Introduction to Algorithms

George Voutsadakis[1]

[1]Mathematics and Computer Science
Lake Superior State University

LSSU Math 400

1. Functions
   - Description of Quicksort
   - Performance of Quicksort
   - A Randomized Version of Quicksort

Subsection 1

Description of Quicksort

Description of QUICKSORT

- Quicksort, like merge sort, applies the divide-and-conquer paradigm.
- The three-step divide-and-conquer process for sorting a typical subarray $A[p \dots r]$:
  - Divide: Partition (rearrange) the array $A[p \dots r]$ into two (possibly empty) subarrays $A[p \dots q-1]$ and $A[q+1 \dots r]$, such that each element of $A[p \dots q-1]$ is less than or equal to $A[q]$, which is, in turn, less than or equal to each element of $A[q+1 \dots r]$. Compute the index q as part of this partitioning procedure.
  - Conquer: Sort the two subarrays $A[p \dots q-1]$ and $A[q+1 \dots r]$ by recursive calls to quicksort.
  - Combine: Because the subarrays are already sorted, no work is needed to combine them: The entire array $A[p \dots r]$ is now sorted.

The Procedure QUICKSORT

QUICKSORT(A, p, r)

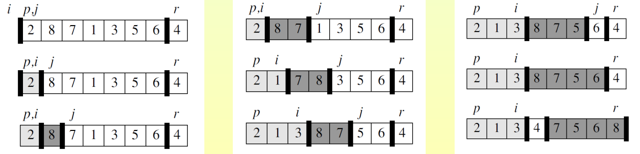
1. if $p < r$
2. $\quad q = \text{PARTITION}(A, p, r)$
3. $\quad \text{QUICKSORT}(A, p, q-1)$
4. $\quad \text{QUICKSORT}(A, q+1, r)$

- To sort an array A, the initial call is $\text{QUICKSORT}(A, 1, A.\text{length})$.
- The key to the algorithm is the PARTITION procedure, which rearranges the subarray $A[p \ldots r]$ in place.

The PARTITION Procedure

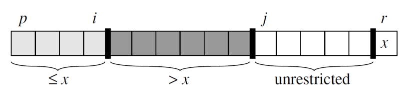
PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
    if $A[j] \leq x$
        $i = i + 1$
        exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
return $i + 1$



How PARTITION Works

- PARTITION always selects an element $x = \mathrm{A}[r]$ as a **pivot** element around which to partition the subarray $A[p \dots r]$.
- As the procedure runs, it partitions the array into four (possibly empty) regions that satisfy a loop invariant:

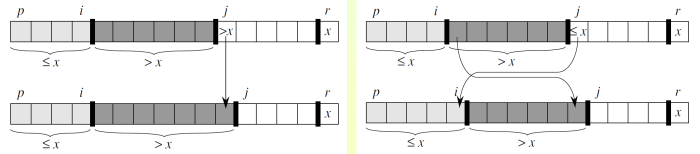


  For any array index k,
  1. If $p \leq k \leq i$, then $\mathrm{A}[k] \leq x$.
  2. If $i + 1 \leq k \leq j - 1$, then $\mathrm{A}[k] > x$.
  3. If $k = r$, then $\mathrm{A}[k] = x$.

  The indices between j and $r - 1$ are not covered by any of the three cases, and these values have no particular relationship to the pivot x.
- We need to show that this loop invariant (a) is true prior to the first iteration, (b) is maintained by each iteration, (c) provides a helps in showing correctness when the loop terminates.

Initialization and Maintenance

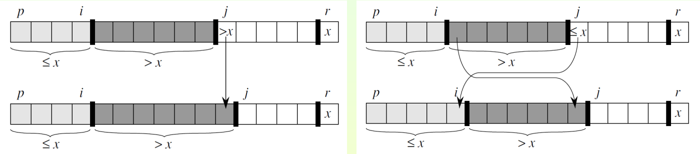
- Initialization: Prior to the first iteration of the loop, $i = p - 1$, and $j = p$. There are no values between p and i, and no values between $i + 1$ and $j - 1$, so the first two conditions of the loop invariant are trivially satisfied. The assignment in Line 1 satisfies the third condition.
- Maintenance: There are two cases to consider, depending on the outcome of the test in Line 4:



  - If $A[j] > x$, the only action in the loop is to increment j. After j is incremented, Condition 2 holds for $A[j-1]$ and all other entries remain unchanged.

Maintenance (Cont'd)

- Maintenance: There are two cases to consider, depending on the outcome of the test in Line 4:



  - If A[j] $\leq x$, i is incremented, A[i] and $A[j]$ are swapped, and then j is incremented. Because of the swap, we now have that $A[i] \leq x$, and Condition 1 is satisfied. Similarly, we also have that A[$j-1$] $> x$, since the item that was swapped into A[$j-1$] is, by the loop invariant, greater than x.

Termination and Running Time

- Termination: At termination, $j = r$. Therefore, every entry in the array is in one of the three sets described by the invariant, and we have partitioned the values in the array into three sets:
  - those less than or equal to x;
  - those greater than x;
  - a singleton set containing x.
- The final two lines of PARTITION move the pivot element into its place in the middle of the array by swapping it with the leftmost element that is greater than x.

  The output of PARTITION now satisfies the specifications given for the divide step.
- The running time of PARTITION on the subarray $A[p \ldots r]$ is $\Theta(n)$, where $n = r - p + 1$.

Subsection 2

Performance of Quicksort

Worst-Case Partitioning

- The worst-case behavior for quicksort occurs when the partitioning produces one subproblem with $n-1$ and one with 0 elements.
  Let us assume that this unbalanced partitioning arises in each recursive call.
  - The partitioning costs $\Theta(n)$ time.
  - A recursive call on an array of size 0 takes $T(0)=\Theta(1)$.

  Thus, the recurrence for the running time is
  $$T(n)=T(n-1)+T(0)+\Theta(n)=T(n-1)+\Theta(n).$$
- Using the substitution method we can prove that the recurrence $T(n)=T(n-1)+\Theta(n)$ has the solution
  $$T(n)=\Theta(n^2).$$
- Thus, if the partitioning is maximally unbalanced at every recursive level of the algorithm, the running time is $\Theta(n^2)$.

Best-Case Partitioning

- In the most even possible split, PARTITION produces two subproblems, each of size no more than $\frac{n}{2}$, since one is of size $\lfloor \frac{n}{2} \rfloor$ and one of size $\lceil \frac{n}{2} \rceil - 1$.

  In this case, quicksort runs much faster.

  The recurrence for the running time is then

  $$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n),$$

  where we tolerate the sloppiness from ignoring the floor and ceiling and from subtracting 1.
- This recurrence has the solution

  $$T(n) = n \log n.$$
- By equally balancing the two sides of the partition at every level of the recursion, we get an asymptotically faster algorithm.

Balanced Partitioning

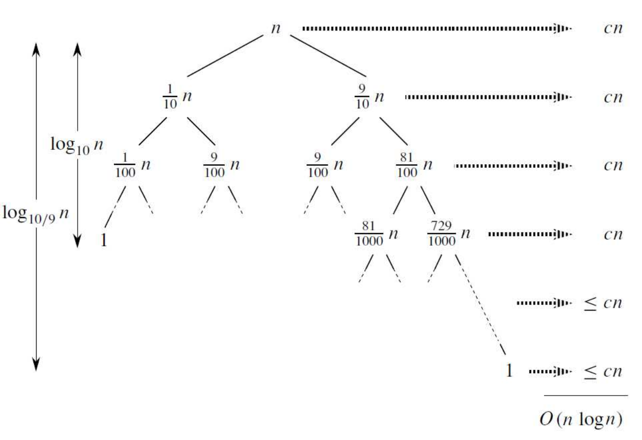
- The average-case running time of quicksort is much closer to the best case than to the worst case.
- The key to understand why is to understand how the balance of the partitioning is reflected in the recurrence for the running time.
- Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which seems quite unbalanced. We then obtain the recurrence

$$T(n) = T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$$

  on the running time of quicksort, where we have explicitly included the constant c hidden in the $\Theta(n)$ term.

  Every level of the tree has cost cn, until a boundary condition is reached at depth $\log_{10} n = \Theta(\log n)$. Then the levels have cost at most cn. The recursion terminates at depth $\log_{10/9} n = \Theta(\log n)$. The total cost of quicksort is therefore $O(n \log n)$.

Balanced Partitioning (Cont'd)



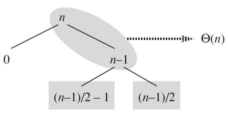
- Even a 99-to-1 split yields an $O(n \log n)$ running time, since any split of constant proportionality yields a recursion tree of depth $\Theta(\log n)$, where the cost at each level is $O(n)$.

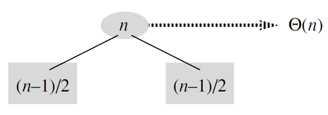
Intuition for the Average Case

- To develop a clear notion of the average case for quicksort, we must make an assumption about how frequently we expect to encounter the various inputs.
- The behavior of quicksort is determined by the relative ordering of the values in the array and not by the particular values.
- We will assume for now that all permutations of the input numbers are equally likely.
- When we run quicksort on a random input array, it is unlikely that the partitioning always happens in the same way at every level: We expect that some of the splits will be reasonably well balanced ("good") and that some will be fairly unbalanced ("bad").
- In a recursion tree for an average-case execution of PARTITION, the good and bad splits are distributed randomly throughout the tree.

Intuition for the Average Case (Cont'd)

- Suppose that the good and bad splits alternate levels, the good splits are best-case splits and the bad splits are worst-case splits:





- At the root of the tree, the cost is n for partitioning, and the subarrays produced have sizes $n-1$ and 0: the worst case.
- At the next level, the subarray of size $n-1$ is best-case partitioned into subarrays of size $\frac{n-1}{2}-1$ and $\frac{n-1}{2}$.
- If the boundary-condition cost is 1 for the subarray of size 0, the combination produces three subarrays of sizes 0, $\frac{n-1}{2}-1$ and $\frac{n-1}{2}$ at a combined partitioning cost of $\Theta(n)+\Theta(n-1)=\Theta(n)$.
- This situation is no worse than a single level of balanced partitioning that produces two subarrays of size $\frac{n-1}{2}$, at a cost of $\Theta(n)$!

Subsection 3

A Randomized Version of Quicksort

Introducing Random Sampling in Quicksort

- We employ a randomization technique, called **random sampling**, to analyze a randomized version of quicksort.
- Instead of always using A[r] as the pivot, we will select a randomly chosen element from the subarray A[$p \ldots r$].
- We do so by first exchanging element A[r] with an element chosen at random from A[$p \ldots r$].
- By randomly sampling the range $p, \ldots, r$, we ensure that the pivot element $x = A[r]$ is equally likely to be any of the $r - p + 1$ elements in the subarray.
- Because we randomly choose the pivot element, we expect the split of the input array to be reasonably well balanced on average.

RandomizedQuickSort

- The changes to Partition and QuickSort are small.
  - In the new partition procedure, we simply implement the swap before actually partitioning:

RandomizedPartition(A, p, r)

```
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

  - The new quicksort calls RandomizedPartition in place of Partition:

RandomizedQuickSort(A, p, r)

```
if p < r
    q = RANDMIZEDPARTITION(A, p, r)
    RANDOMIZEDQUICKSORT(A, p, q − 1)
    RANDOMIZEDQUICKSORT(A, q + 1, r)
```