Decrease & Conquer
Selection Problem

CS 350 – Algorithms and Complexity
Paul Doliotis – Adjunct Assistant Professor
Portland State University

Lomuto Partition – Variable Size Decrease

ALGORITHM $LomutoPartition(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
//　　　indices l and r $(l \le r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l+1$ **to** r **do**
　　if $A[i] < p$
　　　　$s \leftarrow s+1$;　swap($A[s]$, $A[i]$)
swap($A[l]$, $A[s]$)
return s

Lomuto Partition Example

s	i							
4	1	10	8	7	12	9	2	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \leq r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l+1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s+1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto Partition Example

	s, i							
4	1	10	8	7	12	9	2	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \leq r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l + 1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s + 1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto Partition Example

	s, i							
4	1	10	8	7	12	9	2	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \leq r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot

$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l + 1$ **to** r **do**
    if $A[i] < p$
        $s \leftarrow s + 1$;   swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto Partition Example

	s	i						
4	1	10	8	7	12	9	2	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \le r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l+1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s+1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto Partition Example

	s		i					
4	1	10	8	7	12	9	2	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \leq r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l + 1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s + 1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto Partition Example

	s			i				
4	1	10	8	7	12	9	2	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \le r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l+1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s+1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto Partition Example

	s				i			
4	1	10	8	7	12	9	2	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \le r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l+1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s+1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto Partition Example

	s					i		
4	1	10	8	7	12	9	2	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \leq r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l+1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s+1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto Partition Example

	s						i	
4	1	10	8	7	12	9	**2**	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \le r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l + 1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s + 1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto Partition Example

		s					i	
4	1	10	8	7	12	9	2	15

ALGORITHM $LomutoPartition(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r ($l \le r$)
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l+1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s+1$; swap($A[s]$, $A[i]$)
swap($A[l]$, $A[s]$)
return s

Lomuto Partition Example

		s					i	
4	1	2	8	7	12	9	10	15

ALGORITHM $LomutoPartition(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r ($l \leq r$)
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l + 1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s + 1$; swap($A[s]$, $A[i]$)
swap($A[l]$, $A[s]$)
return s

Lomuto Partition Example

		s						i
4	1	2	8	7	12	9	10	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \le r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l + 1$ **to** r **do**
  if $A[i] < p$
    $s \leftarrow s + 1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto Partition Example

		s						i
2	1	4	8	7	12	9	10	15

ALGORITHM *LomutoPartition*$(A[l..r])$

//Partitions subarray by Lomuto's algorithm using first element as pivot
//Input: A subarray $A[l..r]$ of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l \le r)$
//Output: Partition of $A[l..r]$ and the new position of the pivot
$p \leftarrow A[l]$
$s \leftarrow l$
for $i \leftarrow l+1$ **to** r **do**
    if $A[i] < p$
        $s \leftarrow s+1$; swap$(A[s], A[i])$
swap$(A[l], A[s])$
return s

Lomuto vs Hoare Partition

- Hoare’s partition is more efficient than Lomuto

Lomuto vs Hoare Partition

- Hoare’s partition is more efficient than Lomuto
  - Three times fewer swaps on average

Lomuto vs Hoare Partition

- Hoare's partition is more efficient than Lomuto
  - Three times fewer swaps on average
  - Better partition with equal values

Lomuto vs Hoare Partition

- Hoare's partition is more efficient than Lomuto
  - Three times fewer swaps on average
  - Better partition with equal values
- Both have worst case performance $O(n^2)$ when array is sorted.

The Selection Problem

- The selection problem is defined as follows:
- Given a set of N numbers, find the K-th smallest value.

The Selection Problem

- The selection problem is defined as follows:
- Given a set of N numbers, find the K-th smallest value.
- K can be anything.

The Selection Problem

- The selection problem is defined as follows:
- Given a set of N numbers, find the K-th smallest value.
- K can be anything.
- Special cases:

The Selection Problem

- The selection problem is defined as follows:
- Given a set of N numbers, find the K-th smallest value.
- K can be anything.
- Special cases:
  - K = 1: we are looking for the minimum value.

The Selection Problem

- The selection problem is defined as follows:
- Given a set of N numbers, find the K-th smallest value.
- K can be anything.
- Special cases:
  - K = 1: we are looking for the minimum value.
  - K = N/2: we are looking for the median value.

The Selection Problem

- The selection problem is defined as follows:
- Given a set of N numbers, find the K-th smallest value.
- K can be anything.
- Special cases:
  - K = 1: we are looking for the minimum value.
  - K = N/2: we are looking for the median value.
  - K = N: we are looking for the maximum value.

The Selection Problem

- The selection problem is defined as follows:
- Given a set of N numbers, find the K-th smallest value.
- K can be anything.
- Special cases:
  - K = 1: we are looking for the minimum value.
  - K = N/2: we are looking for the median value.
  - K = N: we are looking for the maximum value.
- However, K can take other values as well.

Solving The Selection Problem

- Special cases:
- K = 1: we are looking for the minimum value.
  - We can find the solution in linear time, by just going through the array once.

Solving The Selection Problem

- Special cases:
- K = 1: we are looking for the minimum value.
  - We can find the solution in linear time, by just going through the array once.
- K = N: we are looking for the maximum value.
  - Again, we can find the solution in linear time.

Solving The Selection Problem

- Special cases:
- K = 1: we are looking for the minimum value.
  - We can find the solution in linear time, by just going through the array once.
- K = N: we are looking for the maximum value.
  - Again, we can find the solution in linear time.
- What about K = N/2, i.e., for finding the median?

Solving The Selection Problem

- Special cases:
- K = 1: we are looking for the minimum value.
  - We can find the solution in linear time, by just going through the array once.
- K = N: we are looking for the maximum value.
  - Again, we can find the solution in linear time.
- What about K = N/2, i.e., for finding the median?
- An easy (but not optimal) approach would be:
  - Sort the numbers using quicksort.
  - Return the middle position in the array.
  - Average time complexity: $\Theta(N \lg N)$.

Solving The Selection Problem

- It turns out we can solve the selection problem in linear time (on average), using an algorithm very similar to quicksort.

ALGORITHM *Quickselect*$(A[l..r], k)$

//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r-l+1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k-1$ **return** $A[s]$
else if $s > l+k-1$ $Quickselect(A[l..s-1], k)$
else $Quickselect(A[s+1..r], k-1-s)$

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
    $s \leftarrow Partition(A[l..r])$ //s is a split position
    $Quicksort(A[l..s-1])$
    $Quicksort(A[s+1..r])$

Solving The Selection Problem

ALGORITHM *Quickselect*$(A[l..r], k)$
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ $Quickselect(A[l..s-1], k)$
else $Quickselect(A[s+1..r], k - 1 - s)$

- Why does this work?

Solving The Selection Problem

ALGORITHM *Quickselect*$(A[l..r], k)$
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ $Quickselect(A[l..s-1], k)$
else $Quickselect(A[s+1..r], k - 1 - s)$

- Why does this work?
- Suppose that some partition(a, L, R) returned k-1.
- That means that:
  - Value a[k-1] was the pivot used in that partition.

Solving The Selection Problem

ALGORITHM $Quickselect(A[l..r], k)$
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ $Quickselect(A[l..s-1], k)$
else $Quickselect(A[s+1..r], k-1-s)$

- Why does this work?
- Suppose that some partition(a, L, R) returned k-1.
- That means that:
  - Value a[k-1] was the pivot used in that partition.
- Everything to the left of a[k-1] is <= a[k-1].
- Everything to the right of a[k-1] is >= a[k-1].
- Thus, a[k-1] is the k-th smallest value.

Solving The Selection Problem

ALGORITHM $Quickselect(A[l..r], k)$
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ $Quickselect(A[l..s-1], k)$
else $Quickselect(A[s+1..r], k-1-s)$

- Suppose that some partition(a, L, R) returned s > k-1. (l = 0)
- Value a[s] was the pivot used in that partition.
- Everything to the left of a[s] is <= a[s].
- Everything to the right of a[s] is >= a[s].
- Thus, a[s] is the (s+1)-th smallest value.
- Since k < s –l + 1, the answer is among items a[L], ..., a[s-1].

Solving The Selection Problem

ALGORITHM *Quickselect*$(A[l..r], k)$
//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ $Quickselect(A[l..s-1], k)$
else $Quickselect(A[s+1..r], k-1-s)$

- Suppose that some partition(a, L, R) returned s < k-1. (l = 0)
- Value a[s] was the pivot used in that partition.
- Everything to the left of a[s] is <= a[s].
- Everything to the right of a[s] is >= a[s].
- Thus, a[s] is the (s+1)-th smallest value.
- Since k > s –l + 1, the answer is among items a[s+1], ..., a[r], looking for k-1-s!.

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

s	i							
4	1	10	8	7	12	9	2	15

ALGORITHM $Quickselect(A[l..r], k)$

//Solves the selection problem by recursive partition-based algorithm
//Input: Subarray $A[l..r]$ of array $A[0..n-1]$ of orderable elements and
// integer k $(1 \le k \le r - l + 1)$
//Output: The value of the kth smallest element in $A[l..r]$
$s \leftarrow LomutoPartition(A[l..r])$ //or another partition algorithm
if $s = k - 1$ **return** $A[s]$
else if $s > l + k - 1$ $Quickselect(A[l..s-1], k)$
else $Quickselect(A[s+1..r], k-1-s)$

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

	s, *i*							
4	1	10	8	7	12	9	2	15

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

	s	*i*						
4	1	10	8	7	12	9	2	15

	s		*i*					
4	1	10	8	7	12	9	2	15

	s			*i*				
4	1	10	8	7	12	9	2	15

	s				*i*			
4	1	10	8	7	12	9	2	15

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

	s					*i*		
4	1	10	8	7	12	9	2	15

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

	s						*i*	
4	1	10	8	7	12	9	2	15

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

	s						*i*	
4	1	10	8	7	12	9	2	15

		s					*i*	
4	1	10	8	7	12	9	2	15

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

	s						*i*	
4	1	10	8	7	12	9	2	15

		s					*i*	
4	1	10	8	7	12	9	2	15

		s					*i*	
4	1	2	8	7	12	9	10	15

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

		s						*i*
4	1	2	8	7	12	9	10	15

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

		s						*i*
4	1	2	8	7	12	9	10	15

		s						*i*
2	1	4	8	7	12	9	10	15

- Partition is over! S = 2.
- More formally: 2 = LomutoPartition (A [0, 8], 5)

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

		s						*i*
2	1	4	8	7	12	9	10	15

- S =2 is smaller than k-1 = 4
- Where do we look next?

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

		s						i
2	1	4	8	7	12	9	10	15

- S =2 is smaller than k-1 = 4
- Where do we look next?
- To the right, for 2^{nd} element (k-1-s)

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

		s						*i*
2	1	4	8	7	12	9	10	15

- S =2 is smaller than k-1 = 4
- Where do we look next?
- To the right, for 2nd element (k-1-s)
- Lets solve QuickSelect(A[3,8], 2)

Solving The Selection Problem - Example

- Lets solve QuickSelect (A [0, 8], 5), i.e. median

		s						*i*
2	1	4	8	7	12	9	10	15

- S =2 is smaller than k-1 = 4
- Where do we look next?
- To the right, for 2nd element (k-1-s)
- Lets solve QuickSelect(A[3,8], 2)

  QuickSelect(A[s+1…r], k-1-s)

Solving The Selection Problem - Example

- Lets solve QuickSelect(A[3,8], 2)

		s						*i*
2	1	4	8	7	12	9	10	15

s	*i*				
8	7	12	9	10	15

	s, *i*				
8	7	12	9	10	15

Solving The Selection Problem - Example

- Lets solve QuickSelect(A[3,8], 2)

		s						*i*
2	1	4	8	7	12	9	10	15

s	*i*				
8	7	12	9	10	15

	s	*i*			
8	7	12	9	10	15

Solving The Selection Problem - Example

- Lets solve QuickSelect(A[3,8], 2)

		s						*i*
2	1	4	8	7	12	9	10	15

s	*i*				
8	7	12	9	10	15

	s	*i*			
8	7	12	9	10	15

	s				*i*
8	7	12	9	10	15

Solving The Selection Problem - Example

- Lets solve QuickSelect(A[3,8], 2)

		s						*i*
2	1	4	8	7	12	9	10	15

s	*i*				
8	7	12	9	10	15

	s	*i*			
8	7	12	9	10	15

Done!

	s				*i*
7	8	12	9	10	15

Solving The Selection Problem - Example

- 4 = QuickSelect(A[3,8], 2)

	s				*i*
7	8	12	9	10	15

S = 4, which makes 8 the 5th element! End of algorithm.

2	1	4	7	8	12	9	10	15

Selection Time Complexity

- The **worst-case** time complexity of selection is equivalent to that for quicksort:
  - The pivot is the smallest or the largest element.
  - Then, we did a lot of work to just eliminate one item.

- Overall, **worst-case** time is $N+(N-1)+(N-2)+\ldots+1 = \Theta(N^2)$.
  - Same as for quicksort.

Best-Case Time Complexity

- The **best case** time complexity for selection is also similar to the one for quicksort:

Best-Case Time Complexity

- The **best case** time complexity for selection is also similar to the one for quicksort:
  - When the array is partitioned in a **perfectly balanced** way.
  - That is, when the pivot is always the median value in the array.

Best-Case Time Complexity

- The **best case** time complexity for selection is also similar to the one for quicksort:
  - When the array is partitioned in a **perfectly balanced** way.
  - That is, when the pivot is always the median value in the array.
- Let $T_Q(N)$ be the best-case running time complexity for quicksort.
- Let T_S be the best-case running time complexity for selection.

Best-Case Time Complexity

- The **best case** time complexity for selection is also similar to the one for quicksort:
  - When the array is partitioned in a **perfectly balanced** way.
  - That is, when the pivot is always the median value in the array.
- Let $T_Q(N)$ be the best-case running time complexity for quicksort.
- Let T_S be the best-case running time complexity for selection.
- $T_Q(N) = N + 2 * T_Q(N/2)$.
- $T_S(N) = N + T_S(N/2)$.

Best-Case Time Complexity

- The **best case** time complexity for selection is also similar to the one for quicksort:
  - When the array is partitioned in a **perfectly balanced** way.
  - That is, when the pivot is always the median value in the array.
- Let $T_Q(N)$ be the best-case running time complexity for quicksort.
- Let T_S be the best-case running time complexity for selection.
- $T_Q(N) = N + 2 * T_Q(N/2)$.
- $T_S(N) = N + T_S(N/2)$.
- Why is the T_S different than the T_Q recurrence?

Best-Case Time Complexity

- The **best case** time complexity for selection is also similar to the one for quicksort:
  - When the array is partitioned in a **perfectly balanced** way.
  - That is, when the pivot is always the median value in the array.
- Let $T_Q(N)$ be the best-case running time complexity for quicksort.
- Let T_S be the best-case running time complexity for selection.
- $T_Q(N) = N + 2 * T_Q(N/2)$.
- $T_S(N) = N + T_S(N/2)$.
- Why is the T_S different than the T_Q recurrence?
- In quicksort, we need to process **both parts** of the partition.
- In selection, we only need to process **one part** of the partition.

Best-Case Time Complexity

- For convenience, let $N = 2^n$.
- Assuming that the partition always splits the set into two equal halves, we get:
- $T_S(2^n) = 2^n + T_S(2^{n-1})$

$$= 2^n + T_S(2^{n-1}) \quad \text{step 1}$$

$$= 2^n + 2^{n-1} + T_S(2^{n-2}) \quad \text{step 2}$$

$$= 2^n + 2^{n-1} + 2^{n-2} + T_S(2^{n-3}) \quad \text{step 3}$$

$$\ldots$$

$$= 2^n + 2^{n-1} + 2^{n-2} + \ldots + 2^1 + T_S(1) \quad \text{step n}$$

$$= 2^{n+1} - 1 + \text{constant} = 2*2^n + \text{constant}$$

$$= 2*N + \text{constant}$$

$$= \Theta(N).$$

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.
- The best-case time complexity is $\Theta(N)$.
- The average time complexity is also $\Theta(N)$.

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.
- The best-case time complexity is $\Theta(N)$.
- The average time complexity is also $\Theta(N)$.
- On average, selection performance is close to that of the best case.
- Why?

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.
- The best-case time complexity is $\Theta(N)$.
- The average time complexity is also $\Theta(N)$.
- On average, selection performance is close to that of the best case.
- Why? A good choice for a pivot is when it lies within 25^{th} to 75^{th} percentile.

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.
- The best-case time complexity is $\Theta(N)$.
- The average time complexity is also $\Theta(N)$.
- On average, selection performance is close to that of the best case.
- Why? A good choice for a pivot is when it lies within 25^{th} to 75^{th} percentile.
  - That is 50% chance of picking a good pivot

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.
- The best-case time complexity is $\Theta(N)$.
- The average time complexity is also $\Theta(N)$.
- On average, selection performance is close to that of the best case.
- Why? A good choice for a pivot is when it lies within 25^{th} to 75^{th} percentile.
  - That is 50% chance of picking a good pivot
  - On average a fair coin needs to be tossed two times before a "heads" is seen

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.
- The best-case time complexity is $\Theta(N)$.
- The average time complexity is also $\Theta(N)$.
- On average, selection performance is close to that of the best case.
- Why? A good choice for a pivot is when it lies within 25^{th} to 75^{th} percentile.
  - That is 50% chance of picking a good pivot
  - On average a fair coin needs to be tossed two times before a “heads” is seen
  - So, usually, the pivot value is "close enough" to the 50-th percentile to achieve a reasonably balanced partition.

Quicksort Overview

- Quicksort is the most popular sorting algorithm.

Quicksort Overview

- Quicksort is the most popular sorting algorithm.
- Extensively used in popular languages (such as C) as the default sorting algorithm.

Quicksort Overview

- Quicksort is the most popular sorting algorithm.
- Extensively used in popular languages (such as C) as the default sorting algorithm.
- The **average** time complexity is $\Theta(N \log N)$.

Quicksort Overview

- Quicksort is the most popular sorting algorithm.
- Extensively used in popular languages (such as C) as the default sorting algorithm.
- The **average** time complexity is $\Theta(N \log N)$.
- Interestingly, the **worst-case** time complexity is $\Theta(N^2)$.

Quicksort Overview

- Quicksort is the most popular sorting algorithm.
- Extensively used in popular languages (such as C) as the default sorting algorithm.
- The **average** time complexity is $\Theta(N \log N)$.
- Interestingly, the **worst-case** time complexity is $\Theta(N^2)$.
- However, if quicksort is implemented appropriately, the probability of the worst case happening is astronomically small.

Worst-Case Time Complexity

- The **worst-case** of quicksort is interesting:

Worst-Case Time Complexity

- The **worst-case** of quicksort is interesting:
- Quicksort has the slowest running time when the input array **is already sorted**.

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	10	17	30	35	42	50	60	70	80	90

- partition(a, 0, 9):
  - scans 10 elements, makes no changes, returns 9.

Worst-Case Time Complexity

- The **worst-case** of quicksort is interesting:
- Quicksort has the slowest running time when the input array **<u>is already sorted</u>**.

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	10	17	30	35	42	50	60	70	80	90

- partition(a, 0, 9):
  – scans 10 elements, makes no changes, returns 9.
- partition(a, 0, 8):
  – scans 9 elements, makes no changes, returns 8.

Worst-Case Time Complexity

- The **worst-case** of quicksort is interesting:
- Quicksort has the slowest running time when the input array **<u>is already sorted</u>**.

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	10	17	30	35	42	50	60	70	80	90

- partition(a, 0, 9):
  - scans 10 elements, makes no changes, returns 9.
- partition(a, 0, 8):
  - scans 9 elements, makes no changes, returns 8.
- partition(a, 0, 7):
  - scans 8 elements, makes no changes, returns 7.

Worst-Case Time Complexity

- The **worst-case** of quicksort is interesting:
- Quicksort has the slowest running time when the input array **is already sorted**.

position	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
value	10	17	30	35	42	50	60	70	80	90

- partition(a, 0, 9):
  - scans 10 elements, makes no changes, returns 9.
- partition(a, 0, 8):
  - scans 9 elements, makes no changes, returns 8.
- partition(a, 0, 7):
  - scans 8 elements, makes no changes, returns 7.
- Overall, **worst-case** time is $N+(N-1)+(N-2)+\ldots+1 = \Theta(N^2)$.

Best-Case Time Complexity

- Overall, the worst-case happens when the array is partitioned in an **imbalanced** way:

Best-Case Time Complexity

- Overall, the worst-case happens when the array is partitioned in an **imbalanced** way:
  - One item, or very few items, on one side.
  - Everything else on the other side.

Best-Case Time Complexity

- Overall, the worst-case happens when the array is partitioned in an **imbalanced** way:
  - One item, or very few items, on one side.
  - Everything else on the other side.
- The **best case** time complexity for quicksort is when the array is partitioned in a **perfectly balanced** way.

Best-Case Time Complexity

- Overall, the worst-case happens when the array is partitioned in an **imbalanced** way:
  - One item, or very few items, on one side.
  - Everything else on the other side.
- The **best case** time complexity for quicksort is when the array is partitioned in a **perfectly balanced** way.
- I.e., when the pivot is always the median value in the array.

Best-Case Time Complexity

- Overall, the worst-case happens when the array is partitioned in an **imbalanced** way:
  - One item, or very few items, on one side.
  - Everything else on the other side.
- The **best case** time complexity for quicksort is when the array is partitioned in a **perfectly balanced** way.
- I.e., when the pivot is always the median value in the array.
- Let T(N) be the best-case running time complexity for quicksort.
- T(N) = N + 2 * T(N/2)

Best-Case Time Complexity

- Overall, the worst-case happens when the array is partitioned in an **imbalanced** way:
  - One item, or very few items, on one side.
  - Everything else on the other side.
- The **best case** time complexity for quicksort is when the array is partitioned in a **perfectly balanced** way.
- I.e., when the pivot is always the median value in the array.
- Let T(N) be the best-case running time complexity for quicksort.
- $T(N) = N + 2 * T(N/2)$
- Why? Because to sort the array:
  - We do N operations for the partition.

Best-Case Time Complexity

- Overall, the worst-case happens when the array is partitioned in an **imbalanced** way:
  - One item, or very few items, on one side.
  - Everything else on the other side.
- The **best case** time complexity for quicksort is when the array is partitioned in a **perfectly balanced** way.
- I.e., when the pivot is always the median value in the array.
- Let T(N) be the best-case running time complexity for quicksort.
- $T(N) = N + 2 * T(N/2)$
- Why? Because to sort the array:
  - We do N operations for the partition.
  - We do two recursive calls, and each call receives half the data.

Best-Case Time Complexity

- For convenience, let $N = 2^n$.
- Assuming that the partition always splits the set into two equal halves, we get:
- $T(2^n) = 2^n + 2 * T(2^{n-1})$

$$
\begin{aligned}
&= 1*2^n + 2^1 * T(2^{n-1}) && \text{step 1} \\
&= 2*2^n + 2^2 * T(2^{n-2}) && \text{step 2} \\
&= 3*2^n + 2^3 * T(2^{n-3}) && \text{step 3} \\
&\ldots \\
&= i*2^n + 2^i * T(2^{n-i}) && \text{step i} \\
&\ldots \\
&= n*2^n + 2^n * T(2^{n-n}) && \text{step n} \\
&= \lg N * N + N * T(1) \\
&= \Theta(N \lg N).
\end{aligned}
$$

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.
- The best-case time complexity is $\Theta(N \lg N)$.

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.
- The best-case time complexity is $\Theta(N \lg N)$.
- It turns out that the average time complexity is also $\Theta(N \lg N)$.
- On average, quicksort performance is close to that of the best case.

Average Time Complexity

- The worst-case time complexity is $\Theta(N^2)$.
- The best-case time complexity is $\Theta(N \lg N)$.
- It turns out that the average time complexity is also $\Theta(N \lg N)$.
- On average, quicksort performance is close to that of the best case.
- Why? Because, usually, the pivot value is "close enough" to the 50-th percentile to achieve a reasonably balanced partition.
  - For example, half the times the pivot value should be between the 25-th percentile and the 75th percentile.

Improving Performance

- The basic implementation of quicksort that we saw, makes a partition using the rightmost element as pivot.
  - This has the risk of giving a pivot that is not that close to the 50th percentile.
  - When the data is already sorted, the pivot is the 100th percentile, which is the worst-case.

Improving Performance

- We can improve performance by using as pivot the median of three values:
  - The leftmost element.
  - The middle element.
  - The rightmost element.

Improving Performance

- We can improve performance by using as pivot the median of three values:
  - The leftmost element.
  - The middle element.
  - The rightmost element.
- Then, the pivot has better chances of being close to the 50th percentile.

Improving Performance

- We can improve performance by using as pivot the median of three values:
  - The leftmost element.
  - The middle element.
  - The rightmost element.
- Then, the pivot has better chances of being close to the 50th percentile.
- If the file is already sorted, the pivot is the median.

Improving Performance

- We can improve performance by using as pivot the median of three values:
  - The leftmost element.
  - The middle element.
  - The rightmost element.
- Then, the pivot has better chances of being close to the 50th percentile.
- If the file is already sorted, the pivot is the median.
- Thus, already sorted data is:
  - The worst case (slowest running time) when the pivot is the rightmost element.
  - The best case (fastest run time) when the pivot is the median of the leftmost, middle, and rightmost elements.

Improving Performance

- We can improve performance by picking pivot randomly

Improving Performance

- We can improve performance by picking pivot randomly
- Quicksort is not stable but is in place! (no need for extra memory like MergeSort)

Improving Performance

- We can improve performance by picking pivot randomly
- Quicksort is not stable but is in place! (no need for extra memory like MergeSort)
- better pivot selection: median-of-three partitioning

Improving Performance

- We can improve performance by picking pivot randomly
- Quicksort is not stable but is in place! (no need for extra memory like MergeSort)
- better pivot selection: median-of-three partitioning
- separate partition for keys equal to pivot

Improving Performance

- We can improve performance by picking pivot randomly
- Quicksort is not stable but is in place! (no need for extra memory like MergeSort)
- better pivot selection: median-of-three partitioning
- separate partition for keys equal to pivot
- switch to insertion sort on small sub-problems

Improving Performance

- We can improve performance by picking pivot randomly
- Quicksort is not stable but is in place! (no need for extra memory like MergeSort)
- better pivot selection: median-of-three partitioning
- separate partition for keys equal to pivot
- switch to insertion sort on small sub-problems
- elimination of recursion

Improving Performance

- We can improve performance by picking pivot randomly
- Quicksort is not stable but is in place! (no need for extra memory like MergeSort)
- better pivot selection: median-of-three partitioning
- separate partition for keys equal to pivot
- switch to insertion sort on small sub-problems
- elimination of recursion
- These combine to give 20–25% improvement