickSort subproblem sizes and total partitioning time](https://ka-perseus-images.s3.amazonaws.com/7da2ac32779bef669a6f05dceb62f219a9132158.png)
QuickSort Time Complexity: Worst case recurrence

\[ T_{\text{worst}}(n) = cn + T_{\text{worst}}(n - 1) \]  
\[ = cn + c(n - 1) + T(n - 2) \]  
\[ = cn + c(n - 1) + c(n - 2) + T(n - 3) \]  
\[ = cn + c(n - 1) + c(n - 2) + c(n - k - 1) + T(n - 1) \]  
\[ = c \sum_{k=0}^{n}, k + T(0), \quad \text{when } n - k = 1 \]  
\[ = c \frac{n(n + 1)}{2} \in \Theta(n^2) \]

- We do a linear amount of work for partition \((cn)\) plus QuickSort on the rest of the array \(T(n - 1)\)
QuickSort Time Complexity: Best case

- Best case is when the pivot divides the array in two each time.
- $T_{\text{best}}(n) \in \Theta(n \log n)$

![Diagram showing the subproblem sizes and total partitioning time for all subproblems of this size.](https://ka-perseus-images.s3.amazonaws.com/21cd0d70813845d67fbb11496458214f90ad7cb8.png)
QuickSort Time Complexity: Best case recurrence

\[ T_{\text{best}}(n) = cn + 2T_{\text{best}}\left(\frac{n}{2}\right) \in \Theta(n \log n) \]  \hspace{1cm} (2)

- We do a linear amount of work for partition \((cn)\) plus QuickSort on the rest of the array \(T(n-1)\)
- Recurrence is the same as MergeSort.
Quicksort Considerations

- **Average case:** if the array is in random order, it is split in roughly even parts: \( t_{\text{average}}(n) \in \Theta(n \log n) \)
- **Advantage over MergeSort**
  - Constants hidden in \( \mathcal{O}(n \log n) \) smaller than in MergeSort \( \rightarrow \) faster by constant factor.
  - QuickSort is easy to do without additional memory. Good for large lists.
- **SelectionSort and InsertionSort** are in-place: all we are doing is moving elements around the array
- **There are strategies to picking pivots to avoid bad running times.**
Good to know

\[
1 + 2 + 3 + 4 + 5 + \ldots + k = \sum_{k=1}^{n} k = \frac{n(n + 1)}{2}
\]

\[
1 + 2 + 4 + \ldots + 2^k = \sum_{k=1}^{n} 2^k = 2^{n+1} - 1 \tag{3}
\]

\[
1 + x + x^2 + x^3 + x^4 + \ldots + x^k = \sum_{k=1}^{n} x^k = \frac{x^{k+1} - 1}{x - 1}
\]