CSE331 Introduction to Algorithm
Lecture 12: Quicksort

Antoine Vigneron
antoine@unist.ac.kr

Ulsan National Institute of Science and Technology

July 23, 2021
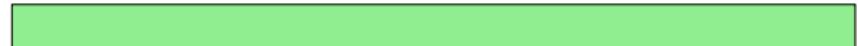
Introduction
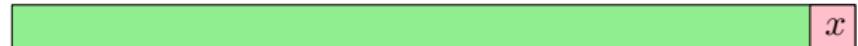
- Today, another sorting algorithm: QUICKSORT.
- A divide and conquer algorithm, but different from MERGE SORT
- An important algorithm as it is used in practice.
- Easy to implement, but analysis is not so easy.
- Reference: Chapter 7 of the textbook Introduction to Algorithms by Cormen, Leiserson, Rivest and Stein.

QUICKSORT: Overview

QUICKSORT: Overview

- If the pivot x falls in the middle (i.e. $q = \lfloor n/2 \rfloor$), then the running time satisfies $T(n) = 2T(n/2) + \Theta(n)$.
- So $T(n) = \Theta(n \log n)$. (Same as MERGE SORT.)
- This is the best case scenario. Unfortunately there is no simple way of doing it.
- We will see that a *random* choice for the pivot is a good option.

QUICKSORT

Pseudocode

1: **procedure** QUICKSORT($A[p \ldots r]$)
2:     **if** $p < r$ **then**
3:         $q \leftarrow$ PARTITION($A[p \ldots r]$)
4:         QUICKSORT($A[p \ldots q - 1]$)
5:         QUICKSORT($A[q + 1 \ldots r]$)

- In the next few slides, we describe two approaches for the PARTITION procedure.

Partitioning the Array
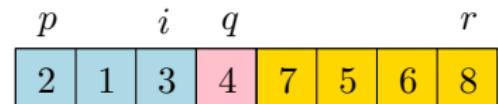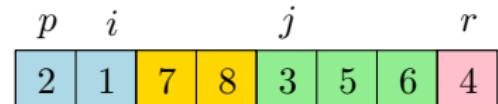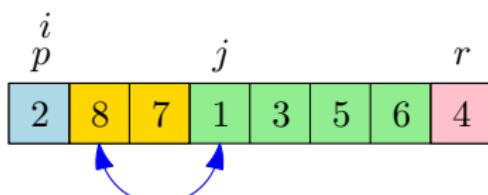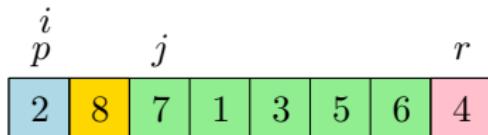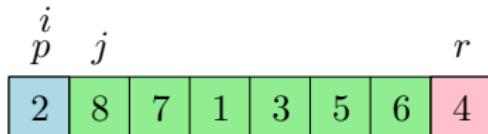
Pseudocode

1: **procedure** NAIVEPARTITION($A[p \ldots r]$)
2:      $x \leftarrow A[r]$ ▷ choose the pivot
3:      $L, R \leftarrow$ new arrays
4:      $j \leftarrow 1, \; k \leftarrow 1$
5:      **for** $i = p, r - 1$ **do**   ▷ fill arrays L and R
6:          **if** $A[i] \leqslant x$ **then**
7:              $L[j] \leftarrow A[i]$
8:              $j \leftarrow j + 1$
9:          **else**
10:              $R[k] \leftarrow A[i]$
11:              $k \leftarrow k + 1$
12:      $A[p \ldots p + j - 2] \leftarrow L[1 \ldots j - 1]$ ▷ copy L into A
13:      $A[p + j - 1] \leftarrow x$ ▷ place the pivot
14:      $A[p + j \ldots r] \leftarrow R[1 \ldots k - 1]$ ▷ copy R into A
15:      **return** $p + j - 1$

Partitioning the Array

- The naive approach above runs in linear time.
- The pivot is chosen to be $A[r]$, but this is not a strong restriction. We could pick it anywhere.
- Drawback of the naive approach:
  - It fills up two auxiliary arrays.
  - i.e. it is not *in place*.
- Next two slides: a shorter, in-place version, that also runs in linear time.

Partitioning the Array: Example

Partitioning the Array

Pseudocode

1: **procedure** PARTITION($A[p \ldots r]$)
2:     $x \leftarrow A[r]$ ▷ Choosing the pivot
3:     $i \leftarrow p - 1$
4:     **for** $j \leftarrow p, r - 1$ **do**
5:         **if** $A[j] \leqslant x$ **then**
6:             $i \leftarrow i + 1$
7:             Exchange $A[i]$ with $A[j]$
8:     Exchange $A[i + 1]$ with $A[r]$
9:     **return** $i + 1$ ▷ Returns the position of the pivot

Proof of Correctness

Loop invariants

At the beginning of each iteration of the loop, for any array index k,

1. If $p \leqslant k \leqslant i$, then $A[k] \leqslant x$.
2. If $i < k < j$, then $A[k] > x$.
3. If $k = r$, then $A[k] = x$.

- We need to prove Initialization, Maintenance, and Termination.
- Done in class. See textbook p. 173

Worst-Case Analysis

- The partition procedure runs in $\Theta(n)$ time.
- So the worst case running time $T(n)$ of QUICKSORT satisfies the relation:

$$T(n) \leqslant \max_{0 \leqslant q \leqslant n-1} (T(q) + T(n - q - 1)) + \Theta(n).$$

- Guess: $T(n) = \Theta(n^2)$.
- Using the substitution method, we prove that, $T(n) = O(n^2)$.
  - ▶ Done in class. See textbook p. 180.
- Example where $T(n) = \Omega(n^2)$:

$$A[1 \ldots n] = [1, 2, \ldots, n],$$

so QUICKSORT performs badly when the input is already sorted.

Randomized QUICKSORT

- In previous slide, we saw that QUICKSORT is slow when the pivot is the largest element in the array.
- An alternative is to pick a *random* pivot.
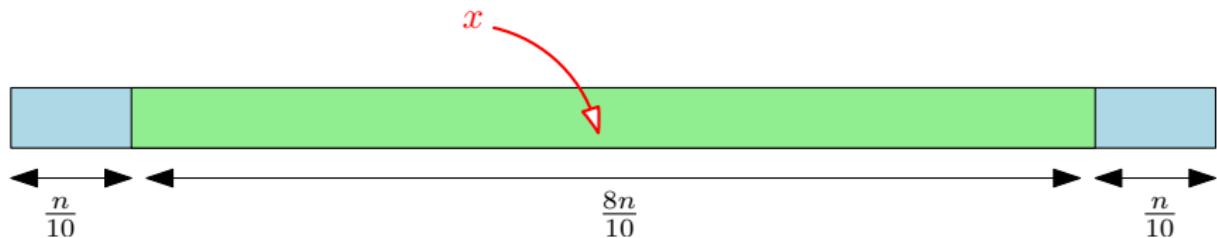
Pseudocode

```
1: procedure RANDOMIZEDQUICKSORT(A[p . . . r])
2:     if p < r then
3:         i ← RANDOM(p, r)
4:         exchange A[r] with A[i]
5:         q ← PARTITION(A[p . . . r])
6:         RANDOMIZEDQUICKSORT(A[p . . . q − 1])
7:         RANDOMIZEDQUICKSORT(A[q + 1 . . . r])
```

- Line 3: The integer i is chosen uniformly at random between p and r.

Randomized QUICKSORT
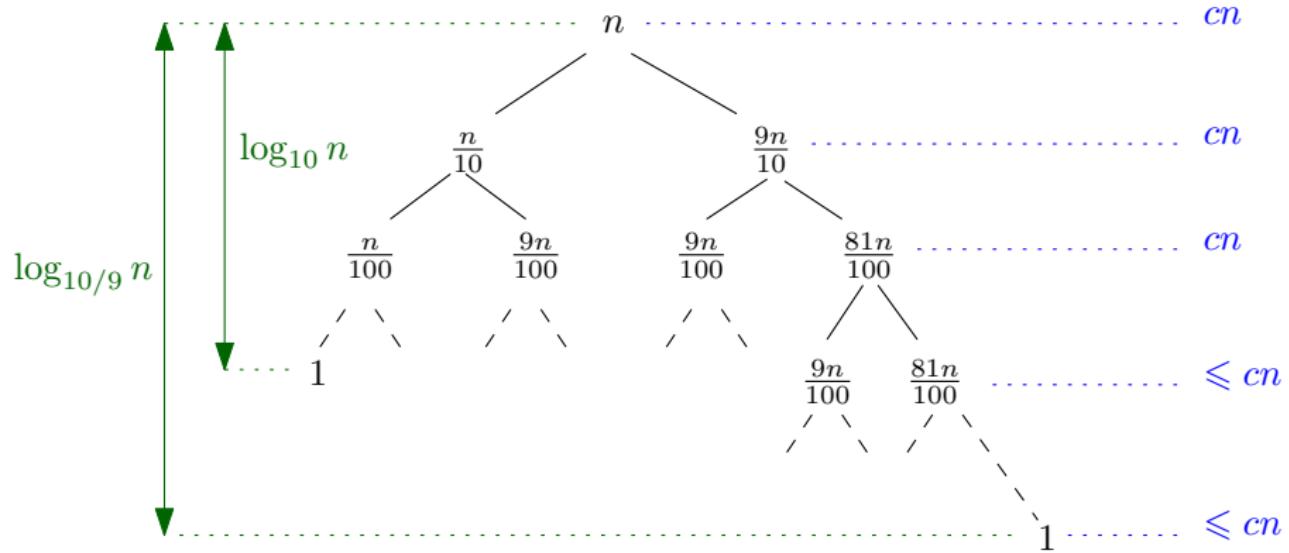
- As $A[r]$ is the pivot, the randomized QUICKSORT algorithm uses as pivot an element of the current array chosen uniformly at random.
- We now explain intuitively why the expected running time should be $O(n \log n)$.

- With probability $8/10$, the pivot falls in the green area.
- So the array is split into subarrays of sizes $\geqslant n/10$.
- We now apply the recursion tree method to

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

Randomized QUICKSORT

Total: $\Theta(n \log n)$

Randomized QUICKSORT

- Every level of the tree has cost $\leqslant cn$.
- Each level is full until depth $\log_{10} n = \Theta(\log n)$.
- The height is $\log_{10/9} n = \Theta(\log n)$.
- So the running time is $\Theta(n \log n)$, even in this very unbalanced case.
- Conjecture: The *expected* running time of randomized QUICKSORT is $\Theta(n \log n)$.
  - Here, "expected" running time means the average running time over all random choices of pivots.
  - It is not an average running time over a distribution of input arrays.
- We will prove it rigorously in the next lecture, while studying a related algorithm.
  - Requires the assumption that input numbers are distinct. (Why?)

C Code (source: *The C programming language* by Kernighan & Ritchie)

```c
void qsort(int v[], int left, int right){
   int i, last;
   void swap(int v[], int i, int j);
   if (left >= right)
      return;
   swap(v, left, (left + right)/2);
   last = left;
   for (i = left + 1; i <= right; i++)
      if (v[i] < v[left])
         swap(v, ++last, i);
   swap(v, left, last);
   qsort(v, left, last-1);
   qsort(v, last+1, right);
}
void swap(int v[], int i, int j){
   int temp; temp = v[i]; v[i] = v[j]; v[j] = temp;
}
```

Conclusion

- QUICKSORT is simple and practical, but its analysis is non-trivial.
- The randomized version runs in expected $\Theta(n \log n)$ time *on any input*.
- It can be shown that this bound holds with high probability.
- It is often the case that simple randomized algorithms can be found for solving a given problem. It is an attractive choice for programmers as these algorithms are easier to implement and faster in practice.
- The deterministic version of QUICKSORT runs in quadratic time on some input, for instance sorted input.
- QUICKSORT is in-place (as opposed to MERGE SORT).