CSE 221: Algorithms

Quicksort

Mumit Khan

Computer Science and Engineering
BRAC University

References

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
2. Erik Demaine and Charles Leiserson, *6.046J Introduction to Algorithms*. MIT OpenCourseWare, Fall 2005. Available from: `ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-046JFall-2005/CourseHome/index.htm`

Last modified: June 21, 2009

Contents

Contents

Quicksort

- Proposed by C. A. R. Hoare in 1962.

Quicksort

- Proposed by C. A. R. Hoare in 1962.
- Divide and Conquer algorithm – like merge sort.

Quicksort

- Proposed by C. A. R. Hoare in 1962.
- Divide and Conquer algorithm – like merge sort.
- In-place algorithm – like insertion and heap sorts.

Quicksort

- Proposed by C. A. R. Hoare in 1962.
- Divide and Conquer algorithm – like merge sort.
- In-place algorithm – like insertion and heap sorts.
- Runs very well with tuning.

Quicksort

- Proposed by C. A. R. Hoare in 1962.
- Divide and Conquer algorithm – like merge sort.
- In-place algorithm – like insertion and heap sorts.
- Runs very well with tuning.
- Worst-case is $O(n^2)$, but for all practical purposes runs in $O(n \lg n)$.

Quicksort

- Proposed by C. A. R. Hoare in 1962.
- Divide and Conquer algorithm – like merge sort.
- In-place algorithm – like insertion and heap sorts.
- Runs very well with tuning.
- Worst-case is $O(n^2)$, but for all practical purposes runs in $O(n \lg n)$.

Why do we want to study Quicksort?

One of the most widely used, and extensively studied, sorting algorithms.

Divide and conquer

Quicksort an *n*-element array:

1. *Divide* Partition the array into subarrays around a *pivot* x such that the elements in lower subarray $\leq x \leq$ elements in the upper subarray.

$\leqslant x$	x	$\geqslant x$

2. *Conquer* Recursively sort the two subarrays.
3. *Combine* Trivial – just concatenate the lower subarray, *pivot*, and the upper subarray.

Divide and conquer

Quicksort an *n*-element array:

1. *Divide* Partition the array into subarrays around a *pivot* x such that the elements in lower subarray $\leq x \leq$ elements in the upper subarray.

2. *Conquer* Recursively sort the two subarrays.
3. *Combine* Trivial – just concatenate the lower subarray, *pivot*, and the upper subarray.

Divide and conquer

Quicksort an *n*-element array:

1. *Divide* Partition the array into subarrays around a *pivot* x such that the elements in lower subarray $\leq x \leq$ elements in the upper subarray.

2. *Conquer* Recursively sort the two subarrays.
3. *Combine* Trivial – just concatenate the lower subarray, *pivot*, and the upper subarray.

Divide and conquer

Quicksort an *n*-element array:

1. *Divide* Partition the array into subarrays around a *pivot* x such that the elements in lower subarray $\leq x \leq$ elements in the upper subarray.

2. *Conquer* Recursively sort the two subarrays.
3. *Combine* Trivial – just concatenate the lower subarray, *pivot*, and the upper subarray.

Divide and conquer

Quicksort an n-element array:

1. *Divide* Partition the array into subarrays around a *pivot* x such that the elements in lower subarray $\leq x \leq$ elements in the upper subarray.

2. *Conquer* Recursively sort the two subarrays.
3. *Combine* Trivial – just concatenate the lower subarray, *pivot*, and the upper subarray.

Key

Linear-time partitioning algorithm.

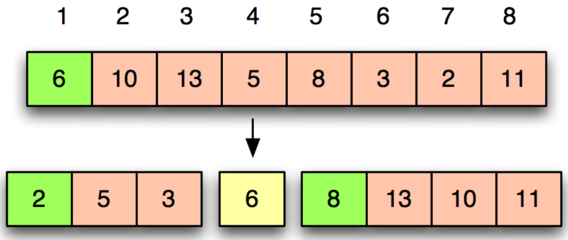
Quicksort in action

1	2	3	4	5	6	7	8
6	10	13	5	8	3	2	11

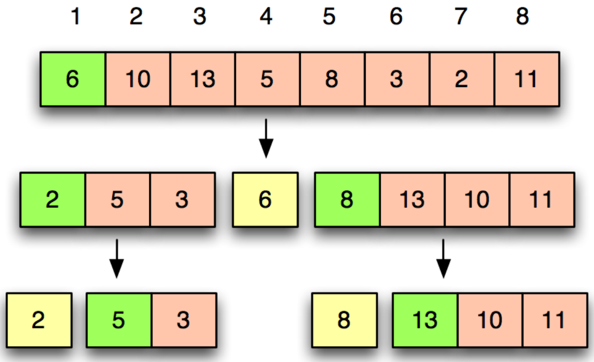
Quicksort in action

1	2	3	4	5	6	7	8
6	10	13	5	8	3	2	11

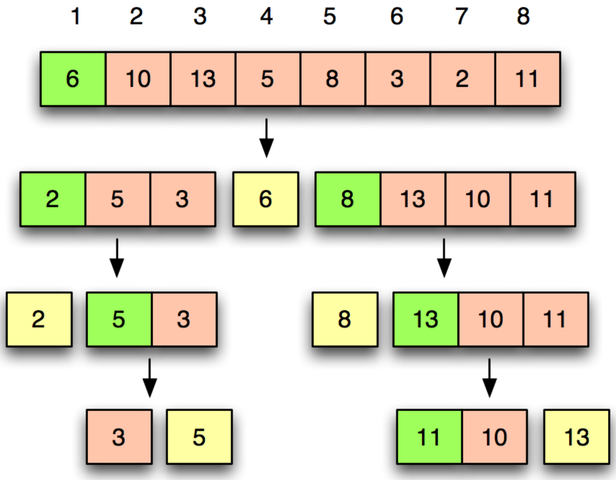
Quicksort in action



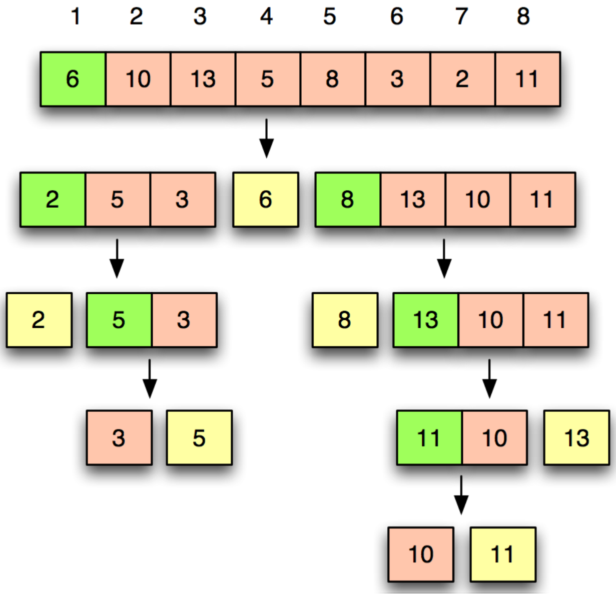
Quicksort in action



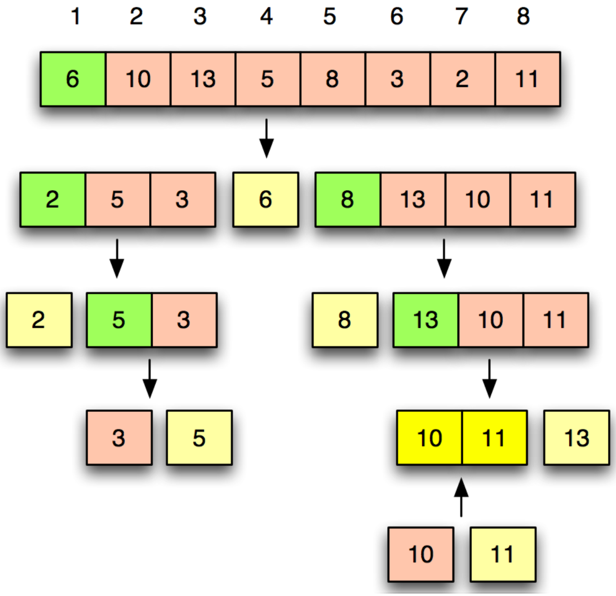
Quicksort in action



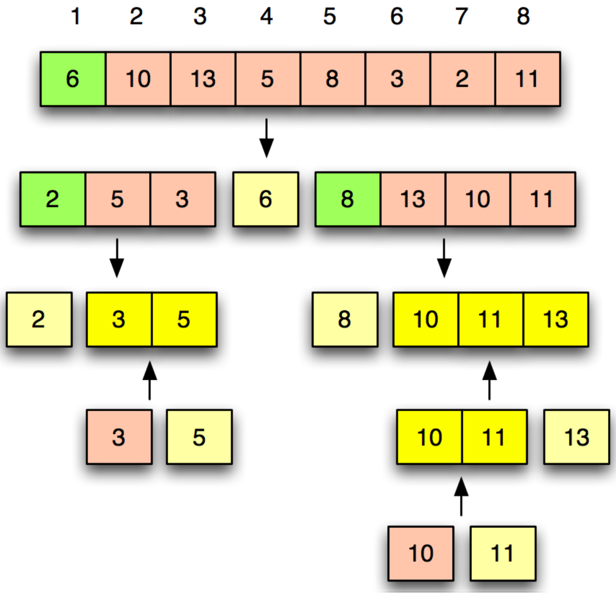
Quicksort in action



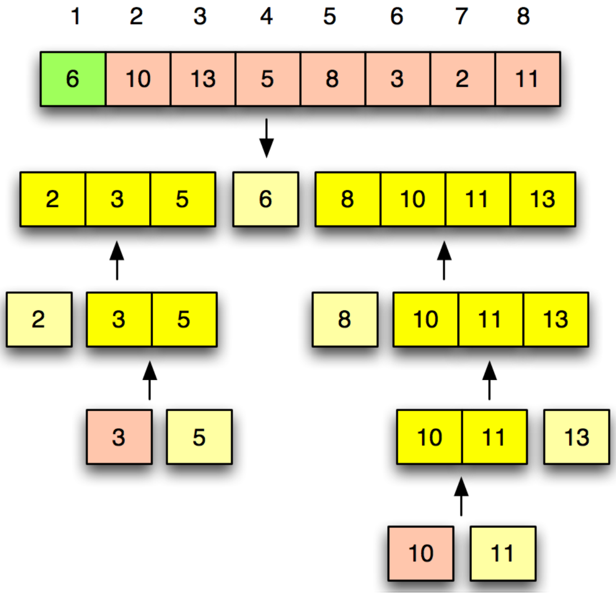
Quicksort in action



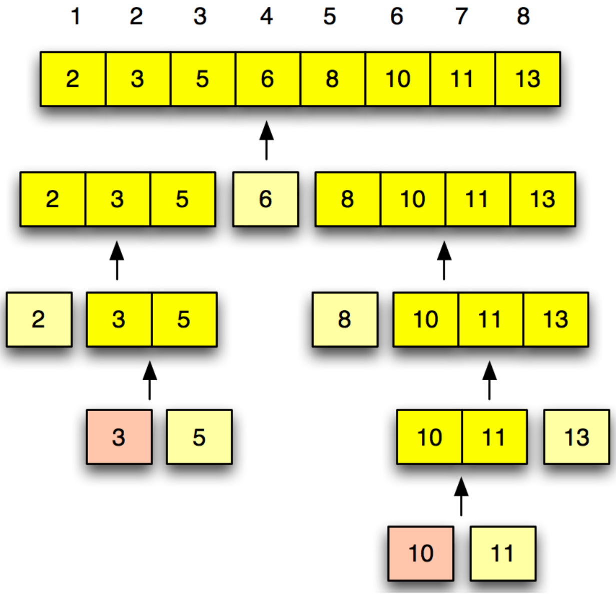
Quicksort in action



Quicksort in action



Quicksort in action



Contents

Partitioning algorithm

Algorithm

```
PARTITION(A, p, q) ▷ A[p . . q]
x ← A[p]                ▷ pivot = A[p]
i ← p
for j ← p + 1 to q
        do if A[j] ≤ x
                then i ← i + 1
                     exchange A[i] ↔ A[j]
exchange A[p] ↔ A[i]
return i
```

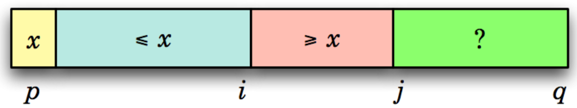
Partitioning algorithm

Algorithm

$\textsc{Partition}(A, p, q) \triangleright A[p \mathinner{.\,.} q]$

```
x ← A[p]                ▷ pivot = A[p]
i ← p
for j ← p + 1 to q
      do if A[j] ≤ x
            then i ← i + 1
                  exchange A[i] ↔ A[j]
exchange A[p] ↔ A[i]
return i
```

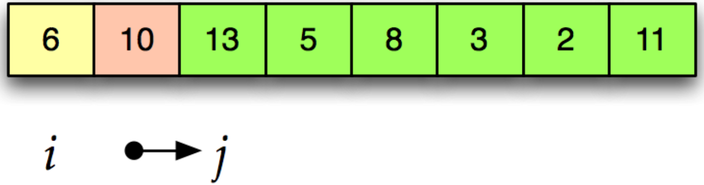
Invariant



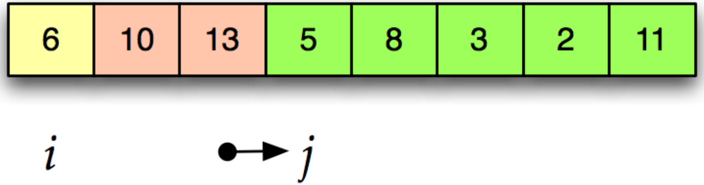
Partitioning in action

6	10	13	5	8	3	2	11
i	j						

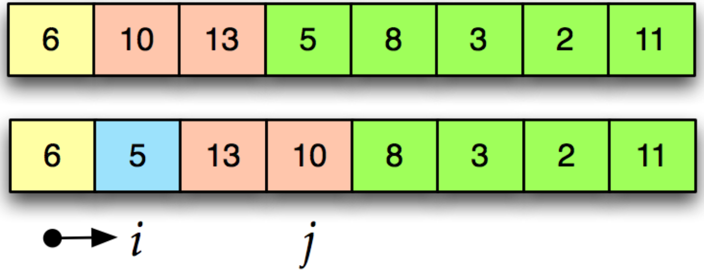
Partitioning in action



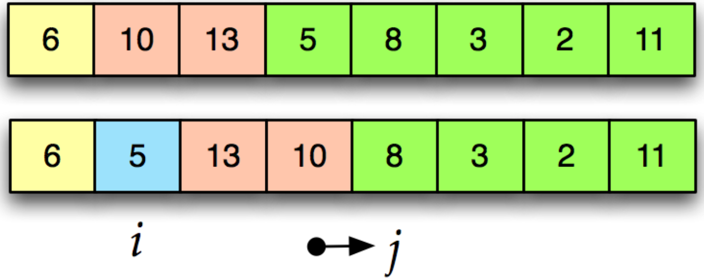
Partitioning in action



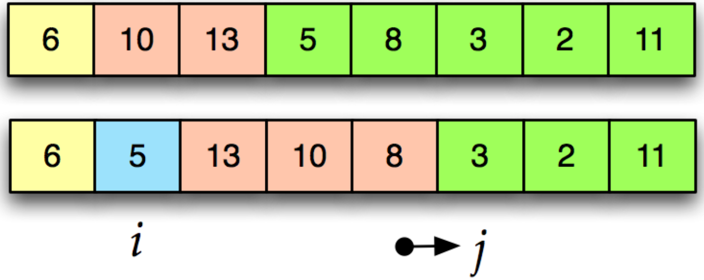
Partitioning in action



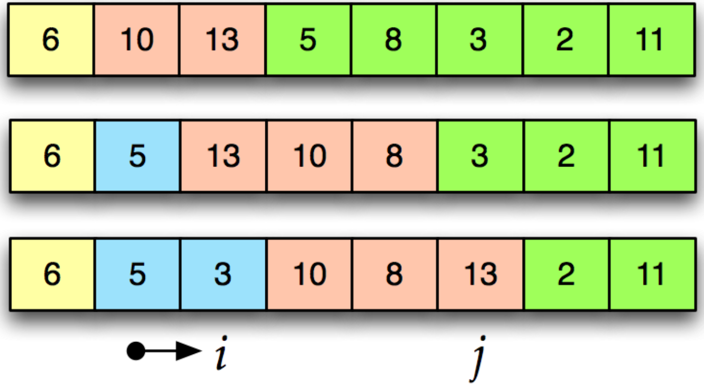
Partitioning in action



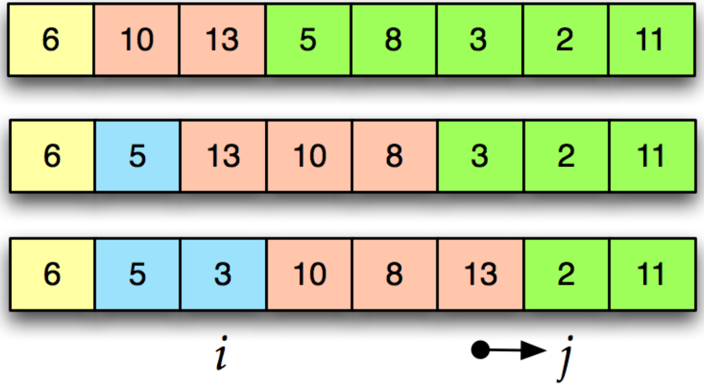
Partitioning in action



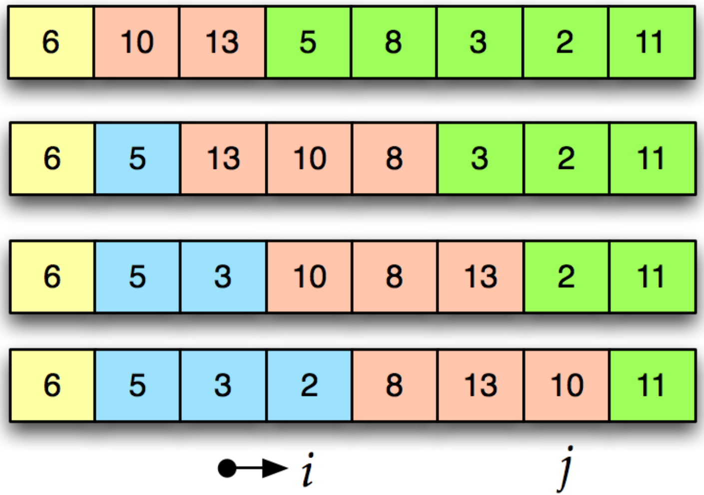
Partitioning in action



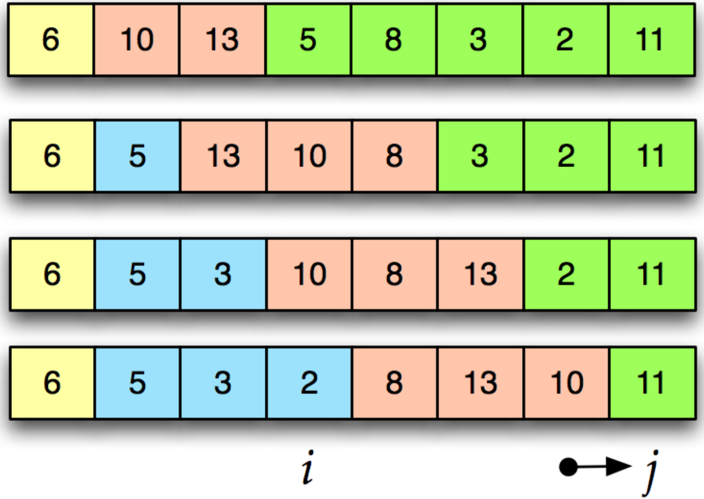
Partitioning in action



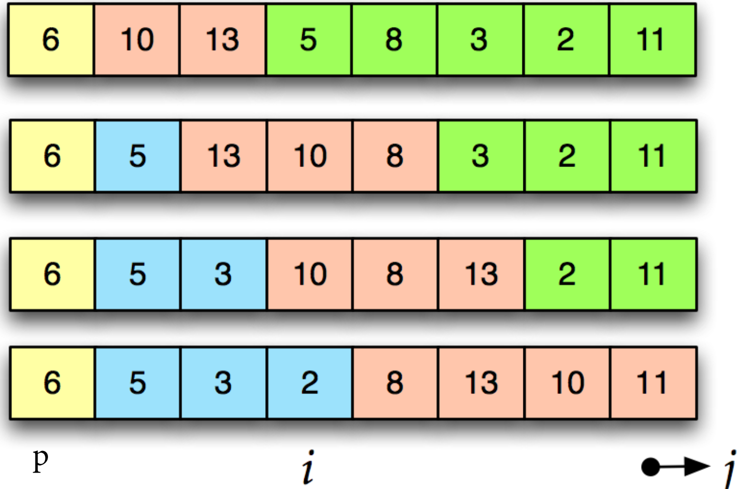
Partitioning in action



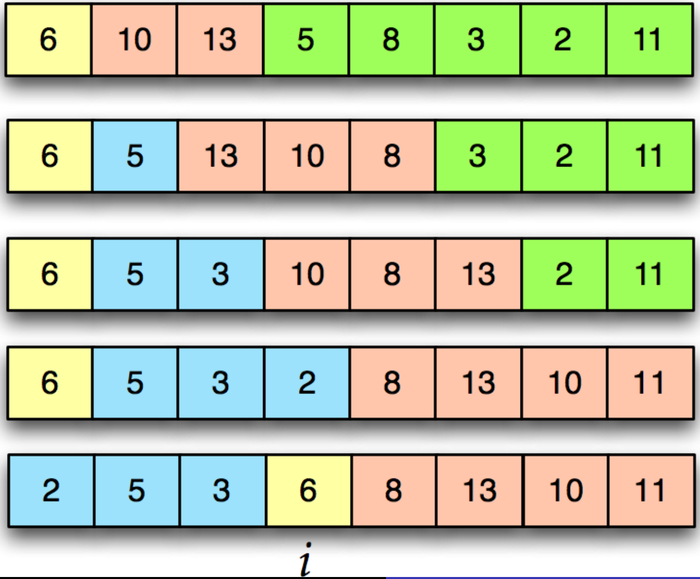
Partitioning in action



Partitioning in action



Partitioning in action



Contents

Quicksort algorithm

Algorithm

$\textsc{Quicksort}(A, p, r) \triangleright A[p \,..\, r]$

```
if p < r
   then q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```


Quicksort algorithm

Algorithm

$\textsc{Quicksort}(A, p, r) \triangleright A[p \,..\, r]$

```
if p < r
   then q ← PARTITION(A, p, r)
        QUICKSORT(A, p, q − 1)
        QUICKSORT(A, q + 1, r)
```

Initial call

$\textsc{Quicksort}(A, 1, n)$

Contents

Analyzing Quicksort - worst-case performance

- Worst-case happens when pivot is always the minimum or maximum element.

Analyzing Quicksort - worst-case performance

- Worst-case happens when pivot is always the minimum or maximum element.
- Result is that one of the partitions is always empty.

Analyzing Quicksort - worst-case performance

- Worst-case happens when pivot is always the minimum or maximum element.
- Result is that one of the partitions is always empty.
- When?

Analyzing Quicksort - worst-case performance

- Worst-case happens when pivot is always the minimum or maximum element.
- Result is that one of the partitions is always empty.
- When? Input sorted (either non-decreasing or non-increasing)

Analyzing Quicksort - worst-case performance

- Worst-case happens when pivot is always the minimum or maximum element.
- Result is that one of the partitions is always empty.
- When? Input sorted (either non-decreasing or non-increasing)

Worst-case analysis

(Note: the worst-case running time for partitioning is $\Theta(n)$.)

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \boxed{\Theta(n^2)} \end{aligned}$$

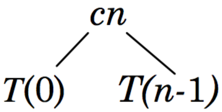
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$

$$T(n)$$

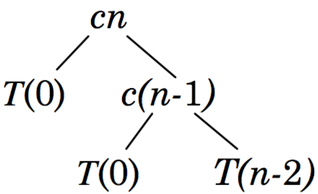
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



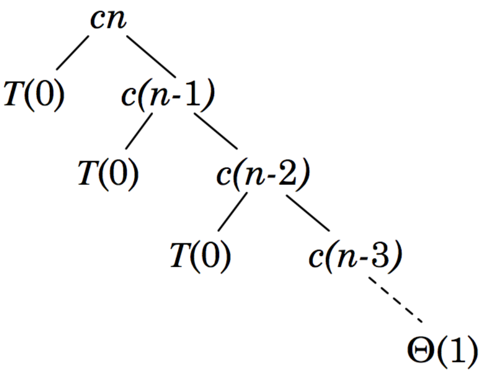
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



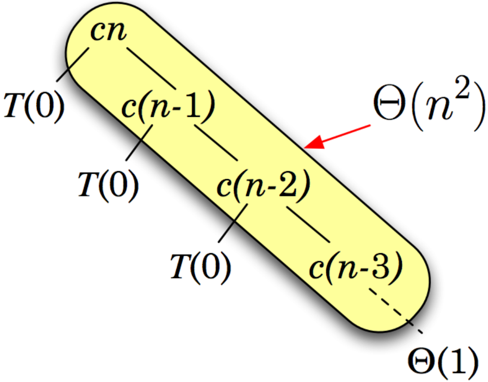
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



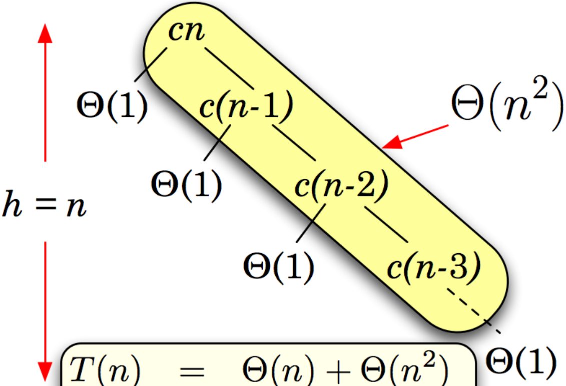
Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + cn$$



$$\begin{aligned} T(n) &= \Theta(n) + \Theta(n^2) \\ &= \Theta(n^2) \end{aligned}$$

Best- and almost-worst case performances

- Best-case happens when pivot is the median element, creating equal size partitions.

Best- and almost-worst case performances

- Best-case happens when pivot is the median element, creating equal size partitions.

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

Best- and almost-worst case performances

- Best-case happens when pivot is the median element, creating equal size partitions.

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

- What if the split is always $\frac{1}{10} : \frac{9}{10}$?

Best- and almost-worst case performances

- Best-case happens when pivot is the median element, creating equal size partitions.

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

- What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$\begin{aligned} T(n) &= T(n/10) + T(9n/10) + \Theta(n) \\ &= \Theta(n \lg n) \qquad \triangleright \text{See text for details} \end{aligned}$$

Best- and almost-worst case performances

- Best-case happens when pivot is the median element, creating equal size partitions.

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

- What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$\begin{aligned} T(n) &= T(n/10) + T(9n/10) + \Theta(n) \\ &= \Theta(n \lg n) \qquad \triangleright \text{See text for details} \end{aligned}$$

Key observation

Very close to worst-case produces $\Theta(n \lg n)$, not $\Theta(n^2)$.

Best- and almost-worst case performances

- Best-case happens when pivot is the median element, creating equal size partitions.

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

- What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$\begin{aligned} T(n) &= T(n/10) + T(9n/10) + \Theta(n) \\ &= \Theta(n \lg n) \qquad \triangleright \text{See text for details} \end{aligned}$$

Key observation

Very close to worst-case produces $\Theta(n \lg n)$, not $\Theta(n^2)$.
How to ensure that we don't *usually* hit the worst-case?

Contents

Randomized Quicksort

- Pick a random pivot and partition around it.

Randomized Quicksort

- Pick a random pivot and partition around it.
- Pivot independent of input order, so no specific input produces worst-case behavior.

Randomized Quicksort

- Pick a random pivot and partition around it.
- Pivot independent of input order, so no specific input produces worst-case behavior.
- The worst-case is determined only by the output of a random number generator.

Randomized Quicksort

- Pick a random pivot and partition around it.
- Pivot independent of input order, so no specific input produces worst-case behavior.
- The worst-case is determined only by the output of a random number generator.

RANDOMIZED-PARTITION(A, p, r) $\triangleright$ $A[p \mathinner{.\,.} r]$

```
1   i ← RANDOM(p, r)                    ▷ i = [p . . r]
2   exchange A[p] ↔ A[i]
3   return PARTITION(A, p, r)
```

Randomized Quicksort

- Pick a random pivot and partition around it.
- Pivot independent of input order, so no specific input produces worst-case behavior.
- The worst-case is determined only by the output of a random number generator.

$\textsc{Randomized-Partition}(A, p, r) \triangleright A[p \mathinner{..} r]$

$i \leftarrow \textsc{Random}(p, r) \qquad \triangleright i = [p \mathinner{..} r]$

exchange $A[p] \leftrightarrow A[i]$

return $\textsc{Partition}(A, p, r)$

$\textsc{Randomized-Quicksort}(A, p, r)$

if $p < r$

  then $q \leftarrow \textsc{Randomized-Partition}(A, p, r)$

    $\textsc{Randomized-Quicksort}(A, p, q-1)$

    $\textsc{Randomized-Quicksort}(A, q+1, r)$

Conclusion

- One of the most widely used sorting algorithm.
- While it runs in $O(n^2)$ time in the worst-case, it runs in $O(n \lg n)$ time on the average.
- Runs almost twice as fast as merge-sort.
- Can be *tuned* substantially.
- Almost all program language runtime library provide some variant of Quicksort (java.util.Arrays.sort() in Java, qsort() in C, std::sort() in C++, etc).

Conclusion

- One of the most widely used sorting algorithm.
- While it runs in $O(n^2)$ time in the worst-case, it runs in $O(n \lg n)$ time on the average.
- Runs almost twice as fast as merge-sort.
- Can be *tuned* substantially.
- Almost all program language runtime library provide some variant of Quicksort (java.util.Arrays.sort() in Java, qsort() in C, std::sort() in C++, etc).

Questions to ask (and remember)

Conclusion

- One of the most widely used sorting algorithm.
- While it runs in $O(n^2)$ time in the worst-case, it runs in $O(n \lg n)$ time on the average.
- Runs almost twice as fast as merge-sort.
- Can be *tuned* substantially.
- Almost all program language runtime library provide some variant of Quicksort (java.util.Arrays.sort() in Java, qsort() in C, std::sort() in C++, etc).

Questions to ask (and remember)

- What are the worst, best and average case performances?

Conclusion

- One of the most widely used sorting algorithm.
- While it runs in $O(n^2)$ time in the worst-case, it runs in $O(n \lg n)$ time on the average.
- Runs almost twice as fast as merge-sort.
- Can be *tuned* substantially.
- Almost all program language runtime library provide some variant of Quicksort (java.util.Arrays.sort() in Java, qsort() in C, std::sort() in C++, etc).

Questions to ask (and remember)

- What are the worst, best and average case performances?
- Is it in-place?

Conclusion

- One of the most widely used sorting algorithm.
- While it runs in $O(n^2)$ time in the worst-case, it runs in $O(n \lg n)$ time on the average.
- Runs almost twice as fast as merge-sort.
- Can be *tuned* substantially.
- Almost all program language runtime library provide some variant of Quicksort (java.util.Arrays.sort() in Java, qsort() in C, std::sort() in C++, etc).

Questions to ask (and remember)

- What are the worst, best and average case performances?
- Is it in-place?
- Is it stable?