Quicksort

CS624 — Analysis of Algorithms

February 8, 2024

Sorting, Revisited

We have seen several sorting algorithms so far:

- Insertion Sort (incremental)
- Merge Sort (divide and conquer, all work in *combine* step)
- Heap Sort

Is there a divide and conquer algorithm for sorting
that does all of the work in the *divide* step instead?

Designing Quicksort

Is there a divide and conquer algorithm for sorting that does all of the work in the *divide* step instead?

- Let's assume there are two sorting sub-problems.
- If all the work is in *divide*, then *combine* must be trivial, such as just concatenating sorted sub-arrays.
- For concatenation to work, one sub-array must be be ordered entirely before the other sub-array.
- So our *divide* step must be to partition the original array such that every element of the first part is $\leq$ every element of the second part.

Quicksort

Algorithm 1 $\text{Quicksort}(A, p, r)$

Ensure: $A[p \mathinner{..} r]$ is sorted

```
if p < r then
    q ← Partition(A, p, r)
    Quicksort(A, p, q − 1)
    Quicksort(A, q + 1, r)
end if
```

Quicksort

The Partition procedure picks an element called the "pivot" and breaks the array into three parts: $\leq$, $=$, $>$ the pivot.

After $\mathrm{Partition}$ has been called the following are true:

1. $p \leq q \leq r$.
2. The number $A[q]$, the pivot, is in its final position.
   It will never be moved again.
3. If $i < q$, then $A[i] \leq A[q]$,
   and if $i > q$, then $A[i] > A[q]$.

Partition

Algorithm 2 $\text{Partition}(A, p, r)$

Ensure: Let $q =$ **result.** $A[p \mathinner{..} q-1] \le A[q] < A[q+1 \mathinner{..} r]$, $p \le q \le r$

$x \leftarrow A[r]$ // x is the "pivot"

$i \leftarrow p - 1$ // i maintains the "left-right boundary"

for $j \leftarrow p$ **to** $r - 1$ **do**

    if $A[j] \le x$ **then**

        $i \leftarrow i + 1$

        exchange $A[i] \leftrightarrow A[j]$

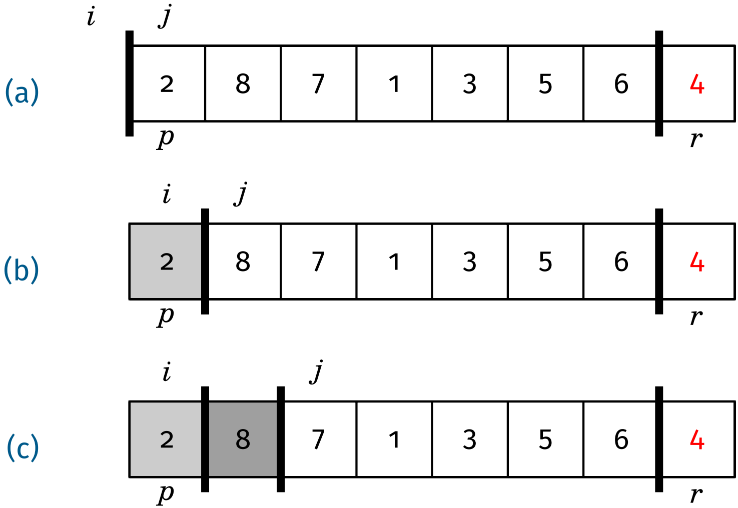
    end if

end for

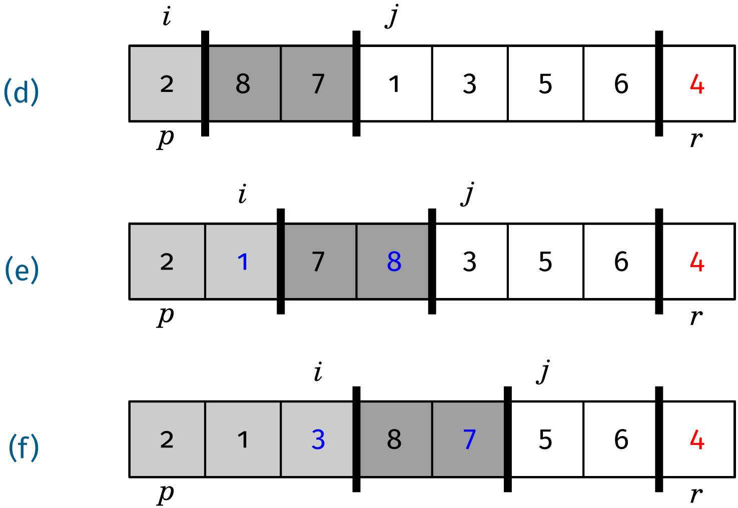
exchange $A[i+1] \leftrightarrow A[r]$

return $i + 1$

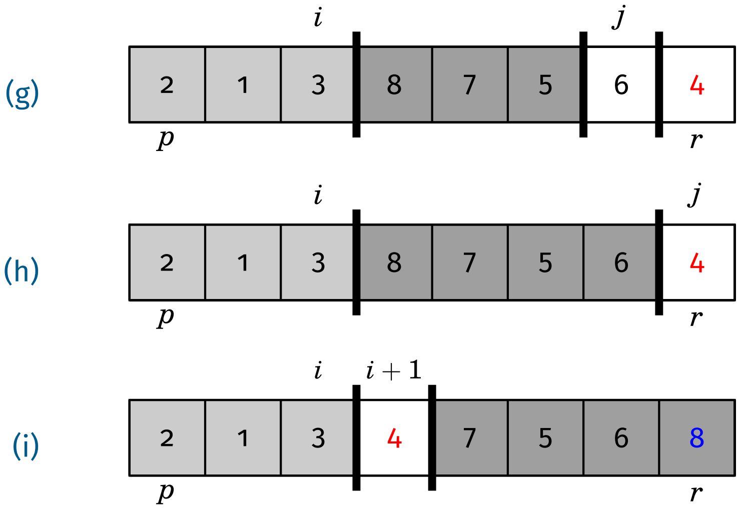
Example: Partition



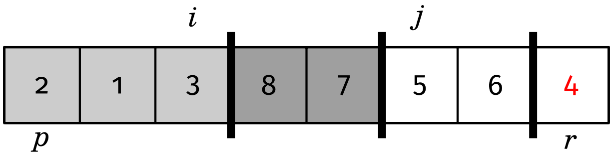
Example: Partition



Example: Partition



Partition, Proof of Correctness



Loop Invariant (Partition)

At the beginning of each iteration:

- $A[p \mathinner{..} i]$ are known to be $\leq \mathit{pivot}$.
- $A[i+1 \mathinner{..} j-1]$ are known to be $> \mathit{pivot}$.
- $A[j, r-1]$ not yet examined.
- $A[r]$ is the pivot.
- $p - 1 \leq i < j$

Partition, Proof of Correctness

Lemma (Partition correctness)

Let $q = \mathrm{Partition}(A, p, r)$. *Then afterwards,*

- $p \leq q \leq r$
- $A[p \mathinner{..} q-1] \leq A[q] < A[q+1 \mathinner{..} r]$

$$\mathrm{LI} : A[p \mathinner{..} i] \leq \mathit{pivot},\ A[i+1 \mathinner{..} j-1] > \mathit{pivot},\ p-1 \leq i < j$$

Proof.

Initialization:

- At the beginning, $i = p - 1$ and $j = p$. Both array ranges simplify to $A[p \mathinner{..} p-1]$ and $A[p \mathinner{..} p-1]$, empty, so LI trivially holds.

Partition, Proof of Correctness

$$\mathrm{LI} : A[p \mathinner{..} i] \le pivot,\ A[i+1 \mathinner{..} j-1] > pivot,\ p-1 \le i < j$$

Proof.

Maintenance:

- Assume LI is true at the start of some j loop.
  In particular: $A[p \mathinner{..} i] \le pivot$ and $A[i+i \mathinner{..} j-1] > pivot$.
- We must show that the execution of the loop body makes LI true for the next j value, $j+1$. There are two cases:
  1. Case $A[j] \le pivot$: *(next page)*
  2. Case $A[j] > pivot$: We don't move it. The $\le$ range stays the same, and $A[j]$ gets absorbed into the $>$ range, and now $A[i+1 \mathinner{..} (j+1)-1] > pivot$, so the LI holds for $j+1$.

Partition, Proof of Correctness

$$\text{LI} : A[p \mathinner{..} i] \leq \textit{pivot},\ A[i+1 \mathinner{..} j-1] > \textit{pivot},\ p-1 \leq i < j$$

Proof.

Maintenance (continued):

1. Case $A[j] \leq \textit{pivot}$: We increment i and exchange $A[i]$ and $A[j]$. I'll write i for the new value and i_0 for the pre-increment value, $i = i_0 + 1$. I'll write $A_0[i]$ and $A_0[j]$ for the pre-exchange array values. ($i_0 < j$ so $i < j + 1$, so that part of LI holds for $j + 1$.)
   - We have added $A_0[j] \leq \textit{pivot}$ to the $\leq$ range and extended its size by incrementing i, so $A[p \mathinner{..} i] \leq \textit{pivot}$ holds.
   - We have moved $A_0[i_0 + 1]$. It was either the first element of the $>$ range, or the $>$ range was empty and it was the first unexamined element (and the "exchange" didn't move it).
   - In either case, the $>$ range (empty or not), moves right one step: it lost $A[i_0 + 1] = A[i]$ and it now starts at $A[i + 1]$ and runs to $A[j]$. That is, $A[i + 1 \mathinner{..} (j + 1) - 1] > \textit{pivot}$, so the LI holds for $j + 1$.

Partition, Proof of Correctness

$$\text{LI} : A[p \mathinner{..} i] \leq pivot,\ A[i+1 \mathinner{..} j-1] > pivot,\ p-1 \leq i < j$$

Proof.

Termination: After the loop ends, $j = r$ (the loop does not cover r), so the loop invariant gives

- $A[p \mathinner{..} i] \leq pivot$
- $A[i+1 \mathinner{..} r-1] > pivot$
- $p-1 \leq i < r$

The algorithm's final step is to exchange $A[i+1]$ and $A[r]$.

This shifts the $>$ range (empty or not) right one index (see reasoning from Maintenance case 1). So $A[i+2 \mathinner{..} r] > pivot = A[i+1]$.

Let $q = i+1$, the return value. Then we have

- $A[p \mathinner{..} i-1] \leq A[q] < A[q+1 \mathinner{..} r]$
- $p \leq q \leq r$

$\square$

Running Time: Best Case

Running time of Partition is clearly $\Theta(n)$ in all cases.

Running time of Quick Sort:

- Best case is when the array is partitioned into two equal parts.
- In this case the recurrence is $T(n) = 2T(n/2) + \Theta(n)$.
- We already know this is $\Theta(n \log n)$.

Running Time: Worst Case

- The worst case happens when the pivot partitions the array into two sub-arrays of size n-1 and 0.
- This happens when the array is already sorted.
- Thus we have:

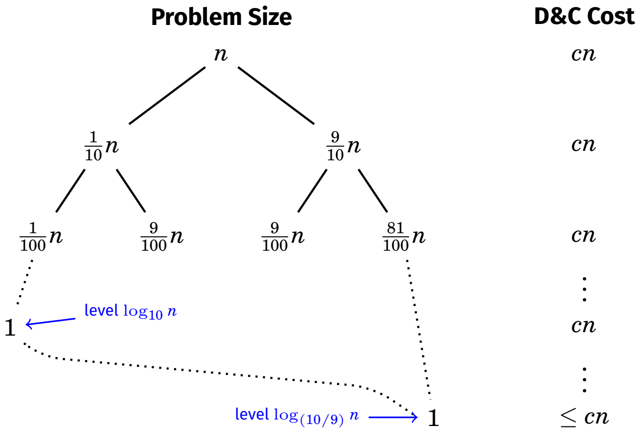
$$\begin{aligned} T(n) &= T(n-1) + T(0) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \sum_{j=0}^{n} \Theta(j) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2) \end{aligned}$$

Running Time: Average Case

- Claim: the average runtime seems to be $O(n \log n)$.
- This means that on average we hit a “good” case.
- This is quite atypical, as usually the average case is no better than the worst case.
- What explains Quick Sort’s luck?

Running Time: Average Case

What happens if the pivot divides the array into two sub-arrays of $0.9n$ and $0.1n$?



Running Time: Average Case

Analysis of Unlucky Case (0.1 – 0.9 split):

- There are $1 + \log_{(10/9)} n$ levels and each has $O(n)$ cost.
- The total cost is therefore $O(n \log n)$.

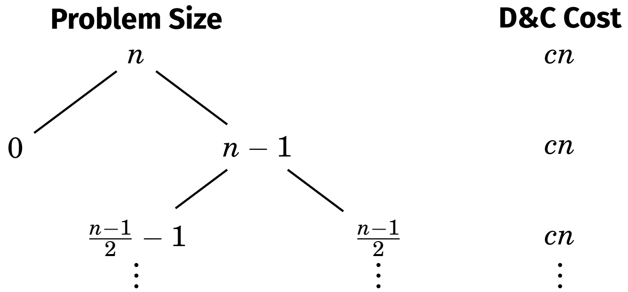
So Quick Sort is not *that* sensitive to how good the pivot is.

What about a different kind of bad luck?

- What happens if occasionally it is as bad as can be?
- Suppose every other iteration the pivot is the largest element.

Running Time: Average Case

Suppose every other iteration the pivot is the largest element.



We simply double the number of levels, it is still $O(n\log(n))$.

Probabilistic vs Randomized Analysis

Probabilistic Analysis

- Remember the average runtime analysis of Insertion Sort.
- We averaged the running time over a particular distribution of inputs — we used a uniform distribution: all inputs equally likely.
- We have to know the distribution of the input — and be able to calculate an average over it!

Randomized Analysis

- We can *change the algorithm* to introduce randomness.
  But it still must *definitely* behave according to its specification.
- By adding randomness, we can make the input distribution *irrelevant*, making it easier to calculate the average (or expected) case behavior.

Randomized Quicksort

- We have a random number generator Random(p,r) which produces numbers between p and r, each with equal probability.

  In practice most random number generators produce pseudo-random numbers.
- The selected number is the pivot index.
- When analyzing the running time of a randomized algorithm we take the expected run time over all inputs.

Randomized Quicksort

Algorithm 3 $\text{RandomizedPartition}(A, p, r)$

Ensure: (same as Partition)

1: $i \leftarrow \text{Random}(p, r)$
2: exchange $A[i] \leftrightarrow A[r]$
3: **return** $\text{Partition}(A, p, r)$

Randomized Quicksort

Algorithm 4 $\text{RandomizedQuicksort}(A, p, r)$

Ensure: (same as Quicksort)

1: **if** $p < r$ **then**
2: $\quad q \leftarrow \text{RandomizedPartition}(A, p, r)$
3: $\quad \text{RandomizedQuicksort}(A, p, q-1)$
4: $\quad \text{RandomizedQuicksort}(A, q+1, r)$
5: **end if**

Rigorous Worst Case Analysis of Quicksort

Let $T(n)$ be the worst case running time for quicksort (or randomized quicksort). It is described by

$$T(n) \leq \max_{0 \leq q \leq n-1} (T(q) + T(n-q-1)) + an$$

for some $a > 0$.

That is, the worst case happens when, on each recursive call, we pick the worst pivot, resulting in the worst (maximum) combined run times on the sub-problems.

We guess that $T(n) = O(n^2)$, and now we'll prove it.

Rigorous Worst Case Analysis of Quicksort

$$T(n) \le cn^2$$

Proof by induction.

- Base case: We must show $T(1) \le c$. Trivial.
- Inductive case: We must show $T(n) \le cn^2$.
- Inductive hypothesis: Assume $T(k) \le ck^2$ for all $1 \le k < n$.
- Calculate:

$$\begin{aligned} T(n) &\le \max_{0 \le q \le n-1} (T(q) + T(n-q-1)) + an \\ &\le c \max_{0 \le q \le n-1} \left(q^2 + (n-q-1)^2\right) + an \end{aligned}$$

- The expression $\left(q^2 + (n-q-1)^2\right)$ is a convex function, achieving a maximum at the endpoints: 0 and $n-1$.
- In those endpoints the value is $(n-1)^2$.

Rigorous Worst Case Analysis of Quicksort

Proof by induction, Cont.

- Therefore:

$$\begin{aligned}
T(n) &\le \max_{0\le q\le n-1} (T(q) + T(n-q-1)) + an \\
&\le c \max_{0\le q\le n-1} \left(q^2 + (n-q-1)^2\right) + an \\
&\le cn^2 - c(2n-1) + an \\
&= cn^2 - (2c-a)n + c \\
&\le cn^2 - (2c-a)n + cn && \text{because } n \ge 1 \\
&= cn^2 - (c-a)n
\end{aligned}$$

- We must pick a large enough c so that $c \ge a$. $\square$

Rigorous Worst Case Analysis of Quicksort

- We just proved an upper bound to the worst case runtime: $T(n) = O(n^2)$.
- Previously we have seen a case where the run time is quadratic. That is, we knew $T(n) = \Omega(n^2)$.
- So when $T(n)$ represents the worst-case performance, $T(n) = \Theta(n^2)$.

Average Case Analysis: Method 1

The average (ie, expected) run time for Randomized-Quicksort on an array of size n is described by the following equation:

$$\begin{aligned} T(n) &= \frac{1}{n}\sum_{q=0}^{n-1}(T(q)+T(n-q-1))+cn+\Theta(1) \\ &= \frac{2}{n}\sum_{q=0}^{n-1}T(q)+cn+\Theta(1) \end{aligned}$$

- We wrote $cn+\Theta(1)$ rather than $\Theta(n)$ since we can assume we do "everything" every time we call Partition.
- This is a worst case assumption that allows us to do something really nice mathematically.

Average Case Analysis: Method 1

$$T(n) = \frac{2}{n}\sum_{q=0}^{n-1} T(q) + cn + \Theta(1)$$

$$nT(n) = 2\sum_{q=0}^{n-1} T(q) + cn^2 + \Theta(n) \qquad \text{multiply by } n$$

$$(n+1)T(n+1) = 2\sum_{q=0}^{n} T(q) + c(n+1)^2 + \Theta(n)$$

multiply by $n+1$

$$(n+1)T(n+1) - nT(n) = 2T(n) + \Theta(n) \qquad \text{subtract}$$

$$(n+1)T(n+1) = (n+2)T(n) + \Theta(n) \qquad \text{simplify}$$

Average Case Analysis: Method 1

- Starting from: $(n+1)T(n+1) = (n+2)T(n) + \Theta(n)$
- Divide by $(n+1)(n+2)$ to get: $\frac{T(n+1)}{n+2} = \frac{T(n)}{n+1} + \Theta\left(\frac{1}{n}\right)$
- Define $g(n) = \frac{T(n)}{(n+1)}$
- So: $g(n+1) = g(n) + \Theta\left(\frac{1}{n}\right)$
- Then: $g(n) = \Theta\left(\sum_{k=1}^{n-1} \frac{1}{k}\right) = \Theta(\log n)$
- Going back: $T(n) = (n+1)g(n) = \Theta(n \log n)$

Average Case Analysis: Method 2

- The total cost is the sum of the costs of all the calls to RandomizedPartition.
- The cost of a call to RandomizedPartition is $O(\#$**for** loop executions$)$, which is $O(\#\text{comparisons})$.
- The expected cost of RandomizedQuicksort is $O(\text{expected } \#\text{comparisons})$.
- Notice that once a key x_k is chosen as pivot, the elements to its left will never be compared to the elements to its right.

Average Case Analysis: Method 2

- Consider $\{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\}$, the set of keys in sorted order.
- Any two keys here are compared only if one of them is pivot and that is the last time they are all in the same partition.
- Each key is equally likely to be chosen as the pivot.
- x_i and x_j can be compared only if one of them is pivot and this will only happen if this is the first pivot from the set $\{x_i, x_{i+1}, \ldots, x_{j-1}, x_j\}$.
- The probability of this is $\frac{2}{(j-i+1)}$.

Average Case Analysis: Method 2

The expected number of comparisons is:

$$\begin{aligned}
\sum_{i<j} \frac{2}{j-i+1} &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&\leq \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= 2(n-1)H_n = O(n \log n)
\end{aligned}$$

where H_n is the nth Harmonic number (see A.7 in the Appendix)