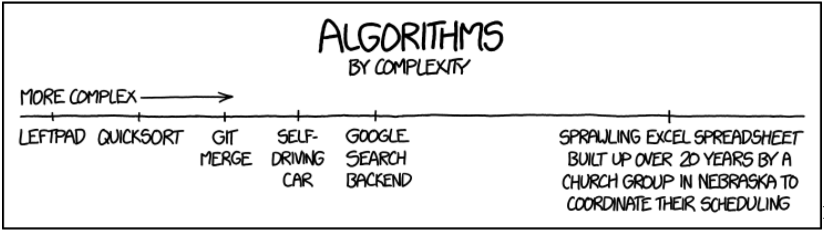
COMP 250: Quicksort

Carlos G. Oliver

February 13, 2018

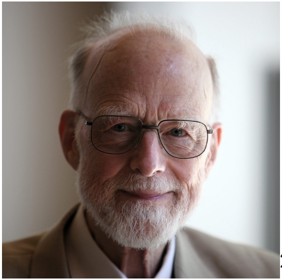


[1]

[1] `https://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.


[2]

- "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.

[2] https://en.wikipedia.org/wiki/Tony_Hoare

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 $\rightarrow$ 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).

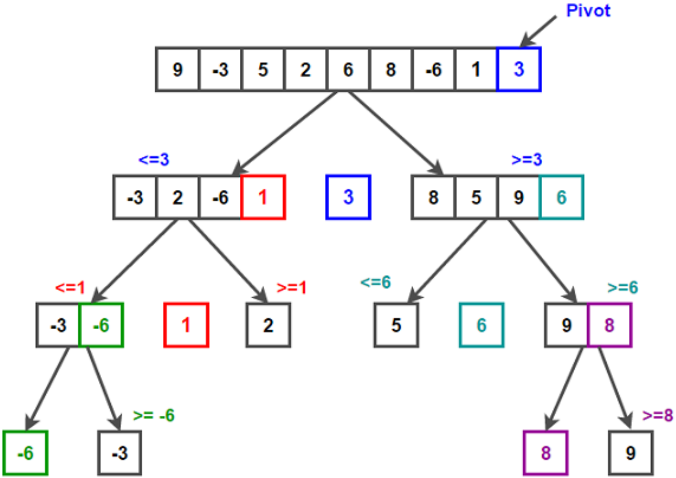
Quicksort (simple, not in place)

Algorithm 1: QuickSort(A)

Result: Sorted array A.

```
if len(A) ≤ 1 then
    return A
else
    x = A.removeFirst()
    list1 = A.getElementsLessThan(x)
    list2 = A.getElementsNotLessThan(x)
    list1 = QuickSort(list1)
    list2 = QuickSort(list2)
    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

if $p \geq q$ **then**

    $x \leftarrow \texttt{Partition}(A, p, q)$

    $\texttt{QuickSort}(A, p, x-1)$

    $\texttt{QuickSort}(A, x+1, q)$

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. $\Theta(n)$

Parition pseudocode (Version 1)

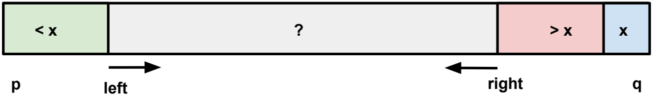
Algorithm 3: Partition(A, p, q)

Result: Index *left* and rearranges elements of A such that $\forall i < left, A[i] \leq A[left]$ and $\forall k > left, A[k] \geq A[left]$.

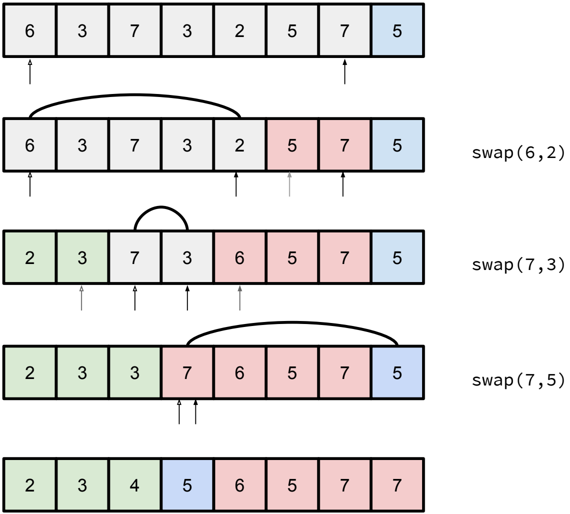
$x \leftarrow A[q]$
$left \leftarrow p$
$right \leftarrow q - 1$
while $left \leq right$ **do**
    while $left \leq right \wedge A[left] < x$ **do**
        $left \leftarrow left + 1$
    while $left \leq right \wedge A[right] \geq x$ **do**
        $right \leftarrow right - 1$
    if $left < right$ **then**
        `swap`(*A[left], A[right]*)
`swap`(*A[p], A[left]*)
return left

Partition (Version 1) Loop invariant intuition

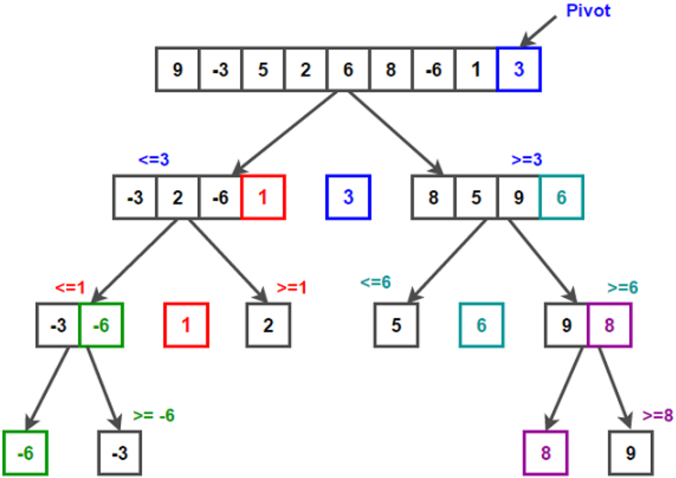
- At each step we ensure that elements to the right of `right` and left of `left` are on the correct side of the pivot.



Partition example



Back to the big picture



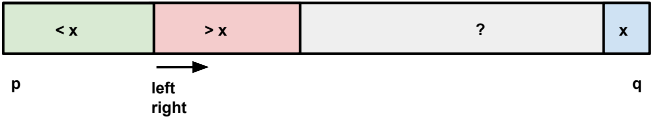
Parition pseudocode (Version 2)

Algorithm 4: Partition(A, p, q)

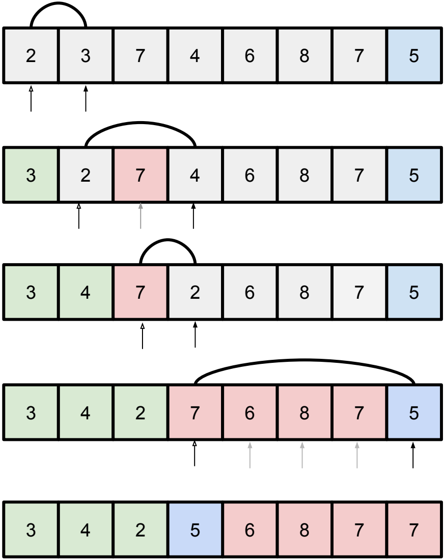
Result: Index *left* and rearranges elements of A such that $\forall i < left, A[i] \leq A[left]$ and $\forall k > left, A[k] \geq A[left]$.

```
x ← A[q]
left ← p
for right ← p + 1 to q do
    if A[right] ≤ x then
        left ← left + 1
        swap(A[left-1], A[right])
swap(A[p], A[left])
return left
```

Partition (Version 2) Loop Invariant



Demo of partition (version 2)



swap(2,3)

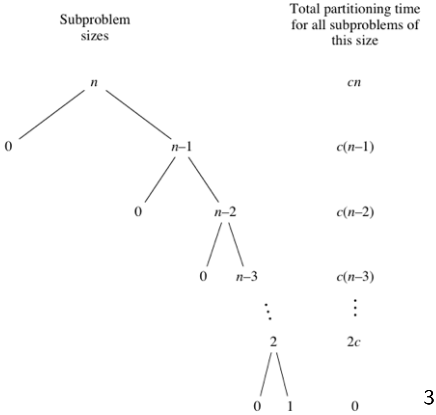
swap(2,4)

swap(2,7)

swap(7,5)

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.
- $T_{\text{worst}}(n) \in \Theta(n^2)$



[3]

[3] https://ka-perseus-images.s3.amazonaws.com/7da2ac32779bef669a6f05decb62f219a9132158.png

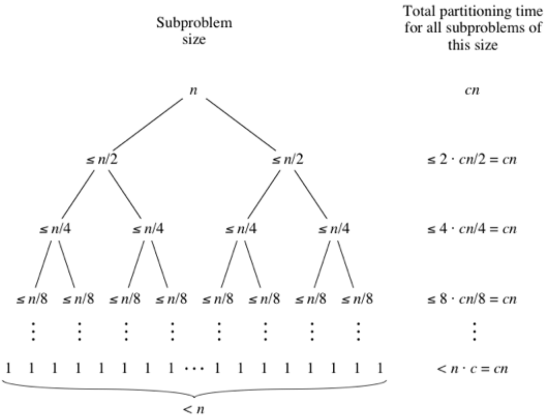
QuickSort Time Complexity: Worst case recurrence

$$
\begin{aligned}
T_{\text{worst}}(n) &= cn + T_{\text{worst}}(n-1) && (1)\\
&= cn + c(n-1) + T(n-2) && (2)\\
&= cn + c(n-1) + c(n-2) + T(n-3) && (3)\\
&= cn + c(n-1) + c(n-2) + c(n-k-1) + T(n-1) && (k)\\
&= c\sum_{k=0}^{n}, k + T(0), \qquad \text{when } n-k=1\\
&= c\frac{n(n+1)}{2} \in \Theta(n^2)
\end{aligned}
\tag{1}
$$

- 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.
- $T_{\text{best}}(n) \in \Theta(n \log n)$



[4]

[4] https://ka-perseus-images.s3.amazonaws.com/21cd0d70813845d67fbb11496458214f90ad7cb8.png

QuickSort Time Complexity: Best case recurrence

$$T_{\text{best}}(n) = cn + 2T_{\text{best}}\left(\frac{n}{2}\right) \in \Theta(n \log n) \tag{2}$$

- 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

$$
\begin{aligned}
1 + 2 + 3 + 4 + 5 + ... + k = \sum_{k=1}^{n} k &= \frac{n(n+1)}{2} \\
1 + 2 + 4 + ... + 2^k = \sum_{k=1}^{n} 2^k &= 2^{n+1} - 1 \\
1 + x + x^2 + x^3 + x^4 + ... + x^k = \sum_{k=1}^{n} x^k &= \frac{x^{k+1} - 1}{x - 1}
\end{aligned} \tag{3}
$$