QuickSort (Ch. 7)

Like Merge-Sort, based on the three-step process of divide-and-conquer.

Input: An array A[1..n] of comparable elements, the starting position and the ending index of A.

Output: A permutation of A such that elements are in ascending order.

QuickSort

Divide:

Conquer:
Sort the two subarrays by recursive calls to QuickSort.

Combine:
No work is needed to combine subarrays since they are sorted in-place.

To sort the subarray A[p..r] (initially, p = 1 and r = A.length):

1. if p < r
2. q = Partition(A, p, r)
3. QuickSort(A, p, q-1)
4. QuickSort(A, q+1, r)

QuickSort(A, p, r)
1. if p < r
2. q = Partition(A, p, r)
3. QuickSort(A, p, q-1)
4. QuickSort(A, q+1, r)

Partition(A, p, r)
1. x = A[r] \ // choose pivot
2. i = p - 1
3. for j = p to r - 1
4. if A[j] ≤ x
5. i = i + 1
6. swap A[i] and A[j]
7. swap A[i+1] and A[r]
8. return i + 1

Partition

Example on an 8-element array:

1. x = A[r] \ // choose pivot
2. i = p - 1
3. for j = p to r - 1
4. if A[j] ≤ x
5. i = i + 1
6. swap A[i] and A[j]
7. swap A[i+1] and A[r]
8. return i + 1

Initial call:
QuickSort(A, 1, A.length)

What does Partition do? What is running time of Partition?

Note that Partition always selects the last element A[r] in the subarray A[p..r] as the pivot.

Partition

Loop invariant:
As Partition executes, the array is divided into 4 regions, some possibly empty, such that
1. All values in A[p..i] are ≤ pivot.
2. All values in A[i+1..j-1] are > pivot.

Although not needed as part of the loop invariant, the 4th region is A[i+1..j-1], whose entries have not yet been examined, so we don't know how they compare to the pivot.
### Partition

- **Partition(A, p, r)**
  1. \( x = A[r] \) // choose pivot
  2. \( i = p - 1 \)
  3. for \( j = p \) to \( r - 1 \)
  4. if \( A[j] \neq x \)
  5. \( i = i + 1 \)
  6. swap \( A[i] \) and \( A[j] \)
  7. swap \( A[i+1] \) and \( A[r] \)
  8. return \( i + 1 \)

The index \( j \) disappears after step 7 because it is no longer needed once the **for** loop is exited.

### Formal Correctness of Partition

As the Partition procedure executes, the array is partitioned into four regions, some of which may be empty.

**Loop invariant:** At the beginning of each iteration of the for loop (lines 3-6), for any array index \( k \),
- If \( p \neq k \neq i \), then \( A[k] \neq x \).
- If \( i+1 \neq k \neq j-1 \), then \( A[k] > x \).
- If \( k = r \), then \( A[k] = x \).

The fourth region is \( A[j \ldots r - 1 \ldots 1] \), whose entries have not yet been examined.

### Quicksort Running Time

The running time of quicksort depends on the partitioning of subarrays.

- If the subarrays are balanced, then quicksort can run as fast as mergesort.
- If the subarrays are unbalanced, then quicksort can run as slowly as insertion sort.

**Worst case** occurs when the array is already sorted in ascending order.

**Best case** occurs when subarrays are completely unbalanced, with 0 elements in one subarray and \( n-1 \) elements in the other.

We get the recurrence

\[
T(n) = T(n-1) + O(n) \\
= T(n-1) + \Theta(n) \\
= \Theta(n^2)
\]

This is the same running time as the worst case of insertion sort.

What input instance would cause the worst-case running time of quicksort to occur?

When the array is already sorted in ascending order.
Quicksort Average-case

The average case of quicksort's running time is much closer to the best case than it is to the worst case.

Imagine that Partition always produces a 9-to-1 split.

Then we get the recurrence $T(n) \leq T(n/3) + T(2n/3) + O(n)$

The recursion tree is like the one for $T(n) = T(n/3) + T(2n/3) + O(n)$

Any split of constant proportionality will yield a recursion tree of depth $O(\log n)$.

Quicksort with Good Average Running Time

How can we modify quicksort to get good average case behavior on all inputs?

2 techniques:
1. randomly permute input prior to running quicksort. Will produce tree of possible executions, most of them finish fast.
2. choose partition randomly at each iteration instead of choosing element in highest array position.

### Randomized-Quicksort

1. if $p < r$
2. $q = \text{Randomized-Partition}(A, p, r)$
3. $\text{Randomized-Quicksort}(A, p, q-1)$
4. $\text{Randomized-Quicksort}(A, q+1, r)$

### Randomized-Partition

1. $i = \text{Random}(p, r)$
2. swap $A[i]$ and $A[r]$
3. return $\text{Partition}(A, p, r)$

Average-case Analysis of Randomized-Quicksort

- Rename elements of $A$ as $z_1, z_2, \ldots, z_n$ and define the set $Z_i$ to be the set of elements between $z_i$ and $z_{i+1}$ inclusive.
- Each pair of elements is compared at most once, because elements are compared only to the pivot and the pivot is not used again.
- Let $X_{ij} = 1$ if $z_i$ is compared to $z_j$
- Since each pair is compared at most once, the total # comparisons $= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$
- Taking the expectations of both sides, use L.5.1 and LoE:
  $$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} Pr(X_{ij})$$
- But what is $Pr(z_i$ is compared to $z_j$)?

Quicksort with Good Average Running Time

Intuition: Some splits will be close to balanced and others close to unbalanced, so good and bad splits will be randomly distributed in recursion tree.

The running time will be (asymptotically) bad only if there are many bad splits in a row.

- A bad split followed by a good split results in a good partitioning after one extra step.

Average-case Analysis of Randomized-Quicksort

Observations:
- Numbers in separate partitions are never compared.
- If the pivot is not $z_i$ or $z_j$, these elements are never compared.
- The probability that $z_i$ is compared to $z_j$ is the probability that either one is chosen as the pivot.
- There are $j-i+1$ elements and the probability that any particular one is chosen as the pivot is $1/(j-i+1)$

Thus $Pr(z_i$ is compared to $z_j) = Pr(z_i$ is the pivot) + $Pr(z_j$ is the pivot) $= 2/(j-i+1)$. Substituting this for $E[X]$ and letting $k = j - i$:

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{k=1}^{n-1} \sum_{i=1}^{k} \frac{2}{k+1} = \sum_{i=1}^{n} O(lg n) = O(n lgn)$$

So the expected running time of Randomized-Quicksort is $O(n lgn)$ when elements in $A$ are distinct.