Chapter 7. Quicksort

Joon Soo Yoo

April 9, 2025

Assignment

- Read §7
- Problems
  - §7.1 - #2, 4
  - §7.2 - #3, 6
  - §7.3 - #2
  - §7.4 - #1, 4

Chapter 7: Quicksort

- **Chapter 7.1: Description of quicksort**
- Chapter 7.2: Performance of quicksort
- Chapter 7.3: A randomized version of quicksort
- Chapter 7.4: Analysis of quicksort

Chapter 7: Quicksort — Introduction

- **Quicksort** is a divide-and-conquer sorting algorithm
- **Worst-case time:** $\Theta(n^2)$
- **Expected time (distinct elements):** $\Theta(n \log n)$
- Despite its worst case, it is often the **most practical sorting algorithm** in practice
- Advantages over merge sort:
  - **In-place** sorting (no extra memory)
  - Performs well even in virtual memory environments
  - Small hidden constants in the $\Theta(n \log n)$ bound

Quicksort is a Divide-and-Conquer Algorithm

Quicksort applies the **divide-and-conquer** paradigm introduced in Section 2.3.1:

- **Divide:** Partition the subarray $A[p \ldots r]$ into two sides:
  - Elements $\leq A[q]$ go to the left (**low side**)
  - Elements $\geq A[q]$ go to the right (**high side**)
  - q is the final position of the pivot
- **Conquer:** Recursively apply quicksort to both subarrays
- **Combine:** Do nothing — the two sides are already sorted

Why No Combine Step in Quicksort?

Unlike Merge Sort, Quicksort has a trivial combine step.

- After partitioning:

$$A[p \ldots q-1] \leq A[q] \leq A[q+1 \ldots r]$$

- Recursive calls sort each side independently
- Since all elements on the left are $\leq A[q]$, and those on the right are $\geq A[q]$,

The full subarray $A[p \ldots r]$ becomes sorted

No merging step is needed!

`QUICKSORT(A, p, r)` Pseudocode

```
QUICKSORT(A, p, r):
    if p < r:
        q = PARTITION(A, p, r)
        // Recursively sort left and right parts
        QUICKSORT(A, p, q - 1)
        QUICKSORT(A, q + 1, r)
```

Initial call: `QUICKSORT(A, 1, n)` to sort the entire array.

Partitioning the Array

PARTITION(A, p, r) is the key subroutine in Quicksort.

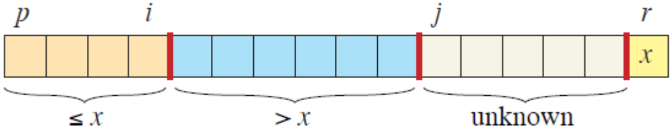
- It rearranges the subarray $A[p \dots r]$ **in-place**
- The pivot is chosen as $x = A[r]$
- After execution:
  - All elements $\leq x$ appear to the left of the pivot
  - All elements $> x$ appear to the right of the pivot
  - The pivot is placed in its final sorted position
- Returns index q, the pivot's final position

PARTITION(A, p, r) Pseudocode

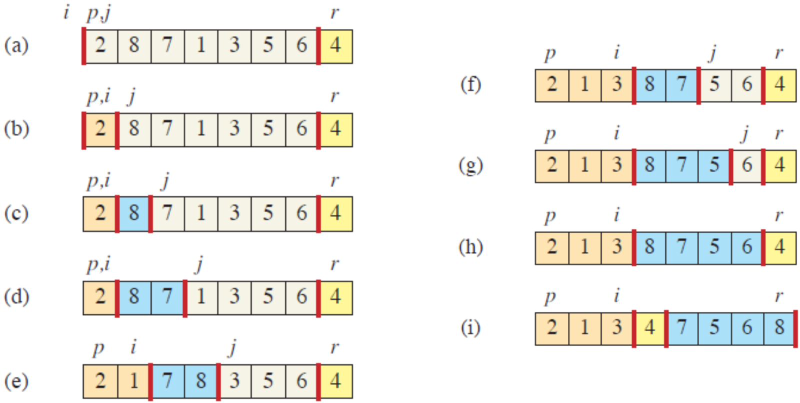
```
PARTITION(A, p, r)
x = A[r]                              // the pivot
i = p - 1                             // highest index into the low side
for j = p to r - 1                    // process each element other than the pivot
    if A[j] ≤ x                       // does this element belong on the low side?
        i = i + 1                     // index of a new slot in the low side
        exchange A[i] with A[j]       // put this element there
exchange A[i + 1] with A[r]           // pivot goes just to the right of the low side
return i + 1                          // new index of the pivot
```

Partitioning: Four Regions

- The array $A[p \dots r]$ is divided into four regions during `PARTITION`:
  - Brown region $A[p \dots i]$: All values $\leq x$
  - Blue region $A[i+1 \dots j-1]$: All values $> x$
  - White region $A[j \dots r-1]$: Values not yet examined
  - Yellow cell $A[r] = x$: The pivot
- These regions form the loop invariant for the `for` loop in `PARTITION`



Visualization



Loop Invariant for PARTITION

At the beginning of each iteration of the `for` loop:

1. If $p \leq k \leq i$, then $A[k] \leq x$ — the tan region
2. If $i + 1 \leq k \leq j - 1$, then $A[k] > x$ — the blue region
3. If $k = r$, then $A[k] = x$ — the yellow region (pivot)

Goal: Prove the loop invariant holds:

- Before the first iteration (Initialization)
- Maintained during each iteration (Maintenance)
- Guarantees correctness at the end (Termination)

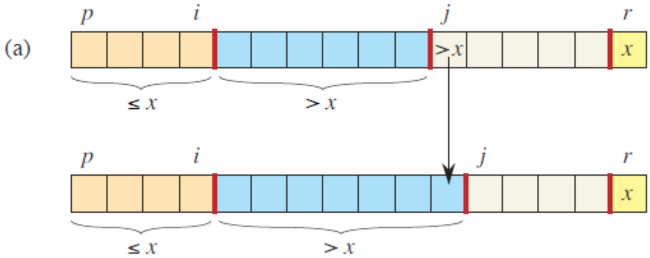
Loop Invariant — Initialization

Before the first iteration:

- $i = p - 1$, $j = p$
- There are no elements between p and i, or between $i + 1$ and $j - 1$
- Therefore:
  - Conditions 1 and 2 of the invariant are trivially satisfied
  - Line 1 sets $x = A[r]$, so condition 3 is satisfied

The invariant holds before the loop starts.

Loop Invariant — Maintenance Case 1: $A[j] > x$



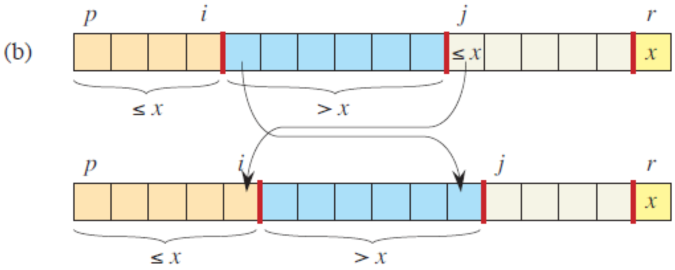
What happens: $A[j] > x$

- We do not increment i, no swap is performed
- We increment j

All loop invariant conditions continue to hold

- $A[j-1]$ (just examined) now belongs in the blue region

Loop Invariant — Maintenance Case 2: $A[j] \leq x$



What happens: $A[j] \leq x$

- We increment i
- Swap $A[i]$ and $A[j]$
- Then increment j

Why it's correct:

- The new $A[i] \leq x \rightarrow$ condition 1 satisfied
- $A[j-1] > x$ (from blue region) was just moved $\rightarrow$ condition 2 holds

Loop Invariant — Termination

At termination:

- Loop runs for $r - p$ iterations $\to$ ends with $j = r$
- The unexamined region $A[j \ldots r-1]$ is now empty
- All values are partitioned into three regions:
  - $A[p \ldots i] \leq x$
  - $A[i+1 \ldots r-1] > x$
  - $A[r] = x$

The final swap places x at position $i + 1$, and we return that index.
Loop invariant guarantees correctness of the partition.

Running Time of `PARTITION`

Claim: The running time of `PARTITION` on a subarray of size n is $\Theta(n)$

- The `for` loop runs from $j = p$ to $r - 1 \rightarrow$ exactly $n - 1$ iterations
- Each iteration performs a constant number of operations:
  - A comparison: $A[j] \leq x$
  - Maybe an increment of i
  - Maybe a swap between $A[i]$ and $A[j]$
- One final swap after the loop (pivot placement)

Conclusion: The total work is proportional to n, so the running time is:

$$\boxed{\Theta(n)}$$

Chapter 7: Quicksort

Quicksort Performance Depends on Pivot Balance

- The running time of Quicksort depends on how well the pivot splits the array.
- **Balanced partition:** Both sides of the partition are roughly equal in size
  - Quicksort behaves like Merge Sort: $\Theta(n \log n)$
- **Unbalanced partition:** One side is much larger than the other
  - Quicksort degrades to $\Theta(n^2)$, similar to Insertion Sort

Key Insight: The choice of pivot greatly affects performance

Best vs. Worst Case Intuition

Balanced Partitioning:

- Each pivot splits the array into two halves
- Recursion tree is roughly of height $\log n$
- Total time is $\Theta(n \log n)$

Unbalanced Partitioning:

- Each pivot gives a 1-element side and $n - 1$ element side
- Recursion tree becomes a long chain
- Total time is $\Theta(n^2)$

Quicksort Memory Usage

In-place sorting:

- Quicksort sorts the array in place (no extra arrays)
- This satisfies the definition of in-place sorting

But it still uses stack memory:

- Each recursive call uses stack space
- Stack depth = maximum depth of recursion tree
- In worst case (unbalanced), recursion depth = $\Theta(n)$
- In best case (balanced), depth = $\Theta(\log n)$

Worst-Case Partitioning in Quicksort

Worst case: Partitioning produces:

$$\text{One subproblem of size } n-1, \quad \text{the other of size } 0$$

- This occurs when the pivot is the smallest or largest element
- Happens consistently if input is already sorted (ascending or descending)

Time per partition: $\Theta(n)$
Recursive pattern:

$$T(n) = T(n-1) + \Theta(n)$$

Solving the Worst-Case Recurrence

Given:

$$T(n) = T(n-1) + \Theta(n)$$

Unrolling:

$$= T(n-2) + \Theta(n-1) + \Theta(n)$$

$$= \cdots = T(1) + \sum_{k=2}^{n} \Theta(k) = \Theta(n^2)$$

Result: Worst-case time is:

$$T(n) = \Theta(n^2)$$

Best-Case Partitioning in Quicksort

Best case: Every call to PARTITION splits the array into two equal halves.

- One subarray of size $\left\lfloor \frac{n-1}{2} \right\rfloor \leq \frac{n}{2}$
- One subarray of size $\left\lceil \frac{n-1}{2} \right\rceil \leq \frac{n}{2}$
- Partitioning work per level: $\Theta(n)$

Running time recurrence:

$$T(n) = 2T(n/2) + \Theta(n)$$

Apply Master Theorem (Case 2):

$$T(n) = \Theta(n \log n)$$

Conclusion: When Quicksort always partitions evenly, its runtime matches Merge Sort: $\Theta(n \log n)$

Balanced Partitioning in Quicksort

Idea: Even when partitioning is *not perfectly even*, Quicksort performs well.

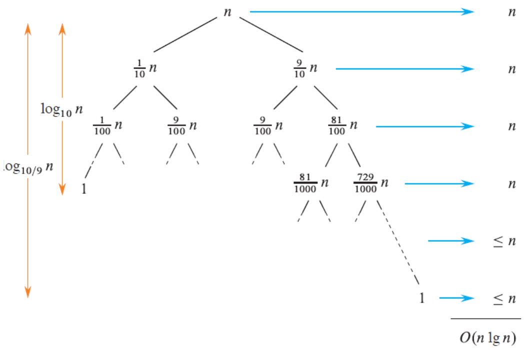
- Example: PARTITION always splits the array in a 9-to-1 ratio
- This seems unbalanced — but Quicksort still achieves:

$$T(n) = T(9n/10) + T(n/10) + \Theta(n)$$

- The cost at each level is $\Theta(n)$

Goal: Understand how such a split leads to $O(n \log n)$ time

Recursion Tree for a 9-to-1 Partitioning



- Each level of the recursion tree does $O(n)$ total work
- The depth of the tree is:

$$\Theta(\log_{10/9} n) = \Theta(\log n)$$

- Total work:

$$O(n) \cdot \Theta(\log n) = O(n \log n)$$

Constant Proportional Splits Yield $O(n \log n)$

- Even a 99-to-1 partition gives $O(n \log n)$ time
- As long as each partition is reduced by a constant factor at every level
- The key: tree depth is still $\Theta(\log n)$, and each level does $O(n)$ work

Therefore:

$$T(n) = O(n \log n) \quad \text{for any constant-ratio split}$$

The split ratio affects only the constant factor, not the asymptotic growth

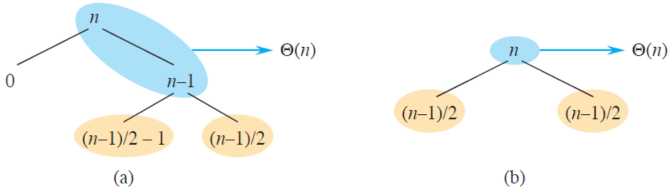
Intuition for the Average Case

Goal: Understand why Quicksort is fast *on average*, despite some bad splits.

- Input is modeled as a random permutation of distinct elements
- Quicksort's performance depends on the relative ordering, not values
- Like the hiring problem (Chapter 5), we assume all orderings are equally likely

Key idea: Average-case behavior is a mix of good and bad splits, but still leads to efficient sorting.

Alternating Splits



- First level: worst-case split $\rightarrow$ sizes $n - 1$ and 0
- Second level: good split $\rightarrow$ halves of $n - 1$
- Total work for two levels:

$$\Theta(n) + \Theta(n-1) = \Theta(n)$$

- Even though the first split was bad, the good split catches up

Conclusion: Average Case is $O(n \log n)$

- In the average case, good and bad splits are **randomly mixed**
- Even alternating between worst-case and best-case still gives:

$$\Theta(n \log n)$$

- Bad splits get "absorbed" by good ones
- So average-case cost is close to best case — with only a slightly larger constant

Result: Quicksort is fast on average, even with occasional bad pivots!

Chapter 7: Quicksort

Why Randomize Quicksort?

Problem: Our average-case analysis assumes the input is a random permutation

- In real-world scenarios, input may be sorted, reverse-sorted, or adversarial
- In such cases, deterministic Quicksort can degrade to $\Theta(n^2)$
- **Solution:** Use randomization to guard against worst-case inputs

Benefit: With random pivot selection, all inputs behave like "random" inputs on average

How Randomized Quicksort Works

Idea: Instead of always choosing the last element $A[r]$ as the pivot:

- Choose a random index $i \in [p, r]$
- Swap $A[i]$ with $A[r]$
- Then proceed with normal partitioning

Effect: Every element in $A[p \dots r]$ has equal probability of being the pivot

RANDOMIZED-QUICKSORT and PARTITION

```
RANDOMIZED-PARTITION(A, p, r)
    i = RANDOM(p, r)
    exchange A[r] with A[i]
    return PARTITION(A, p, r)

RANDOMIZED-QUICKSORT(A, p, r)
    if p < r
        q = RANDOMIZED-PARTITION(A, p, r)
        RANDOMIZED-QUICKSORT(A, p, q - 1)
        RANDOMIZED-QUICKSORT(A, q + 1, r)
```

Only change: one random swap before partitioning

Why Randomization Helps

- Avoids dependence on input order
- Every pivot has equal probability $\rightarrow$ partitions are likely balanced on average
- All inputs behave like random permutations
- Prevents adversarial inputs from triggering worst-case behavior

Result: Expected runtime becomes $O(n \log n)$ for *all* inputs

Chapter 7: Quicksort

- Chapter 7.1: Description of quicksort
- Chapter 7.2: Performance of quicksort
- Chapter 7.3: A randomized version of quicksort
- **Chapter 7.4: Analysis of quicksort**

Formal Analysis of Quicksort

Goal: Prove that the worst-case running time of Quicksort is $\Theta(n^2)$

- Section 7.2 gave an intuition for how bad partitioning leads to $\Theta(n^2)$
- Now we'll use the **substitution method** to formally prove:

$$T(n) = O(n^2)$$

- This worst-case applies to both `QUICKSORT` and `RANDOMIZED-QUICKSORT`

Worst-Case Recurrence for Quicksort

Let $T(n)$ be the worst-case time for QUICKSORT on input of size n.
In the worst case:

- The pivot causes a maximally unbalanced split: $q = 0$ or $q = n - 1$
- PARTITION does $\Theta(n)$ work

Worst-case recurrence:

$$T(n) = \max_{0 \leq q \leq n-1} \{T(q) + T(n - 1 - q)\} + \Theta(n) \quad \text{(Equation 7.1)}$$

Using the Substitution Method

Goal: Prove $T(n) = O(n^2)$ using substitution.

Step 1: Guess the bound (inductive hypothesis)
Assume:

$$T(m) \leq cm^2 \quad \text{for all } m < n, \text{ where } c > 0$$

Step 2: Apply the recurrence to $T(n)$
From the recurrence:

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

Apply the inductive hypothesis to the recursive calls:

$$T(n) \leq \max_{0 \leq q \leq n-1} \left\{cq^2 + c(n-1-q)^2\right\} + \Theta(n)$$

Now simplify and bound the maximum to complete the proof.

Bounding the Maximum

Focus on the inner term:

$$q^2 + (n-1-q)^2$$

Expand:

$$= q^2 + (n-1)^2 - 2q(n-1) + q^2 = (n-1)^2 + 2q(q-(n-1))$$

The second term is always ≤ 0, so:

$$q^2 + (n-1-q)^2 \leq (n-1)^2$$

So:

$$T(n) \leq c(n-1)^2 + \Theta(n) \leq cn^2 - c(2n-1) + \Theta(n) \leq cn^2$$

(for large enough c)

Conclusion: Worst-Case is $\Theta(n^2)$

- We showed $T(n) = O(n^2)$ via substitution method
- We also know that $T(n) = \Omega(n^2)$ in the worst-case input (exercise 7.4-1)

Therefore:

$$T(n) = \Theta(n^2)$$

Running Time and Comparisons in Quicksort

```
PARTITION(A, p, r)
1  x = A[r]                         // the pivot
2  i = p − 1                        // highest index into the low side
3  for j = p to r − 1               // process each element other than the pivot
4      if A[j] ≤ x                  // does this element belong on the low side?
5          i = i + 1                    // index of a new slot in the low side
6          exchange A[i] with A[j]      // put this element there
7  exchange A[i + 1] with A[r]      // pivot goes just to the right of the low side
8  return i + 1                     // new index of the pivot
```

Observation: Quicksort's total running time is primarily due to comparisons in the `PARTITION` procedure.

Focus: Count comparisons of elements (not indices)

- Occur in line 4 of `PARTITION`
- Each compares an element to the current pivot

Let: $X =$ Total number of element comparisons
Then the total running time is tied to X

Lemma 7.1 – Running Time in Terms of Comparisons

Lemma 7.1: The running time of `QUICKSORT` on an n-element array is:

$$\boxed{O(n + X)}$$

where X is the number of element comparisons made in line 4 of `PARTITION`

This lemma applies to both:

- `QUICKSORT` (deterministic)
- `RANDOMIZED-QUICKSORT`

Proof Idea of Lemma 7.1

- Each call to `PARTITION` removes the pivot from future calls
- There are at most n calls to `PARTITION` (one per pivot)
- Each `QUICKSORT` call can make up to 2 recursive calls $\rightarrow$ at most $2n$ total calls

For each `PARTITION` call:

- Outside the loop: $O(1)$
- Loop iterations: one comparison per iteration

Total time:

$$O(n) \text{ (overhead) } + O(X) \text{ (comparisons)} = O(n + X)$$

Goal: Expected Number of Comparisons in Quicksort

Let X = total number of element comparisons made by `PARTITION`
Goal: Compute $\mathbb{E}[X]$

- Each comparison happens in line 4 of `PARTITION`
- To understand $\mathbb{E}[X]$, we must ask:
  - When are two elements compared?
  - How many such comparisons occur?

Tool: Analyze using the order of elements in the sorted array $z_1 < z_2 < \cdots < z_n$

Lemma 7.2 – When Are Two Elements Compared?

Let $z_i < z_j$ be two elements in the sorted array. Then:
Lemma 7.2:

z_i and z_j are compared if and only if

either z_i or z_j is the first pivot chosen from $Z_{ij} = \{z_i, \ldots, z_j\}$

Consequences:

- If any z_k with $i < k < j$ is chosen first $\rightarrow$ z_i and z_j fall into different sides $\rightarrow$ no comparison
- Comparison happens only once — when the pivot is chosen

No Two Elements Are Compared Twice

Lemma 7.2 (continued): Once two elements z_i and z_j are compared, they are never compared again.

Why?

- The comparison happens only if one of z_i or z_j is chosen first as pivot in Z_{ij}
- After that pivot is chosen, it is removed from the array
- So z_i and z_j can never appear together in the same recursive call again

Example: Comparing Elements in Randomized Quicksort

Suppose the input is $\{1, 2, \ldots, 10\}$ (random order)

- First pivot chosen is 7 $\rightarrow$ `PARTITION` splits array into:

$$\{1, 2, 3, 4, 5, 6\} \quad \text{and} \quad \{8, 9, 10\}$$

- 7 is compared with every element — it's the first pivot in each Z_{ij}
- 2 and 9 will **never be compared** because 7 splits them
- 7 and 9 **are compared** because 7 is the first pivot in $Z_{7,9}$

Takeaway: A pair (z_i, z_j) is only compared if no element in between was chosen as pivot first.

Lemma 7.3 – Probability of Comparing Two Elements

Let $z_1 < z_2 < \cdots < z_n$ be distinct elements in sorted order.

Lemma 7.3: For any $i < j$, the probability that z_i and z_j are compared during Randomized Quicksort is:

$$\boxed{\Pr[z_i \text{ compared with } z_j] = \frac{2}{j - i + 1}}$$

Why Are z_i and z_j Compared?

- Let $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$
- z_i and z_j are compared **iff** one of them is chosen first as pivot from Z_{ij}
- Each element in Z_{ij} has equal chance to be the first pivot

$$\Pr[\text{first pivot is } z_i] = \Pr[\text{first pivot is } z_j] = \frac{1}{j-i+1}$$

$$\Pr[z_i \text{ and } z_j \text{ are compared}] = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \boxed{\frac{2}{j-i+1}}$$

Theorem 7.4 – Expected Time of Randomized Quicksort

Theorem 7.4: The expected running time of `RANDOMIZED-QUICKSORT` on n distinct elements is:

$$\boxed{O(n \log n)}$$

Strategy: We'll compute the expected number of comparisons $\mathbb{E}[X]$, and use:

$$\text{Total time} = O(n + X) \quad \text{(Lemma 7.1)}$$

Step 1: Define Indicator Random Variables

Let elements be sorted: $z_1 < z_2 < \cdots < z_n$
Define:

$$X_{ij} = \begin{cases} 1 & \text{if } z_i \text{ is compared with } z_j \\ 0 & \text{otherwise} \end{cases} \quad \text{for } 1 \leq i < j \leq n$$

Then the total number of comparisons:

$$X = \sum_{1 \leq i < j \leq n} X_{ij}$$

We will compute $\mathbb{E}[X]$ using linearity of expectation.

Step 2: Apply Linearity of Expectation

$$\mathbb{E}[X] = \mathbb{E}\left[\sum_{1 \leq i < j \leq n} X_{ij}\right] = \sum_{1 \leq i < j \leq n} \mathbb{E}[X_{ij}]$$

From Lemma 5.1 (Chapter 5), $\mathbb{E}[X_{ij}] = \Pr[z_i \text{ compared with } z_j]$
And from Lemma 7.3:

$$\mathbb{E}[X_{ij}] = \frac{2}{j - i + 1}$$

So:

$$\mathbb{E}[X] = \sum_{1 \leq i < j \leq n} \frac{2}{j - i + 1}$$

Step 3: Change of Variables to Simplify the Sum

Change variable: let $k = j - i$, so:

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} < \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

The inner sum is a harmonic series:

$$\sum_{k=1}^{n} \frac{1}{k} = O(\log n)$$

So the total becomes:

$$\mathbb{E}[X] = O(n \log n)$$

Conclusion: Expected Running Time

We proved:

$$\mathbb{E}[X] = O(n \log n) \quad \text{(expected number of comparisons)}$$

And from Lemma 7.1:

$$\text{Expected running time of Quicksort} = O(n + X) = O(n \log n)$$

Quicksort is fast on average for any input!

Question?