Quicksort

CSE 5311: Design and Analysis of Algorithms

Alex Dillhoff

The University of Texas at Arlington

Introduction

Quicksort is a popular sorting algorith mimplemented in many language libraries.

It has a worst-case running time of $\Theta(n^2)$...

Quicksort

Why is it so popular if it has a worst-case running time of $\Theta(n^2)$?

- It has an average-case running time of $\Theta(n \log n)$ if all values are distinct.
- It is an in-place sorting algorithm.
- It is cache-efficient.

Basic Quicksort

Basic Implementation

- Quicksort is a divide-and-conquer algorithm.
- **Input:** An array A and indices p and r.
- **Output:** The array A with elements in sorted order.

Basic Implementation

```python
def quicksort(arr, p, r):
    q = partition(arr, p, r)
    quicksort(arr, p, q - 1)
    quicksort(arr, q + 1, r)
```

Basic Implementation

- The details are in the `partition` function.
- This function rearranges the elements in the array such that:
    - The pivot element is in its correct position.

Basic Implementation

- The details are in the `partition` function.
- This function rearranges the elements in the array such that:
  - The pivot element is in its correct position.
  - All elements less than the pivot are to the left of it.

Basic Implementation

- The details are in the `partition` function.
- This function rearranges the elements in the array such that:
    - The pivot element is in its correct position.
    - All elements less than the pivot are to the left of it.
    - All elements greater than the pivot are to the right of it.

Partitioning

- The first or last element is chosen as the pivot.
- Picking it this way yields a fairly obvious recurrence of $T(n) = T(n-1) + O(n)$, which is $\Theta(n^2)$.
- There is no need for additional memory to store the sub-arrays: it is done through a clever use of indices.

Partitioning

$p \qquad\qquad\qquad r = x$

$[1, 2, 3, 5, 4]$

$i \quad j \qquad\quad j$

```
def partition(arr, p, r):
    x = arr[r]
    i = p - 1
    for j in range(p, r):
        if arr[j] <= x:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i + 1], arr[r] = arr[r], arr[i + 1]
    return i + 1
```

Partitioning

p - start
r - end

The indices are used to define the following loop invariant. x is the last element.

- **Left:** if $p \leq k \leq i$, then $A[k] \leq x$
- **Middle:** if $i + 1 \leq k \leq j - 1$, then $A[k] > x$
- **Right:** if $k = r$, then $A[k] = x$

Partitioning

p $\qquad\qquad\qquad\qquad$ $r = x$

Example: Partition the array $A = [2, 8, 7, 1, 3, 5, 6, 4]$.

i $\quad$ j

$[2, 1, 3, 4, 7, 5, 6, 8]$

j

Partitioning

Result: Partitioning produces the array $A = [2, 1, 3, 4, 7, 5, 6, 8]$.

Quicksort

Example: Given that the first partitioning step is complete, complete the sorting of the array $A = [2, 1, 3, 4, 7, 5, 6, 8]$ using quicksort.

[2, 1, 3]

[2, 1]   [3]

x

[1, 2, 3, 4]

0 1 2 3

[7, 5, 6, 8]

[5, 6, 7]   [8]

4 5 6   7

Performance Analysis

Recursion Tree Analysis

Picking the smallest or largest value as the partition yields a bad split.

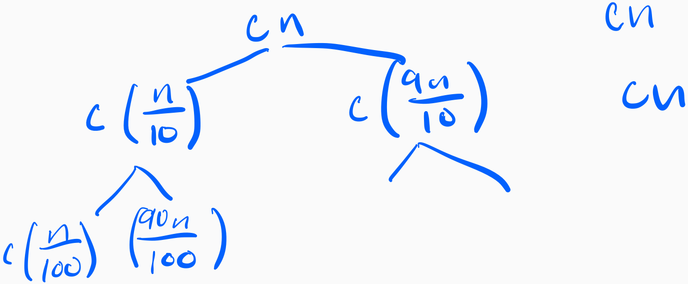
When does a split become acceptable?

Recursion Tree Analysis

Suppose we always get a 9-to-1 split.

Intuitively this seems pretty bad, and you may think that there is no way we would ever get this unlucky.

Exercise: Draw the recursion tree for the case where we always get a 9-to-1 split.



Recap

- The subtree for the $\frac{1}{10}$ split bottoms out after being called $\log_{10} n$ times.
- Until this happens, the cost of each level of the tree is n.
- The right tree continues with an upper bound of $\leq n$.
- The right tree completes after $\log_{10/9} n = \Theta(\lg n)$ levels.

Analysis

Best-case

In the best-case, the pivot is the median of the array and two balanced subarrays are created:

1. one of size $n/2$ and
2. one of size $\lfloor(n-1)/2\rfloor$.

The recurrence is $T(n) = 2T(n/2) + \Theta(n)$, which is...

Best-case

In the best-case, the pivot is the median of the array and two balanced subarrays are created:

1. one of size $n/2$ and
2. one of size $\lfloor (n-1)/2 \rfloor$.

The recurrence is $T(n) = 2T(n/2) + \Theta(n)$, which is...$\Theta(n \log n)$.

Lower bound

Using the substitution method, we can establish a lower bound.

Start with the fact that the partitioning produces two subproblems with a total size of $n-1$.

This gives the following recurrence:

$$T(n) = \min_{0 \leq q \leq n-1} \{T(q) + T(n-q-1)\} + \Theta(n).$$

Lower bound

This gives the following recurrence:

$$T(n) = \min_{0 \leq q \leq n-1} \{T(q) + T(n - q - 1)\} + \Theta(n).$$

The minimum function here means we are looking for the value of q that minimizes the sum of the two subproblems.

Our **hypothesis** will be that

$$T(n) \geq cn \lg n = \Omega(n \lg n).$$

Lower Bound

$3x^2$

Plugging in our hypothesis, we get

$$\begin{aligned} T(n) &\geq \min_{0\leq q\leq n-1}\{cq\lg q + c(n-q-1)\lg(n-q-1)\} + \Theta(n) \\ &= c\min_{0\leq q\leq n-1}\{q\lg q + (n-q-1)\lg(n-q-1)\} + \Theta(n). \end{aligned}$$

If we take the derivative of the function inside the minimum with respect to q, we get

$$\begin{aligned}&\frac{d}{dq}\{q \lg q + (n-q-1)\lg(n-q-1)\}\\&= c\{\frac{q}{q} + \lg q - \lg(n-q-1) - \frac{(n-q-1)}{(n-q-1)}\}.\end{aligned}$$

Lower Bound

Setting this equal to zero and solving for q yields

$$q = \frac{n-1}{2}.$$

Lower Bound

We can then plug this value of q into the original function to get

$$
\begin{aligned}
T(n) &\geq c\frac{n-1}{2}\lg\frac{n-1}{2} + c\frac{n-1}{2}\lg\frac{n-1}{2} + \Theta(n) \\
&= cn\lg(n-1) + c(n-1) + \Theta(n) \\
&= cn\lg(n-1) + \Theta(n) \\
&\geq cn\lg n \\
&= \Omega(n\lg n).
\end{aligned}
$$

Average Case

The average-case running time is $\Theta(n \log n)$.

Quicksort is highly dependent on the relative ordering of the input.

Average Case

Consider the case of a randomly ordered array.

- The cost of partitioning the original input is $O(n)$.
- Let's say that the pivot was the last element, yielding a split of 0 and $n - 1$.

What if we get lucky on the next iteration and get a balanced split?

What if we get lucky on the next iteration and get a balanced split?

Even if the rest of the algorithm splits between the median and the last element, the upper bound on the running time is $\Theta(n \log n)$.

It is highly unlikely that the split will be unbalanced on every iteration given a random initial ordering.

Average Case

Formalize this by defining a lucky $L(n) = 2U(n/2) + \Theta(n)$ and an unlucky split $U(n) = L(n-1) + \Theta(n)$.

Solve for $L(n)$ by plugging in the definition of $U(n)$.

$$
\begin{aligned}
L(n) &= 2U(n/2) + \Theta(n) \\
&= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\
&= 2L(n/2-1) + \Theta(n) \\
&= \Theta(n \log n)
\end{aligned}
$$

Randomized Quicksort

One would have to be extremely unlucky to get a quadratic running time if the input is randomly ordered.

Randomized quicksort builds on this intuition by selecting a random pivot on each iteration.

Randomized Quicksort

```python
def randomized_partition(arr, p, r):
    i = random.randint(p, r)
    arr[i], arr[r] = arr[r], arr[i]
    return partition(arr, p, r)

def randomized_quicksort(arr, p, r):
    if p < r:
        q = randomized_partition(arr, p, r)
        randomized_quicksort(arr, p, q - 1)
        randomized_quicksort(arr, q + 1, r)
```

As long as each split puts a constant amount of elements to one side of the split, then the running time is $\Theta(n \log n)$.

We can understand this analysis simply by asking the right questions.

Analysis

1. What is the running time of Quicksort dependent on?

Analysis

1. **What is the running time of Quicksort dependent on?**
    - The biggest bottleneck is the partitioning function.
    - At most, we get really unlucky and the first pivot is picked every time.
    - This means it is called n times yielding $O(n)$.

1. **What is the running time of Quicksort dependent on?**
    - The biggest bottleneck is the partitioning function.
    - At most, we get really unlucky and the first pivot is picked every time.
    - This means it is called n times yielding $O(n)$.
    - The variable part of this is figuring out X: the number of comparisons made.
    - The running time is then $O(n + X)$.

Expected value of X

The number of comparisons can be expressed as

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij},$$

where X_{ij} is the indicator random variable that is 1 if $A[i]$ and $A[j]$ are compared and 0 otherwise.

Expected value of X

This works with our worst case analysis.

If we always get a split of 0 and $n-1$, then the indicator random variable is 1 for every comparison, yielding $O(n^2)$.

Expected value of X

Taking the expectation of both sides:

$$
\begin{aligned}
E[X] &= E\left[\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}\right] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} P(X_{ij} = 1).
\end{aligned}
$$

What is $P(X_{ij} = 1)$?

- Let $z_i, \ldots, z_j$ be the indices of elements in a sorted version of the array.

Expected value of X

What is $P(X_{ij} = 1)$?

- Let $z_i, \ldots, z_j$ be the indices of elements in a sorted version of the array.
- Under this assumption, z_i is compared to z_j only if z_i or z_j is the first pivot chosen from the subarray $A[i \ldots j]$.

Expected value of X

What is $P(X_{ij} = 1)$?

- Let $z_i, \ldots, z_j$ be the indices of elements in a sorted version of the array.
- Under this assumption, z_i is compared to z_j only if z_i or z_j is the first pivot chosen from the subarray $A[i \ldots j]$.
- In a set of distinct elements, the probability of picking any pivot from the array from i to j is $\frac{1}{j-i+1}$.

Expected value of X

What is $P(X_{ij} = 1)$?

- Let $z_i, \ldots, z_j$ be the indices of elements in a sorted version of the array.
- Under this assumption, z_i is compared to z_j only if z_i or z_j is the first pivot chosen from the subarray $A[i \ldots j]$.
- In a set of distinct elements, the probability of picking any pivot from the array from i to j is $\frac{1}{j-i+1}$.
- This means that the probability of comparing z_i and z_j is $\frac{2}{j-i+1}$.

Expected value of X

$$
\begin{aligned}
E[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \quad \text{change of variable } k = j - i \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k} \quad \text{bounded by harmonic series} \\
&= \sum_{i=1}^{n-1} O(\log n) \\
&= O(n \log n).
\end{aligned}
$$

Paranoid Quicksort

Paranoid Quicksort

Repeat the following until the partitioning until the left or right subarray is less than or equal to $\frac{3}{4}$ of the original array.

1. Choose a random pivot.
2. Partition the array.
3. Verify that the left and right subarrays are less than or equal to $\frac{3}{4}$ of the original array; if not, repeat the partitioning.
4. Recursively sort the subarrays.

Analysis

Most of the analysis of Paranoid Quicksort follows that of randomized quicksort.

The focus is on the expected number of calls times `partition` is called until no side of the split is greater than $\frac{3}{4}$ of the input.

Analysis

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

Consider a sorted array of n *distinct* elements.

- The first and last $\frac{n}{4}$ elements would produce a bad split.
- That means there are $n/2$ values that provide a good split, implying that $p(\text{good split}) = \frac{1}{2}$.
- Knowing the probability of this event means we can calculated the expected number of times we should call `partition` before getting a *good* split, which is 2.

Analysis

Continuing on with this analysis, we need to state the recurrence:

$$T(n) \leq 2cn + T(\lfloor 3n/4) \rfloor) + T(\lceil n/4 \rceil) + O(1)$$

The addition of $2cn$ is not enough to change our analysis from above.

Thus, the expected running time of Quicksort is $O(n \log n)$.