Quick Sort

Mina Gabriel

Harrisburg University

March 7, 2024

Introduction and Advantages of Quicksort

Definition

Quicksort is a fast sorting algorithm that employs a divide-and-conquer strategy to organize elements around a pivot, achieving an average-case performance of $O(n \log n)$. It is favored for its efficiency and in-place sorting capability, despite a worst-case runtime of $O(n^2)$, Unlike merge sort, it also has the advantage of sorting in place.

Algorithm, Randomized Quicksort, and Analysis

- Quicksort algorithm details and partitioning subroutine.
- Randomized Quicksort avoids worst-case behavior, maintaining expected $O(n \log n)$ time for distinct elements.
- Analysis shows worst-case: $O(n^2)$, expected: $O(n \log n)$ for distinct elements, confirming efficiency.

The Quicksort Process

- Applies **divide-and-conquer** strategy for sorting an array $A[p..r]$.
- **Divide**: Partition $A[p..r]$ into $A[p..q-1]$ (less than pivot) and $A[q+1..r]$ (greater than pivot) around pivot $A[q]$.
- **Conquer**: Recursively apply quicksort to the subarrays.
- **Combine**: No work needed as subarrays are sorted during partitioning.

Quicksort Algorithm Implementation

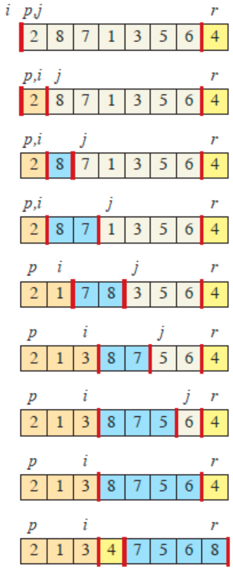
- Start with `QUICKSORT(A, 1, n)` to sort an entire array $A[1..n]$.
- Partitioning places the pivot in its correct sorted position at $A[q]$.
- Recursively sort subarrays $A[p..q-1]$ and $A[q+1..r]$.

Quicksort Pseudocode

```
QUICKSORT(A, p, r)
  if p < r
      q = PARTITION(A, p, r)
      QUICKSORT(A, p, q - 1) // Sort left side
      QUICKSORT(A, q + 1, r) // Sort right side
```

Partitioning the array

```
PARTITION(A, p, r)
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] <= x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



The PARTITION Procedure Regions

The four regions maintained by the procedure PARTITION on a subarray $A[p:r]$ are defined as follows:

- The tan values in $A[p:i]$ are all less than or equal to x.
- The blue values in $A[i+1:j-1]$ are all greater than x.
- The white values in $A[j:r-1]$ have unknown relationships to x.
- $A[r]=x$.

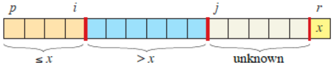


Figure: Visualization of PARTITION regions

Example

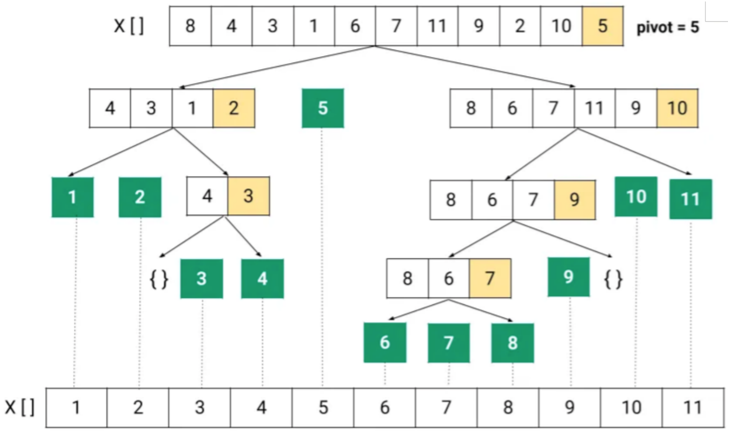


Figure: Quick Sort Example

PARTITION Procedure: One Iteration

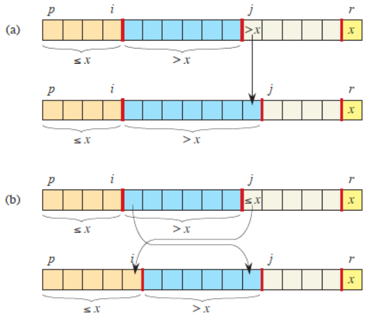


Figure: The two cases in the PARTITION procedure.

The two cases for one iteration of procedure PARTITION:

- (a) If $A[j] > x$, the only action is to increment j, which maintains the loop invariant.
- (b) If $A[j] \leq x$, index i is incremented, $A[i]$ and $A[j]$ are swapped, and then j is incremented. Again, the loop invariant is maintained.

Worst-Case Scenario of Quicksort

- The worst-case occurs when partitioning results in one subproblem with $n-1$ elements and one with 0.
- This unbalanced partitioning leads to the most inefficient sorting, where the algorithm does not evenly split the array.
- Partitioning Cost: $O(n)$ for each step, as it goes through all elements.
- Recursive Calls: The recurrence relation becomes $T(n) = T(n-1) + O(n)$, leading to an arithmetic series in total cost.

Quicksort Worst-Case Example

- Example list: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
- Choosing the last element as pivot (10), the partitioning yields:
  - Left subarray: [1, 2, 3, 4, 5, 6, 7, 8, 9]
  - Pivot: 10
  - Right subarray: [] (empty)
- The empty right subarray leads to immediate return on recursive call.
- This process repeats, causing a series of one-sided partitions and illustrating the worst-case scenario.
- Result: Quicksort's time complexity becomes $O(n^2)$ in such cases.

Analysis of Recursion $T(n) = T(n-1) + O(n)$

- At the top level (level 0), we have a problem of size n: $T(n)$.
- The work done at this level is $O(n)$.
- The first recursive call (level 1): $T(n) = T(n-1) + O(n)$.
- The second recursive call (level 2): $T(n-1) = T(n-2) + O(n-1)$.
- Continuing this pattern, we accumulate the following:

$$\begin{aligned}
T(n) &= T(n-1) + O(n) \\
&= T(n-2) + O(n-1) + O(n) \\
&= T(n-3) + O(n-2) + O(n-1) + O(n) \\
&\vdots \\
&= T(1) + O(2) + O(3) + \ldots + O(n-1) + O(n) \\
&= T(1) + \sum_{i=2}^{n} O(i) \\
&= O(1) + O(n^2) \text{ (since the summation is an arithmetic series)}
\end{aligned}$$

Implications and Comparison

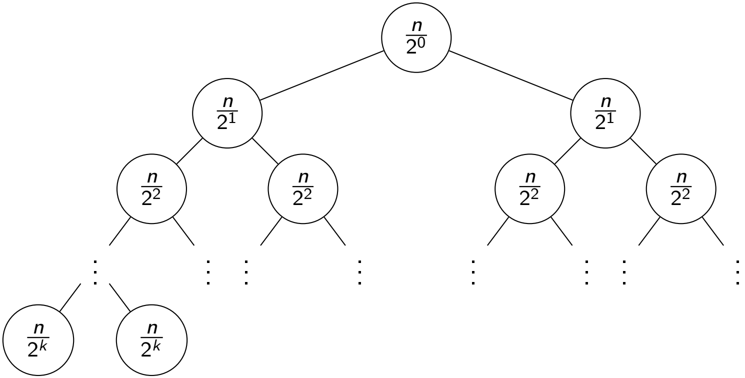
- The total running time in this scenario evaluates to $O(n^2)$, similar to the worst case of insertion sort.
- Notably, when the array is already sorted (a quicksort worst-case), insertion sort runs in $O(n)$, outperforming quicksort.
- This highlights that while quicksort is efficient on average, its performance can significantly worsen with consistently unbalanced partitions.

Best Case Time Complexity of Quicksort

- In the best case, the pivot is the median element in each call of the partition algorithm, resulting in balanced partitions.
- At the top level (level 0), we have a problem of size n: $T(n)$.
- The work done at this level is $O(n)$.
- Each recursive call results in two subproblems of size $\frac{n}{2}$: $T\left(\frac{n}{2}\right)$.

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + O(n) \\
&= 2\left(2T\left(\frac{n}{4}\right) + O\left(\frac{n}{2}\right)\right) + O(n) \\
&= 4T\left(\frac{n}{4}\right) + 2O\left(\frac{n}{2}\right) + O(n) \\
&\vdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + kO(n) \quad (\text{until } n = 2^k) \\
&= 2^{\log_2(n)} T(1) + \log_2(n) O(n) \\
&= nT(1) + O(n \log n) \\
&= O(n \log n) \quad (\text{since } T(1) \text{ is constant})
\end{aligned}$$

Binary Tree Representation



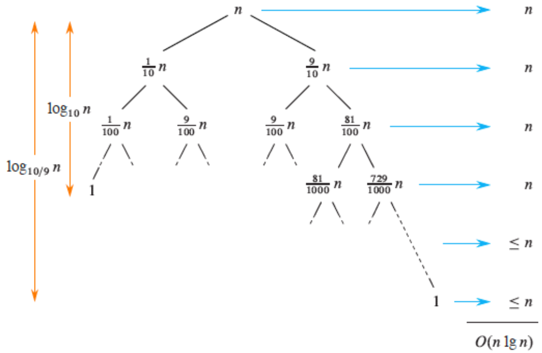
$$k = \log_2 n \quad \text{given} \quad \frac{n}{2^k} = 1$$

Average-Case Running Time of Quicksort

The average-case running time of quicksort is much closer to the best case than to the worst case. By appreciating how the balance of the partitioning affects the recurrence describing the running time, we can gain an understanding of why.
Suppose, for example, that the partitioning algorithm always produces a 9-to-1 proportional split, which at first blush seems quite unbalanced. We then obtain the recurrence:

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

PARTITION Procedure: 9-to-1 split



Given $\frac{1}{10^{k_1}} n = 1$, we have $10^{k_1} = n$, and thus, $\log_{10} n = k_1$.

For $n = 100$, we find $k_1 = 2$.

Similarly, given $\left(\frac{9}{10}\right)^{k_2} n = 1$, it follows that $n = \left(\frac{10}{9}\right)^{k_2}$, and $\log_{\frac{10}{9}} n = k_2$.

For $n = 100$, we determine $k_2 \approx 43.709$.

Hoare partition: The original partitioning algorithm.

```
HOARE-PARTITION(A, p, r)
  x = A[p]
  i = p - 1
  j = r + 1
  while true
      repeat
          j = j - 1
      until A[j] <= x
      repeat
          i = i + 1
      until A[i] >= x
      if i < j
         then exchange A[i] with A[j]
      else return j

QUICKSORT(A, p, r)
  if p < r
      q = HOARE-PARTITION(A, p, r)
      QUICKSORT(A, p, q)
      QUICKSORT(A, q + 1, r)
```