COMP90038
Algorithms and Complexity

Lecture 24: Some Revision
(with thanks to Harald Søndergaard)

Toby Murray

✉ toby.murray@unimelb.edu.au

DMD 8.17 (Level 8, Doug McDonell Bldg)

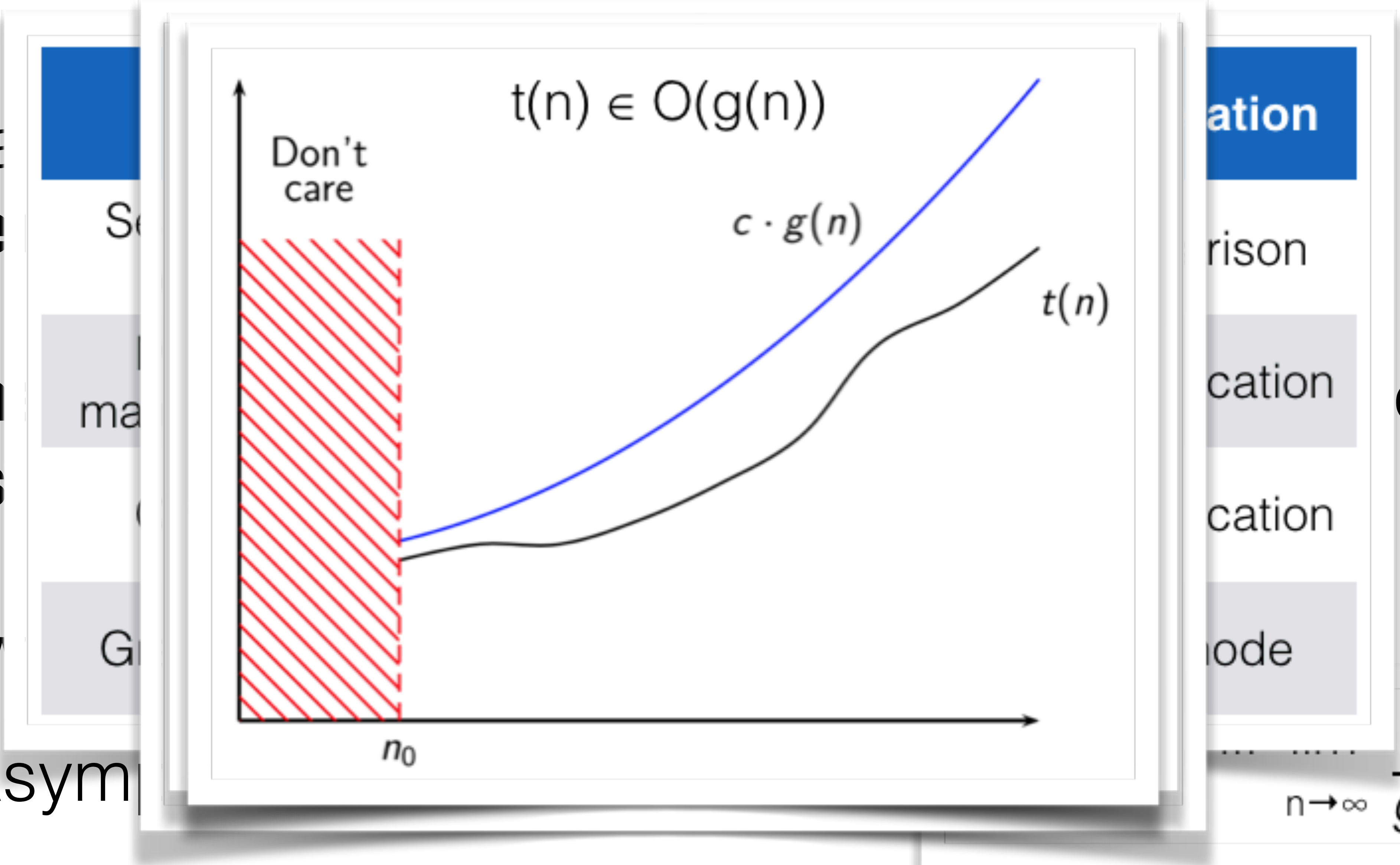
http://people.eng.unimelb.edu.au/tobym

@tobycmurray

COMPLEXITY ANALYSIS

Time Complexity

- Measure **input size** by natural number *n*
- Mea
  ope
- **Tim** of
  bas
- How
  - Asym
  - $O(g(n))$, $\Omega(g(n))$, $\Theta(g(n))$



$$_{n\to\infty} \quad \frac{f(n)}{g(n)} = 0$$

O, Ω, Θ

What this tells us about how $f(n)$ relates to $O(g(n))$, $\Omega(g(n))$ and $\Theta(g(n))$

- $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \begin{cases} 0 & f \text{ grows asymptotically slower than } g \\ c & f \text{ and } g \text{ have same order of growth} \\ \infty & f \text{ grows asymptotically faster than } g \end{cases}$$

$f(n) \in \Theta(g(n))$

Analysing Recursive Algorithms

```
function $F(n)$
    if $n = 0$ then return 1
    else return $F(n - 1) \cdot n$
```

Basic operation: multiplication

We express the cost **recursively** (as a **recurrence relation**)

$$\begin{aligned} M(0) &= \quad 0 \\ M(n) &= \qquad\qquad\qquad 1 \end{aligned}$$

Need to express M(n) in **closed form** (i.e. non-recursively)

Try: **“telescoping”** aka **“backward substitution”**

Telescoping (aka Backward Substitution)

$$\boxed{M(n) = M(n-1) + 1} \qquad \boxed{M(0) = 0}$$

What is M(n-1) ?

$$\begin{aligned} M(n-1) &= M((n-1)-1) + 1 \\ &= M(n-2) + 1 \end{aligned}$$

$$\begin{aligned} M(n) &= (M(n-2)+1)+1 \\ &= M(n-2)+2 \\ &= (M(n-3)+1)+2 \\ &= M(n-3)+3 \\ &\dots \\ &= M(n-n)+n \\ &= M(0)+n \\ &= n \end{aligned}$$

Closed form:

$$\boxed{M(n) = n}$$

Complexity:

$$\boxed{M(n) \in \Theta(n)}$$

The Master Theorem

- (A proof is in Levitin's Appendix B.)

- For integer constants $a \geq 1$ and $b > 1$, and function f with $f(n) \in \Theta(n^d)$, $d \geq 0$, the recurrence

$$T(n) = aT(n/b) + f(n)$$

(with T(1) = c) has solutions, and

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

- Note that we also allow a to be greater than b.

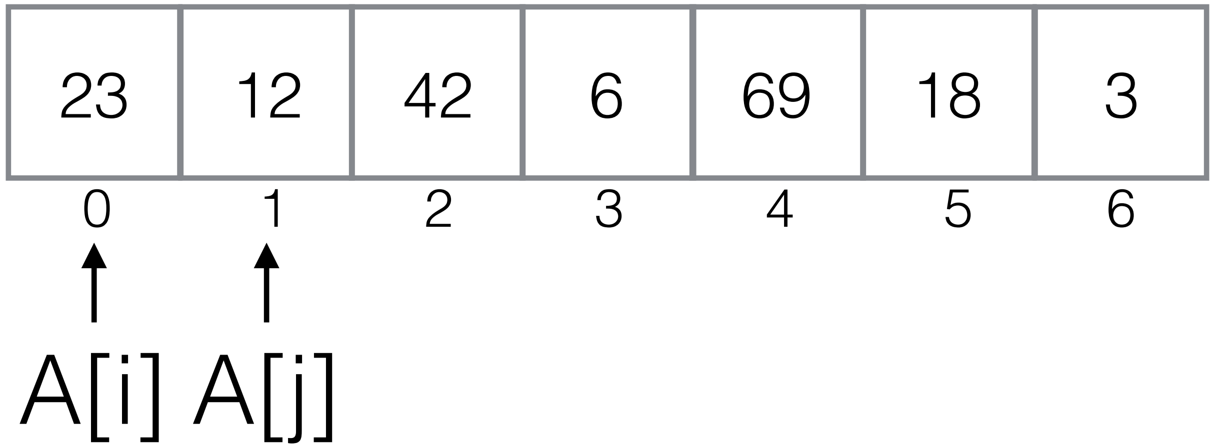
SORTING ALGORITHMS

Properties of Sorting Algorithms

- A Sorting algorithm is:
  - **in-place** if it does not require additional memory except, perhaps, for a few units of memory
  - **stable** if it preserves the relative order of elements with identical keys
  - **input-insensitive** if its running time is fairly independent of input properties other than size

Example: Selection Sort

```
function SELSORT($A[\cdot], n$)
    for $i \leftarrow 0$ to $n - 2$ do
        $min \leftarrow i$
        for $j \leftarrow i + 1$ to $n - 1$ do
            if $A[j] < A[min]$ then
                $min \leftarrow j$
        swap $A[i]$ and $A[min]$
```



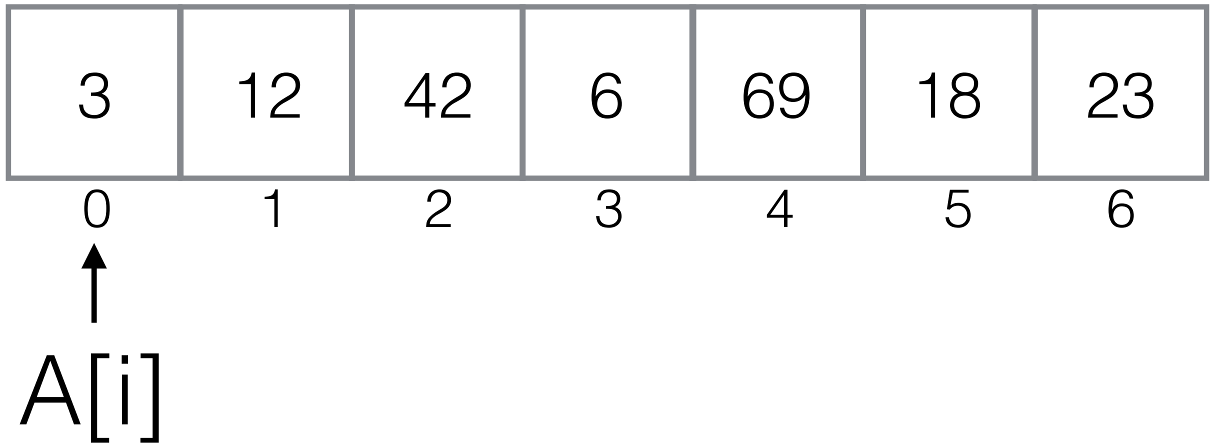
23	12	42	6	69	18	3
0	1	2	3	4	5	6

A[i] A[j]

min: 6

Example: Selection Sort

```
function SelSort($A[\cdot], n$)
    for $i \leftarrow 0$ to $n - 2$ do
        $min \leftarrow i$
        for $j \leftarrow i + 1$ to $n - 1$ do
            if $A[j] < A[min]$ then
                $min \leftarrow j$
        swap $A[i]$ and $A[min]$
```



3	12	42	6	69	18	23
0	1	2	3	4	5	6

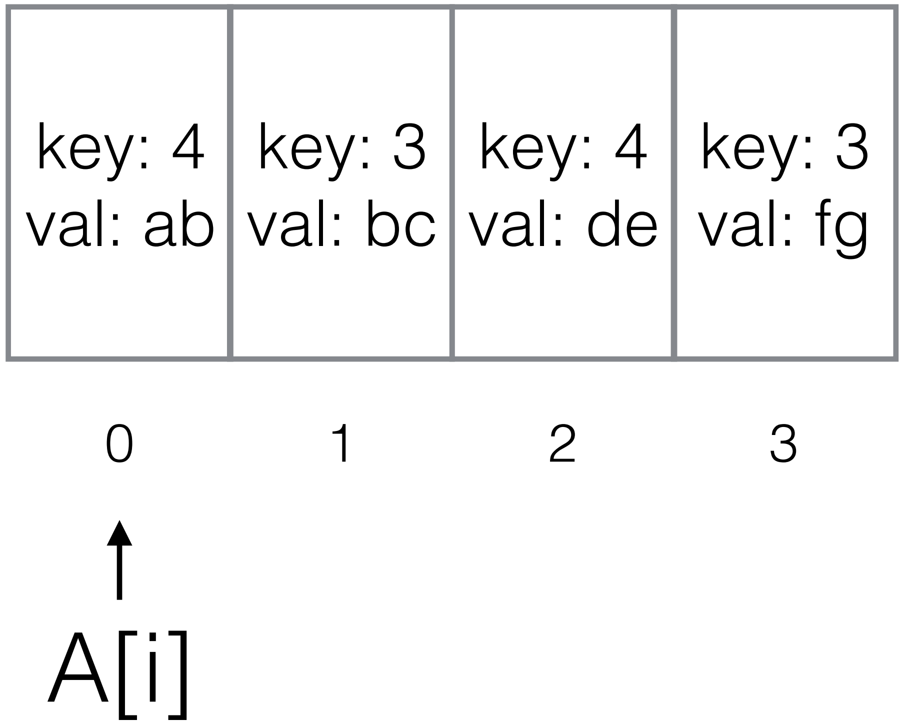
A[i]

min: 6

Properties of Selection Sort

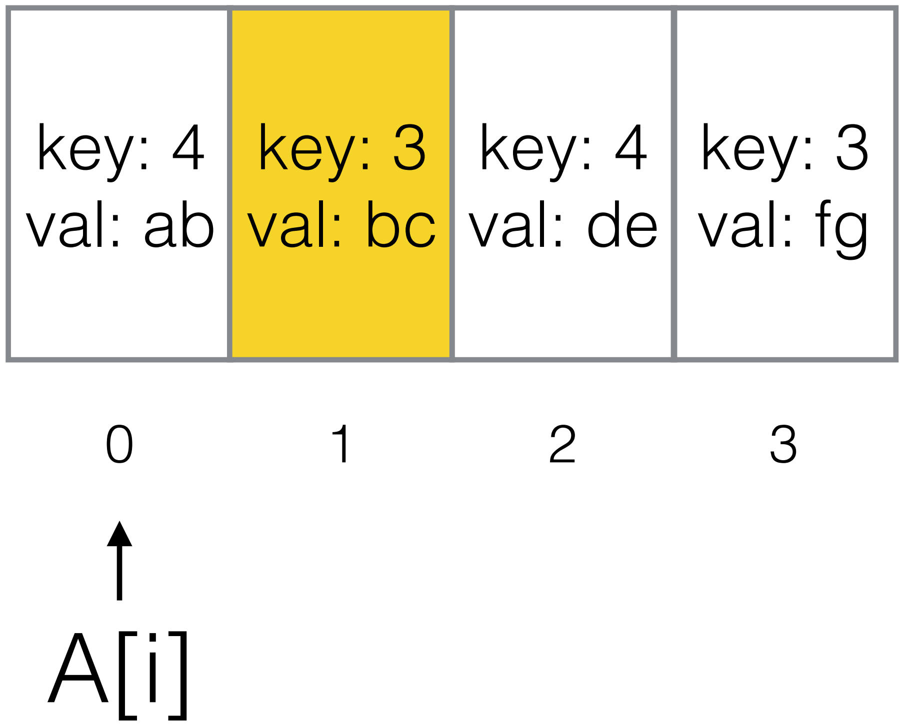
- While running time is quadratic, selection sort makes only about ***n*** **exchanges**.
- So: **selection sort is a good algorithm for sorting small collections of large records.**
- In-place? yes
- Stable? ?
- Input-insensitive? yes

Selection Sort Stability



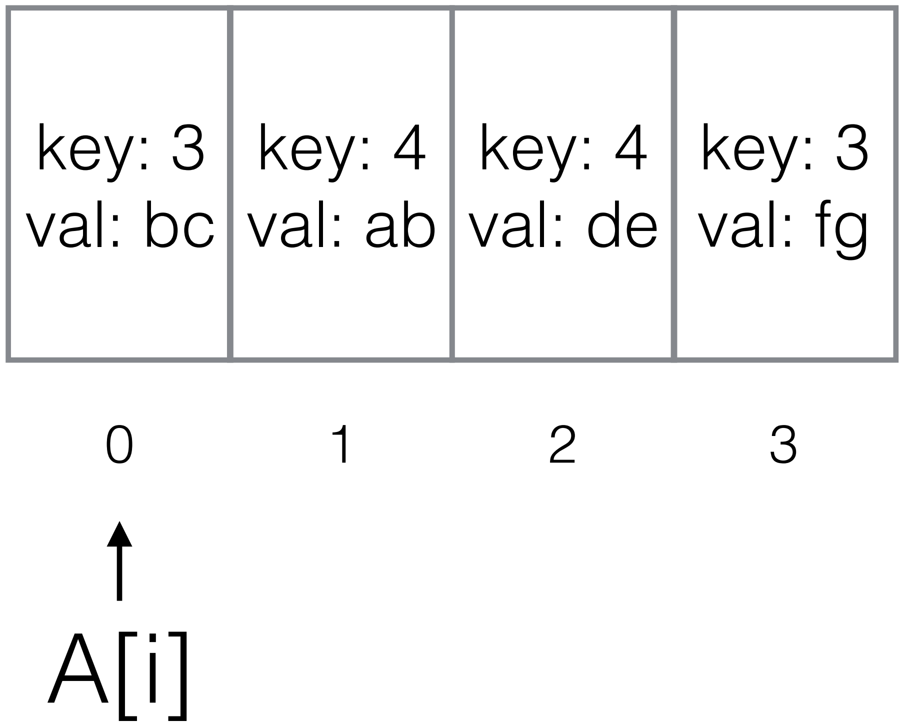
key: 4 val: ab	key: 3 val: bc	key: 4 val: de	key: 3 val: fg
0	1	2	3

Selection Sort Stability



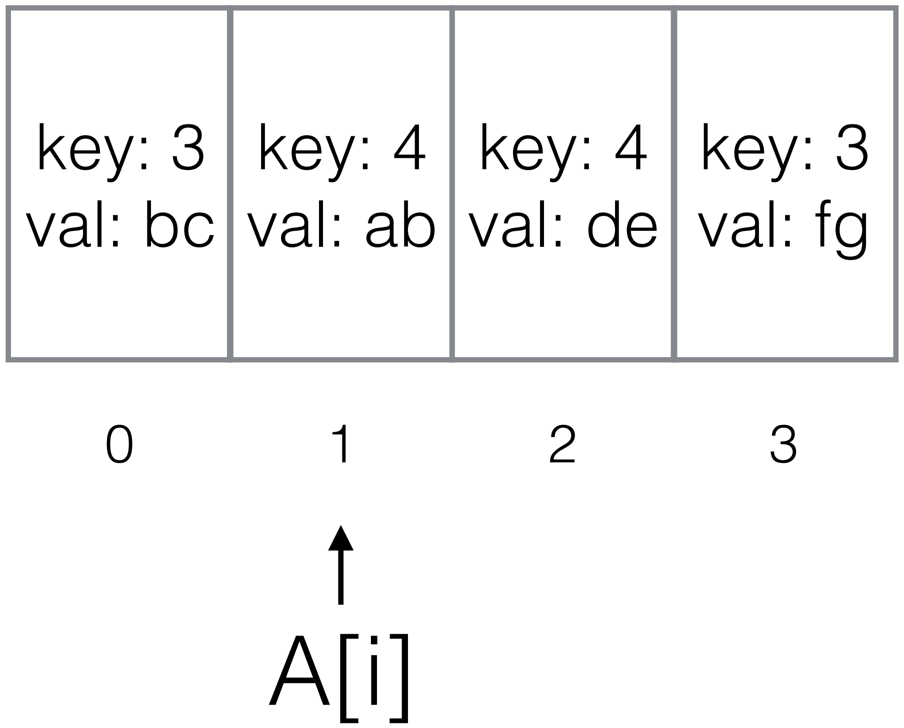
key: 4 val: ab	key: 3 val: bc	key: 4 val: de	key: 3 val: fg
0	1	2	3

Selection Sort Stability



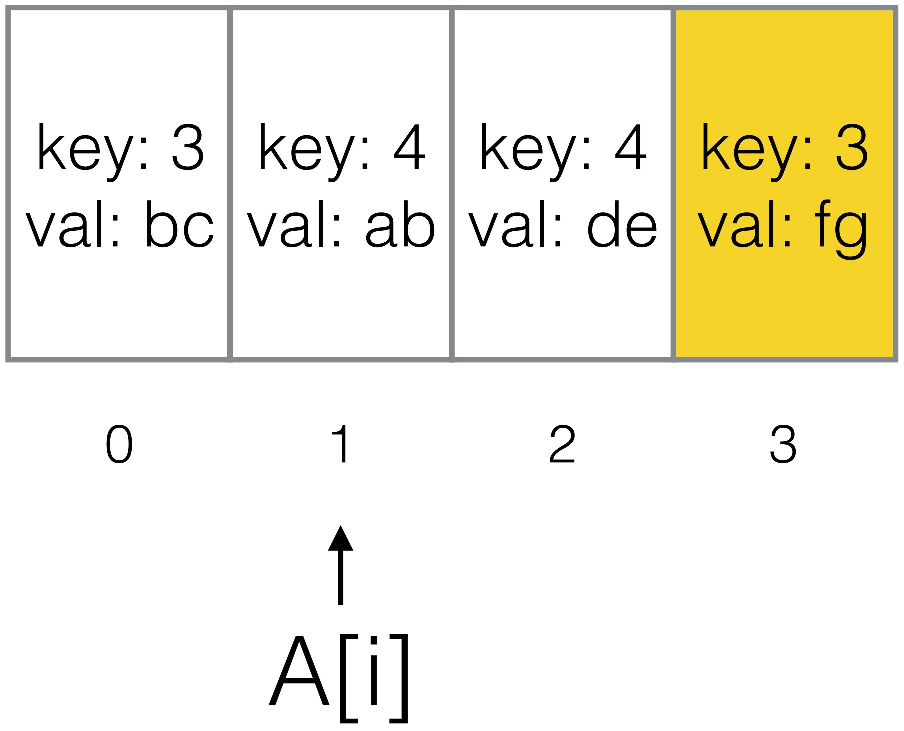
key: 3 val: bc	key: 4 val: ab	key: 4 val: de	key: 3 val: fg
0	1	2	3

Selection Sort Stability



key: 3 val: bc	key: 4 val: ab	key: 4 val: de	key: 3 val: fg
0	1	2	3

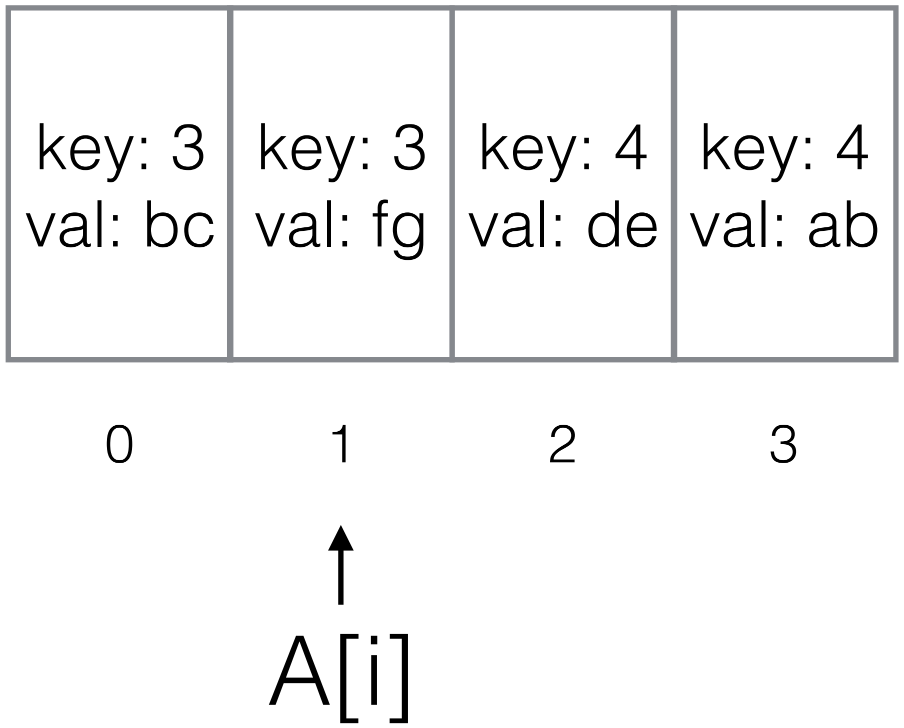
Selection Sort Stability



key: 3 val: bc	key: 4 val: ab	key: 4 val: de	key: 3 val: fg
0	1	2	3

A[i]

Selection Sort Stability



key: 3 val: bc	key: 3 val: fg	key: 4 val: de	key: 4 val: ab
0	1	2	3

the relative order
of the two "4" records
has changed!

Properties of Selection Sort

- While running time is quadratic, selection sort makes only about ***n*** **exchanges**.

- So: **selection sort is a good algorithm for sorting small collections of large records.**

- In-place? yes

- Stable? no

- Input-insensitive? yes

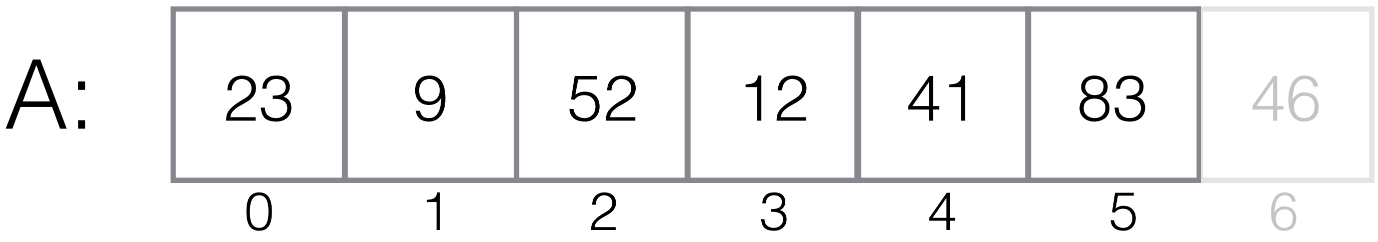
Insertion Sort

- Sorting an array A[0]..A[n – 1]:

- To sort A[0] .. A[i] first sort A[0] .. A[i-1], then insert A[i] in its proper place

```
function InsertionSort($A[\cdot], n$)
    for $i \leftarrow 1$ to $n - 1$ do
        $v \leftarrow A[i]$
        $j \leftarrow i - 1$
        while $j \geq 0$ and $v < A[j]$ do
            $A[j + 1] \leftarrow A[j]$
            $j \leftarrow j - 1$
        $A[j + 1] \leftarrow v$
```

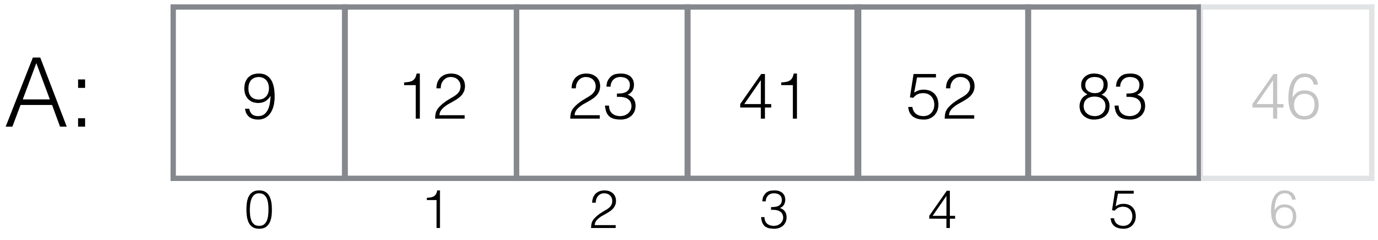
Sorting n items



A:							
	23	9	52	12	41	83	46
	0	1	2	3	4	5	6

Sort first n-1 items

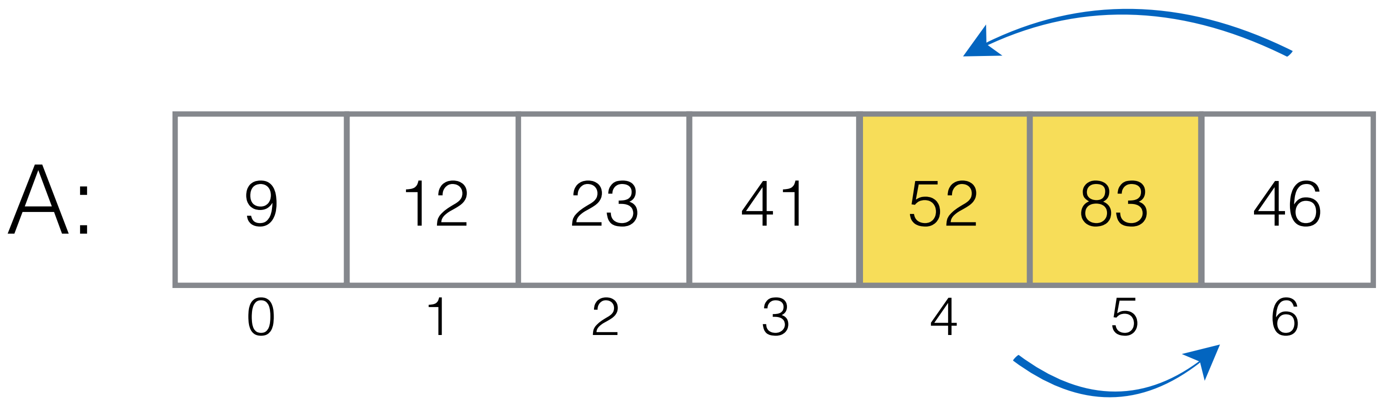
Sorting n items



A:

9	12	23	41	52	83	46
0	1	2	3	4	5	6

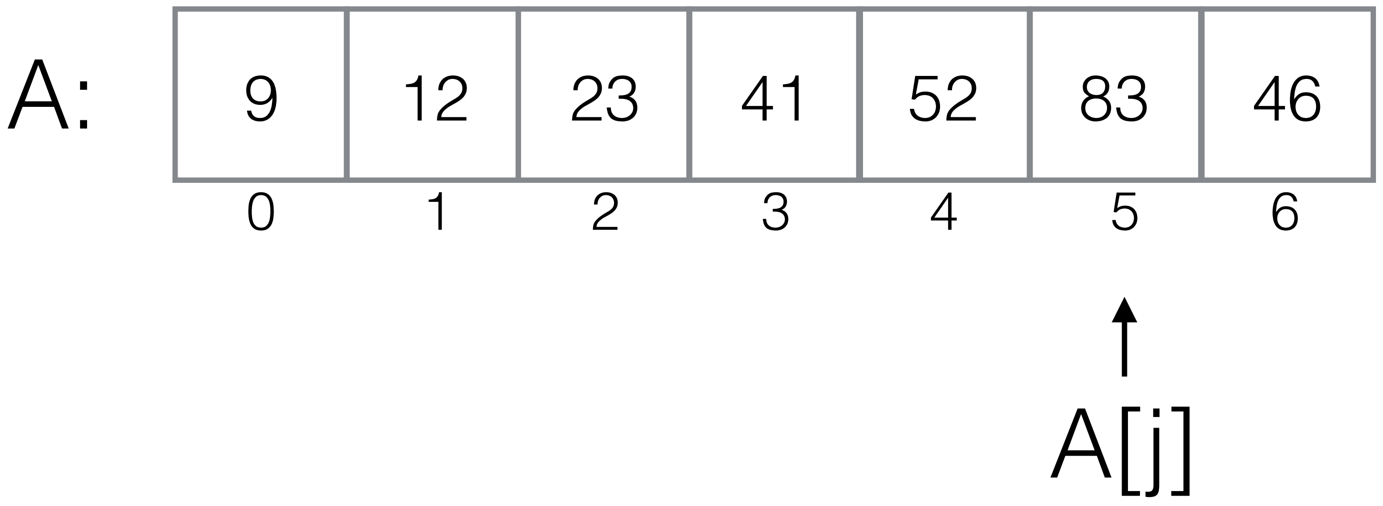
Sorting n items



A:

9	12	23	41	52	83	46
0	1	2	3	4	5	6

Sorting n items



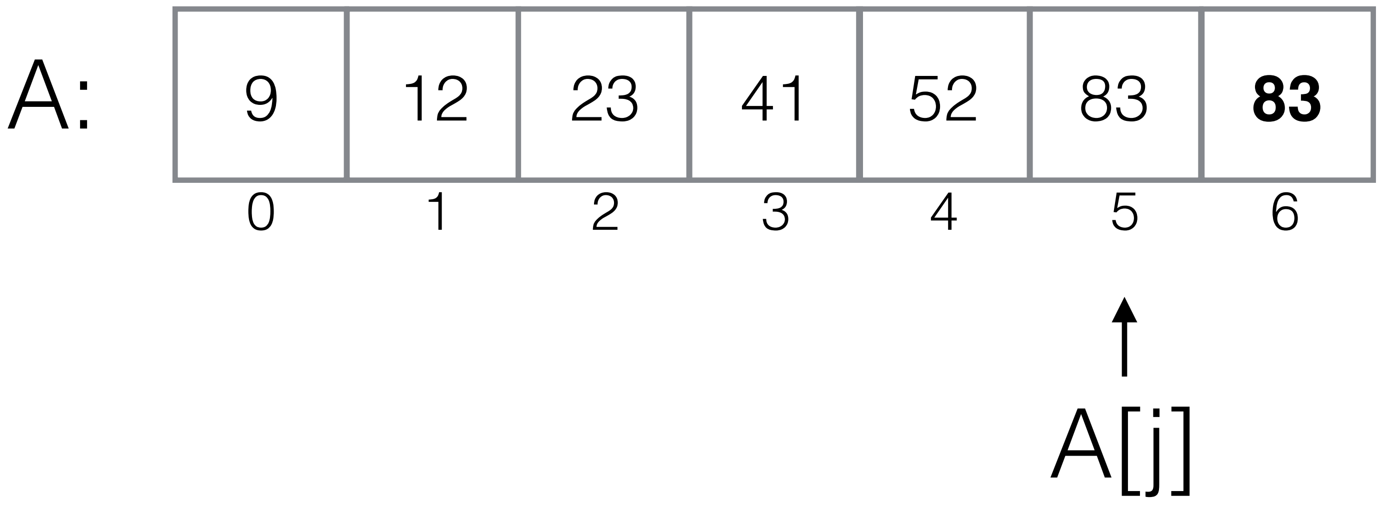
A:

9	12	23	41	52	83	46
0	1	2	3	4	5	6

A[j]

v: 46

Sorting n items



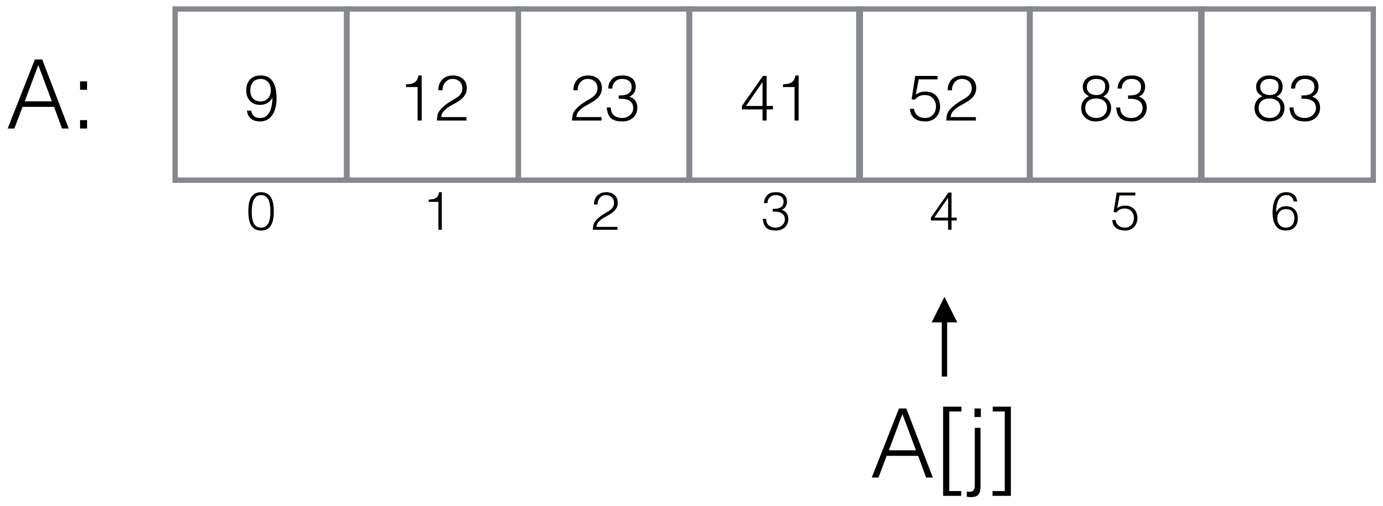
A:

9	12	23	41	52	83	**83**
0	1	2	3	4	5	6

A[j]

v: 46

Sorting n items

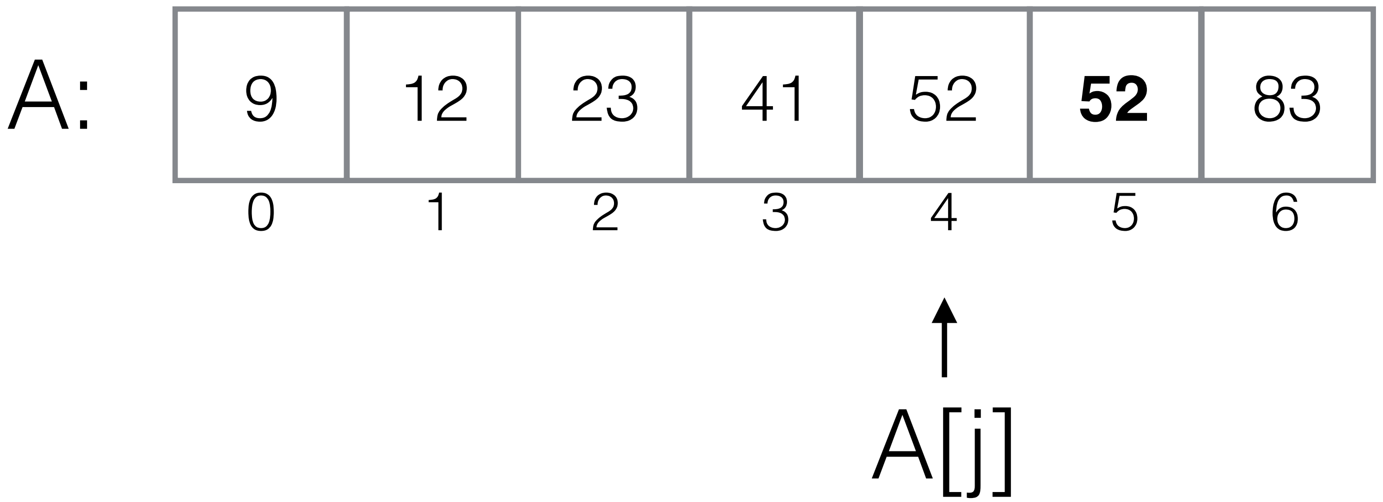


A:

9	12	23	41	52	83	83
0	1	2	3	4	5	6

v: 46

Sorting n items



A:

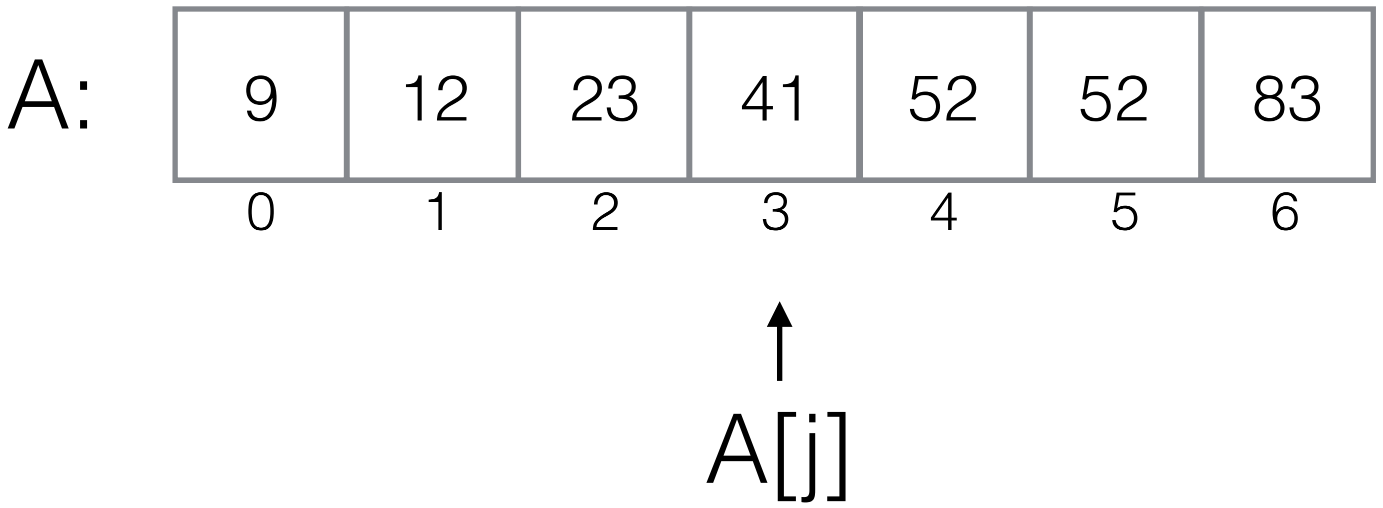
9	12	23	41	52	**52**	83
0	1	2	3	4	5	6

↑

A[j]

v: 46

Sorting n items



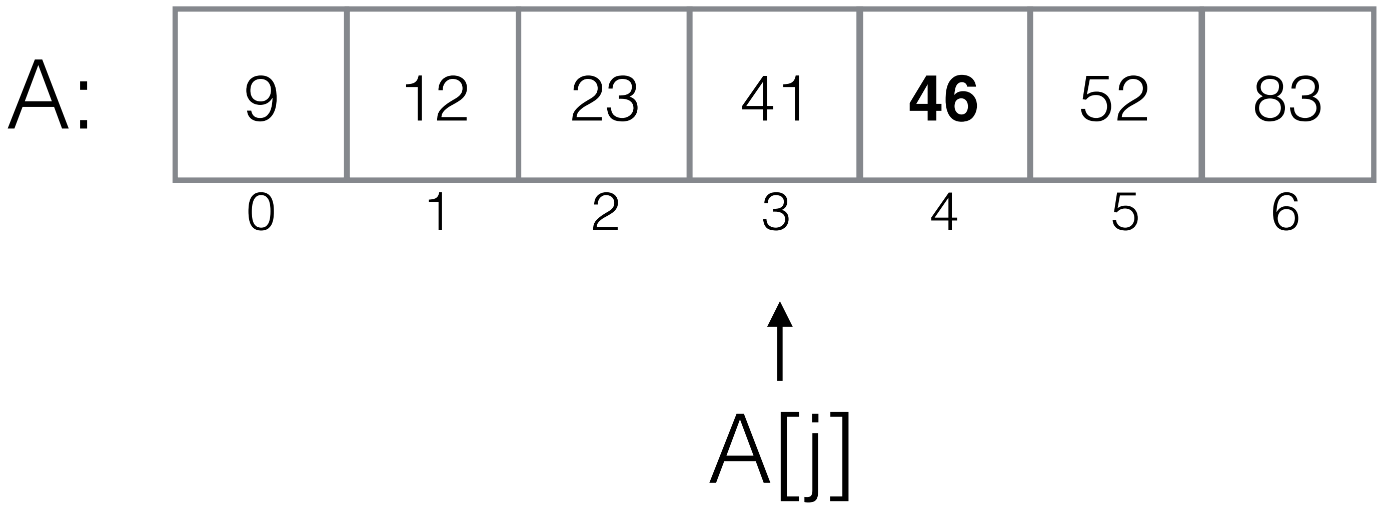
A:

9	12	23	41	52	52	83
0	1	2	3	4	5	6

A[j]

v: 46

Sorting n items



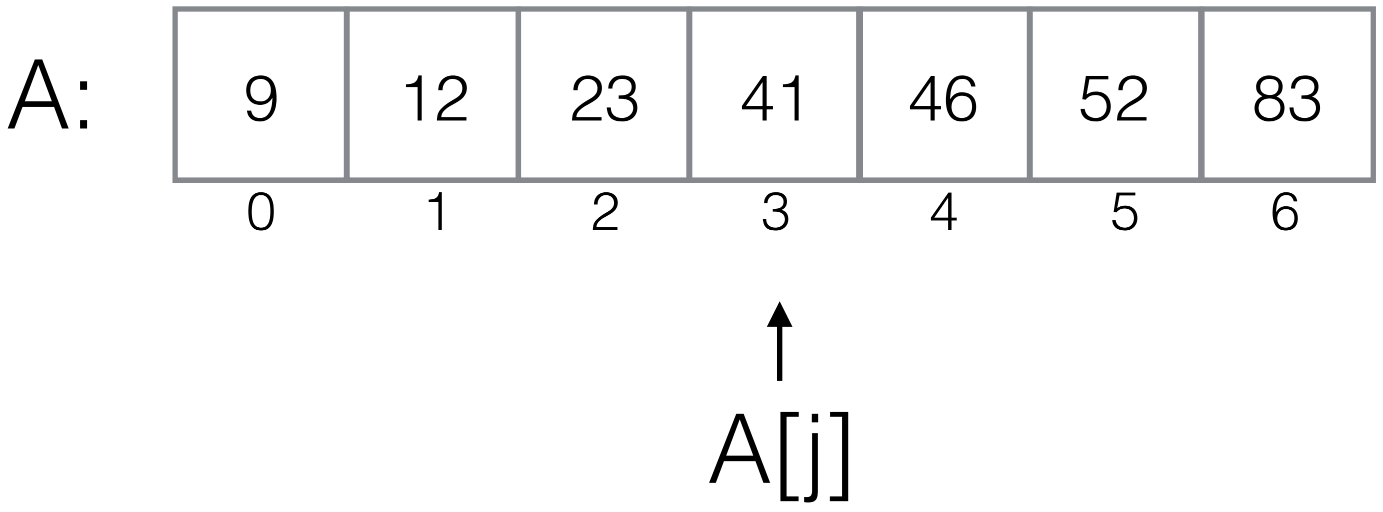
A:

9	12	23	41	**46**	52	83
0	1	2	3	4	5	6

A[j]

v: 46

Sorting n items



A:	9	12	23	41	46	52	83
	0	1	2	3	4	5	6

A[j]

v: 46

Complexity of Insertion Sort

- The for loop is traversed n – 1 times. In the ith round, the test v < A[j] is performed i times, in the worst case.

- Hence the worst-case running time is

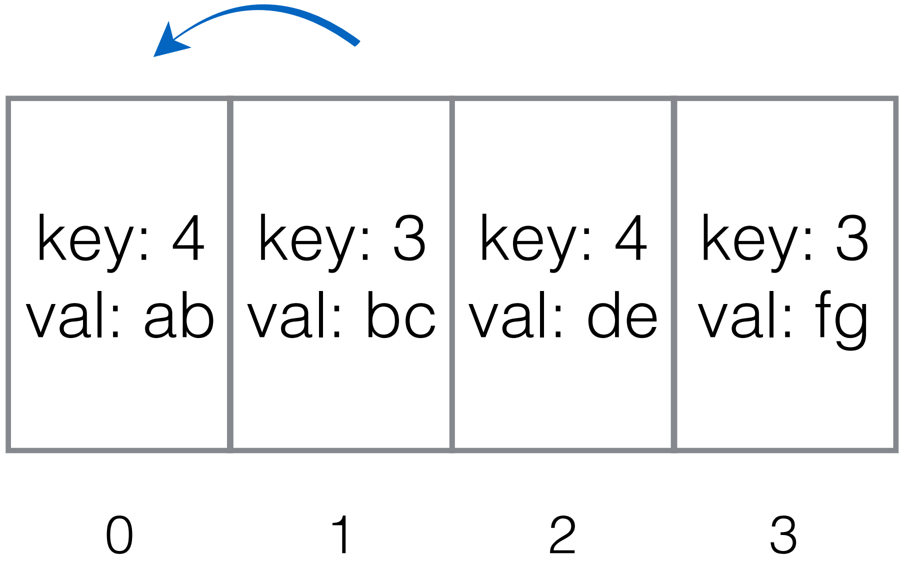
$$\sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

- What does input look like in the worst case?

Properties of Insertion Sort

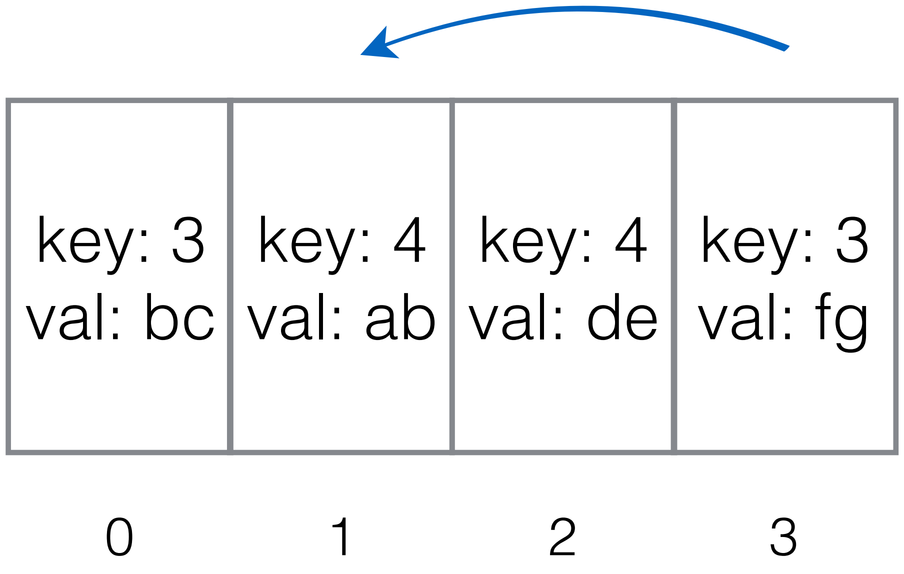
- Easy to understand and implement.
- Average-case and worst-case complexity both quadratic.
- However, linear for almost-sorted input.
- Some cleverer sorting algorithms perform almost-sorting and then let insertion sort take over.
- Very good for small arrays (say, a couple of hundred elements).
- In-place? yes
- Stable? ?

Insertion Sort Stability



key: 4 val: ab	key: 3 val: bc	key: 4 val: de	key: 3 val: fg
0	1	2	3

Insertion Sort Stability



key: 3 val: bc	key: 4 val: ab	key: 4 val: de	key: 3 val: fg
0	1	2	3

Insertion Sort Stability

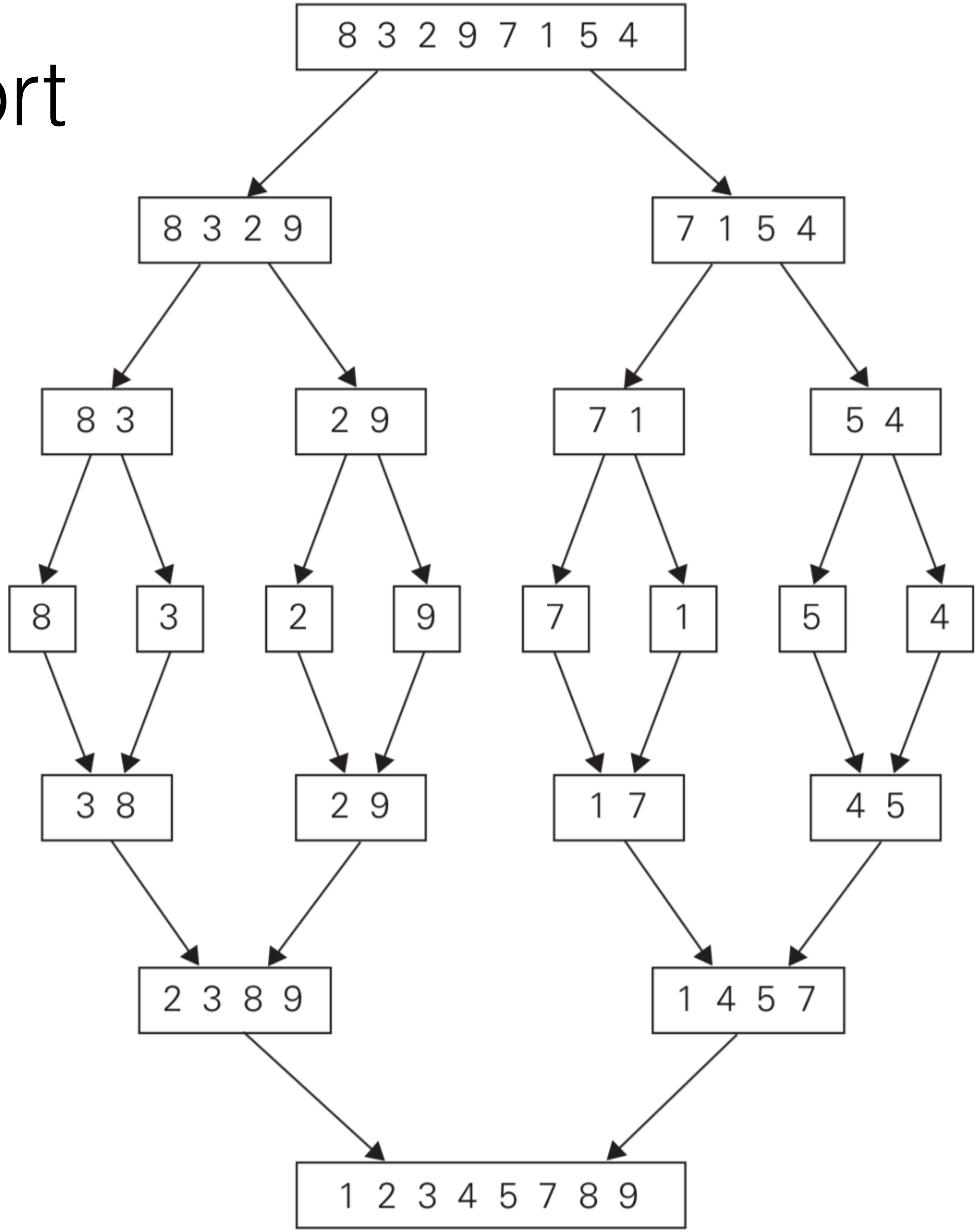
key: 3 val: bc	key: 3 val: fg	key: 4 val: ab	key: 4 val: de
0	1	2	3

Stable

Properties of Insertion Sort

- Easy to understand and implement.
- Average-case and worst-case complexity both quadratic.
- However, linear for almost-sorted input.
- Some cleverer sorting algorithms perform almost-sorting and then let insertion sort take over.
- Very good for small arrays (say, a couple of hundred elements).
- In-place? yes
- Stable? yes

Mergesort


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

Mergesort: Analysis

- How many comparisons will MERGE need to make in the worst case, when given arrays of size ⌊n/2⌋ and ⌈n/2⌉ ?

- If the largest and second-largest elements are in different arrays, then n – 1 comparisons. Hence the cost equation for Mergesort is

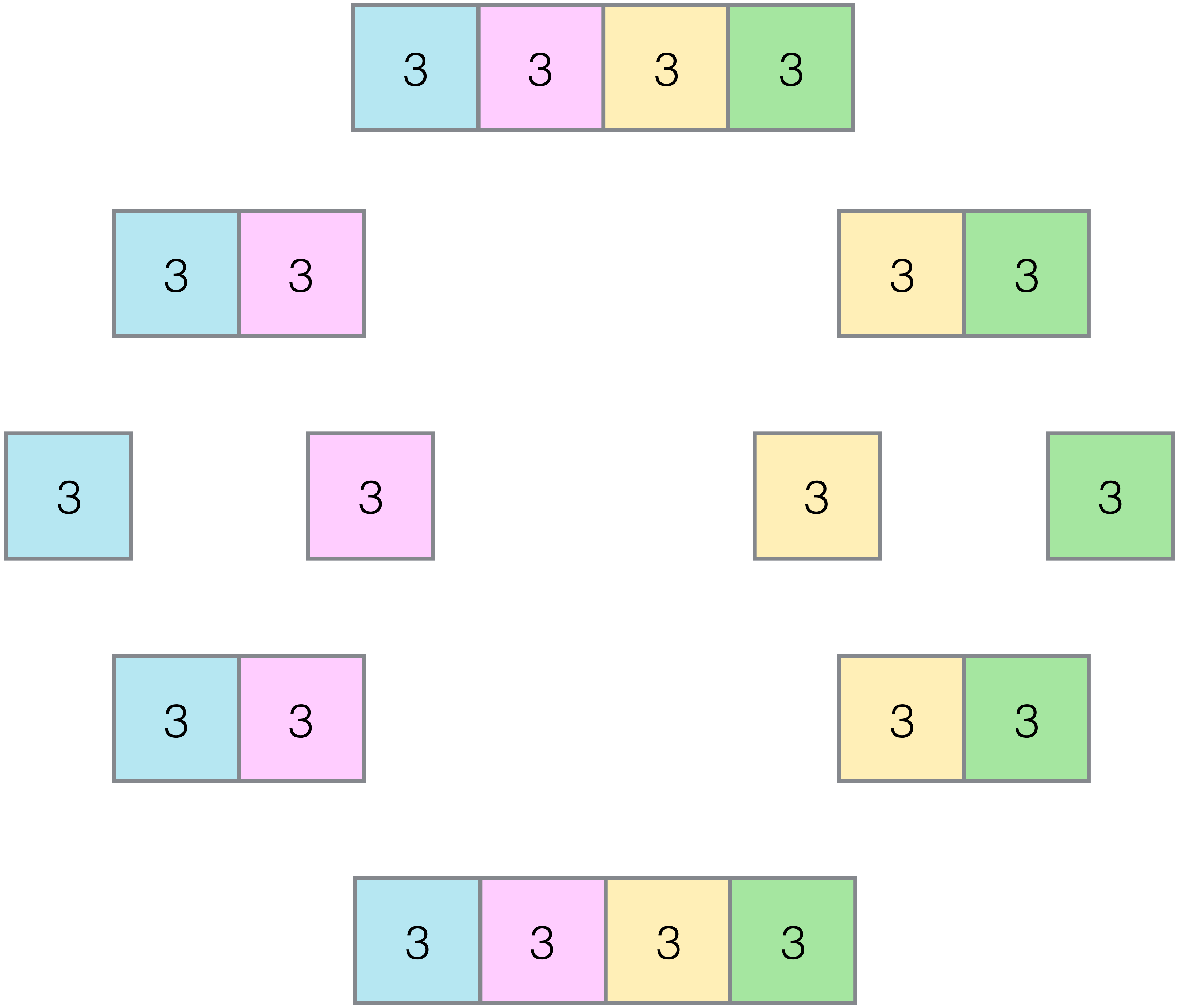
$$C(n) = \begin{cases} 0 & \text{if } n < 2 \\ 2C(n/2) + n - 1 & \text{otherwise} \end{cases}$$

- By the Master Theorem, C(n) ∊ Θ(n log n).

Mergesort: Properties

- For large n, the number of comparisons made tends to be around 75% of the worst-case scenario.
- Is mergesort stable? ?
- Is mergesort in-place? no
- If comparisons are fast, mergesort ranks between quicksort and heapsort (covered next week) for time, assuming random data.
- Mergesort is the method of choice for linked lists and for very large collections of data.

Mergesort: Stability



Mergesort: Properties

- For large n, the number of comparisons made tends to be around 75% of the worst-case scenario.
- Is mergesort stable? yes
- Is mergesort in-place? no
- If comparisons are fast, mergesort ranks between quicksort and heapsort (covered next week) for time, assuming random data.
- Mergesort is the method of choice for linked lists and for very large collections of data.

Quicksort

- Quicksort takes a divide-and-conquer approach that is different to mergesort's.

- It uses the **partitioning** idea from QuickSelect, picking a pivot element, and partitioning the array around that, so as to obtain this situation:

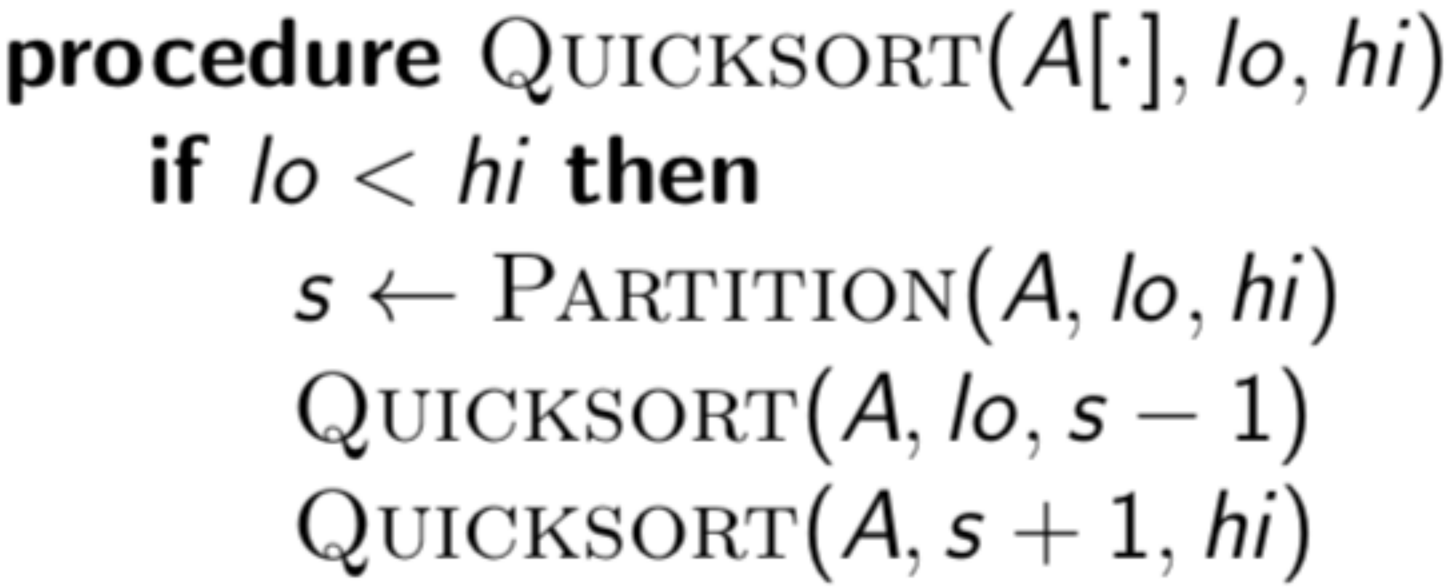
$$\underbrace{A[0]\ldots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are } \geq A[s]}$$

- The element A[s] will be in its final position (it is the (s + 1)th smallest element).

- All that then needs to be done is to sort the segment to the left, recursively, as well as the segment to the right.

Quicksort

- Very short and elegant:

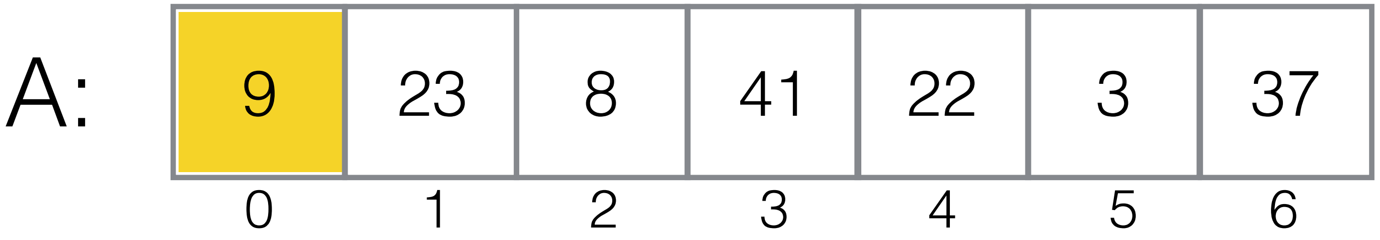


```
procedure Quicksort($A[\cdot], lo, hi$)
    if $lo < hi$ then
        $s \leftarrow$ Partition($A, lo, hi$)
        Quicksort($A, lo, s - 1$)
        Quicksort($A, s + 1, hi$)
```

- Initial call: Quicksort(A, 0, n – 1).

Quicksort: Example

```
procedure QUICKSORT($A[\cdot], lo, hi$)
    if $lo < hi$ then
        $s \leftarrow$ PARTITION($A, lo, hi$)
        QUICKSORT($A, lo, s - 1$)
        QUICKSORT($A, s + 1, hi$)
```

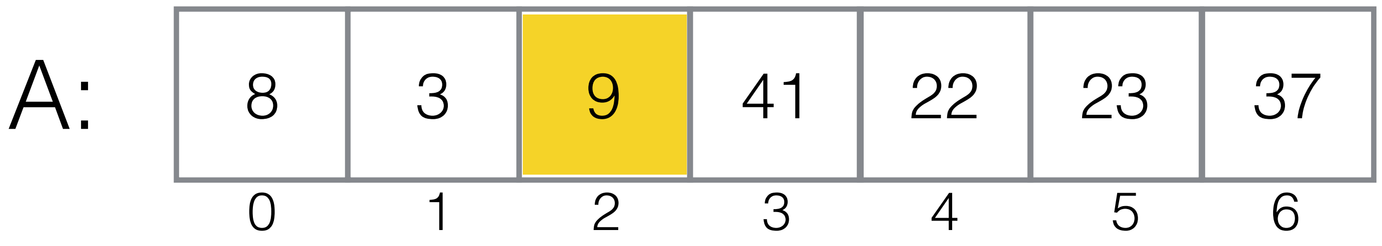


A:

9	23	8	41	22	3	37
0	1	2	3	4	5	6

Quicksort: Example

```
procedure QUICKSORT($A[\cdot]$, $lo$, $hi$)
    if $lo < hi$ then
        $s \leftarrow$ PARTITION($A$, $lo$, $hi$)
        QUICKSORT($A$, $lo$, $s - 1$)
        QUICKSORT($A$, $s + 1$, $hi$)
```

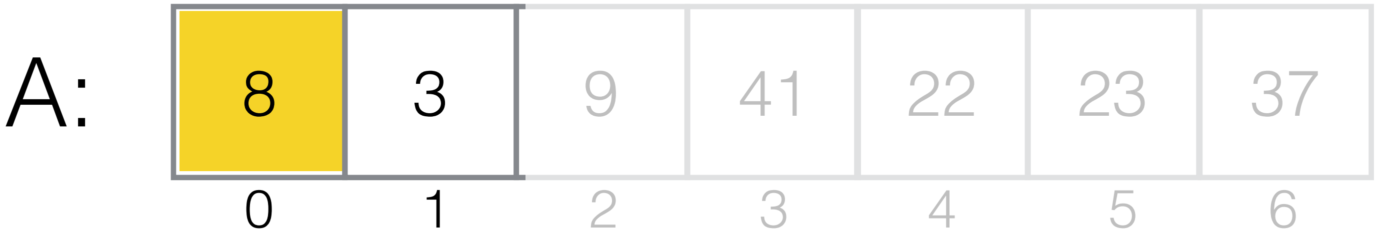


A:

8	3	9	41	22	23	37
0	1	2	3	4	5	6

Quicksort: Example

```
procedure QUICKSORT($A[\cdot]$, $lo$, $hi$)
    if $lo < hi$ then
        $s \leftarrow$ PARTITION($A$, $lo$, $hi$)
        QUICKSORT($A$, $lo$, $s - 1$)
        QUICKSORT($A$, $s + 1$, $hi$)
```

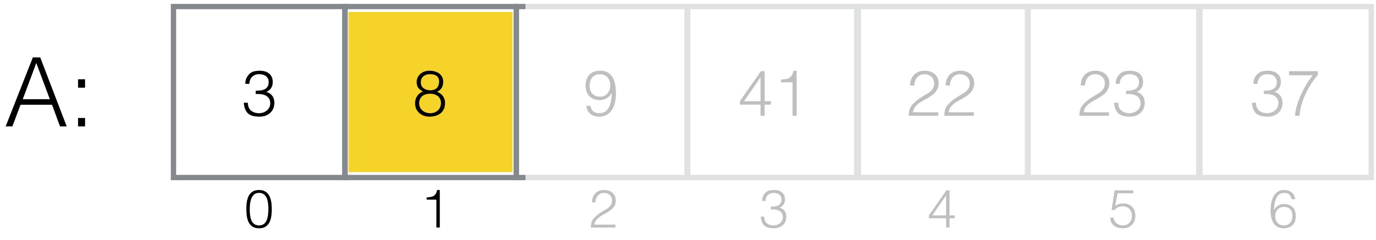


A:

8	3	9	41	22	23	37
0	1	2	3	4	5	6

Quicksort: Example

```
procedure QUICKSORT($A[\cdot], lo, hi$)
    if $lo < hi$ then
        $s \leftarrow$ PARTITION($A, lo, hi$)
        QUICKSORT($A, lo, s - 1$)
        QUICKSORT($A, s + 1, hi$)
```

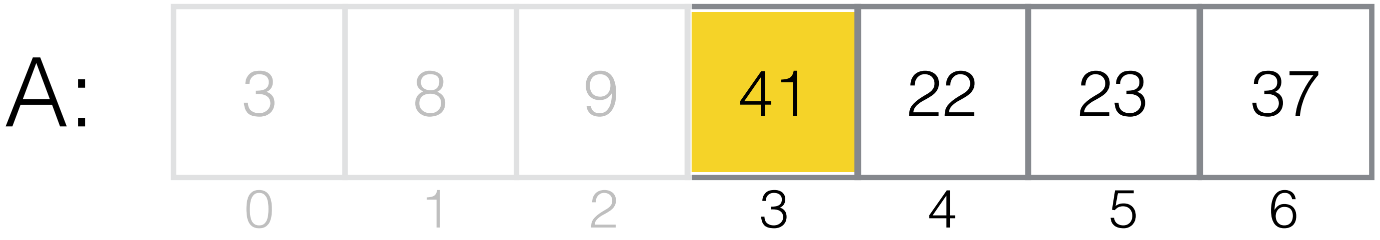


A:

3	8	9	41	22	23	37
0	1	2	3	4	5	6

Quicksort: Example

```
procedure QUICKSORT($A[\cdot], lo, hi$)
    if $lo < hi$ then
        $s \leftarrow$ PARTITION($A, lo, hi$)
        QUICKSORT($A, lo, s - 1$)
        QUICKSORT($A, s + 1, hi$)
```

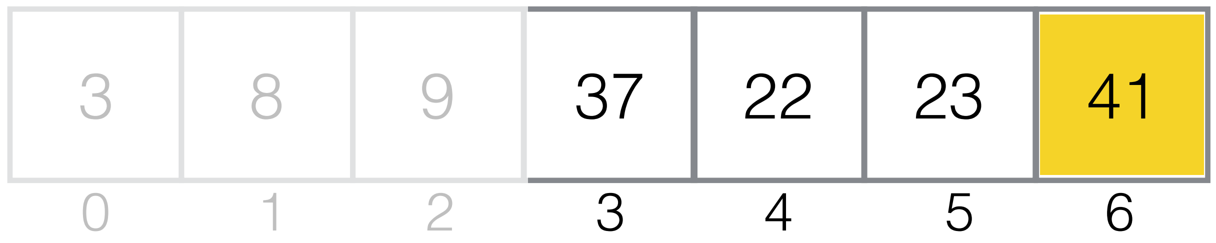


A:

3	8	9	41	22	23	37
0	1	2	3	4	5	6

Quicksort: Example

```
procedure QUICKSORT($A[\cdot], lo, hi$)
    if $lo < hi$ then
        $s \leftarrow$ PARTITION($A, lo, hi$)
        QUICKSORT($A, lo, s - 1$)
        QUICKSORT($A, s + 1, hi$)
```

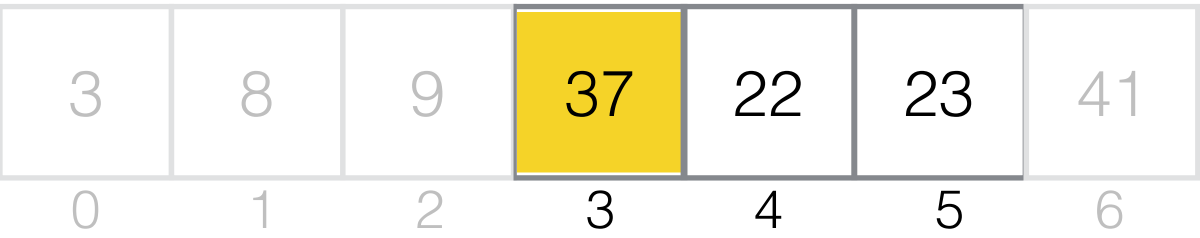


A:

3	8	9	37	22	23	41
0	1	2	3	4	5	6

Quicksort: Example

```
procedure QUICKSORT($A[\cdot], lo, hi$)
    if $lo < hi$ then
        $s \leftarrow$ PARTITION($A, lo, hi$)
        QUICKSORT($A, lo, s - 1$)
        QUICKSORT($A, s + 1, hi$)
```

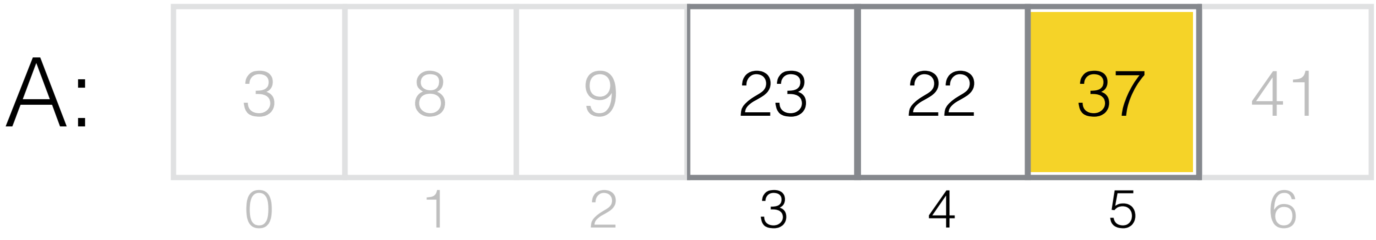


A:

3	8	9	37	22	23	41
0	1	2	3	4	5	6

Quicksort: Example

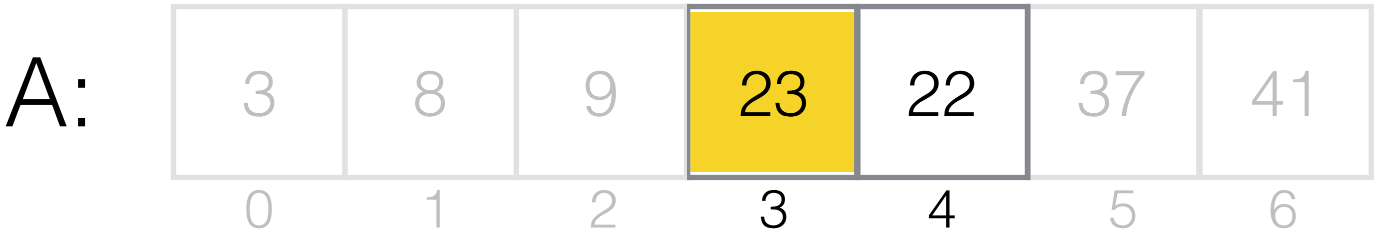
```
procedure QUICKSORT($A[\cdot], lo, hi$)
    if $lo < hi$ then
        $s \leftarrow$ PARTITION($A, lo, hi$)
        QUICKSORT($A, lo, s - 1$)
        QUICKSORT($A, s + 1, hi$)
```



A:	3	8	9	23	22	37	41
	0	1	2	3	4	5	6

Quicksort: Example

```
procedure $\textsc{Quicksort}(A[\cdot], lo, hi)$
    if $lo < hi$ then
        $s \leftarrow \textsc{Partition}(A, lo, hi)$
        $\textsc{Quicksort}(A, lo, s - 1)$
        $\textsc{Quicksort}(A, s + 1, hi)$
```

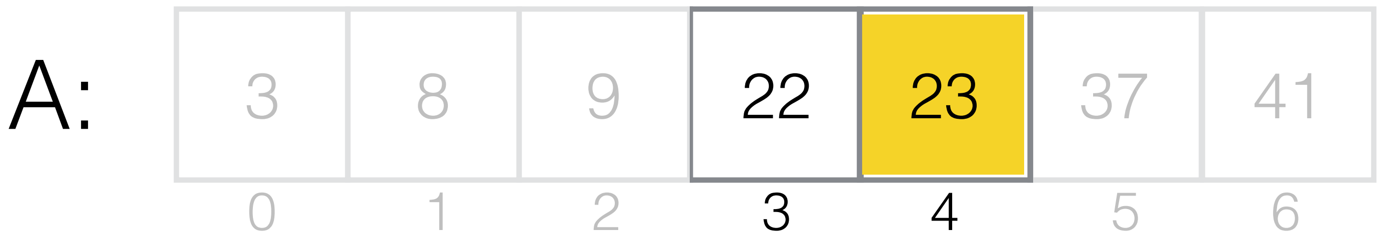


A:

3	8	9	23	22	37	41
0	1	2	3	4	5	6

Quicksort: Example

```
procedure QUICKSORT($A[\cdot]$, $lo$, $hi$)
    if $lo < hi$ then
        $s \leftarrow$ PARTITION($A$, $lo$, $hi$)
        QUICKSORT($A$, $lo$, $s - 1$)
        QUICKSORT($A$, $s + 1$, $hi$)
```



A:

3	8	9	22	23	37	41
0	1	2	3	4	5	6

Quicksort: Example

```
procedure QUICKSORT($A[\cdot], lo, hi$)
    if $lo < hi$ then
        $s \leftarrow$ PARTITION($A, lo, hi$)
        QUICKSORT($A, lo, s - 1$)
        QUICKSORT($A, s + 1, hi$)
```

A:

3	8	9	22	23	37	41
0	1	2	3	4	5	6

Quicksort Analysis: Best Case Analysis

- The best case happens when the pivot is the median; that results in two sub-tasks of equal size.

$$C_{best}(n) = \begin{cases} 0 & \text{if } n < 2 \\ 2C_{best}(n/2) + n & \text{otherwise} \end{cases}$$

The ‘n’ is for the n key comparisons performed by Partition.

- By the Master Theorem, $C_{best}(n) \in \Theta(n \log n)$, just as for mergesort, so quicksort’s best case is (asymptotically) no better than mergesort’s worst case.

Quicksort Analysis: Worst Case Analysis

- The worst case happens if the array is already sorted.

- In that case, we don't really have divide-and-conquer, because each recursive call deals with a problem size that has only been decremented by 1:

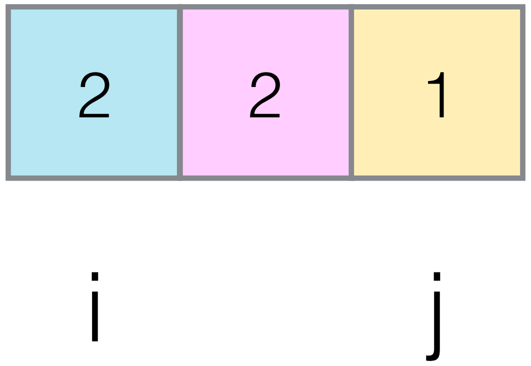
$$C_{worst}(n) = \begin{cases} 0 & \text{if } n < 2 \\ C_{worst}(n-1) + n & \text{otherwise} \end{cases}$$

- That is, $C_{worst}(n) = n + (n - 1) + \cdots + 3 + 2 \in \Theta(n^2)$.

Quicksort Properties

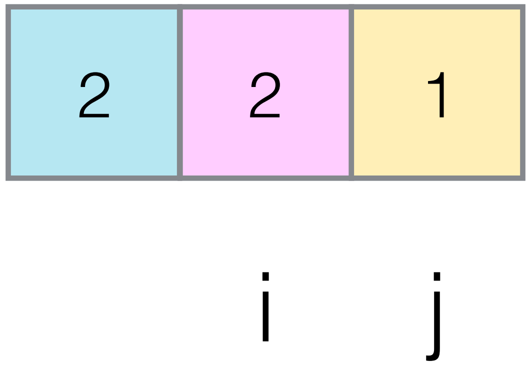
- With these (and other) improvements, quicksort is considered the best available sorting method for arrays of random data.
- A major reason for its speed is the very tight inner loop in PARTITION.
- Although mergesort has a better performance guarantee, quicksort is faster on average.
- In the best case, we get ⌈log2 n⌉ recursive levels. It can be shown that on random data, the expected number is $2 \log_e n \approx 1.38 \log_2 n$. So quicksort's average behaviour is very close to the best-case behaviour.
- Is quicksort stable? ?
- Is it in-place? yes

Quicksort Stability

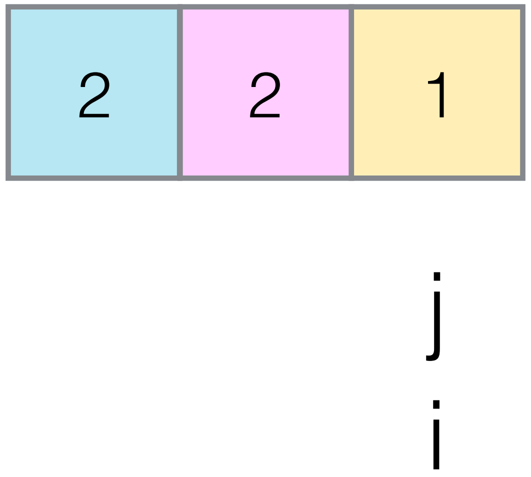


2	2	1
i		j

Quicksort Stability

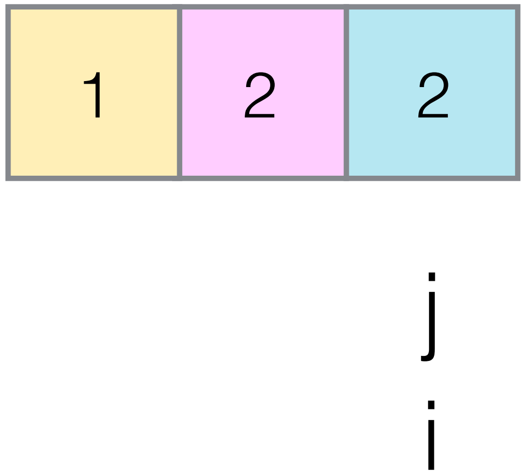


Quicksort Stability



2	2	1

Quicksort Stability

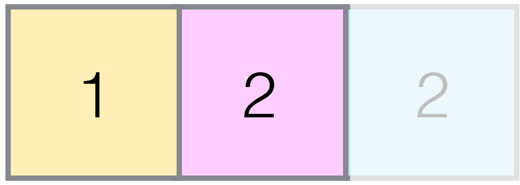


1	2	2

j

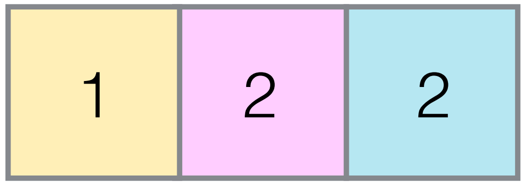
i

Quicksort Stability



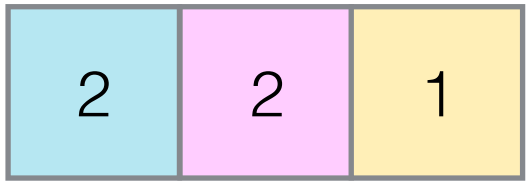
1	2	2

Quicksort Stability



1	2	2

This is where we finished



2	2	1

This is where we started

Not stable

Quicksort Properties

- With these (and other) improvements, quicksort is considered the best available sorting method for arrays of random data.
- A major reason for its speed is the very tight inner loop in PARTITION.
- Although mergesort has a better performance guarantee, quicksort is faster on average.
- In the best case, we get $\lceil \log 2\ n \rceil$ recursive levels. It can be shown that on random data, the expected number is $2 \log_e n \approx 1.38 \log_2 n$. So quicksort's average behaviour is very close to the best-case behaviour.
- Is quicksort stable? no
- Is it in-place? yes