Faculty of Information Technology, Monash University

FIT2004: Algorithms and Data Structures

Week 3: Quick Sort and its Analysis

Lecturer: Reza Haffari

These slides are prepared by M. A. Cheema and are based on the material developed by Arun Konagurthu and Lloyd Allison.

Things to note/remember

- Consultations
  - Reza: Thursday 4:15pm-5PM
  - Vishwajeet: Wednesday 2PM-3PM
  - Tharindu: Friday 1PM-2PM

- Assignment 1
  - Do not forget add **docstring** to each of function, shown in announcement.
  - Deadline 24-March-2019 23:55:00

- Assignment 2 to be released later next week
  - Requires dynamic programming (taught in week 4) – don't miss the lecture
  - Deadline 7-April-2019 23:55:00

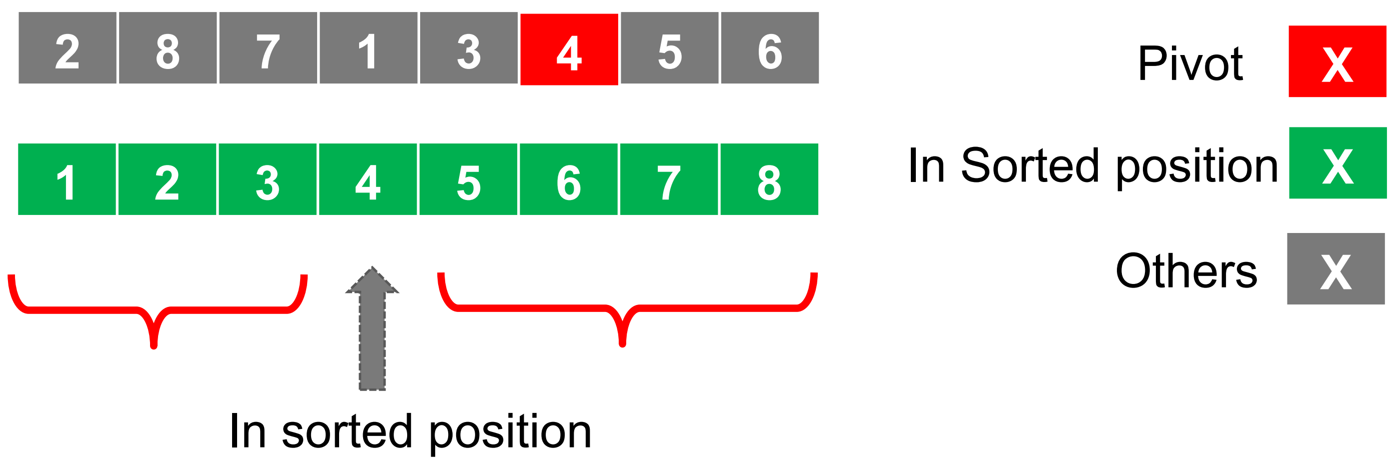
Quick Sort and its Analysis

1. Algorithm
2. Complexity Analysis
3. Improving Worst-case complexity
   A. Quick Select
   B. Quick Sort in O(N log N) worst-case

Quicksort

Partitioning

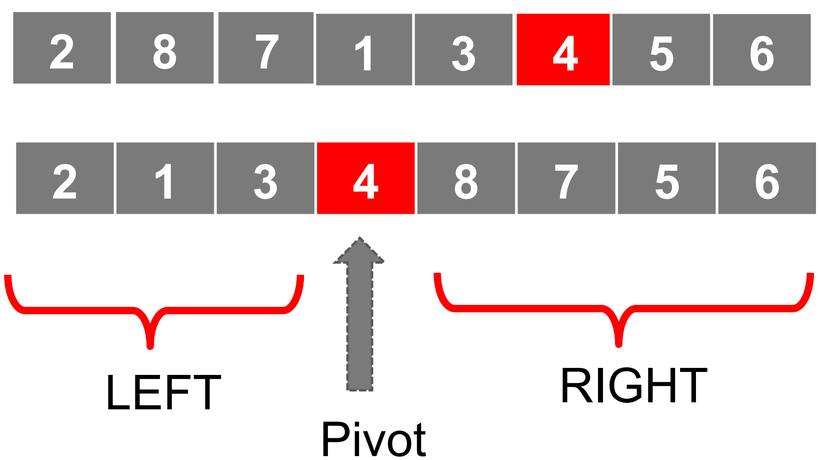
- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p
- QuickSort(LEFT)
- QuickSort(RIGHT)



Partitioning: An out-of-place version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy {LEFT, pivot, RIGHT} to the array

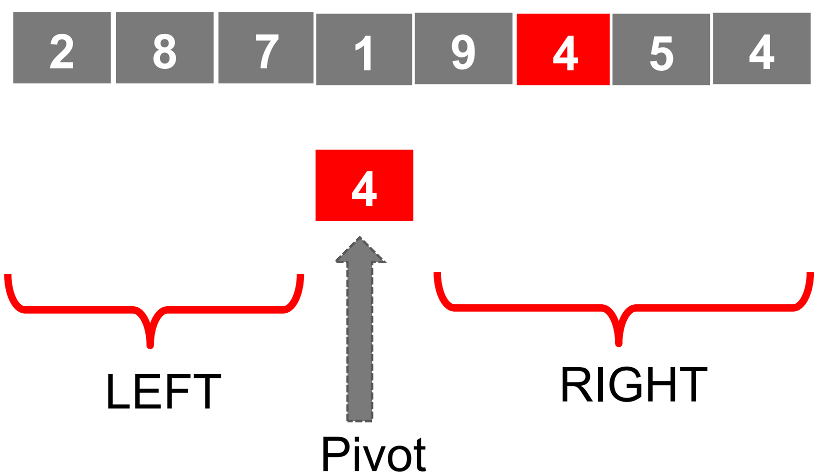
This is clearly not in-place.
Will this result in stable sorting?



Partitioning: An out-of-place version

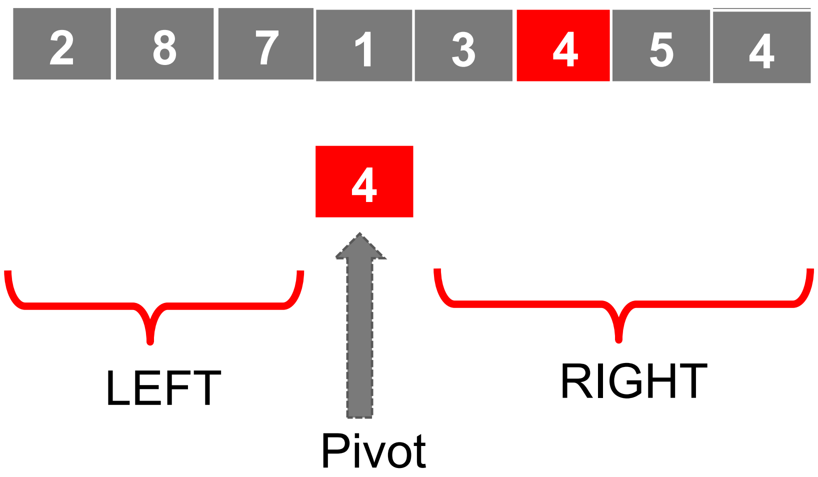
- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy {LEFT, pivot, RIGHT} to the array

This version is unstable but it can be made stable!



Partitioning: A stable version

- Initialize two lists LEFT and RIGHT
- For each element e (except pivot)
  - If e ≤ pivot
    - If e == pivot and e.index > pivot.index
      - Insert e in RIGHT
    - Else
      - Insert e in LEFT
  - If e > pivot
    - Insert e in RIGHT
- Copy {LEFT, pivot, RIGHT} to the array



In-Place Partitioning

- num ← the number of elements smaller than or equal to pivot O(N)
- Swap pivot with element at num
- Repeat until no bad element is found O(N)
  - Find a bad element (LBAD) on the L.H.S. of pivot
  - Find a bad element (RBAD) on the R.H.S. of pivot
  - Swap LBAD and RBAD

2 1 3 4 7 8 5 6

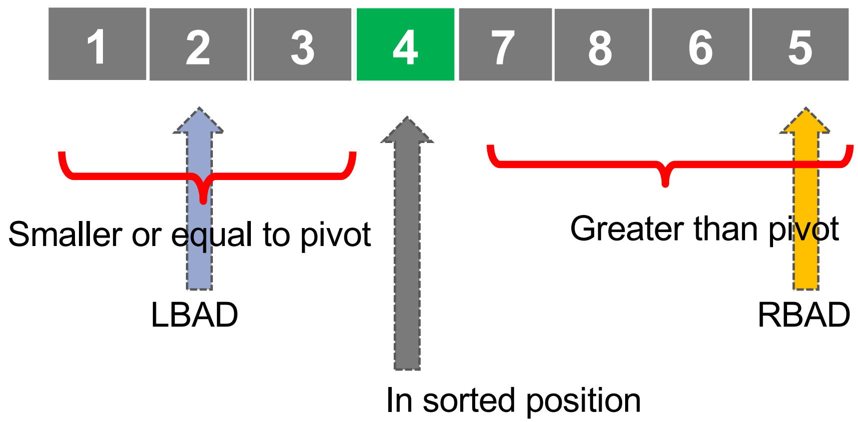
LBAD

RBAD

In sorted position

In-Place Partitioning (Improved)

- Swap pivot with the left most element
- LBAD points to the second element from left
- RBAD points to the right most element
- Repeat until LBAD “crosses” RBAD
  - Move LBAD towards right until it points to an element e > pivot
  - Move RBAD towards left until it points to an element e ≤ pivot
  - Swap elements pointed by LBAD and RBAD
- Swap pivot with the element pointed by RBAD



This partitioning is in-place but unstable.

Python Implementation

Review it at your own time

```
def partition(alist,first,last):
    pivot = alist[first]
    LBAD = first+1
    RBAD = last-1
    # continue until pointers cross
    while LBAD <= RBAD:
        # move LBAD until it points to a bad element or crosses RBAD
        while LBAD <= RBAD and alist[LBAD] <= pivot:
            LBAD = LBAD + 1
        # move RBAD until it points to a bad element or crosses LBAD
        while LBAD<=RBAD and alist[RBAD] > pivot:
            RBAD = RBAD - 1
        #only swap if they have not crossed
        if LBAD <= RBAD:
            # Python shorthand for swapping
            alist[LBAD],alist[RBAD] = alist[RBAD],alist[LBAD]
    # if they have crossed, swap element at RBAD with element at pivot
    alist[first],alist[RBAD] = alist[RBAD],alist[first]
    return RBAD # return pivot position at after partitioning
```

Python Implementation

```
def quickSort(alist,first,last):
    # we need to sort only if the list contains at least two elements
    if (last - first) > 1:
        # partition the list from first to last (exclusive)
        print("partitioning", alist[first:last], "pivot",alist[first])
        pivot_pos = partition(alist,first,last)
        print("partitioned:", alist[first:last])

        # recursively sort the two halves of the list
        print("Splitting into two", alist[first:pivot_pos],alist[pivot_pos+1:last])
        quickSort(alist,first,pivot_pos)
        quickSort(alist,pivot_pos+1,last)

alist = [26,93,44,20,77,31,36,28, 55,17]
quickSort(alist,0,len(alist))
print(alist)
```

Review at your own time

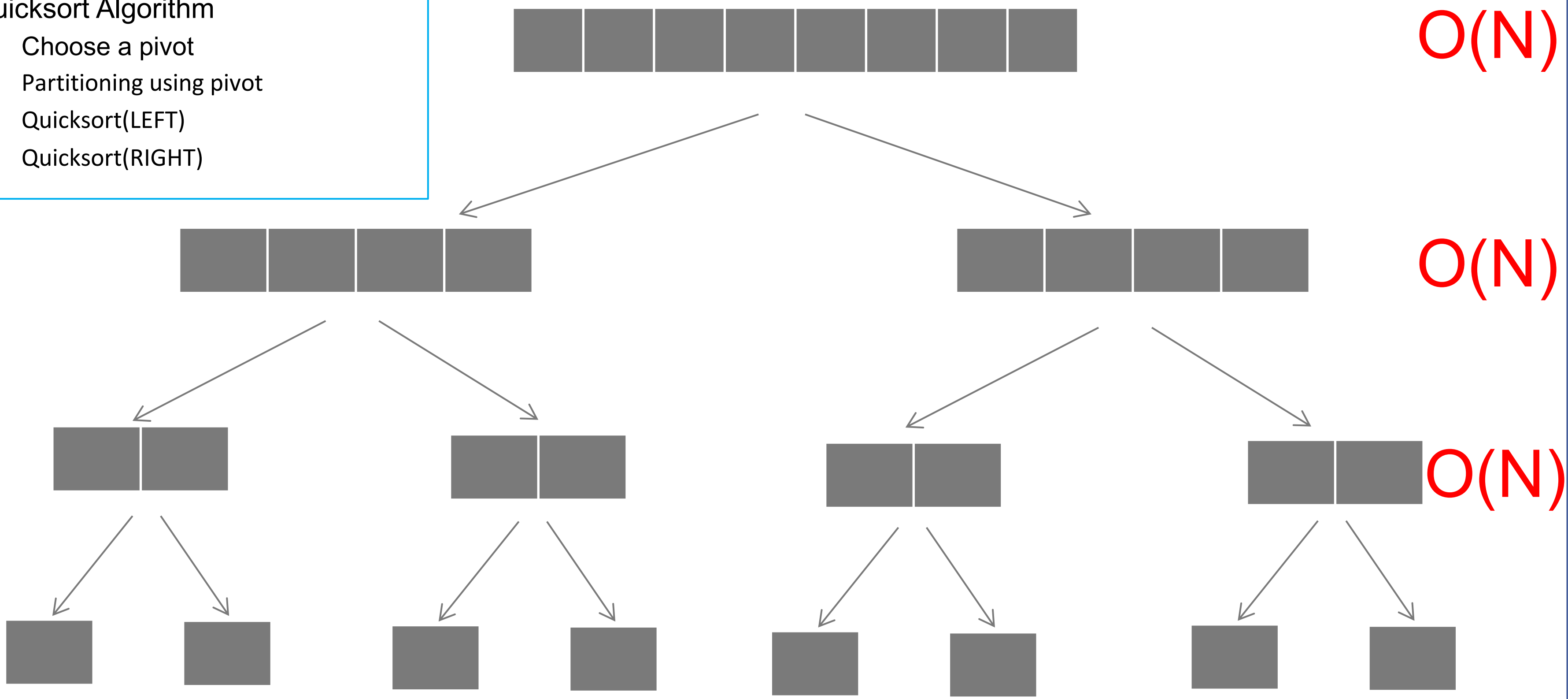
Quick Sort and its Analysis

1. Algorithm
2. Complexity Analysis
3. Improving Worst-case complexity
   A. Quick Select
   B. Quick Sort in O(N log N) worst-case

Best-case time complexity

Quicksort Algorithm

- Choose a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

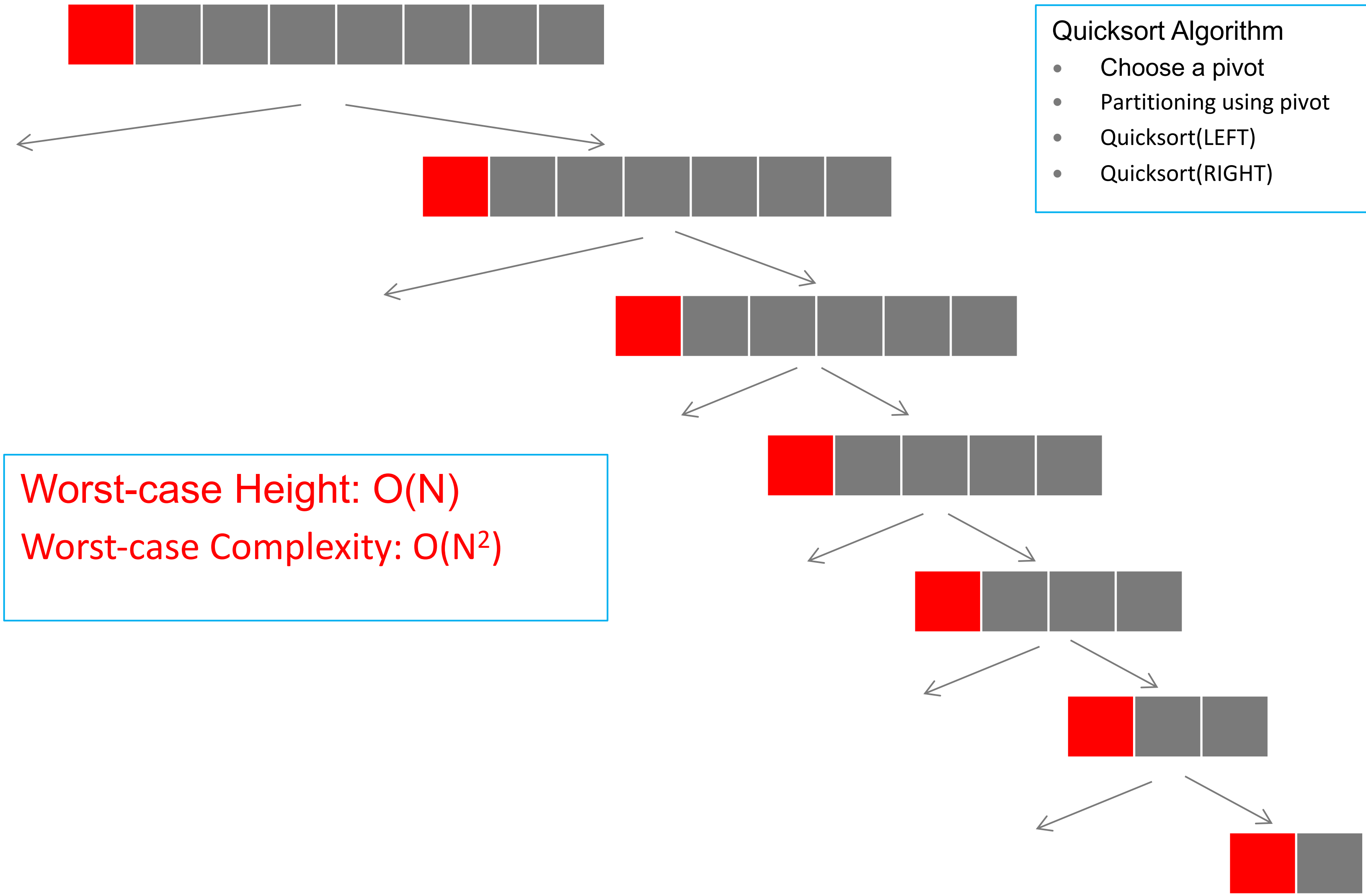


Best-case Height: O(log N)
Best-case complexity: O(N log N)

Important: Quicksort is not in-place even when in-place partitioning is used. Why?

Recursion depth is at least O(log N)

Worst-case Time Complexity



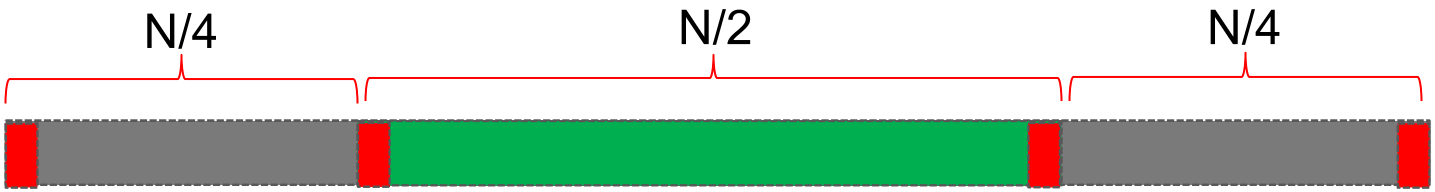
Quicksort Algorithm

- Choose a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

Worst-case Height: $O(N)$

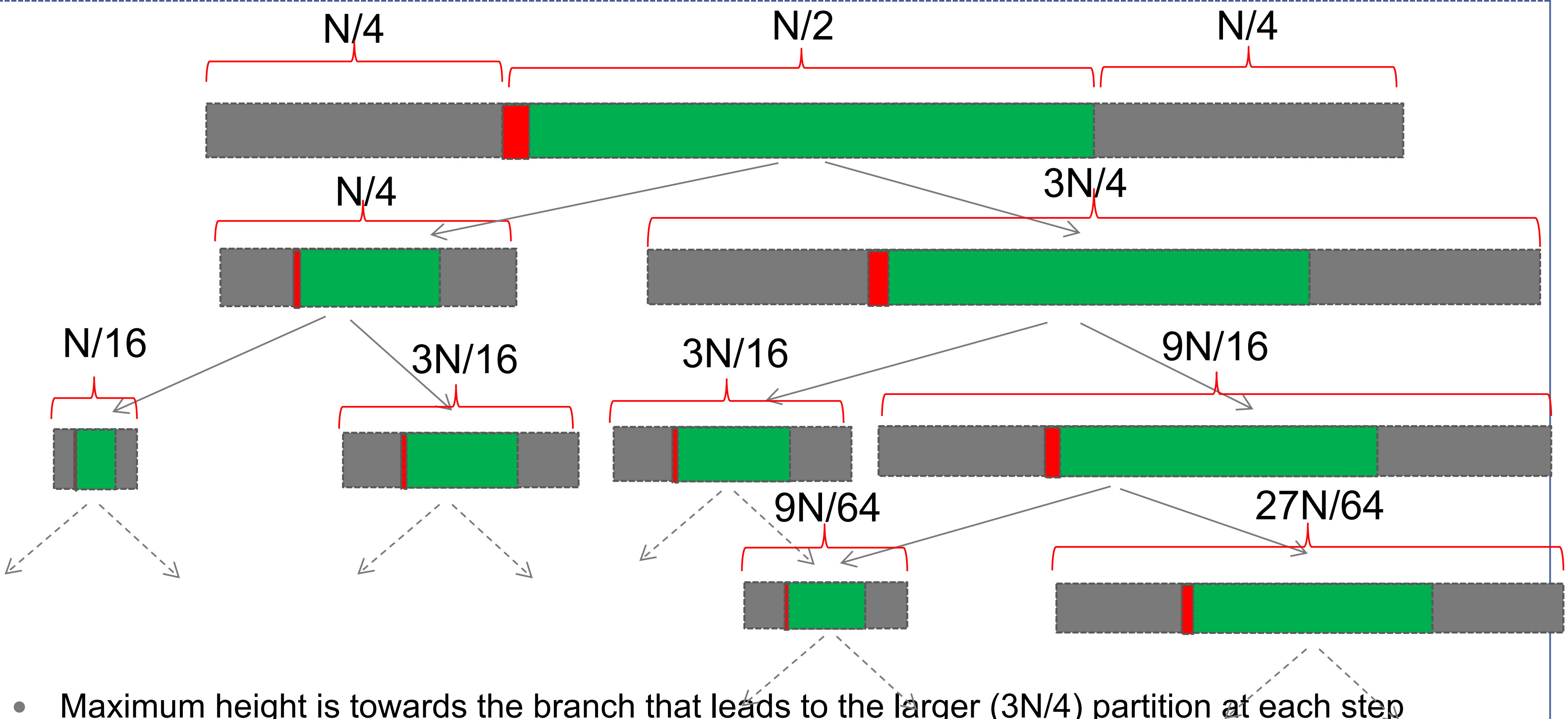
Worst-case Complexity: $O(N^2)$

Average-case Time complexity



- After partitioning, pivot has 50% probability to be in the green sub-array and has 50% probability to be in one of the two grey sub-arrays.
  - i.e., on average, pivot will be in green half of the times and in grey half of the times
- If pivot is in grey sub-array
  - The worst-case (most unbalanced) partition sizes will be 1 and N-1
- If pivot is in green sub-array
  - The worst-case partition sizes will be N/4 and 3N/4
- Let h be the height when pivot is **always** in green.
- Let's see what is h???

Height when pivot always in green



- Maximum height is towards the branch that leads to the larger (3N/4) partition at each step
- At level h, the size of the larger partition is $(3/4)^h$ N
- Partitioning stops when the size is 1, i.e, at level h such that $(3/4)^h N = 1$
- $(3/4)^h N = 1 \rightarrow N = (4/3)^h \rightarrow h = \log_{4/3} N$
- Therefore, the maximum height when pivot is **always** in green is $\log_{4/3} N$

Average case Time complexity

- So, the height h when pivot is always in green is $\log_{4/3} N$
- What is the maximum possible height when pivot is in green only half the times?
  - The height when pivot is in green only half the times (average case) is $2. \log_{4/3} N$
- Therefore, height in average case is O(log N)
- Like before, the cost at each level is O(N)
- The average case complexity is thus O(N log N)

Is $O(\log_a N) = O(\log_b N)$ where a and b are constants?

$$\log_a N = \frac{\log_b N}{\log_b a}$$

Best-case time complexity using recurrence

Quicksort Algorithm
- Choose a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

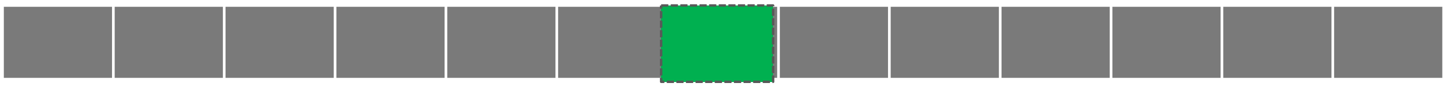
Recurrence relation:

$$T(1) = b$$

$$T(N) = c*N + T(N/2) + T(N/2) = 2*T(N/2) + c*N$$

Solution (exercise in last week):

$$O(N \log N)$$



Worst-case complexity using recurrence

Quicksort Algorithm
- Choose a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

Recurrence relation:

$$T(1) = b$$

$$T(N) = T(N-1) + c*N$$

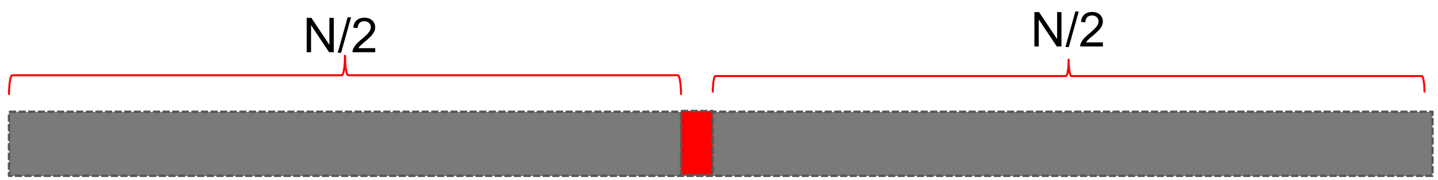
Solution:

$$O(N^2)$$

Quick Sort and its Analysis

1. Algorithm
2. Complexity Analysis
3. Improving Worst-case complexity
   A. Quick Select
   B. Quick Sort in O(N log N) worst-case

Quicksort with O(N log N) in worst-case



Quicksort Algorithm
- Choose median as a pivot
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

Idea:

- Don’t choose pivot randomly!
  - Instead, always choose median as the pivot.
  - If we can find median in O(N), the worst-case cost of quicksort would be?
    - O(N log N)
- How do we choose median in O(N)?
- First, we take a detour and see algorithms to answer k-th order statistics

K-th Order Statistics

- Problem: Given an **unsorted** array, find k-th smallest element in the array
  - If k=1 (i.e., find the smallest), we can easily do this in O(N) using the linear algorithm we saw in the last week.
- Median can be computed by setting k appropriately (e.g., k = len(array)/2)
- For general k, how can we solve this efficiently?
  - Sort the elements and return k-th element – takes O(N log N)
  - Can we do better?
    - Yes, Quick Select

Quick Sort and its Analysis

1. Algorithm
2. Complexity Analysis
3. Improving Worst-case complexity
   A. Quick Select
   B. Quick Sort in O(N log N) worst-case

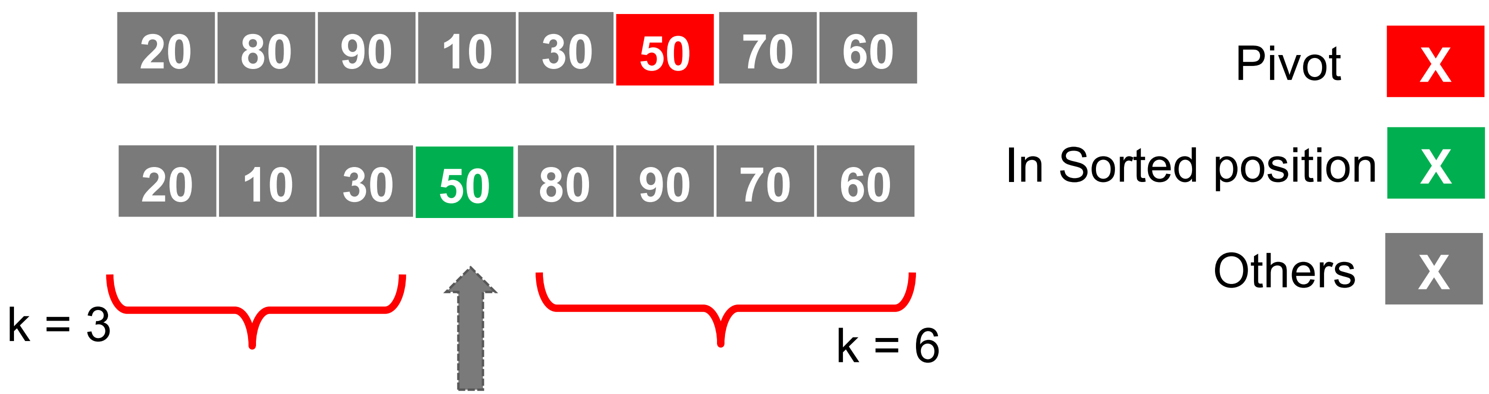
Quick Select

- Choose a pivot p
- Partition the array in two sub-arrays w.r.t. p (same partitioning as in quicksort)
  - LEFT ← elements smaller than or equal to p
  - RIGHT ← elements greater than p
- If index(pivot) == k:
  - Return pivot
- If k > index(pivot)
  - QuickSelect(RIGHT)
- Else:
  - QuickSelect(LEFT)

Best-case time complexity?
- O(N)

Worst-case time complexity?
- $O(N^2)$

Average-case time complexity?
- O(N) – same arguments as for quicksort

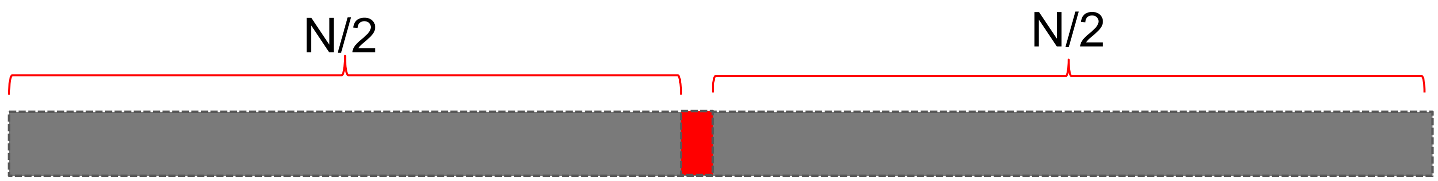


In sorted position (at index 4, i.e., 4th smallest)

Quick Sort and its Analysis

1. Algorithm
2. Complexity Analysis
3. Improving Worst-case complexity
   A. Quick Select
   B. Quick Sort in O(N log N) worst-case

Quicksort with O(N log N) in worst-case



Quicksort Algorithm
- Use quick select to find **median**
- Partitioning using median as pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

- Call Quick Select with k=len(array)/2?
- The value returned by Quick Select will be median.
- Choose this as the pivot.
- What will be the best-case cost of such quick sort?
  - O(N log N)
- What is the worst-case cost?

Quick Sort Worst-case when using Quick Select to choose pivot

Quicksort Algorithm

- Use quick select to find **me**
- Partitioning using median as
- Quicksort(LEFT)
- Quicksort(RIGHT)

N^2

N $^2/4$

$N^2/2$

$N^2/16$ $N^2/16$

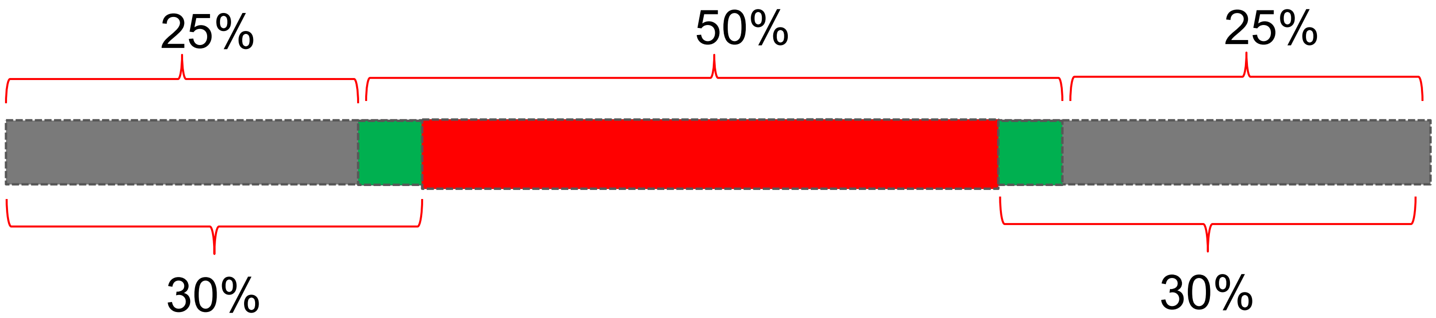
$N^2/4$

Worst-case cost

Total cost: $N^2 + N^2/2 + N^2/4 + \ldots + 1 = N^2(1 + ½ + ¼ + \ldots)$

$= O(N^2)$

Quicksort with O(N log N) in worst-case



Idea:

- Instead of choosing median, we relax the criteria and find a pivot that is guaranteed to be in the green sub-array.
- If we can find such a pivot in O(N) worst-case, what will be the cost of quick sort in the worst-case?
  - O(N log N) – similar arguments as in average case analysis
- **Median of medians** algorithm takes O(N) in worst-case and returns an element that is greater than 30% elements and smaller than 30% elements in the array.
  - i.e., it can be used to find a pivot in green sub-array in O(N)
  - Using this algorithm, the worst-case cost of quicksort is O(N log N)
    - Yay !!!!
    - Note: Median of medians algorithm uses quickselect as a subroutine
- We will not cover median of medians algorithm in this unit – but it is worth looking at.

Quicksort Algorithm

- Choose a pivot in green sub-array
- Partitioning using pivot
- Quicksort(LEFT)
- Quicksort(RIGHT)

Anticlimax

- Although using “median of medians” reduces worst-case complexity to O(N log N), in practice choosing random pivots works better.
  - However, theoretical improvement in worst-case is quite satisfying.

Concluding Remarks

Summary

- Quicksort and its analysis. Quicksort can be made O(N log N) in worst-case which is mostly of theoretical interest but does not usually improve performance in practice.

Coming Up Next

- Dynamic Programming – (super important and powerful tool, assignment 2 is all about dynamic programming)

Things to do <u>before</u> next lecture

- Make sure you understand this lecture completely especially the average-case complexity analysis of quicksort