

Algorithms and Data Structures: Average-Case Analysis of Quicksort

11th October, 2011

Quicksort

Divide-and-Conquer algorithm for sorting an array. It works as follows:

1. If the input array has less than two elements, nothing to do ...
Otherwise, do the following **partitioning** subroutine: Pick a particular key called the **pivot** and divide the array into two subarrays as follows:



2. Sort the two subarrays recursively.

Quicksort Algorithm

Algorithm QUICKSORT(A, p, r)

1. **if** $p < r$ **then**
2. $q \leftarrow$ PARTITION(A, p, r)
3. QUICKSORT($A, p, q - 1$)
4. QUICKSORT($A, q + 1, r$)

Partitioning

Algorithm PARTITION(A, p, r)

1. $pivot \leftarrow A[r]$
2. $i \leftarrow p - 1$
3. **for** $j \leftarrow p$ **to** $r - 1$ **do**
4. **if** $A[j] \leq pivot$ **then**
5. $i \leftarrow i + 1$
6. exchange $A[i], A[j]$
7. exchange $A[i + 1], A[r]$
8. **return** $i + 1$

Analysis of Quicksort

- ▶ The **size** of an instance (A, p, r) is $n = r - p + 1$.
- ▶ Basic operations for sorting are **comparisons of keys**. We let

$$C(n)$$

be the *worst-case number of key-comparisons* performed by $\text{QUICKSORT}(A, p, r)$. We shall try to determine $C(n)$ as precisely as possible.

- ▶ It is easy to verify that the worst-case running time $T(n)$ of $\text{QUICKSORT}(A, p, r)$ is $\Theta(C(n))$ if a single comparison requires time $\Theta(1)$.
(ie, for QUICKSORT , comparisons *dominate* the running time).
In any case,

$$T(n) = \Theta(C(n) \cdot \text{cost per comparison}).$$

Analysis of PARTITION

- ▶ $\text{PARTITION}(A, p, r)$ does *exactly* $n - 1$ comparisons for every input of size n .
- ▶ There is also an initial comparison $p < r$ done on line 1 of `QUICKSORT`.
- ▶ Therefore in our theoretical analysis, we can assume, that apart from recursive calls, we do n comparisons on every input.

Worst-case Analysis of QUICKSORT

- ▶ We get the following recurrence for $C(n)$:

$$C(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \max_{1 \leq k \leq n} (C(k-1) + C(n-k)) + n & \text{if } n \geq 2 \end{cases}$$

- ▶ Intuitively, worst-case seems to be $k = 1$ or $k = n$, i.e., everything falls on one side of the partition. This happens, e.g., if the array is sorted.

Worst-Case Analysis (cont'd)

- ▶ *Lower Bound:* $C(n) \geq \frac{1}{2}n(n+1) = \Omega(n^2)$.

Proof:

$$\begin{aligned}C(n) &\geq C(n-1) + n \\ &\geq C(n-2) + n + n - 1 \\ &\quad \vdots \\ &\geq \sum_{i=0}^{n-1} i + 1 = \sum_{i=1}^n i = \frac{1}{2}n(n+1).\end{aligned}$$

- ▶ *Upper Bound:* $C(n) \leq O(n^2)$.
See [CLRS] p.155 (or [CLR] p.163) for a proof.
- ▶ Thus

$$C(n) = \Theta(n^2).$$

Best-Case Analysis

- ▶ $B(n)$ = number of comparisons done by QUICKSORT in the best case.
- ▶ *Recurrence:*

$$B(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \min_{1 \leq k \leq n} (B(k-1) + B(n-k)) + n & \text{if } n \geq 2 \end{cases}$$

- ▶ Intuitively, the best case is if the array is always partitioned into two parts of the same size. This means

$$B(n) \approx 2B(n/2) + \Theta(n),$$

which implies $B(n) = \Theta(n \lg(n))$.

Average-Case Analysis

- ▶ $A(n)$ = number of comparisons done by QUICK-SORT on average if all input arrays of size n are considered equally likely.
- ▶ **Intuition:** The average case is closer to the best case than to the worst case, because only **repeatedly very unbalanced** partitions lead to the worst case.
- ▶ *Recurrence:*

$$A(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ \sum_{k=1}^n \frac{1}{n} (A(k-1) + A(n-k)) + n & \text{if } n \geq 2 \end{cases}$$

- ▶ *Solution:*

$$A(n) \approx 2n \ln(n).$$

Average Case Analysis in Detail

We shall prove that for all $n \geq 1$ we have

$$A(n) \leq 2(\ln(n) + 1/3)(n + 1). \quad (\star)$$

Clearly, (\star) holds for $n = 1$. So assume that $n \geq 2$. We have

$$\begin{aligned} A(n) &= \sum_{1 \leq k \leq n} \frac{1}{n} (A(k-1) + A(n-k)) + n \\ &= \frac{2}{n} \sum_{k=0}^{n-1} A(k) + n. \end{aligned}$$

Thus

$$nA(n) = 2 \sum_{k=0}^{n-1} A(k) + n^2. \quad (\star\star)$$

Note that $(\star\star)$ does not hold for $n = 1$.

(For $n = 1$ the l.h.s. $1A(1) = 1$, but the r.h.s. is 3.)

Average Case Analysis in Detail (cont'd)

Applying (**) to $(n-1)$ for $n \geq 3$, we obtain

$$(n-1)A(n-1) = 2 \sum_{k=0}^{n-2} A(k) + (n-1)^2.$$

Subtracting this equation from (**) (when $n \geq 3$)

$$nA(n) - (n-1)A(n-1) = 2A(n-1) + n^2 - (n-1)^2,$$

thus

$$nA(n) = (n+1)A(n-1) + 2n - 1,$$

and therefore

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

We now apply unfold-and-sum to this recurrence (stopping at $n=2$):

$$\begin{aligned} \frac{A(n)}{n+1} &\leq \frac{A(n-1)}{n} + \frac{2}{n} \\ &\vdots \end{aligned}$$

Average Case Analysis in Detail (cont'd)

$$\begin{aligned}\frac{A(n)}{n+1} &\leq \frac{A(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n-1} \\ &\vdots \\ &\leq \frac{A(2)}{3} + 2 \sum_{k=3}^n \frac{1}{k} \\ &= \frac{4}{3} + 2 \sum_{k=3}^n \frac{1}{k} = 1/3 + 2 \sum_{k=2}^n \frac{1}{k}.\end{aligned}$$

It is easy to verify this result by induction. Thus

$$\frac{A(n)}{n+1} \leq 1/3 + 2 \sum_{k=2}^n \frac{1}{k} = 1/3 + 2 \sum_{k=1}^{n-1} \frac{1}{k+1} \leq 1/3 + 2 \int_1^n \frac{1}{x} = 1/3 + 2 \ln(n).$$

Multiplying by $(n+1)$ completes the proof of (\star) .

Improvements

- ▶ Use insertion sort for small arrays.
- ▶ Iterative implementation.

Main Question

Is there a way to avoid the bad worst-case performance, and in particular the bad performance on sorted (or almost sorted) arrays?

Different strategies for choosing the pivot-element help (in practice).

Median-of-Three Partitioning

Idea: Use the median of the first, middle, and last key as the pivot.

Algorithm M3PARTITION(A, p, r)

1. exchange $A[(p + r)/2]$, $A[r - 1]$
2. **if** $A[p] > A[r - 1]$ **then** exchange $A[p]$, $A[r - 1]$
3. **if** $A[p] > A[r]$ **then** exchange $A[p]$, $A[r]$
4. **if** $A[r - 1] > A[r]$ **then** exchange $A[r - 1]$, $A[r]$
5. PARTITION($A, p + 1, r - 1$)

Note that M3PARTITION(A, p, r) only requires 1 more comparison than PARTITION(A, p, r)

Median-of-Three Partitioning (cont'd)

Algorithm M3QUICKSORT(A, p, r)

1. **if** $p < r$ **then**
2. $q \leftarrow$ M3PARTITION(A, p, r)
3. M3QUICKSORT($A, p, q - 1$)
4. M3QUICKSORT($A, q + 1, r$)

It can be shown that the worst-case running time of M3QUICKSORT is still $\Theta(n^2)$, but at least in the case of an almost sorted array (and in most other cases that are relevant in practice) it is very efficient.

Randomized Quicksort

Idea: Use key of random element as the pivot.

Algorithm RPARTITION(A, p, r)

1. $k \leftarrow \text{RANDOM}(p, r)$ \triangleright choose k randomly from $\{p, \dots, r\}$
2. exchange $A[k], A[r]$
3. PARTITION(A, p, r)

Algorithm RANDOMIZED QUICKSORT(A, p, r)

1. **if** $p < r$ **then**
2. $q \leftarrow \text{RPARTITION}(A, p, r)$
3. RANDOMIZED QUICKSORT($A, p, q - 1$)
4. RANDOMIZED QUICKSORT($A, q + 1, r$)

Analysis of Randomized Quicksort

The running time of RANDOMIZED QUICKSORT on an input of size n is a **random variable**.

An analysis similar to the average case analysis of QUICKSORT shows:

Theorem

*For all inputs (A, p, r) , the **expected number of comparisons** performed during a run of RANDOMIZED QUICKSORT on input (A, p, r) , is at most $2(\ln(n) + 1/3)(n + 1)$, where $n = r - p + 1$.*

Corollary

*Thus the **expected running time** of RANDOMIZED QUICKSORT on any input of size n is $\Theta(n \lg(n))$.*

Reading Assignment

Sections 7.3, 7.4 of [CLRS] (edition 2 or 3)

Problems

1. Convince yourself that PARTITION works correctly by working a few examples, or (better) try to prove that it works correctly. Find an example that shows what may go wrong if we omit the test $j > p$ in Line 6. Is this test also needed when we call PARTITION from M3PARTITION?
2. In our proof of the Average-running time $A(n)$, we can think of the input as being some permutation of $(1, \dots, n)$, and assume all permutations are equally likely. Why does this explain the $1/n$ factor in the recurrence on slide 10?
3. Exercise 7.3-2 of [CLRS] (edition 2 or 3) or 8.3-2 of [CLR].