Algorithms: Hashing and Quick Sort

Alessandro Chiesa, Ola Svensson

EPFL School of Computer and Communication Sciences

Lecture 24, 14.05.2025

Hash tables: summary

HASH-tables efficiently implement:

- INSERT: $O(1)$
- DELETE: $O(1)$
- SEARCH: Expected $O(n/m)$ (if good hash function)

Cannot avoid collisions without having $m \gg n^2$

Instead deal with collisions using for example chaining

Quick Sort

- The sorting algorithm of choice in many computer systems
- Easy to implement
- Fast in practice (and as we will see in theory)
- As merge-sort, based on divide-and-conquer paradigm

DIVIDE-AND-CONQUER

Quick Sort

DIVIDE-AND-CONQUER

Quick Sort

DIVIDE-AND-CONQUER

Quick Sort

DIVIDE-AND-CONQUER

Quick Sort

Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$

5	8	4	7	1	2	3	6

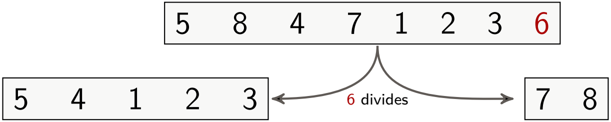
Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$

5	8	4	7	1	2	3	6

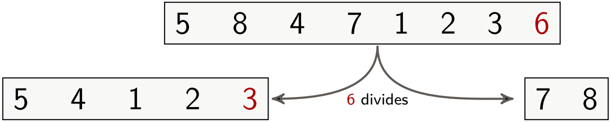
Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$



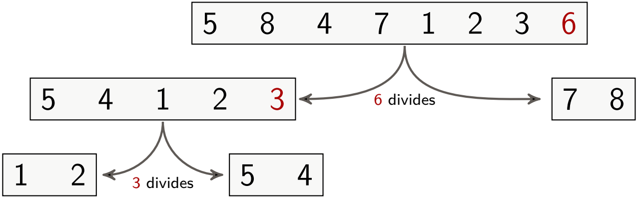
Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$



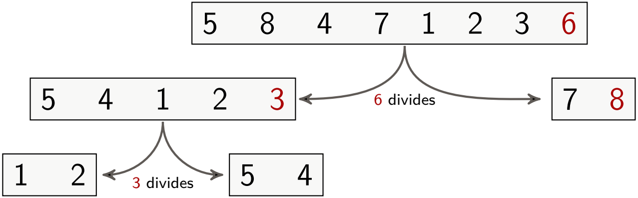
Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$



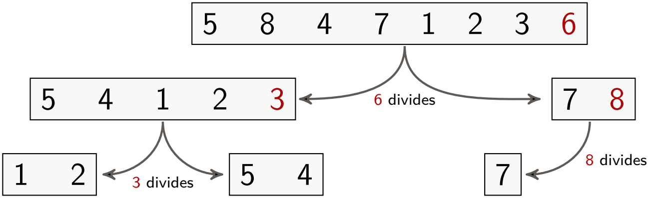
Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$



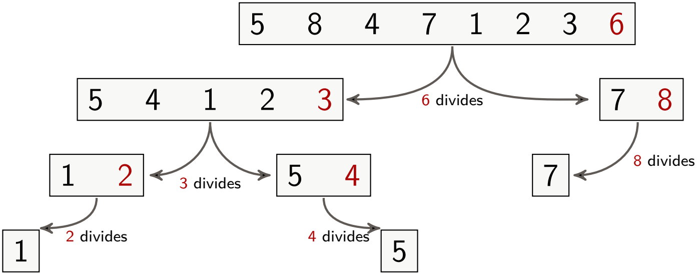
Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$



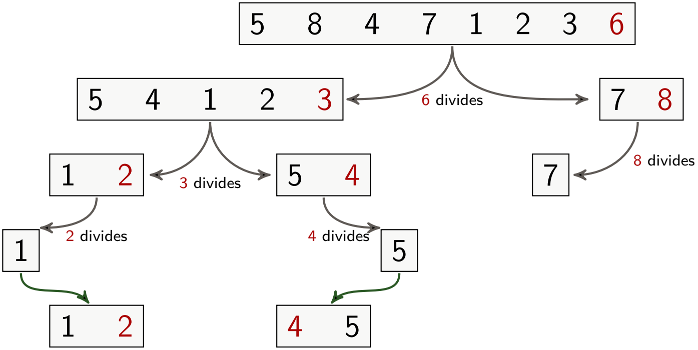
Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$



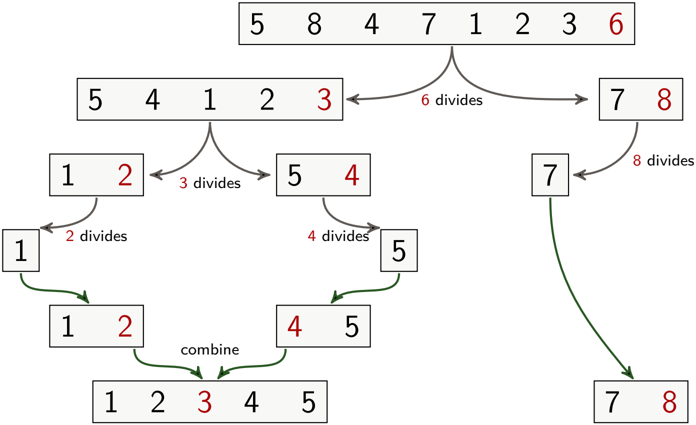
Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$



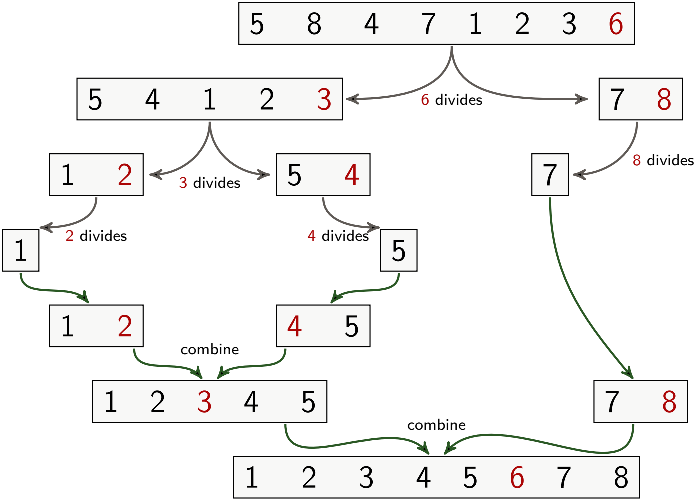
Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$



Quick Sort Idea

Example $\langle 5, 8, 4, 7, 1, 2, 3, 6 \rangle$



Quick Sort — Divide-and-Conquer

To sort the subarray $A[p \dots r]$:

Divide: Partition $A[p \dots r]$, into two (possibly empty) subarrays $A[p \dots q-1]$ and $A[q+1 \dots r]$, such that each element in the first subarray is $\leq A[q]$ and each element in the second subarray is $\geq A[q]$

Quick Sort — Divide-and-Conquer

To sort the subarray $A[p \dots r]$:

Divide: Partition $A[p \dots r]$, into two (possibly empty) subarrays $A[p \dots q-1]$ and $A[q+1 \dots r]$, such that each element in the first subarray is $\leq A[q]$ and each element in the second subarray is $\geq A[q]$

Conquer: Sort the two subarrays by recursive calls to QUICKSORT

Quick Sort — Divide-and-Conquer

To sort the subarray $A[p \dots r]$:

Divide: Partition $A[p \dots r]$, into two (possibly empty) subarrays $A[p \dots q-1]$ and $A[q+1 \dots r]$, such that each element in the first subarray is $\leq A[q]$ and each element in the second subarray is $\geq A[q]$

Conquer: Sort the two subarrays by recursive calls to QUICKSORT

Quick Sort — Divide-and-Conquer

To sort the subarray $A[p \dots r]$:

Divide: Partition $A[p \dots r]$, into two (possibly empty) subarrays $A[p \dots q-1]$ and $A[q+1 \dots r]$, such that each element in the first subarray is $\leq A[q]$ and each element in the second subarray is $\geq A[q]$

Conquer: Sort the two subarrays by recursive calls to QUICKSORT

Combine: No work is needed to combine the subarrays, because they are sorted in place

Partitioning (divide step)

PARTITION always selects the last element $A[r]$ in the subarray $A[p \dots r]$ as the **pivot** — the element around which to partition

```
PARTITION(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

	$A[p]$			$A[q]$				$A[r]$
A:	8	4	5	7	1	2	3	6

Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

x: 6

	$A[p]$			$A[q]$				$A[r]$
A:	8	4	5	7	1	2	3	6

Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \ldots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \ldots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

PARTITION(A, p, r)

```
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

x: 6

	$A[p]$			$A[q]$				$A[r]$
A:	8	4	5	7	1	2	3	6

▲ i

Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

x: 6

	$A[p]$			$A[q]$				$A[r]$
A:	8	4	5	7	1	2	3	6

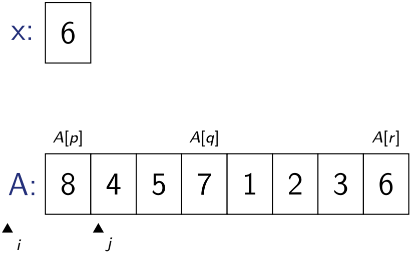
i (before $A[p]$), j (at $A[p]$)

Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

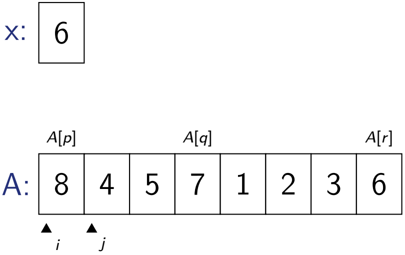


Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```



Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \ldots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \ldots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

PARTITION(A, p, r)

```
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

x: 6

A:

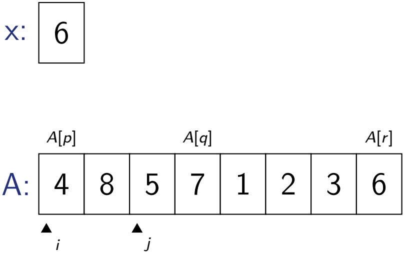
$A[p]$			$A[q]$				$A[r]$
4	8	5	7	1	2	3	6
▲ i	▲ j						

Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

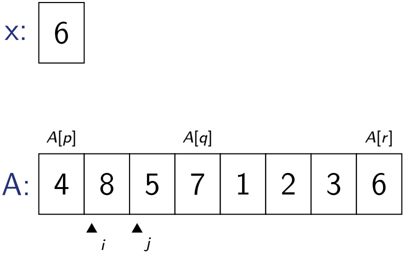


Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

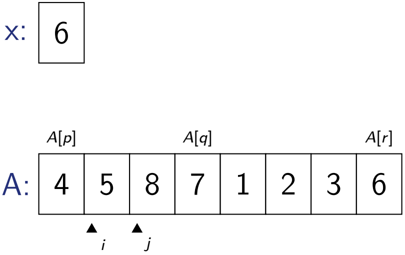


Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r]$ = pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```



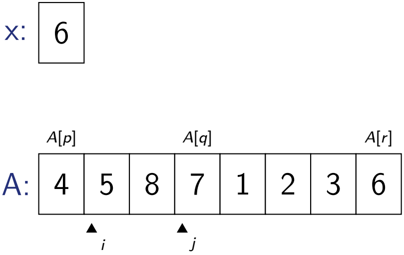
Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

PARTITION(A, p, r)

```
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```



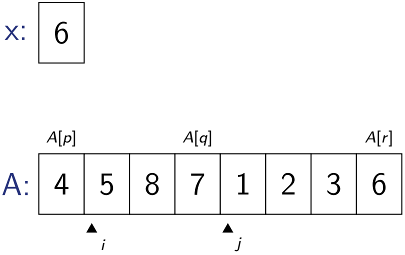
Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
  if $A[j] \leq x$
    $i = i + 1$
    exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
return $i + 1$

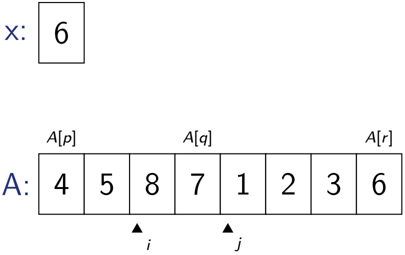


Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \ldots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \ldots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```



Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \ldots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \ldots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

x: 6

	$A[p]$			$A[q]$				$A[r]$
A:	4	5	1	7	8	2	3	6
			▲ i		▲ j			

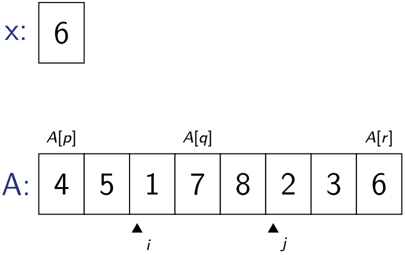
Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
  if $A[j] \leq x$
    $i = i + 1$
    exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
return $i + 1$



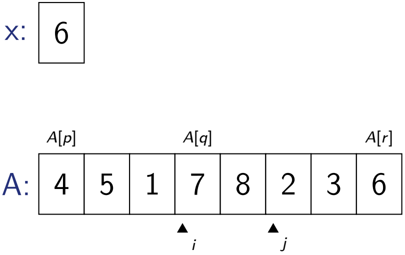
Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r]$ = pivot

PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
  if $A[j] \leq x$
    $i = i + 1$
    exchange $A[i]$ with $A[j]$
exchange $A[i + 1]$ with $A[r]$
return $i + 1$

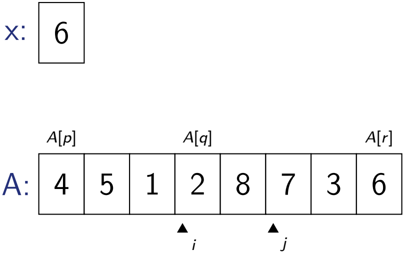


Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```



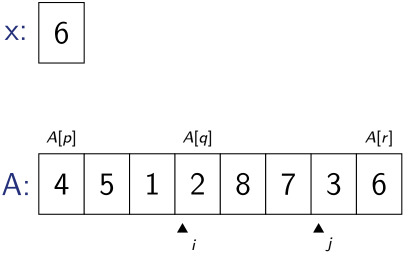
Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
  if $A[j] \leq x$
    $i = i + 1$
    exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
return $i + 1$

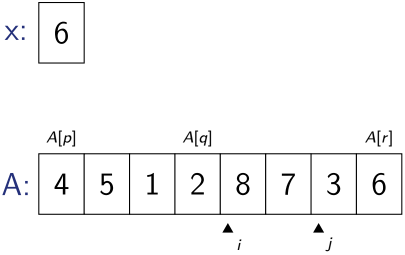


Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

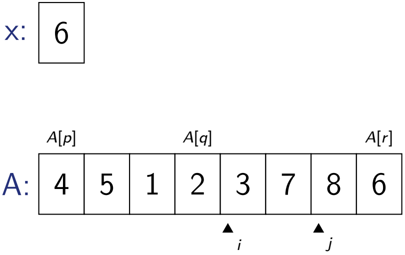


Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```



Partitioning (divide step)

Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

PARTITION(A, p, r)

$x = A[r]$
$i = p - 1$
for $j = p$ **to** $r - 1$
  if $A[j] \leq x$
    $i = i + 1$
    exchange $A[i]$ with $A[j]$
exchange $A[i+1]$ with $A[r]$
return $i + 1$

x: 6

A:

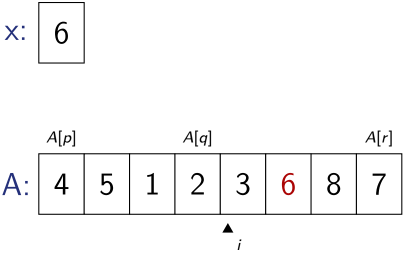
$A[p]$			$A[q]$				$A[r]$
4	5	1	2	3	7	8	6
				▲ i			

Partitioning (divide step)

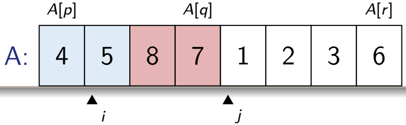
Loop Invariant:

1. All entries in $A[p \ldots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \ldots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

```
PARTITION(A, p, r)
  x = A[r]
  i = p - 1
  for j = p to r - 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```



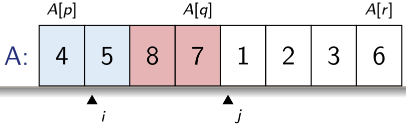
Correctness of Partitioning



Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

Correctness of Partitioning

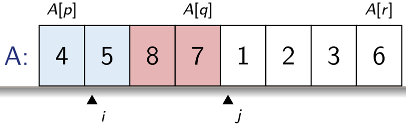


Loop Invariant:

1. All entries in $A[p \ldots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \ldots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

Initialization: Before the loop starts, loop invariant satisfied, because r is the pivot and the subarrays $A[p \ldots i]$ and $A[i+1 \ldots j-1]$ are empty

Correctness of Partitioning



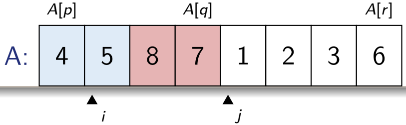
Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

Initialization: Before the loop starts, loop invariant satisfied, because r is the pivot and the subarrays $A[p \dots i]$ and $A[i+1 \dots j-1]$ are empty

Maintenance: If $A[j] \leq$ pivot, then $A[j]$ and $A[i+1]$ are swapped and then i and j are incremented. If $A[j] >$ pivot then increment only j

Correctness of Partitioning



Loop Invariant:

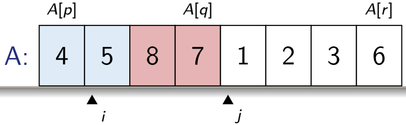
1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] = \text{pivot}$

Initialization: Before the loop starts, loop invariant satisfied, because r is the pivot and the subarrays $A[p \dots i]$ and $A[i+1 \dots j-1]$ are empty

Maintenance: If $A[j] \leq$ pivot, then $A[j]$ and $A[i+1]$ are swapped and then i and j are incremented. If $A[j] >$ pivot then increment only j

Termination: When the loop terminates, $j = r$ so all elements in A are partitioned into $A[p \dots i] \leq$ pivot, $A[i+1 \dots r-1] >$ pivot and $A[r] = \text{pivot}$

Correctness of Partitioning



Loop Invariant:

1. All entries in $A[p \dots i]$ are $\leq$ pivot
2. All entries in $A[i+1 \dots j-1]$ are $>$ pivot
3. $A[r] =$ pivot

Initialization: Before the loop starts, loop invariant satisfied, because r is the pivot and the subarrays $A[p \dots i]$ and $A[i+1 \dots j-1]$ are empty

Maintenance: If $A[j] \leq$ pivot, then $A[j]$ and $A[i+1]$ are swapped and then i and j are incremented. If $A[j] >$ pivot then increment only j

Termination: When the loop terminates, $j = r$ so all elements in A are partitioned into $A[p \dots i] \leq$ pivot, $A[i+1 \dots r-1] >$ pivot and $A[r] =$ pivot

The last two lines of PARTITION moves the pivot element to the "right" place by swapping $A[i+1]$ and $A[r]$

Time for partitioning

PARTITION(A, p, r)

```
x = A[r]
i = p - 1
for j = p to r - 1
    if A[j] ≤ x
        i = i + 1
        exchange A[i] with A[j]
exchange A[i + 1] with A[r]
return i + 1
```

Time for partitioning

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
        if A[j] ≤ x
              i = i + 1
              exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

- **for** loop runs $\approx n := r - p + 1$ times.
- Each iteration takes time $\Theta(1)$
- Total running time is $\Theta(n)$ for an array of length n.

Time for partitioning

```
PARTITION(A, p, r)
  x = A[r]
  i = p − 1
  for j = p to r − 1
      if A[j] ≤ x
          i = i + 1
          exchange A[i] with A[j]
  exchange A[i + 1] with A[r]
  return i + 1
```

- **for** loop runs $\approx n := r - p + 1$ times.
- Each iteration takes time $\Theta(1)$
- Total running time is $\Theta(n)$ for an array of length n.
- Note that the number of comparisons made is $\approx n$

Quick Sort Algorithm

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

Worst case running time of quick sort

1	2	3	4	...	n-2	n-1	n

Worst case running time of quick sort

1	2	3	4	...	n-2	n-1	n

1	2	3	4	...	n-2	n-1

Worst case running time of quick sort

1	2	3	4	...	n-2	n-1	n

1	2	3	4	...	n-2	n-1

1	2	3	4	...	n-2

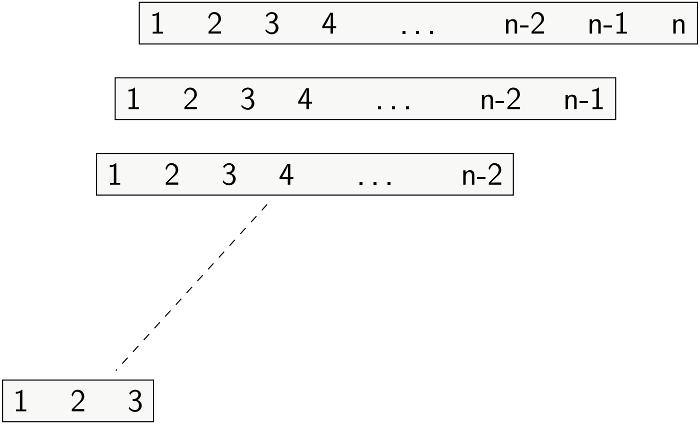
Worst case running time of quick sort

1 2 3 4 ... n-2 n-1 n

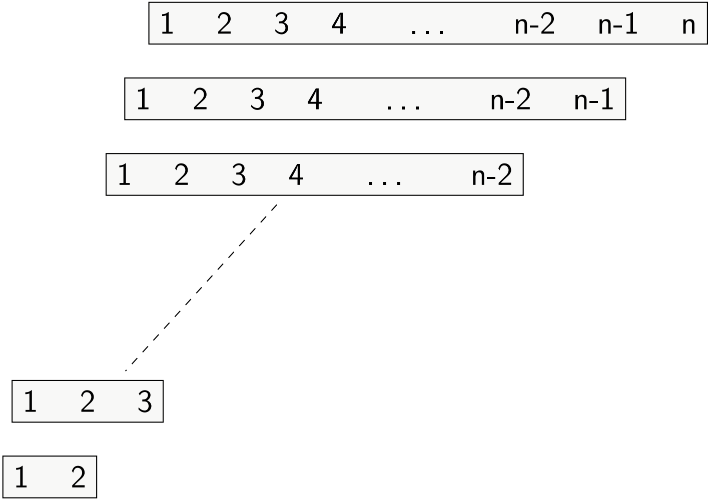
1 2 3 4 ... n-2 n-1

1 2 3 4 ... n-2

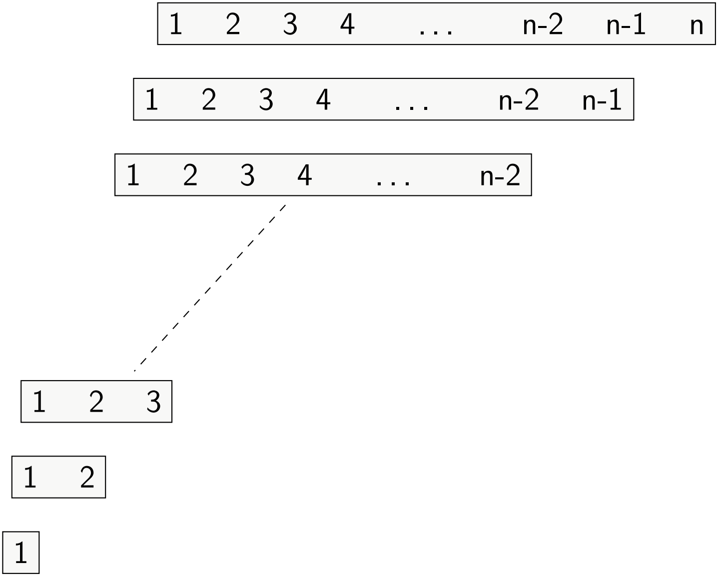
Worst case running time of quick sort



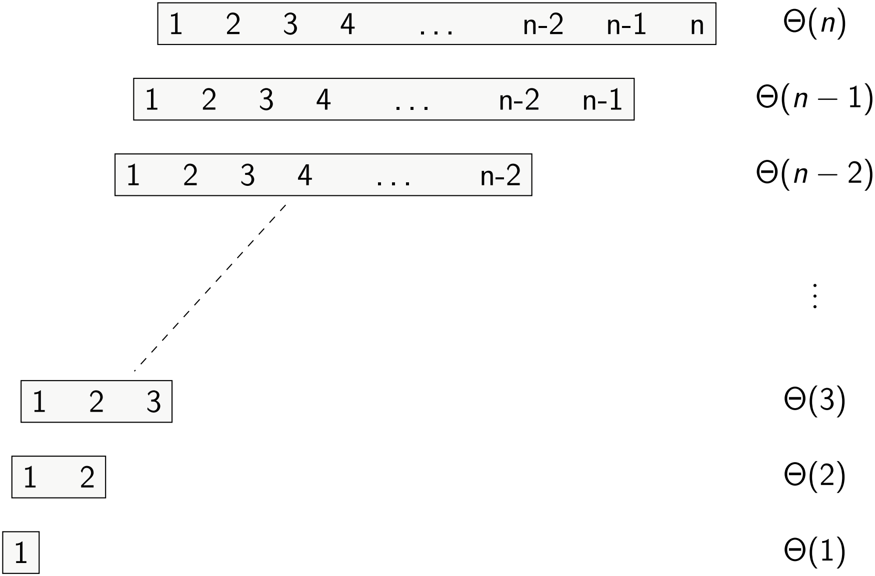
Worst case running time of quick sort



Worst case running time of quick sort



Worst case running time of quick sort



Worst case running time of quick sort

1 2 3 4 … n-2 n-1 n	$\Theta(n)$
1 2 3 4 … n-2 n-1	$\Theta(n-1)$
1 2 3 4 … n-2	$\Theta(n-2)$
	$\vdots$
1 2 3	$\Theta(3)$
1 2	$\Theta(2)$
1	$\Theta(1)$

Total running time: $\Theta(n^2)$

Best case running time of quick sort

Best case running time of quick sort

- Occurs when the subarrays are completely balanced every time $=$ the pivots always split the array into two subarrays of equal size

Best case running time of quick sort

- Occurs when the subarrays are completely balanced every time = the pivots always split the array into two subarrays of equal size

- Get the recurrence

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

Best case running time of quick sort

- Occurs when the subarrays are completely balanced every time = the pivots always split the array into two subarrays of equal size
- Get the recurrence

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

$$= \Theta(n \lg n)$$

Average case over all inputs

Intuition

- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence $T(n) = T(9n/10) + T(n/10) + \Theta(n)$

Average case over all inputs

Intuition

- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence $T(n) = T(9n/10) + T(n/10) + \Theta(n) = \Theta(n \lg n)$

Average case over all inputs

Intuition

- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence $T(n) = T(9n/10) + T(n/10) + \Theta(n) = \Theta(n \lg n)$
- Splits in the recursion tree will not always be good, there will be usually a mix of good and bad splits

Average case over all inputs

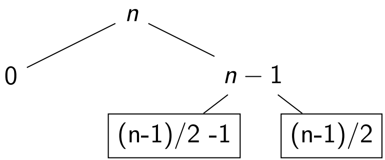
Intuition

- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence $T(n) = T(9n/10) + T(n/10) + \Theta(n) = \Theta(n \lg n)$
- Splits in the recursion tree will not always be good, there will be usually a mix of good and bad splits
- For intuition why this not affect the running time, suppose we alternate between best-case and worst-case flips

Average case over all inputs

Intuition

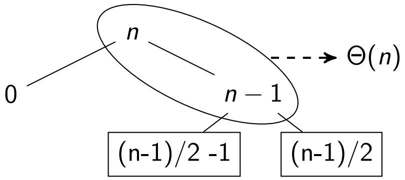
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence $T(n) = T(9n/10) + T(n/10) + \Theta(n) = \Theta(n \lg n)$
- Splits in the recursion tree will not always be good, there will be usually a mix of good and bad splits
- For intuition why this not affect the running time, suppose we alternate between best-case and worst-case flips



Average case over all inputs

Intuition

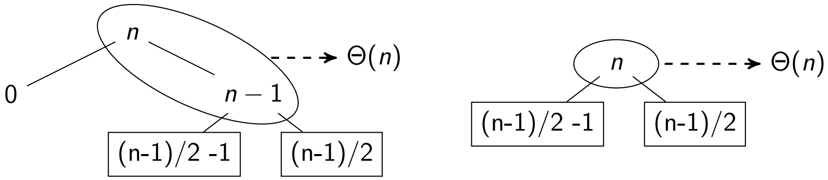
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence $T(n) = T(9n/10) + T(n/10) + \Theta(n) = \Theta(n \lg n)$
- Splits in the recursion tree will not always be good, there will be usually a mix of good and bad splits
- For intuition why this not affect the running time, suppose we alternate between best-case and worst-case flips



Average case over all inputs

Intuition

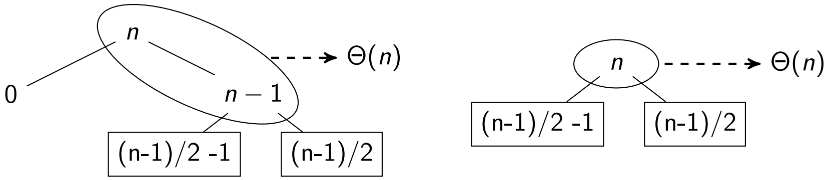
- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence $T(n) = T(9n/10) + T(n/10) + \Theta(n) = \Theta(n \lg n)$
- Splits in the recursion tree will not always be good, there will be usually a mix of good and bad splits
- For intuition why this not affect the running time, suppose we alternate between best-case and worst-case flips



Average case over all inputs

Intuition

- Imagine that PARTITION always produces a 9-to-1 split.
- Get the recurrence $T(n) = T(9n/10) + T(n/10) + \Theta(n) = \Theta(n \lg n)$
- Splits in the recursion tree will not always be good, there will be usually a mix of good and bad splits
- For intuition why this not affect the running time, suppose we alternate between best-case and worst-case flips



Both trees have the same asymptotic running time: $\Theta(n \lg n)$

RANDOMIZED VERSION

OF QUICK SORT

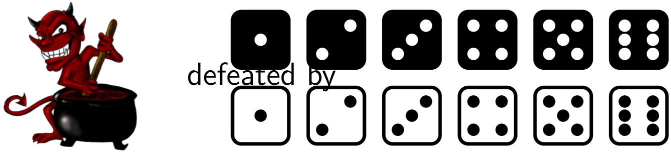
RANDOMIZED VERSION

OF QUICK SORT

defeated by

RANDOMIZED VERSION

OF QUICK SORT



RANDOMIZED VERSION

OF QUICK SORT

Randomized version of quick sort

Advantages

- We saw intuition for good running time when all permutations of input are equally likely
- This is not always true

Randomized version of quick sort

Advantages

- We saw intuition for good running time when all permutations of input are equally likely
- This is not always true
- To correct this and remove the possibilities for enemies we add randomization

Randomized version of quick sort

Advantages

- We saw intuition for good running time when all permutations of input are equally likely
- This is not always true
- To correct this and remove the possibilities for enemies we add randomization
- **HUGE difference between**

Expected running time over all inputs

and

Expected running time for any input

How to use randomization

- We could randomly permute input array

How to use randomization

- We could randomly permute input array
- Instead we use **random sampling** or picking one element in random

How to use randomization

- We could randomly permute input array
- Instead we use **random sampling** or picking one element in random
- Don't always use $A[r]$ as the pivot. Instead, randomly pick an element from the subarray that is being sorted

Randomized quick sort

```
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)
```

Analysis

Time to wake up!

Example

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
7	6	2	3	1	5	10	12	9	15	4

Example

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
7	6	2	3	1	5	10	12	9	15	4

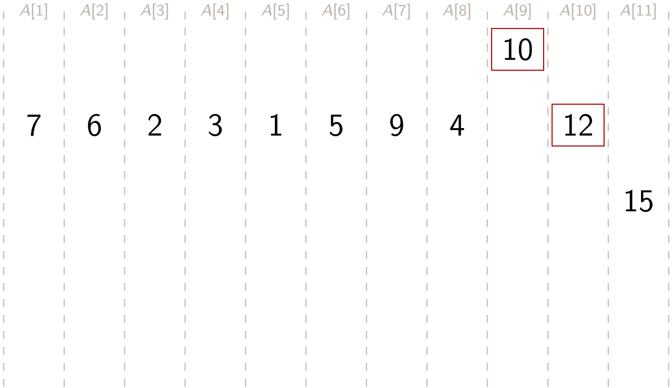
Example

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
								10		
7	6	2	3	1	5	9	4		15	12

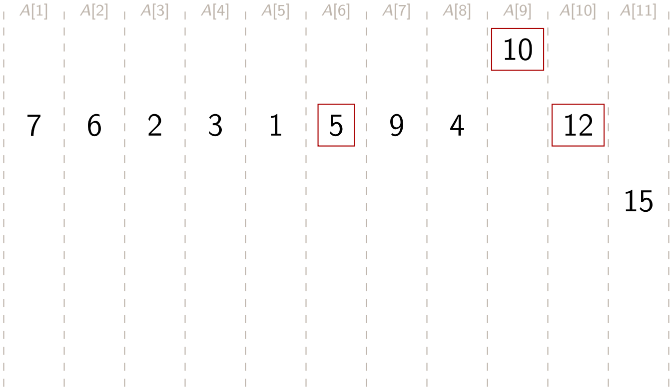
Example

$A[1]$	$A[2]$	$A[3]$	$A[4]$	$A[5]$	$A[6]$	$A[7]$	$A[8]$	$A[9]$	$A[10]$	$A[11]$
								10		
7	6	2	3	1	5	9	4		15	12

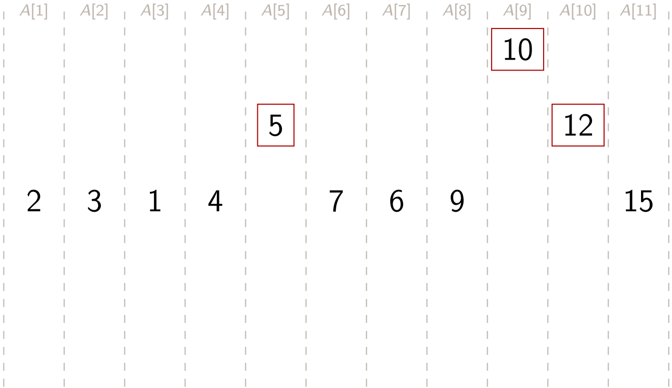
Example



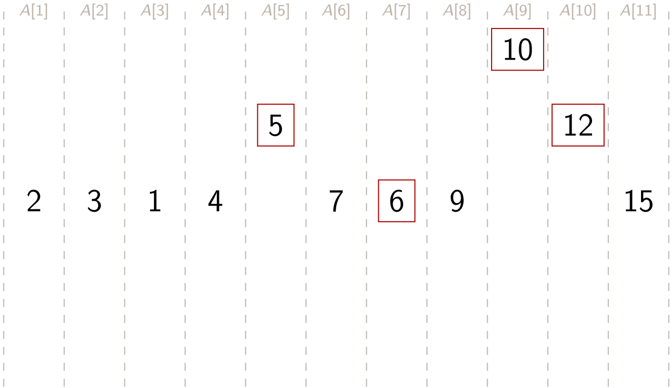
Example



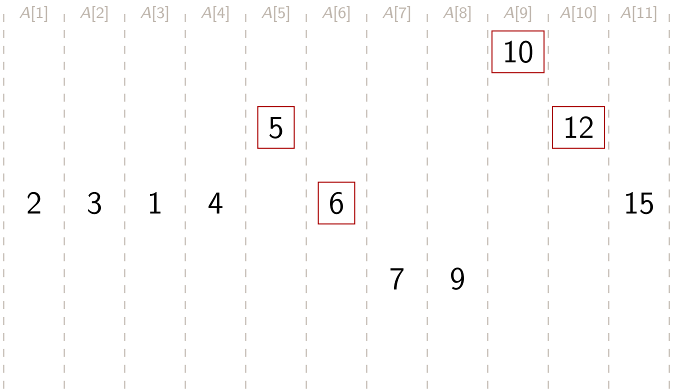
Example



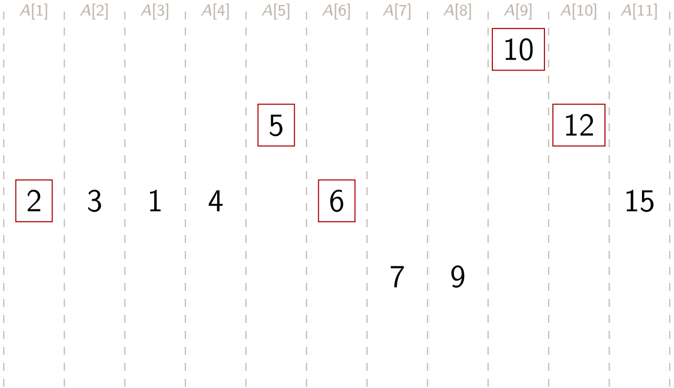
Example



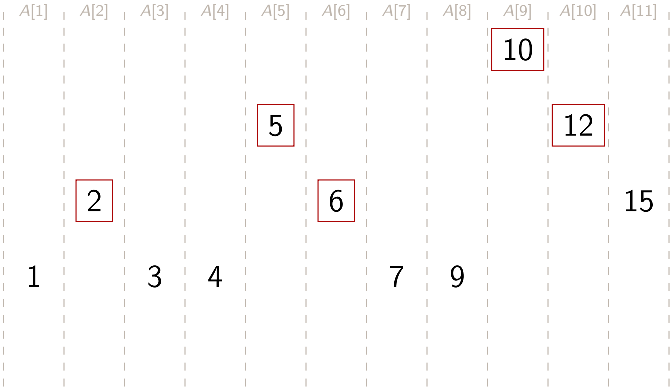
Example



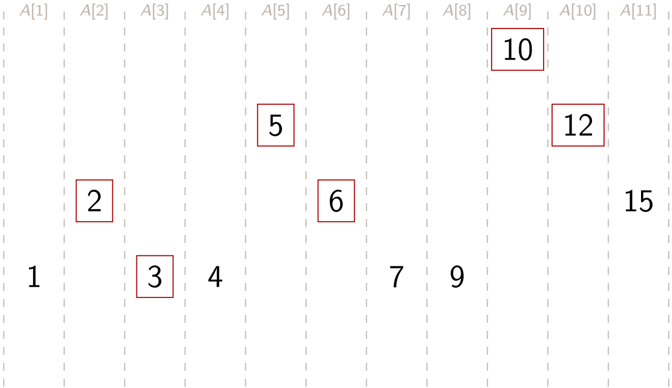
Example



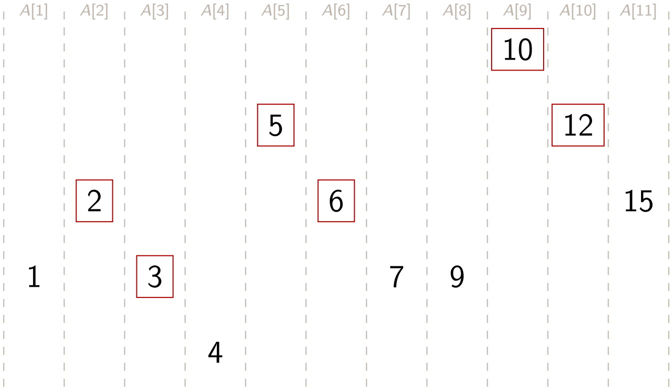
Example



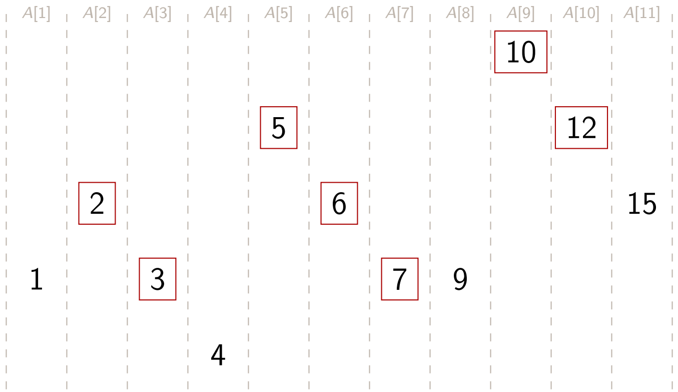
Example



Example

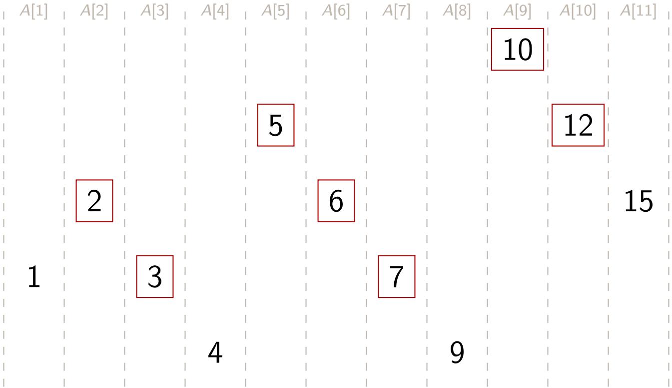


Example



Time it takes is number of calls to PARTITION + total number of comparisons

Example



Time it takes is number of calls to PARTITION + total number of comparisons

Average-case analysis

- The dominant cost of the algorithm is partitioning

Average-case analysis

- The dominant cost of the algorithm is partitioning
- Total amount of work of each call to PARTITION is a constant plus the number of comparisons that are performed in the **for** loop

Average-case analysis

- The dominant cost of the algorithm is partitioning
- Total amount of work of each call to PARTITION is a constant plus the number of comparisons that are performed in the **for** loop
- An element is a pivot at most once $\Rightarrow$ PARTITION is called at most n times

Average-case analysis

- The dominant cost of the algorithm is partitioning
- Total amount of work of each call to PARTITION is a constant plus the number of comparisons that are performed in the **for** loop
- An element is a pivot at most once $\Rightarrow$ PARTITION is called at most n times
- Let $X =$ the total number of comparisons performed in *all calls* to PARTITION

Average-case analysis

- The dominant cost of the algorithm is partitioning
- Total amount of work of each call to PARTITION is a constant plus the number of comparisons that are performed in the **for** loop
- An element is a pivot at most once $\Rightarrow$ PARTITION is called at most n times
- Let $X =$ the total number of comparisons performed in *all calls* to PARTITION
- Then the total work done over the entire execution is $O(n + X)$

Average-case analysis

- The dominant cost of the algorithm is partitioning
- Total amount of work of each call to PARTITION is a constant plus the number of comparisons that are performed in the **for** loop
- An element is a pivot at most once $\Rightarrow$ PARTITION is called at most n times
- Let $X =$ the total number of comparisons performed in *all calls* to PARTITION
- Then the total work done over the entire execution is $O(n + X)$
- We proceed by bounding (the expected value of) X

Bound on the overall number of comparisons

For ease of notation:

- Rename elements of A as $z_1, \ldots, z_n$, with z_i being the ith smallest element
- Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$

Bound on the overall number of comparisons

For ease of notation:

- Rename elements of A as $z_1, \ldots, z_n$, with z_i being the ith smallest element
- Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$

Random indicator variables:

- Let $X_{ij} = I\{z_i \text{ is compared to } z_j\}$

Bound on the overall number of comparisons

For ease of notation:

- Rename elements of A as $z_1, \ldots, z_n$, with z_i being the ith smallest element
- Define the set $Z_{ij} = \{z_i, z_{i+1}, \ldots, z_j\}$

Random indicator variables:

- Let $X_{ij} = I\{z_i \text{ is compared to } z_j\}$
- As each pair is compared at most once (when one of them is the pivot), the total number of comparisons formed by the algorithm is

$$X = \sum_{i=1}^{n} \sum_{j=i+1}^{n} X_{ij}$$

Applying linearity of expectation

The expected total number of comparisons is

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

Applying linearity of expectation

The expected total number of comparisons is

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

$$= \sum_{i=1}^{n}\sum_{j=i+1}^{n} \mathbb{E}\left[X_{ij}\right]$$

Applying linearity of expectation

The expected total number of comparisons is

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

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \mathbb{E}\left[X_{ij}\right]$$

$$= \sum_{i=1}^{n} \sum_{j=i+1}^{n} \Pr[z_i \text{ is compared to } z_j]$$

Probability that z_i is compared to z_j

- If a pivot x such that $z_i < x < z_j$ is chosen, then z_i and z_j will never be compared at any later time

Probability that z_i is compared to z_j

- If a pivot x such that $z_i < x < z_j$ is chosen, then z_i and z_j will never be compared at any later time
- If either z_i or z_j is chosen before any other element of Z_{ij}, then it will be compared to all the elements of Z_{ij}, except itself

Probability that z_i is compared to z_j

- If a pivot x such that $z_i < x < z_j$ is chosen, then z_i and z_j will never be compared at any later time
- If either z_i or z_j is chosen before any other element of Z_{ij}, then it will be compared to all the elements of Z_{ij}, except itself
- The probability that z_i is compared to z_j is the probability that either z_i or z_j is the element first chosen.

Probability that z_i is compared to z_j

- If a pivot x such that $z_i < x < z_j$ is chosen, then z_i and z_j will never be compared at any later time
- If either z_i or z_j is chosen before any other element of Z_{ij}, then it will be compared to all the elements of Z_{ij}, except itself
- The probability that z_i is compared to z_j is the probability that either z_i or z_j is the element first chosen.
- There are $j - i + 1$ elements and pivots are chosen randomly and independently. Thus the probability that any particular one of them is the first one chose is $1/(j - i + 1)$.

Probability that z_i is compared to z_j

- If a pivot x such that $z_i < x < z_j$ is chosen, then z_i and z_j will never be compared at any later time
- If either z_i or z_j is chosen before any other element of Z_{ij}, then it will be compared to all the elements of Z_{ij}, except itself
- The probability that z_i is compared to z_j is the probability that either z_i or z_j is the element first chosen.
- There are $j - i + 1$ elements and pivots are chosen randomly and independently. Thus the probability that any particular one of them is the first one chose is $1/(j - i + 1)$.
- Therefore

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

Wrapping up

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[z_i \text{ is compared to } z_j]$$

Wrapping up

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[z_i \text{ is compared to } z_j]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j - i + 1}$$

Wrapping up

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[z_i \text{ is compared to } z_j]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

Wrapping up

$$\mathbb{E}[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[z_i \text{ is compared to } z_j]$$

$$= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1}$$

$$= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k}$$

Wrapping up

$$
\begin{aligned}
\mathbb{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[z_i \text{ is compared to } z_j] \\
&= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n)
\end{aligned}
$$

Summary of quick sort

- We have proved that randomized quick sort has expected running time $O(n \lg n)$ for any input.

Summary of quick sort

- We have proved that randomized quick sort has expected running time $O(n \lg n)$ for any input.
- The algorithm is in-place

Summary of quick sort

- We have proved that randomized quick sort has expected running time $O(n \lg n)$ for any input.
- The algorithm is in-place
- Very efficient and easy to implement in practice