Lecture 6: Quicksort

MCS/CS 401 Computer Algorithms I — June 27, 2022
Instructor: Danko Adrovic

Lecture Outline

- Sorting and Order Statistics
- Quicksort
  - Quicksort Algorithm Description
  - QUICKSORT Algorithm Illustration
  - Partitioning Type resulting in the Worst-case Scenario
  - Partitioning Type resulting in the Best-case Scenario

1 Sorting and Order Statistics

We continue with this topic by covering Chapter 7 and the Quicksort.

2 Quicksort

The worst-case running time of the Quicksort algorithm is $\Theta(n^2)$ for an input array of length n. Even though the worst case performance appears to be far from ideal, this slow worst-case running time does not occur frequently. Instead, Quicksort is often the best practical choice for sorting because it is remarkably efficient on the average, with an expected running time of $\Theta(n\ lg(n))$. It can also be shown in the analysis that the constant factors hidden in the $\Theta(n\ lg(n))$ notation are very small.

Quicksort also has the advantage of sorting an array A **in place**! That is, it rearranges the numbers (keys) within the array A, with at most a constant number of them stored outside the array at any time. This has a low and favorable impact on the memory consumption, and hence, *memory complexity*. This attribute becomes important when working *virtual environments*.

2.1 Quicksort Algorithm Description

Just like Merge Sort, the Quicksort applies the *divide-and-conquer* paradigm. For an array A[p..r], the components of the paradigm consist of the following:

Divide: Partition (rearrange) the array A[p..r] into two, possibly empty, subarrays A[p..q-1] and A[q+1..r] such that each element of A[p..q-1] is less than or equal to A[q], which is, in turn, less than or equal to each element of A[q+1..r]. Compute the index q as part of this partitioning procedure.

Conquer: Sort the two subarrays A[p..q-1] and A[q+1..r] by making recursive calls to Quicksort.

Combine: Because Quicksort is an *in-place* sorting algorithm, the entire subarray is already sorted when the algorithm has finished. As a result, the entire array A[p..r] is sorted.

The following is the algorithmic representation of the process that we just described above. In order to sort an entire array, QUICKSORT(A, 1, A.length) is called with the specified arguments: array A, index location of the first element in array A, index location of the last element in the array A.

```
QUICKSORT(A, p, r)
1  if p < r
2      q = PARTITION(A, p, r)
3      QUICKSORT(A, p, q − 1)
4      QUICKSORT(A, q + 1, r)
```

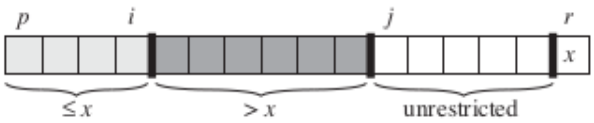
Figure 1: Recursive $QUICKSORT$ Algorithm.

While the algorithm is relatively short, the brunt of the work is done by the $PARTITION$ procedure. The crucial aspect of the $PARTITION$ procedure is that it rearranges the subarray A[p..r] in place!

```
PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4      if A[j] ≤ x
5          i = i + 1
6          exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

Figure 2: $PARTITION$ procedure rearranges the subarray A[p..r] in place.

The $PARTITION$ procedure partitions the array A[p..r] into four, possibly empty, regions.



After selecting an element x = A[r] as a pivot element, the $QUICKSORT$ algorithm proceeds by selecting value from the **unrestricted region**. It places these values in light-gray if they are *smaller or equal* to x. Otherwise, it places these values in dark-gray region if they are *greater* than x. This is an elementary description, however. On next page we look at a more complicated example.

2.2 *QUICKSORT* Algorithm Illustration

The following illustration shows the steps of the *QUICKSORT* algorithm, executed on an 8-element array A. Note the pivot element A[r] = x = 4, in this example.

Light-gray shaded array elements are all in the first partition with values *smaller or equal* to x. **Dark-gray** shaded elements are in the second partition with values *greater* than x. The **unshaded** elements have not yet been put in either one of two first two partitions, and the final white element is the pivot x.

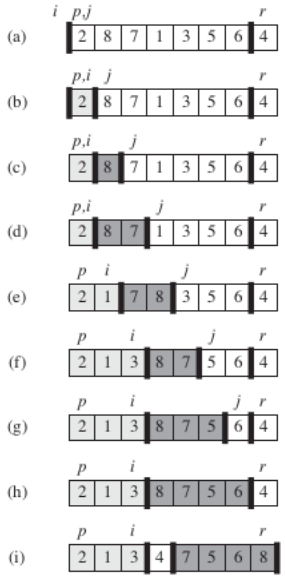


Figure 3: *QUICKSORT* illustration

(a) The initial array with its variable settings. None of the elements have been placed in either of the first two partitions. The pivot value x = A[r] = 4 has been selected. Then the algorithm starts modifying the array in place on lines (b) through (i).

(b) The value 2 is swapped with itself and put in the partition of smaller values (light-gray).

(c)-(d) The values 8 and 7 are added to the partition of larger values (dark-gray).

(e) The values 1 and 8 are **swapped** (not just moved), and the smaller partition grows.

(f) The values 3 and 7 are swapped, and the smaller partition grows.

(g)-(h) The larger partition grows to include 5 and 6, and the loop terminates.

(i) In lines 7-8, the pivot element is swapped so that it lies between the two partitions.

Proposition 1. *Running time of $PARTITION(A, p, r)$ procedure on an subarray of size n is $\Theta(n)$.*

Proof. Determining if $A[j] \leq x$ takes constant time. Exchanging elements $A[i]$ with $A[j]$ in the array A also takes constant time. Both of these operations are carried out in a single for loop, running exactly $r - p = n - 1$ iterations. Outside of the for loop, we have a single exchange between A[i+1] and A[r], which also takes constant time. Therefore, $\Theta(1)+\Theta(1)+\Theta(n-1) + \Theta(1) = \Theta(n)$. □

2.3 Partitioning Type resulting in the Worst-case Scenario

Consider an array A, consisting of n elements. The worst-case scenario for the partitioning procedure, and hence for the Quicksort algorithm, occurs when the array A has the configuration in which one partitioning will result in a subproblem with $n - 1$ elements and the other subproblem with 0 elements. The pivot is an additional and separate element, hence $n - 1$ size of one partition and 0 for the other.

For example, the array $A = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$ will fall into this category, as well as $A = [10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$, and $A = [5, 5, 5, 5, 5, 5, 5, 5, 5, 5]$. That is, the cases where elements of the array A are already sorted, or sorted in reverse order, or all elements are the same. These kinds of configurations cause the largest imbalance in the two partitions (or subproblems). This is why it was remarked earlier that such cases are rare in practice, and the worst-case scenarios are seldom encountered.

Proposition 2. *The worst-case running time T(n) of $QUICKSORT(A, p, r)$ algorithm on an array A of size n is $T(n) = T(n - 1) + \Theta(n)$.*

Proof. For the worst-case, we are assuming that the imbalanced partitioning occurs in each recursive call (ie. so that worst-case scenario is maintained throughout the running of the algorithm).
We first focus on the partition function in the $QUICKSORT$ algorithm. Because we have to partition n elements into smaller/equal than pivot, or larger than pivot, we have to process n elements. Hence, this operation contributes $\Theta(n)$ to the total running time of the algorithm.

In the worst-case scenario, one subproblem is of size $n-1$, and therefore contributes $T(n-1)$ to the total running time of the algorithm. The second problem is of size 0, and therefore contributes $T(0) = \Theta(1)$, a constant time, to the total running time of the algorithm.

Adding up the individual running times, we obtain the following recurrence

$$
\begin{aligned}
T(n) &= \Theta(n) + T(0) + T(n-1) \\
T(n) &= \Theta(n) + \Theta(1) + T(n-1) \\
T(n) &= \Theta(n+1) + T(n-1) \\
T(n) &= \Theta(n) + T(n-1) \\
T(n) &= T(n-1) + \Theta(n)
\end{aligned}
$$

□

The Proposition above only provided us with the relation, which describes the running time of the $QUICKSORT$ algorithm recursively. Now we show the bound for the running time by solving the recurrence.

Proposition 3. *The worst-case running time* $T(n) = T(n-1) + \Theta(n)$ *is* $O(n^2)$.

However, before we prove the Proposition, it is important to know how such a "guess", that $T(n) = T(n-1) + \Theta(n)$ is $\Theta(n^2)$, was derived in the first place.

First, rewrite the recurrence as $T(n) = T(n-1) + cn$, for some constant $c > 0$. Then we unroll the recurrence.

$$\begin{aligned}
T(1) &= T(0) + 1c \\
T(2) &= T(1) + 2c = T(0) + 3c \\
T(3) &= T(2) + 3c = T(0) + 6c \\
T(4) &= T(3) + 4c = T(0) + 10c \\
&... = ... \\
&... = ... \\
T(n) &= T(0) + \frac{cn(n+1)}{2} \\
T(n) &= T(0) + \frac{cn^2 + cn}{2} \\
T(n) &= O(n^2)
\end{aligned}$$

The sequence $1, 3, 6, 10, ...$ comes from a well-known arithmetic sequence, whose n^{th} term can be computed by $\frac{n(n+1)}{2}$.

Proposition 4. *The worst-case running time* $T(n) = T(n-1) + \Theta(n)$ *is* $O(n^2)$.

Proof. Assume that $T(n) \leq n^2 d$, for $d > 0$. Then

$$\begin{aligned}
T(n) = T(n-1) + cn &\leq n^2 d \\
d(n-1)^2 + cn &\leq n^2 d \\
d(n^2 - 2n + 1) + cn &\leq n^2 d \\
n^2 d - 2nd + d + cn &\leq n^2 d \\
-2nd + d + cn &\leq 0 \\
2nd - d &\geq cn \\
(2n-1)d &\geq cn \\
d &\geq \frac{cn}{2n-1}
\end{aligned}$$

For any $n \geq 1$, we can take any $d \geq c$ and we see that the last inequality will hold. Therefore, $T(n)$ is $O(n^2)$. □

This bound can be made tight, such that T(n) is $\Theta(n^2)$! See Chapter 7.2 of our book.

2.4 Partitioning Type resulting in the Best-case Scenario

The best-case running time of the $QUICKSORT$ algorithm occurs when when the $PARTITION$ procedure produces two evenly split subproblems, where each subproblem is no more than $n/2$ (one subproblem will be $\lfloor n/2 \rfloor$ and the other $\lceil n/2 \rceil - 1$). In this case, we get the recurrence for the running time $T(n) = 2T(n/2) + \Theta(n)$.

Proposition 5. *The best-case running time of the $QUICKSORT$ algorithm is $\Theta(n\ lg(n))$.*

Proof. The recurrence relation for the best-case running time is given by $T(n) = 2T(n/2) + \Theta(n)$. By the Case 2 of the master theorem, $T(n)$ is $\Theta(n\ lg(n))$. □