CS 2500: Algorithms

Lecture 13: Quick Sort

Shubham Chatterjee

Missouri University of Science and Technology, Department of Computer Science

October 1, 2024

Quicksort Algorithm

- A sorting algorithm developed by Tony Hoare in 1959.
- Uses a divide-and-conquer approach to sort elements.
- In merge sort, the list `a[1:n]` was divided at its midpoint into subarrays which were independently sorted and later merged.
- In quicksort, the division into two subarrays is made so that the sorted subarrays do not need to be merged later.

Divide and Conquer in Quicksort

- **Divide**: Partition the array into two subarrays around a pivot element.
- **Conquer**: Recursively apply quicksort to the subarrays.
- **Combine**: No need to merge, as the array elements are sorted in place.

Partitioning in Quicksort

- Choose a pivot element and rearrange the array such that:
- Elements less than the pivot are on the left.
- Elements greater than the pivot are on the right.
- Pivot is then in its correct position.

Partitioning in Quicksort

- **Partitioning problem:** Given an array $a[1:n]$ and a pivot x, partition the array such that:

$$\forall i, 1 \leq i \leq m \Rightarrow a[i] \leq x$$

$$\forall j, m+1 \leq j \leq n \Rightarrow a[j] > x$$

where m is the partition index with $1 \leq m \leq n$.

Hoare Partition

- Developed by C.A.R. Hoare as part of the original quicksort.
- Uses two indices that move towards each other to swap elements around the pivot.
- Stops when indices cross, leaving the pivot element between the partitions.

Hoare Partition: Algorithm

```
Algorithm Partition(a, m, p)
// Within a[m], a[m + 1], ..., a[p − 1] the elements are
// rearranged in such a manner that if initially t = a[m],
// then after completion a[q] = t for some q between m
// and p − 1, a[k] ≤ t for m ≤ k < q, and a[k] ≥ t
// for q < k < p. q is returned. Set a[p] = ∞.
{
    v := a[m]; i := m; j := p;
    repeat
    {
        repeat
            i := i + 1;
        until (a[i] ≥ v);

        repeat
            j := j − 1;
        until (a[j] ≤ v);

        if (i < j) then Interchange(a, i, j);

    } until (i ≥ j);

    a[m] := a[j]; a[j] := v; return j;
}

Algorithm Interchange(a, i, j)
// Exchange a[i] with a[j].
{
    p := a[i];
    a[i] := a[j]; a[j] := p;
}
```

Hoare Partition

- Algorithm Partition accomplishes an in-place partitioning of the elements of a[m:p].
- It is assumed that a[p] $\geq$ a[n] and that a[m] is the partitioning element.
- If $m = 1$ and $p = 1 = n$, then a[n+1] must be defined and must be greater than or equal to all elements in a[1:n].
- The assumption that a[n] is the partition element is merely for convenience; other choices for the partitioning element than the first item in the set are better in practice.
- The function Interchange(a,i,j) exchanges a[i] with a[j].

Hoare Partition: Example

(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i	p
65	70	75	80	85	60	55	50	45	$+\infty$	2	9
65	45	75	80	85	60	55	50	70	$+\infty$	3	8
65	45	50	80	85	60	55	75	70	$+\infty$	4	7
65	45	50	55	85	60	80	75	70	$+\infty$	5	6
65	45	50	55	60	85	80	75	70	$+\infty$	6	5
60	45	50	55	65	85	80	75	70	$+\infty$		

Quick Sort

```
Algorithm QuickSort(p, q)
// Sorts the elements a[p], ..., a[q] which reside in the global
// array a[1 : n] into ascending order; a[n + 1] is considered to
// be defined and must be ≥ all the elements in a[1 : n].
{
    if (p < q) then  // If there are more than one element
    {
        // divide P into two subproblems.
            j := Partition(a, p, q + 1);
                // j is the position of the partitioning element.
        // Solve the subproblems.
            QuickSort(p, j - 1);
            QuickSort(j + 1, q);
        // There is no need for combining solutions.
    }
}
```

Quick Sort: Analysis

- Quick Sort is a divide-and-conquer algorithm.
- The time complexity is determined by the number of element comparisons, $C(n)$.
- The analysis assumes:
  - Elements are distinct.
  - Each element has an equal probability of being the pivot during partition.

Quick Sort: Worst-Case Analysis

- In each recursive call to `Partition(a,m,p)`, the pivot divides the array.
- Worst-case occurs when the pivot consistently partitions the array in a highly unbalanced way (e.g., smallest or largest element).

Quick Sort: Worst-Case Analysis

Recursive Structure and Comparisons at Each Level

- **Level 1:**
  - Start by partitioning the entire array of size n.
  - One Partition call is made.
  - **Comparisons at this level:** n
- **Level 2:**
  - After the first partition, we have two subarrays, but in the worst case, one subarray is empty, and the other has $n - 1$ elements.
  - Two Partition calls are made, but one subarray contains no elements.
  - **Comparisons at this level:** $n - 1$.
- **This process continues:**
  - At each level, the size of the subarray decreases by 1.
  - Comparisons continue until the subarray size reduces to 2, at which point 1 comparison is made.

Quick Sort: Worst-Case Analysis

$$
\begin{aligned}
C_w(n) &= n + (n-1) + (n-2) + (n-3) + \cdots + 2 \\
&= \sum_{k=2}^{n} k \\
&= \frac{n(n+1)}{2} - 1 \\
&= \frac{n^2 + n - 2}{2} \\
&= O(n^2)
\end{aligned}
$$

Quick Sort: Average-Case Analysis

- Let $C_A(n)$ be the average number of comparisons made by Quick Sort to sort an array of size n.
- Assumptions:
  - **Distinct Elements:** All n elements to be sorted are distinct.
  - **Uniform Pivot Selection:** The pivot element $v = a[m]$ in a call to `Partition(a,m,p)` has an equal probability of being any of the $p - m$ elements in the subarray $a[m \ldots p-1]$.

Quick Sort: Average-Case Analysis

Aim:

- Find a recurrence equation for $C_A(n)$.
- Solve the recurrence equation obtained above to determine the order of growth.

Quick Sort: Average-Case Analysis

Question: How many comparisons does Quick Sort make in the first partitioning step?

Answer: Quick Sort makes $n + 1$ comparisons in the first partitioning step.

Quick Sort: Average-Case Analysis

Number of comparisons of pivot with non-pivot elements:

- Total Elements to Partition: $n = p - m$ (the size of the subarray $a[m \dots p-1]$).
- Pivot Element: $v = a[m]$.
- Non-Pivot Elements: $n - 1$ (since the pivot is one element).
- Each Non-Pivot Element is compared with the pivot at least once.
- Total Comparisons: $n - 1$.

Quick Sort: Average-Case Analysis

When the First Inner Loop Ends:

- After passing all elements less than v, the loop increments i one more time. This increment leads to a comparison where $a[i] \geq v$.
- This is the **first extra comparison**, which evaluates to true, causing the loop to exit.

When the Second Inner Loop Ends:

- After passing all elements greater than v, the loop decrements j one more time. This decrement leads to a comparison where $a[j] \leq v$.
- This is the **second extra comparison**, which evaluates to true, causing the loop to exit.

- Total number of comparisons in the first partition step $= (n-1)+1+1 = n+1$

Quick Sort: Average-Case Analysis

Recurrence Relation for Average Comparisons:

- After partitioning, the array is divided into two subarrays:
  - Left Subarray: Elements less than the pivot.
  - Right Subarray: Elements greater than the pivot.
- Size of Subarrays:
  - Left Subarray: $k - 1$ elements.
  - Right Subarray: $n - k$ elements.
- k is the position of the pivot in the sorted array (i.e., it is the k-th smallest element).
- Probability of Pivot Position: Since each element is equally likely to be chosen as the pivot, the probability that the pivot is the k-th smallest element is $\frac{1}{n}$ for $k = 1, 2, \ldots, n$

Quick Sort: Average-Case Analysis

Recurrence Relation for Average Comparisons

$$C_A(n) = (n+1) + \frac{1}{n}\sum_{k=1}^{n}[C_A(k-1) + C_A(n-k)]$$

- $n+1$: The comparisons made in the first partitioning step.
- $\frac{1}{n}$: Probability of pivot being the k-th smallest element.
- $C_A(k-1)$: Expected comparisons to sort the left subarray.
- $C_A(n-k)$: Expected comparisons to sort the right subarray.

Quick Sort: Average-Case Analysis

Solving the Recurrence:

- Multiply both sides of the recurrence by n to simplify:

$$nC_A(n) = n(n+1) + 2\left[C_A(0) + C_A(1) + \cdots + C_A(n-1)\right]$$

- Replace n with $n-1$:

$$(n-1)C_A(n-1) = (n-1)n + 2\left[C_A(0) + C_A(1) + \cdots + C_A(n-2)\right]$$

- Subtract the second equation from the first:

$$nC_A(n) - (n-1)C_A(n-1) = 2n + 2C_A(n-1)$$

Quick Sort: Average-Case Analysis

Solving the Recurrence:

- Simplifying further:

$$
\begin{aligned}
nC_A(n) - (n-1)C_A(n-1) &= 2n + 2C_A(n-1) \\
nC_A(n) - (n-1)C_A(n-1) - 2C_A(n-1) &= 2n \\
nC_A(n) - (n+1)C_A(n-1) &= 2n
\end{aligned}
$$

- Dividing both sides of the equation by $n(n+1)$:

$$\frac{C_A(n)}{(n+1)} = \frac{C_A(n-1)}{n} + \frac{2}{(n+1)}$$

Solving the Recurrence: Substitution Method

- Substituting recursively, we get:

$$\frac{C_A(n-1)}{(n)} = \frac{C_A(n-2)}{(n-1)} + \frac{2}{n}$$
$$\frac{C_A(n-2)}{(n-1)} = \frac{C_A(n-3)}{(n-2)} + \frac{2}{n-1}$$

- Continue this process until reaching $C_A(1)$.
- After $n-1$ steps:

$$\frac{C_A(n)}{(n+1)} = \frac{C_A(1)}{(2)} + 2\left(\frac{1}{n+1} + \frac{1}{n} + \frac{1}{n-1} + \frac{1}{3}\right)$$
$$= \frac{C_A(1)}{(2)} + 2\sum_{k=3}^{n+1} \frac{1}{k}$$

  - Each substitution adds a term $\frac{2}{k+1}$ to the sum.
  - The sum accumulates terms of the form $\frac{2}{k}$ starting from $k = n+1$ down to $k = 3$.

Quick Sort: Average-Case Analysis

Approximation Using Integral

- We need to find the sum of the series: $\sum_{k=3}^{n+1} \frac{1}{k}$.
    - This is a partial sum of the harmonic series, excluding the first two terms ($k = 1$ and $k = 2$).
- Upper bound using integration:
    - The harmonic series can be approximated by the natural logarithm:

$$\sum_{k=3}^{n+1} \frac{1}{k} \leq \int_{2}^{n+1} \frac{1}{x} dx = \log_e(n+1) - \log_e 2$$

- The integral of $\frac{1}{x}$ from $x = a$ to $x = b$ equals $\log_e a - \log_e b$.
- The integral provides an upper bound for the sum.

Quick Sort: Average-Case Analysis

The recurrence equation is:

$$\frac{C_A(n)}{(n+1)} = \frac{C_A(1)}{(2)} + 2\sum_{k=3}^{n+1}\frac{1}{k}$$

Multiply both sides by $n+1$:

$$C_A(n) = (n+1)\left(\frac{C_A(1)}{(2)} + 2\sum_{k=3}^{n+1}\frac{1}{k}\right)$$

Since $C_A(1) = 0$ (sorting one element requires zero comparisons), we have:

$$\begin{aligned} C_A(n) &= (n+1)(0 + 2(\log_e(n+1) - \log_e 2)) \\ &= 2(n+1)(\log_e(n+1) - \log_e 2) \\ &= 2(n+1)\log_e\left(\frac{n+1}{2}\right) \end{aligned}$$

Quick Sort: Average-Case Analysis

From the previous approximation, we conclude:

- The dominant term is $n \log_e n$.
- Therefore, $C_A(n) = O(n \log n)$.
- **Note:** We can use base 2 logarithms for simplicity in computer science contexts.