Analyse!

Week 6

COMP 1201 (Algorithmics)

Dr Andrew Sogokon a.sogokon@soton.ac.uk
ECS, University of Southampton

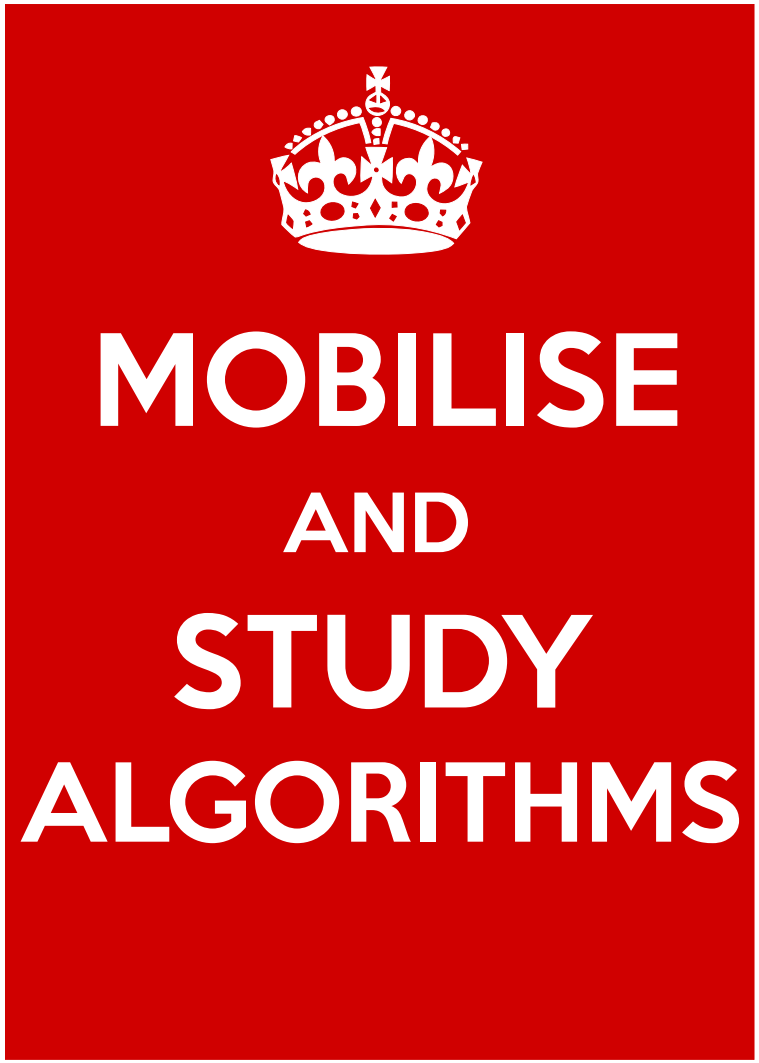
6 March 2020

Previously...

Data structures

- Linked lists, trees, heaps, hash tables, union-find,
- Operations: `search`, `insert`, `remove`, `find` and `union`,
- Goal: build data structures with fast operation running time.

From now on:



Second (Algorithms) part of this course

- Sorting algorithms,
- Graph algorithms,
- Dynamic programming,
- Backtracking,
- Optimisation algorithms,
- Linear programming.

Outline

1. **Algorithm analysis**
   - Pseudocode
2. Search
   - Binary search
3. Simple sorts
   - Insertion sort
   - Selection sort

Why Pseudocode?

- Java code can be difficult to read – everything has to be wrapped in classes, `ArrayLists` are ugly, etc.
- It contains details (e.g. throwing exceptions) that are repetitive and language-specific.
- **Algorithms are not language-dependent**.
- To focus on what is important, we use a stylised "programming language" called **pseudocode**.

> *[The] difference between* **pseudocode** *and* **real code** *is that pseudocode is not typically concerned with issues of software engineering.* – **CLRS**, *Chapter 2.*

Pseudocode: rule of thumb

- Pseudocode is meant to be easily interpreted by **people**.
- There is no recognised standard for pseudocode.
- Imperative: commands are not too dissimilar to Java.
- Arrays are often written in bold, e.g. $\mathbf{a}$ (with elements a_i).
  - Some books (e.g. **CLRS**) use capital letters for arrays A (with elements $A[i]$)

A word about assignments

$$x = x + 1$$

A word about assignments

$$x = x + 1$$

To a mathematician/pure functional programmer, this expression doesn't make a lot of sense!

($x = 1$ is fine, but the equation $x = x + 1$ has **no solution**.)

Using $=$ to denote **assignments** and $==$ to denote equality comes from C.

Historical note: FORTRAN already used $=$ for assignments, but used `.EQ.` for equality.

A word about assignments

> *"The double equals sign == is the C notation for "is equal to" (like Fortran's .EQ.). This symbol is used to distinguish the equality test from the single = used for assignment. Since assignment is about twice as frequent as equality testing in typical C programs, it's appropriate that the operator be half as long."*
>
> – Kernighan & Ritchie, *The C Programming Language*, 1978. Chapter 1, p. 17.

A word about assignments

"The double equals sign == is the C notation for "is equal to" (like Fortran's .EQ.). This symbol is used to distinguish the equality test from the single = used for assignment. Since assignment is about twice as frequent as equality testing in typical C programs, it's appropriate that the operator be half as long."

– Kernighan & Ritchie, *The C Programming Language*, 1978. Chapter 1, p. 17.

A word about assignments

Common notation for assignment in pseudocode:

$$x \leftarrow x + 1 \qquad \text{[**CLRS**, 1st, 2nd ed.]}$$

$$x := x + 1 \qquad \text{[ALGOL-like]}$$

Based on many requests, we changed the syntax (as it were) of our pseudocode. We now use "=" to indicate assignment and "==" to test for equality, just as C, C++, Java and Python do (...) and adopted // as our comment-to-end-of-line symbol. (...) Our pseudocode remains procedural, rather than object-oriented.

– **CLRS**, 3 ed., 2009. Preface.

Pseudocode: other common conventions

▷ comment	**CLRS** 1, 2nd ed.
$x \times y,\ x \cdot y,$ or just $x\,y$	for multiplication
$(x * y$	also sometimes used)

if condition **then**

code ⋮

else if condition

code ⋮

endif

Historical note: The use of * to denote multiplication originated in FORTRAN.

Pseudocode examples

Algorithm 1 Euclid's algorithm

```
1: procedure EUCLID(a, b)                    ▷ The g.c.d. of a and b
2:     r ← a mod b
3:     while r ≠ 0 do                        ▷ We have the answer if r is 0
4:         a ← b
5:         b ← r
6:         r ← a mod b
7:     return b                              ▷ The gcd is b
```

Outline

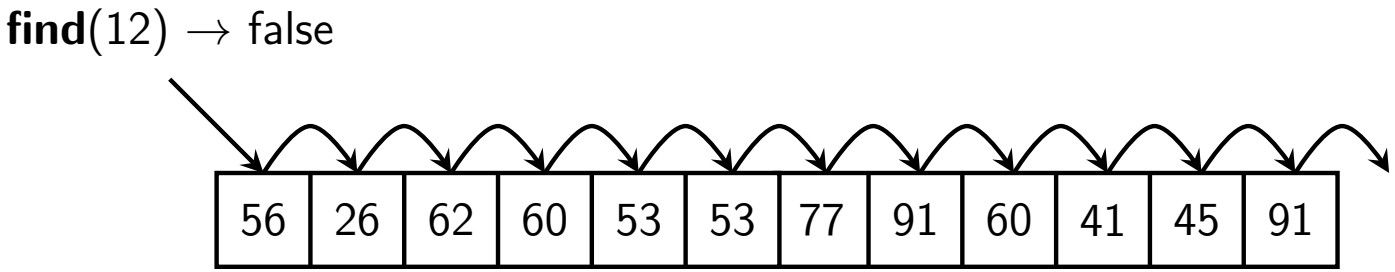
1. Algorithm analysis
   - Pseudocode
2. **Search**
   - Binary search
3. Simple sorts
   - Insertion sort
   - Selection sort

Sequential Search

- Goal: find an element with a given value in a linked list.
- Simple search algorithm: Sequential Search.
- Idea: sequentially check from the first element.

Sequential Search

- Goal: find an element with a given value in a linked list.
- Simple search algorithm: Sequential Search.
- Idea: sequentially check from the first element.



Sequential Search

Algorithm 2 Sequential Search

```
1: procedure SequentialSearch(a,x)
2:     for i ← 1 to a.length do
3:         if a_i = x then
4:             return true                          ▷ item found
5:     return false                                 ▷ item not found
```

Time Complexity of Sequential Search

- **Worst case**:
  - The item is in the last location or not in the list.
  - This takes n comparisons (n is the number of items in the list).
  - Time complexity is $\Theta(n)$.
- **Best case**:
  - The item is at the beginning of the list.
  - Time complexity is $\Theta(1)$
- **Average case**:

$$\frac{(1+2+3+\cdots+n)}{n} = \frac{1}{n} \times \frac{n(n+1)}{2} = \frac{n+1}{2}$$

  - So it is also $\Theta(n)$.

Can we do better?

Binary Search: divide-and-conquer

If the list is **sorted** then we can do better.

- Idea: pick a mid-point element a_m

$$\underbrace{a_1, a_2, a_3, \ldots, a_{m-1}}_{x<a_m}, \overbrace{a_m}^{x=a_m}, \underbrace{a_{m+1}, a_{m+2}, \ldots, a_n}_{x>a_m}$$

- If x equals to the current item $\rightarrow$ `STOP` (found!)
- If x is smaller, continue searching in the left part,
- If x is larger, search in the right part,
- If the range of search becomes empty $\rightarrow$ `STOP` (not found!)

Binary Search

Maintain low and high ends of the range in which the item can be (potentially found).

- If x is smaller, change the high end to current location-1.
- If x is greater, change the low end to current location$+1$.

Intuition: by choosing a location to check, we split the remaining potential candidates into two groups.

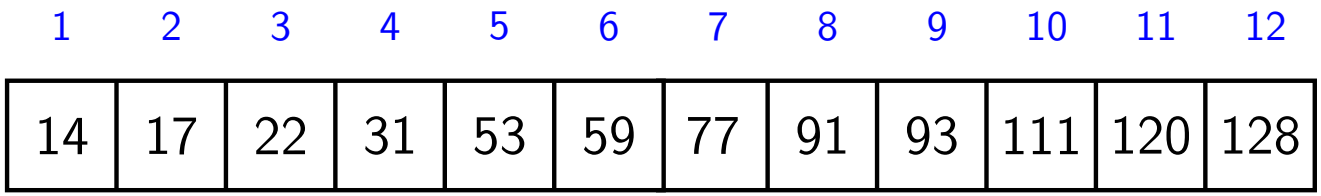
- After the check, we can clearly identify which group the item potentially lies within.
- Worst case: item is in the larger group.
- Balancing: split into two groups by always querying the middle item.
- Repeat the splitting step above.
- Stopping criteria: item found, or high end goes below low end.

Binary Search

Algorithm 3 Binary Search

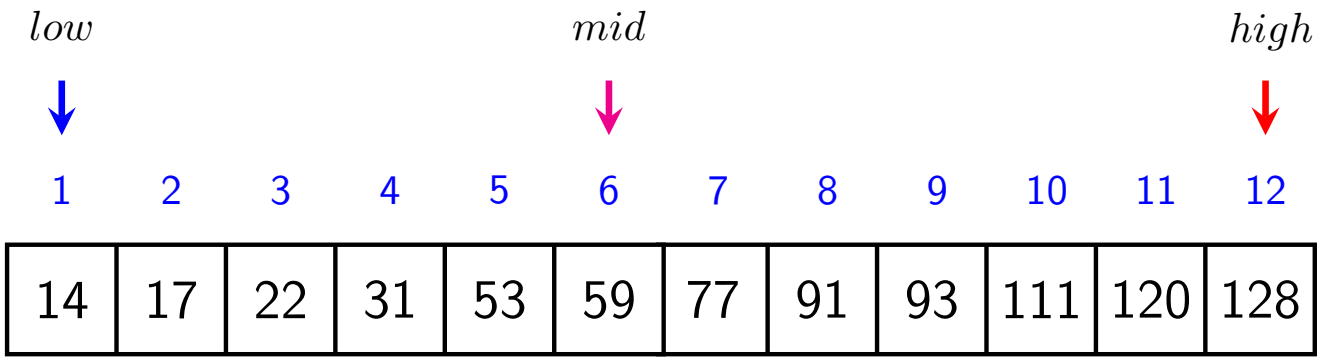
```
1:  procedure BinarySearch(a, x)
2:      low ← 1
3:      high ← a.length
4:      while low ≤ high do
5:          mid ← ⌊(high+low)/2⌋
6:          if x > a_mid then
7:              low ← mid + 1
8:          else if x < a_mid then
9:              high ← mid − 1
10:         else
11:             return true                    ▷ item found
12:     return false                           ▷ item not found
```

Binary Search



Binary Search

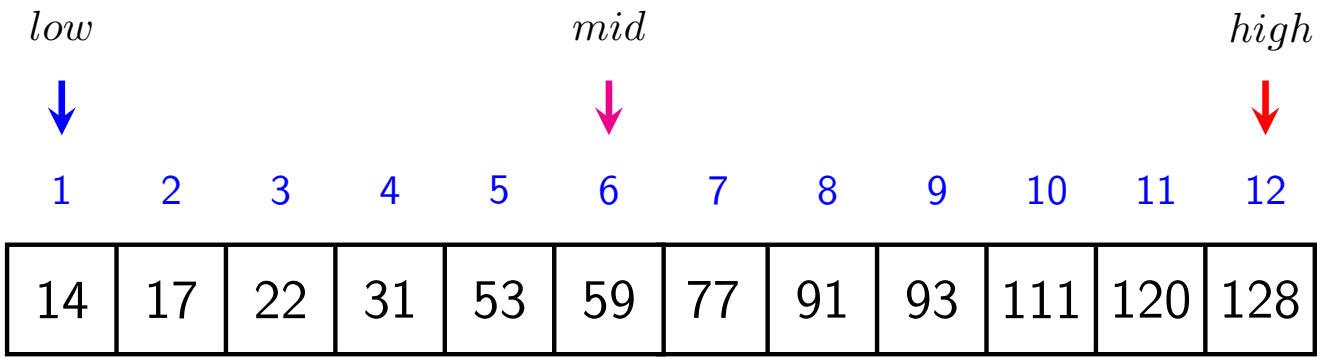
find(120)



Binary Search

find(120)

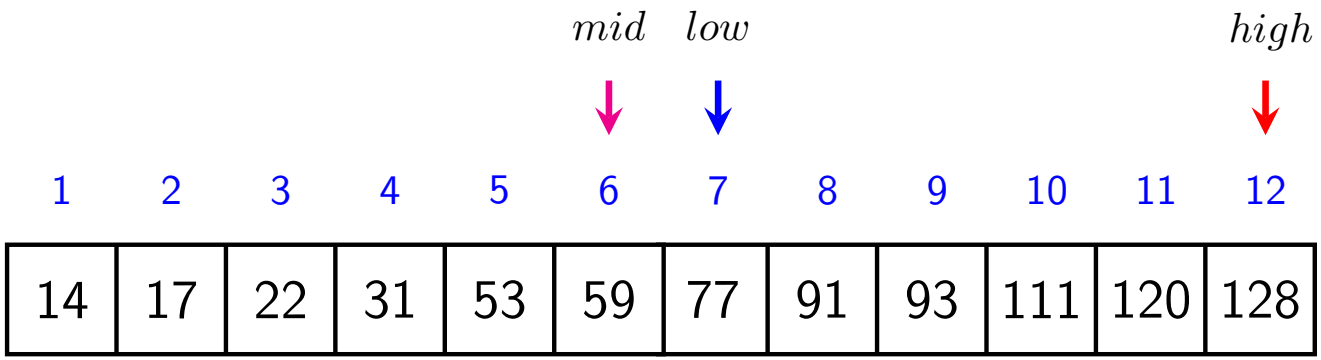
$120 > 59$



Binary Search

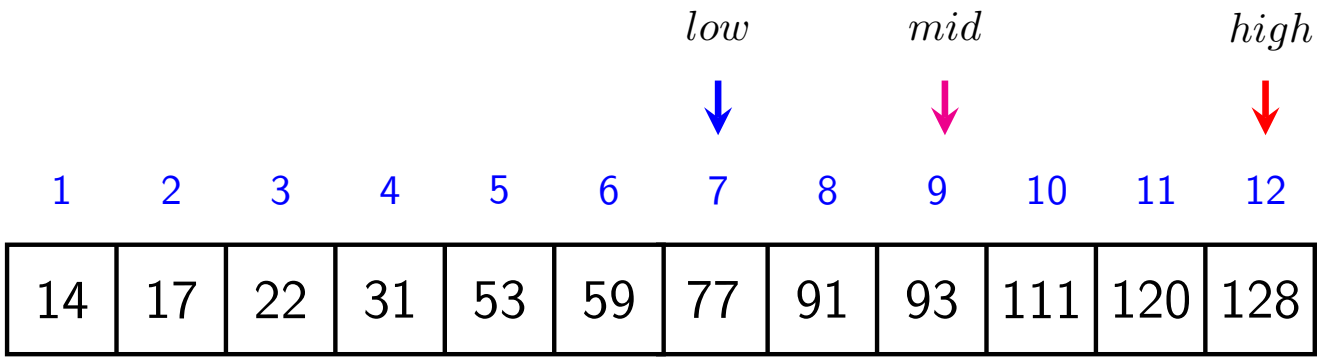
find(120)

$120 > 59$



Binary Search

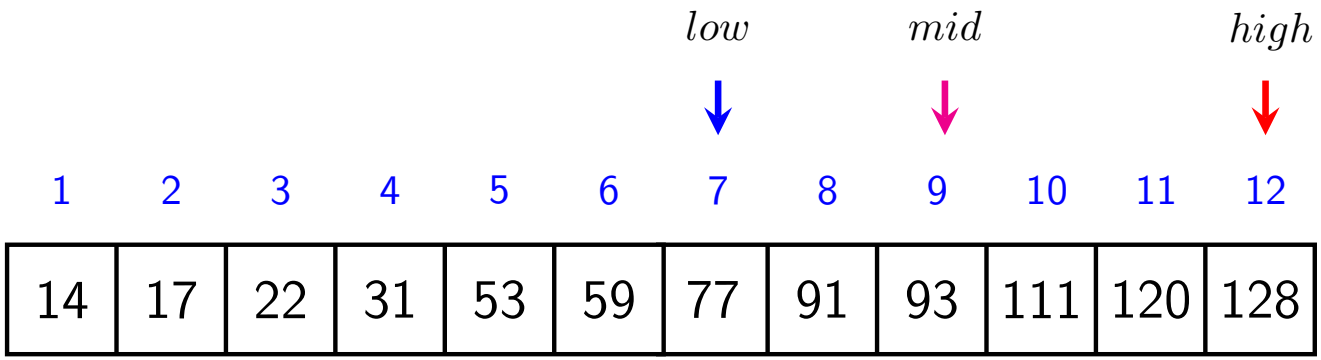
find(120)



Binary Search

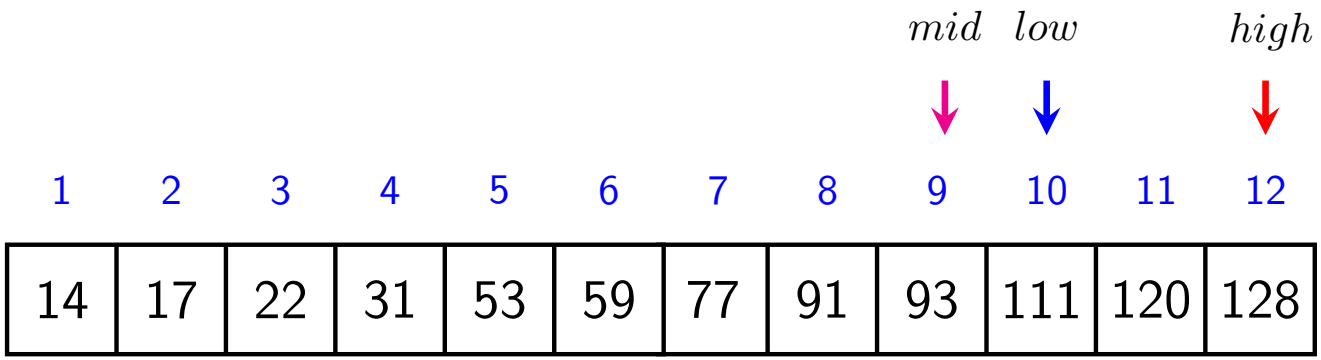
find(120)

$120 > 93$



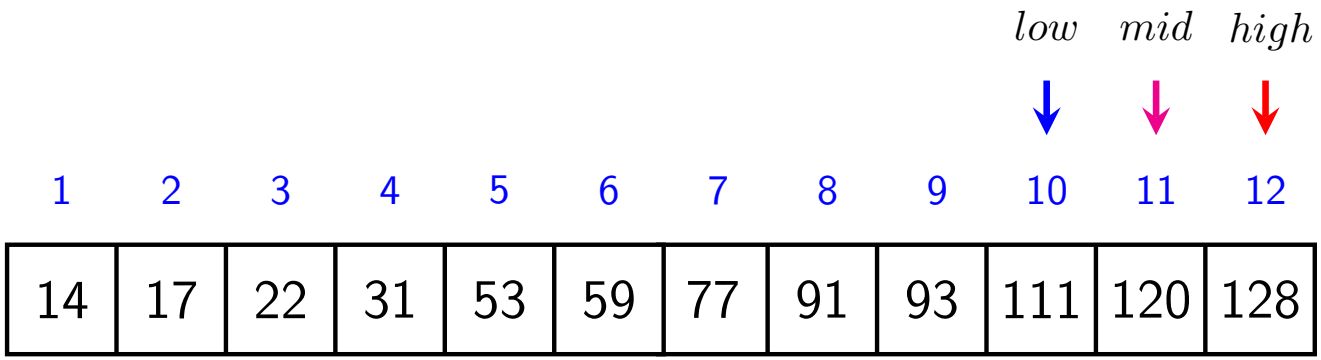
Binary Search

find(120)



Binary Search

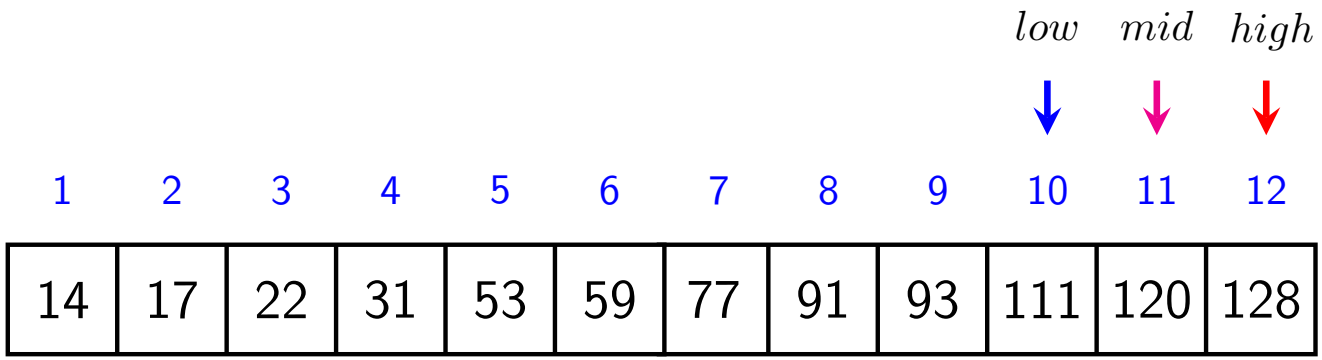
find(120)



Binary Search

find(120)

$120 = 120$



Properties of Binary Search

We count the number of queries (comparisons).

- Query the middle items $\sim$ balanced tree.
- Worst case number of queries $=$ depth of the tree.
- Balanced trees have $O(\log_2(n))$ levels.

Binary Search time complexity is $O(\log_2(n))$.

Outline

1. Algorithm analysis
   - Pseudocode
2. Search
   - Binary search
3. **Simple sorts**
   - Insertion sort
   - Selection sort

Sorting algorithms

Sorting is one of the best-studied problems in algorithms.

Can be applied to data for which there is a (total) ordering.

We care about: **stability** and **complexity** (both time and *space*).

- A sorting algorithm is **stable** if it does not change the order of elements that have the same value.
- A sorting algorithm is **in-place** if the memory used is $O(1)$.
- For time complexity, we're interested in:
  - Worst case
  - Average case
  - Best case

Stability (example: sort a list of people by first name)

unsorted list	
Tony Hoare	
John Hopcroft	
Stephen Cook	
John McCarthy	
Donald Knuth	
John Hennessy	
James Wilkinson	
John Backus	
Leslie Valiant	
Robert Floyd	
Leslie Lamport	
Richard Hamming	
Robert Tarjan	
Richard Stearns	
Robin Milner	
Richard Karp	

Stability (example: sort a list of people by first name)

unsorted list	stable sort	
Tony Hoare	Donald Knuth	
John Hopcroft	James Wilkinson	
Stephen Cook	John Hopcroft	
John McCarthy	John McCarthy	
Donald Knuth	John Hennessy	
John Hennessy	John Backus	
James Wilkinson	Leslie Valiant	
John Backus	Leslie Lamport	
Leslie Valiant	Richard Hamming	
Robert Floyd	Richard Stearns	
Leslie Lamport	Richard Karp	
Richard Hamming	Robert Floyd	
Robert Tarjan	Robert Tarjan	
Richard Stearns	Robin Milner	
Robin Milner	Stephen Cook	
Richard Karp	Tony Hoare	

Stability (example: sort a list of people by first name)

unsorted list	stable sort	unstable sort
Tony Hoare	Donald Knuth	Donald Knuth
John Hopcroft	James Wilkinson	James Wilkinson
Stephen Cook	John Hopcroft	John Hennessy
John McCarthy	John McCarthy	John Backus
Donald Knuth	John Hennessy	John Hopcroft
John Hennessy	John Backus	John McCarthy
James Wilkinson	Leslie Valiant	Leslie Valiant
John Backus	Leslie Lamport	Leslie Lamport
Leslie Valiant	Richard Hamming	Richard Karp
Robert Floyd	Richard Stearns	Richard Stearns
Leslie Lamport	Richard Karp	Richard Hamming
Richard Hamming	Robert Floyd	Robert Floyd
Robert Tarjan	Robert Tarjan	Robert Tarjan
Richard Stearns	Robin Milner	Robin Milner
Robin Milner	Stephen Cook	Stephen Cook
Richard Karp	Tony Hoare	Tony Hoare

Outline

1. Algorithm analysis
   - Pseudocode
2. Search
   - Binary search
3. Simple sorts
   - Insertion sort
   - Selection sort

Insertion Sort

Insertion Sort keeps a subsequence of elements on the left in (correctly) sorted order.

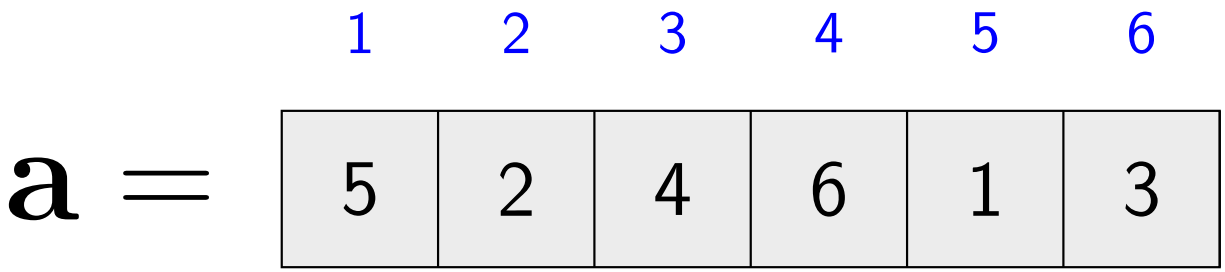
- This subsequence is increased by **inserting** the next element into its (relatively) correct position in the sorted subsequence.
- "Relatively correct": might not be the correct final position, but correct within the sorted subsequence.
- Intuition: *we insert the current element into the already sorted part on the left*.
- With each iteration we move the current element one to the right.

Insertion Sort

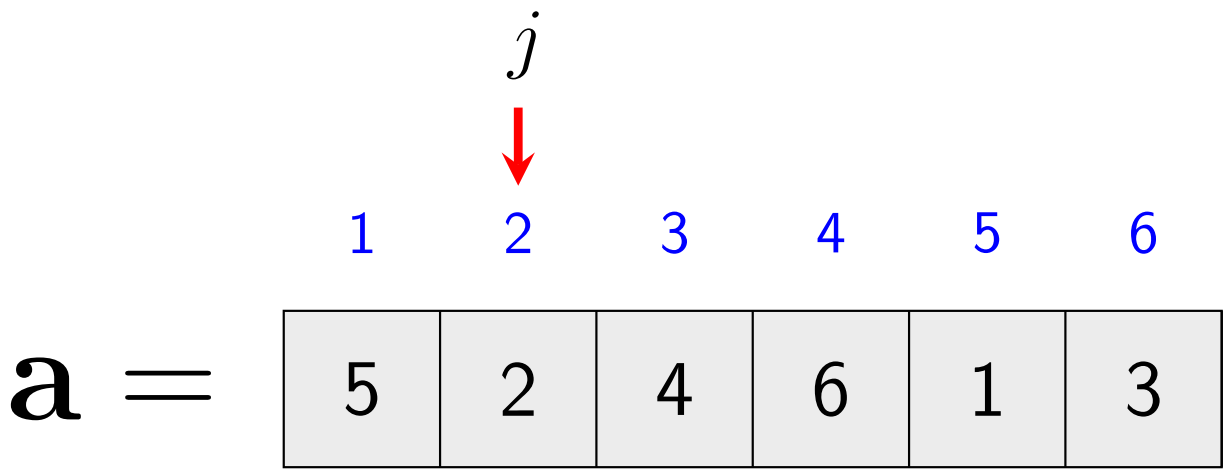
Algorithm 4 Insertion Sort

```
1: procedure InsertionSort(a)
2:     for j ← 2 to a.length do
3:         key ← a_j
4:         i ← j − 1
5:         while i > 0 and a_i > key do
6:             a_{i+1} ← a_i
7:             i ← i − 1
8:         a_{i+1} ← key
9:     return a                                  ▷ Sorted sequence
```

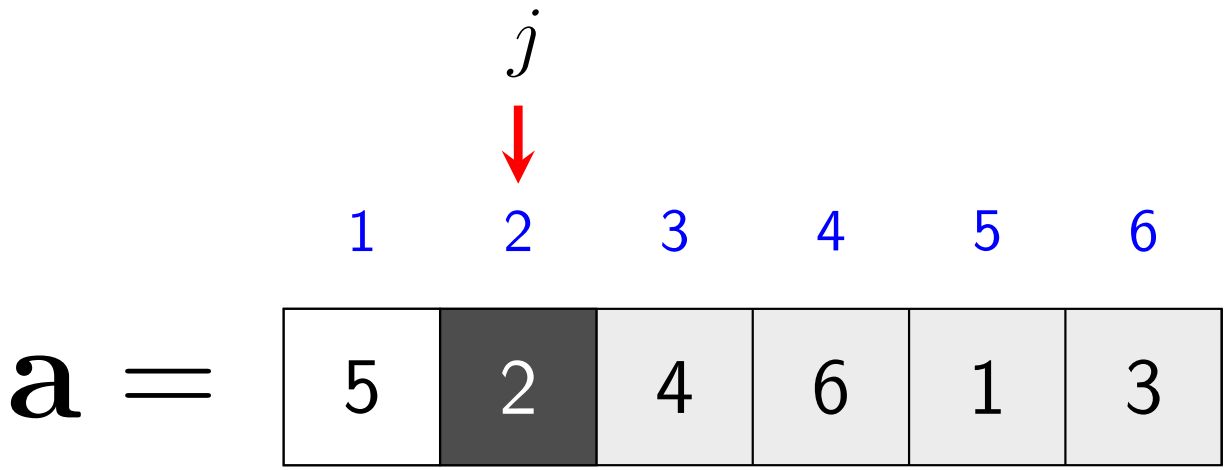
Insertion Sort



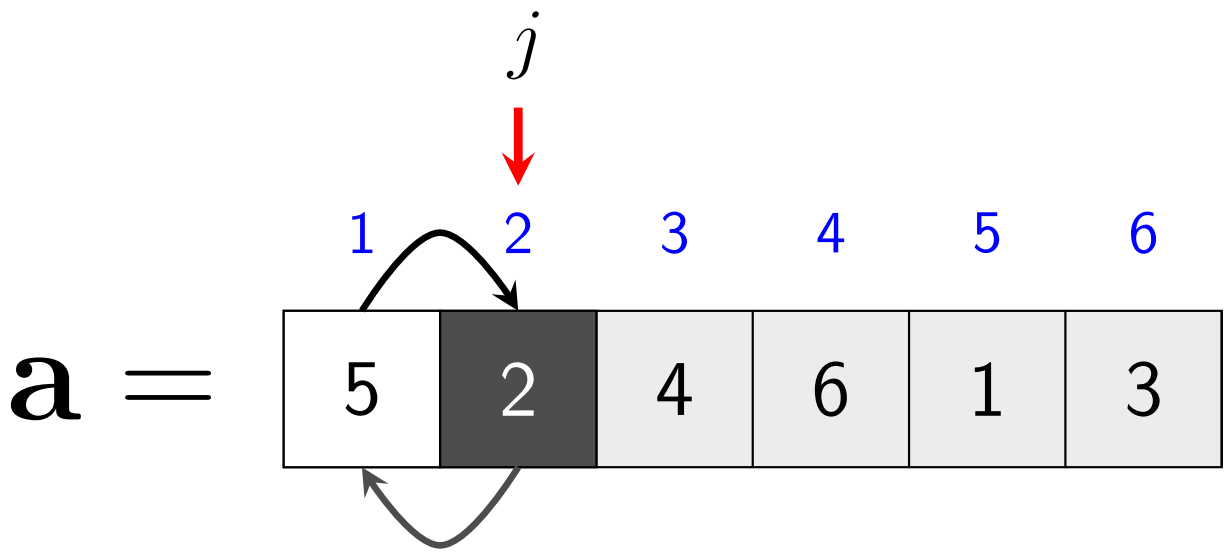
Insertion Sort



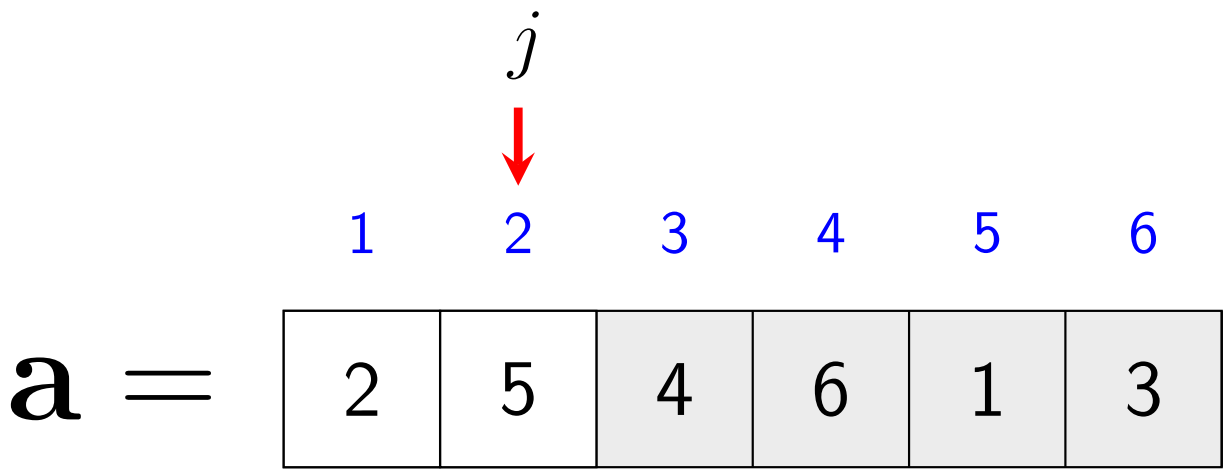
Insertion Sort



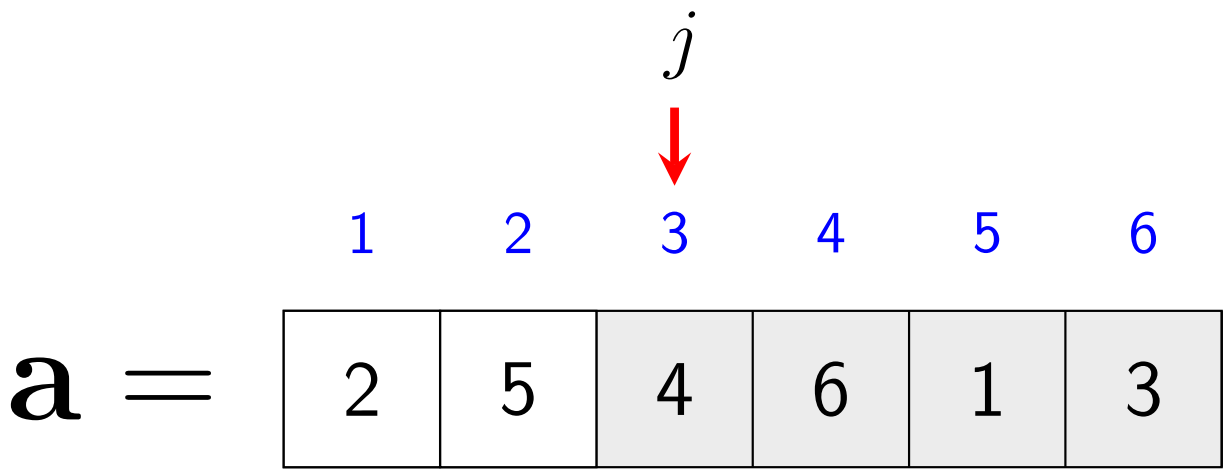
Insertion Sort



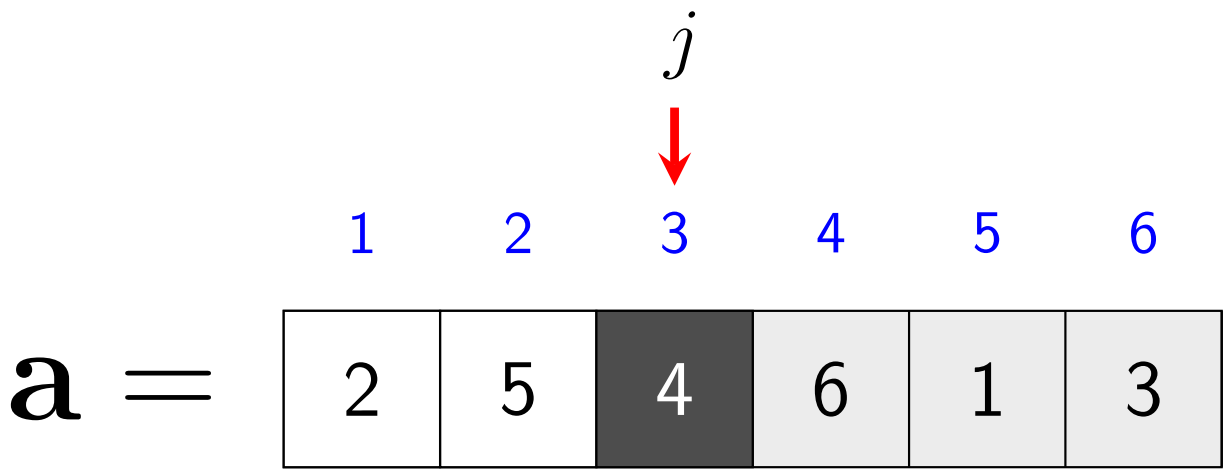
Insertion Sort



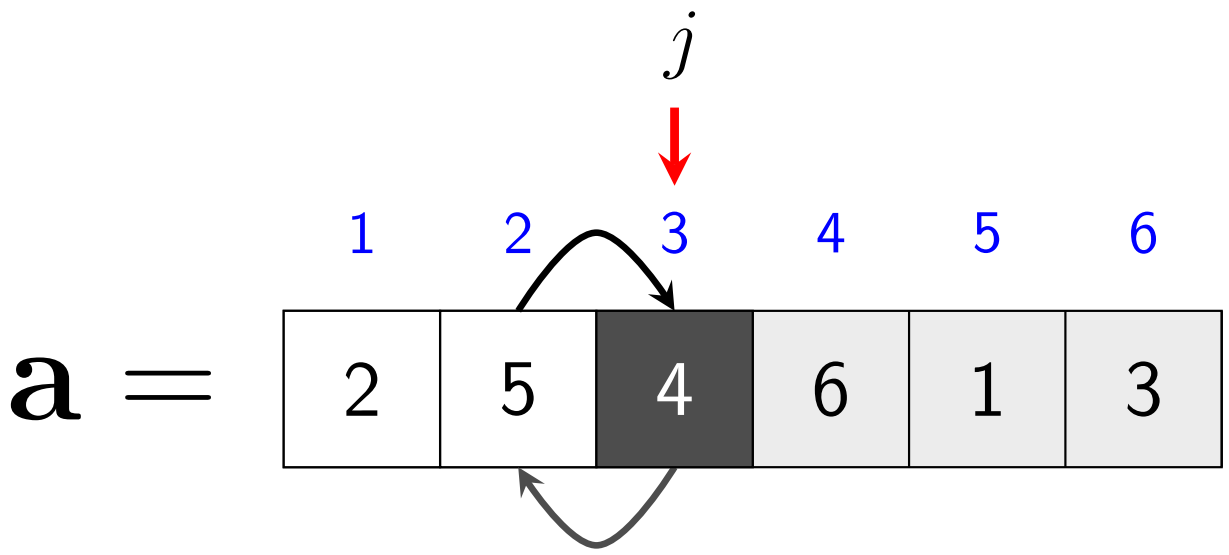
Insertion Sort



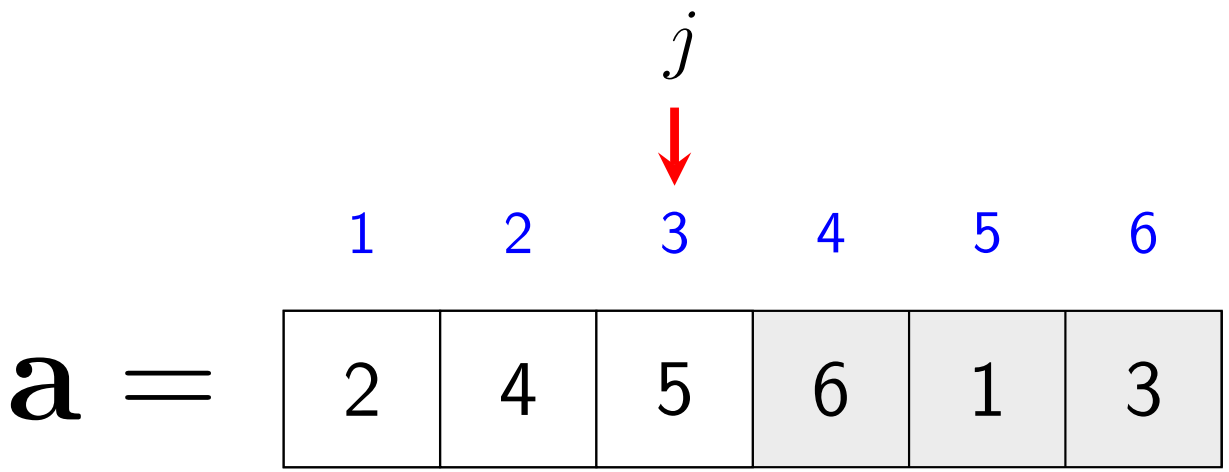
Insertion Sort



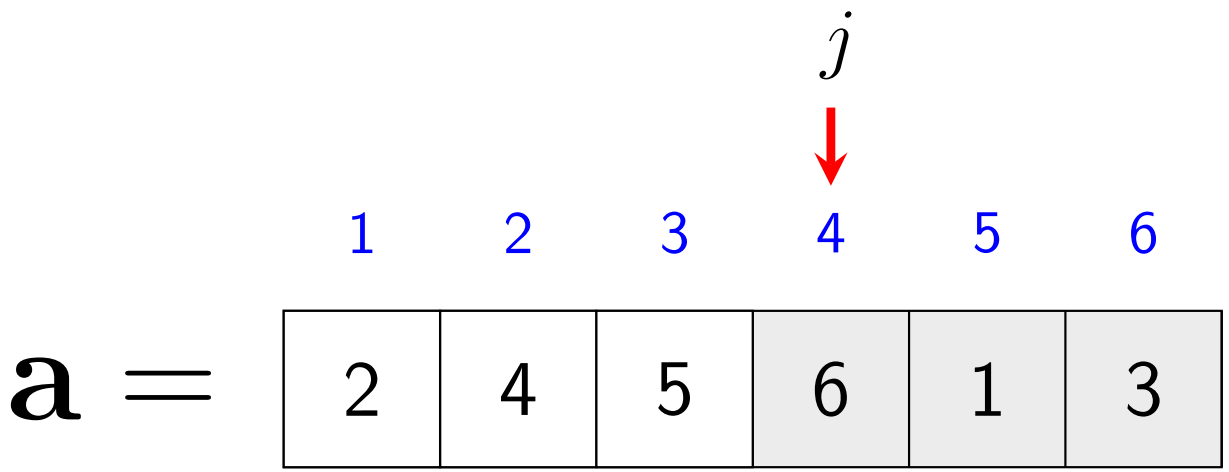
Insertion Sort



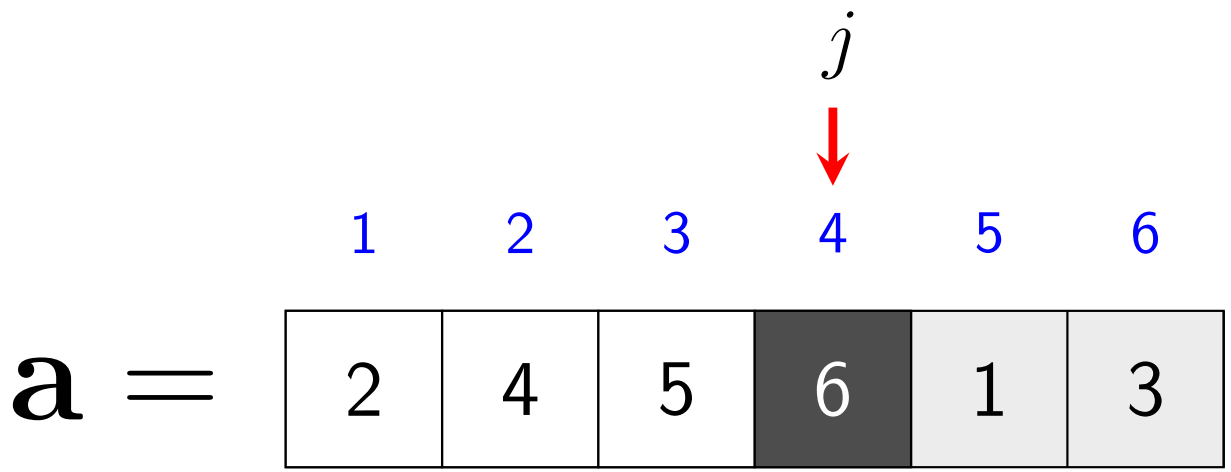
Insertion Sort



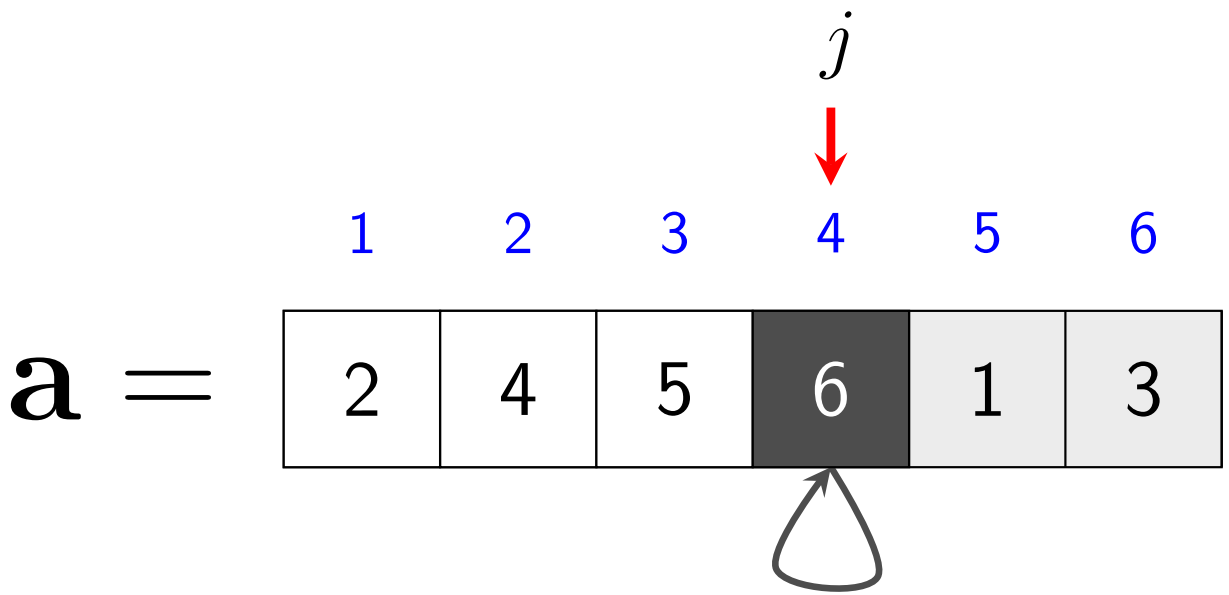
Insertion Sort



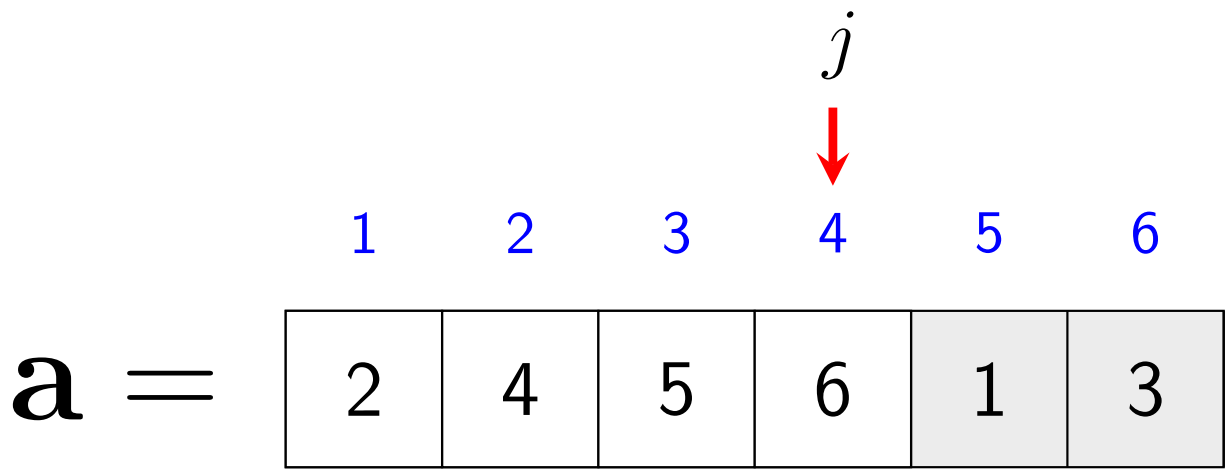
Insertion Sort



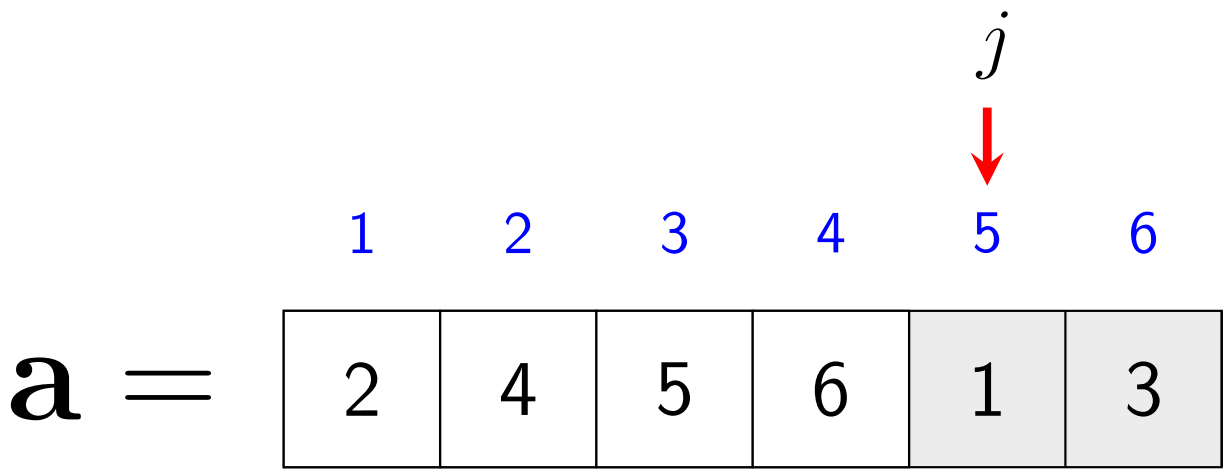
Insertion Sort



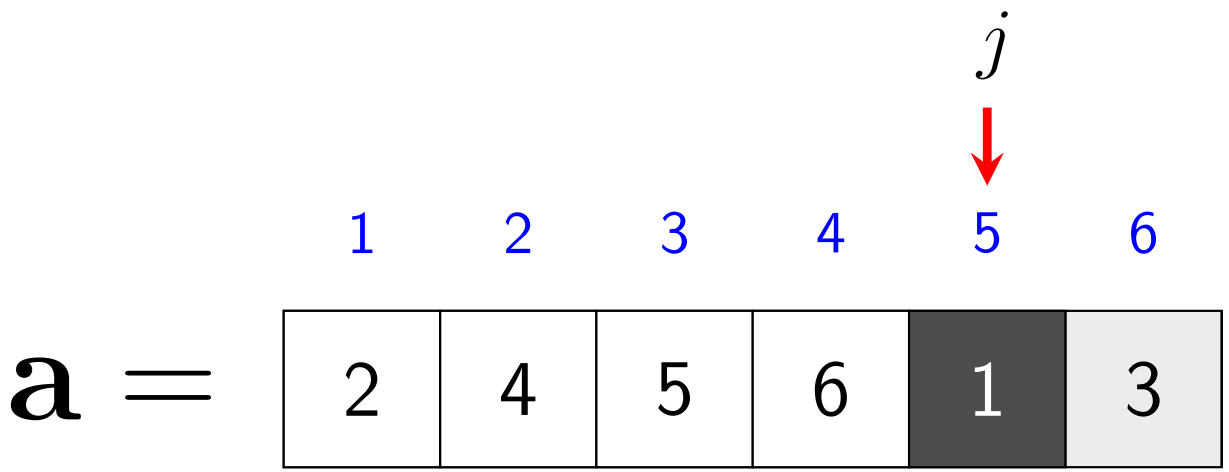
Insertion Sort



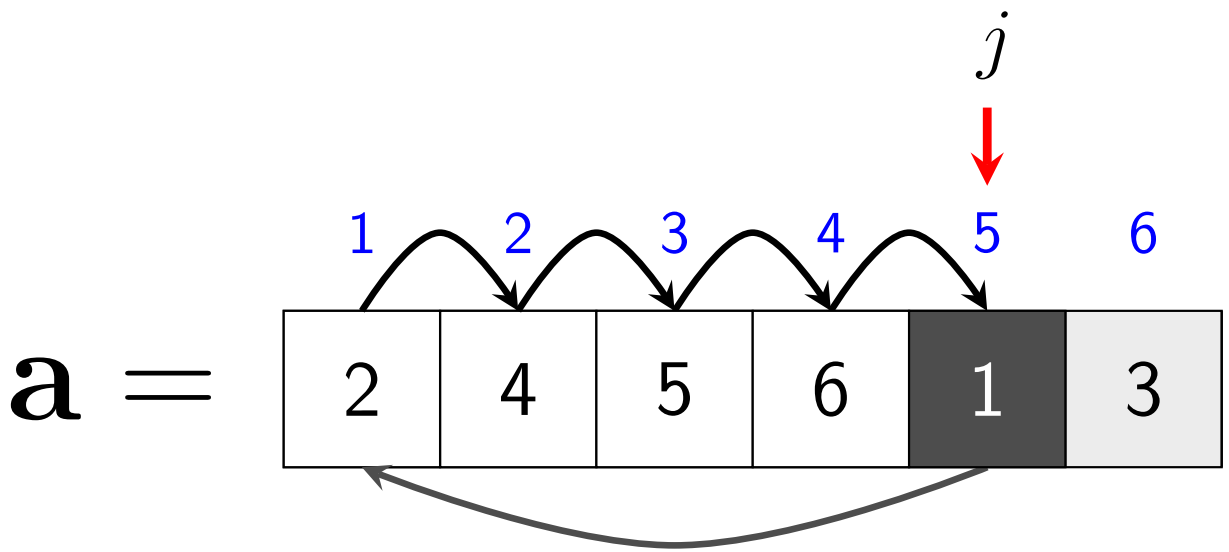
Insertion Sort



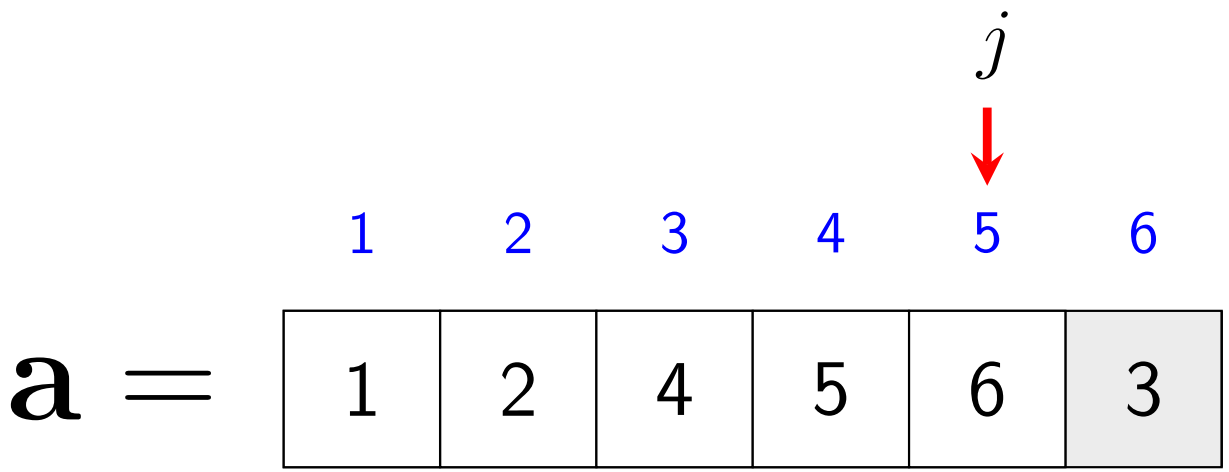
Insertion Sort



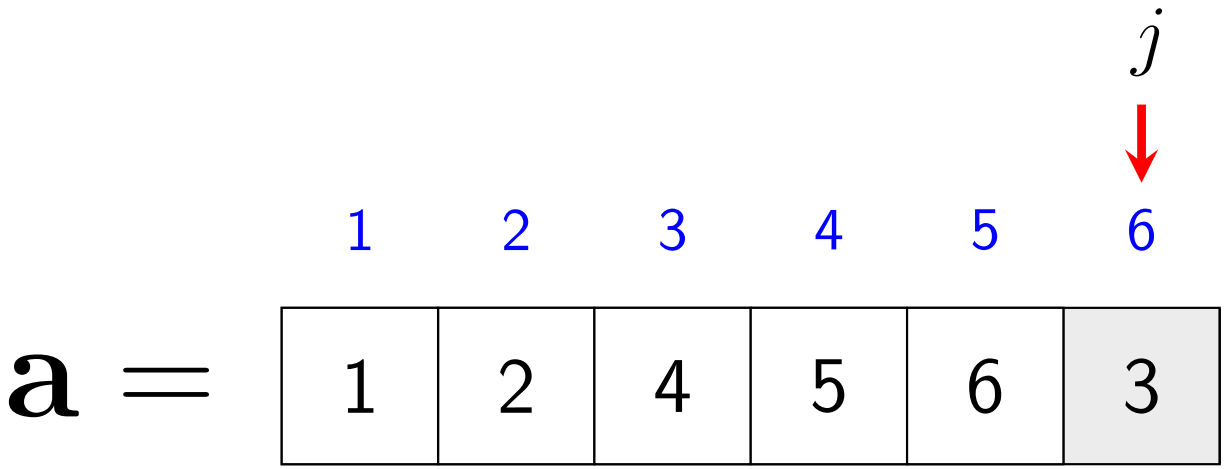
Insertion Sort



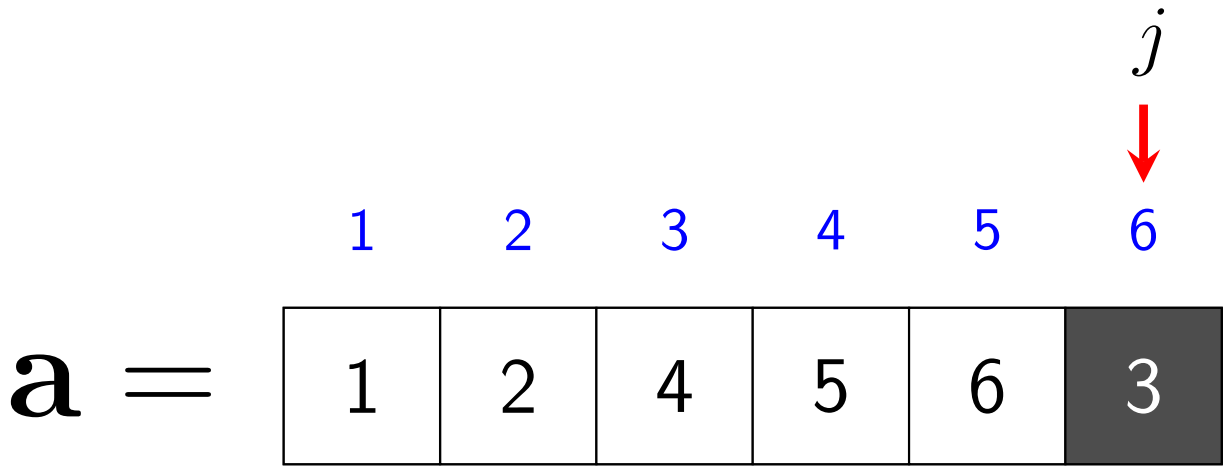
Insertion Sort



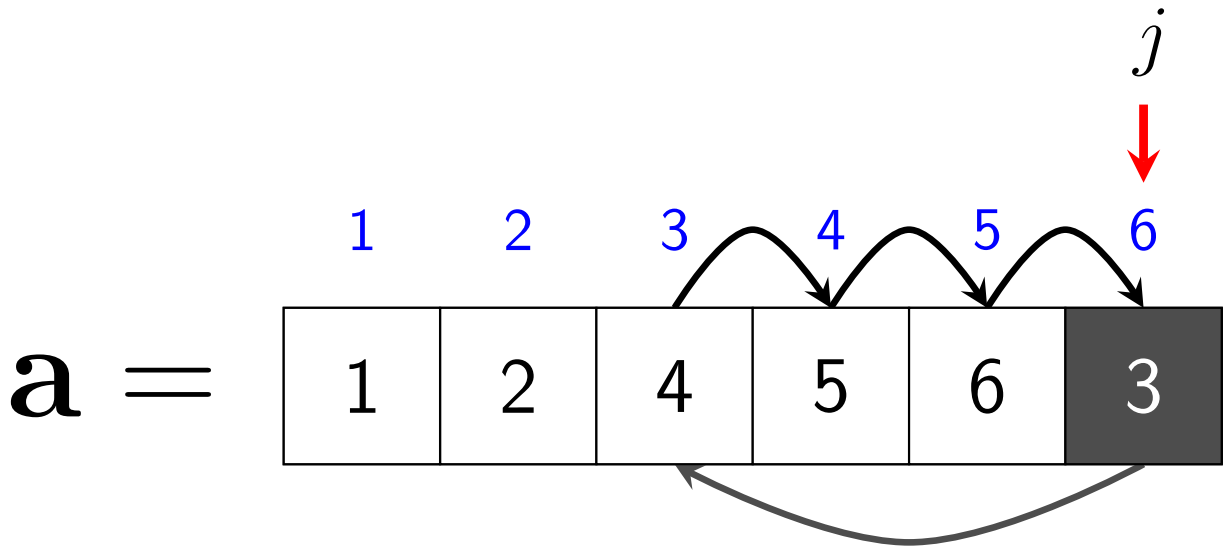
Insertion Sort



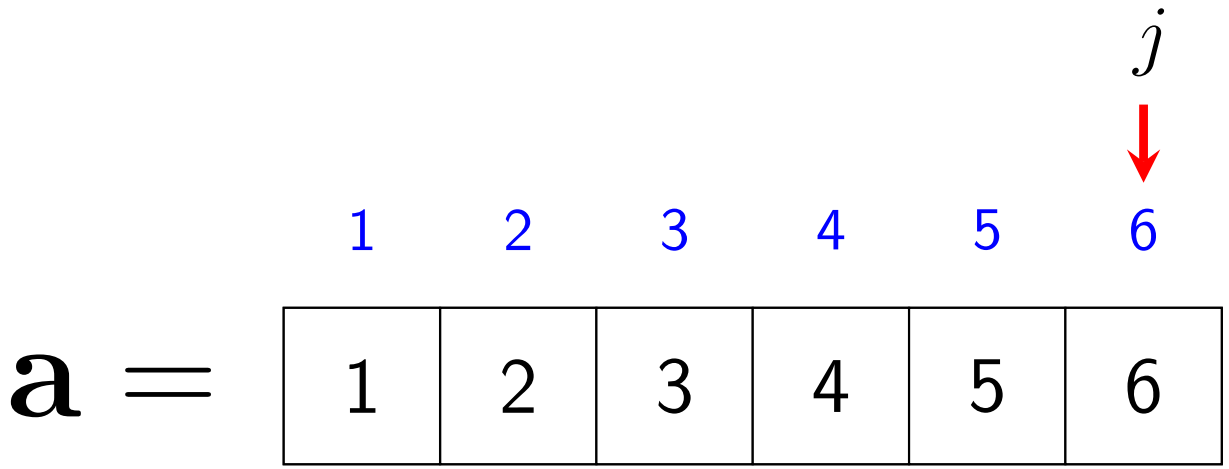
Insertion Sort



Insertion Sort



Insertion Sort



Insertion Sort

Algorithm 5 Insertion Sort

```
1: procedure InsertionSort(a)
2:     for j ← 2 to a.length do
3:         key ← a_j
4:         i ← j − 1
5:         while i > 0 and a_i > key do
6:             a_{i+1} ← a_i
7:             i ← i − 1
8:         a_{i+1} ← key
9:     return a                                  ▷ Sorted sequence
```

Properties of Insertion Sort

- Insertion Sort is **stable**. (We only swap the ordering of two elements if one is strictly less than the other.)
- It is **in-place**.

Worst case time complexity:

- Occurs when the array is sorted in **reverse order**.
- Every element has to be moved to the front.
- Number of comparisons for an array of size n:

$$1 + 2 + \cdots + n - 1 = \frac{n(n-1)}{2}, \quad \text{so } \Theta(n^2).$$

Properties of Insertion Sort

Average case complexity:

- On average we can expect that each new element being sorted moves half way down the sorted subsequence.
- This gives us average time complexity half that of the worst case.
- But it is still $\Theta(n^2)$.

Best case complexity:

- This occurs if the array is **already sorted**.
- Only $n - 1$ comparisons, so $\Theta(n)$.

Overall: Insertion Sort is a good choice for sorting **small arrays** because it is **stable** (no unnecessary moves), **in-place** (low memory requirement) and is fast when the arrays are almost sorted.

Selection Sort

A more direct "brute force" method.

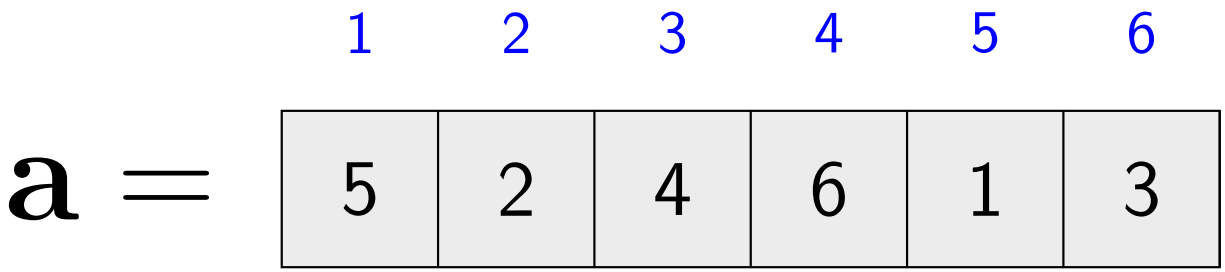
- Intuition: *search for the smallest element and put it into its correct final position; then continue with the second smallest, etc.*
- This method can be made **in-place** by only **swapping** the smallest element with the element in the first position, the second smallest element with the one in the second position, etc.

Selection Sort

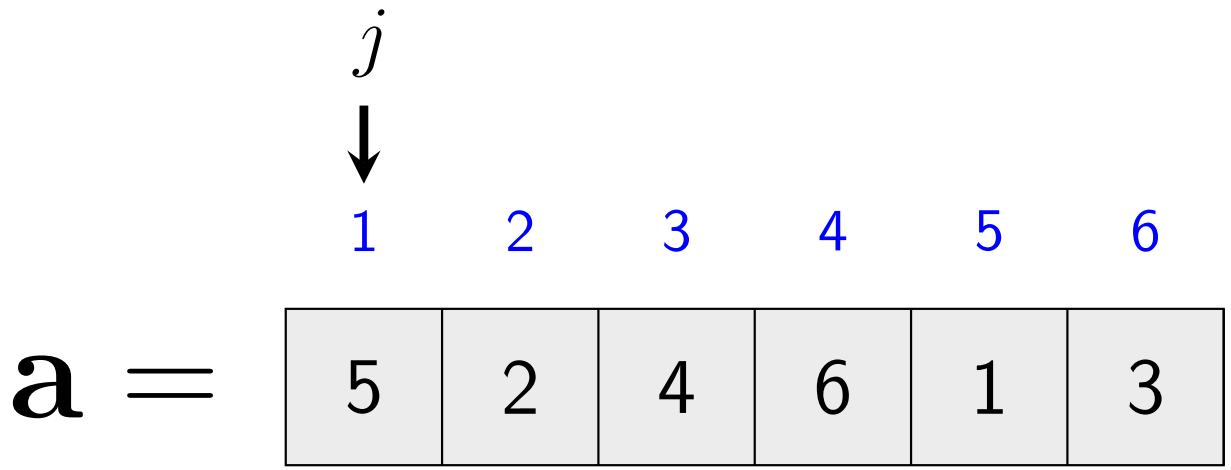
Algorithm 6 Selection Sort

```
1: procedure SelectionSort(a)
2:     for j ← 1 to a.length−1 do
3:         min ← j
4:         for i ← j + 1 to a.length do
5:             if a_i < a_min then
6:                 min ← i
7:         swap a_j and a_min
8:     return a                                  ▷ Sorted sequence
```

Selection Sort

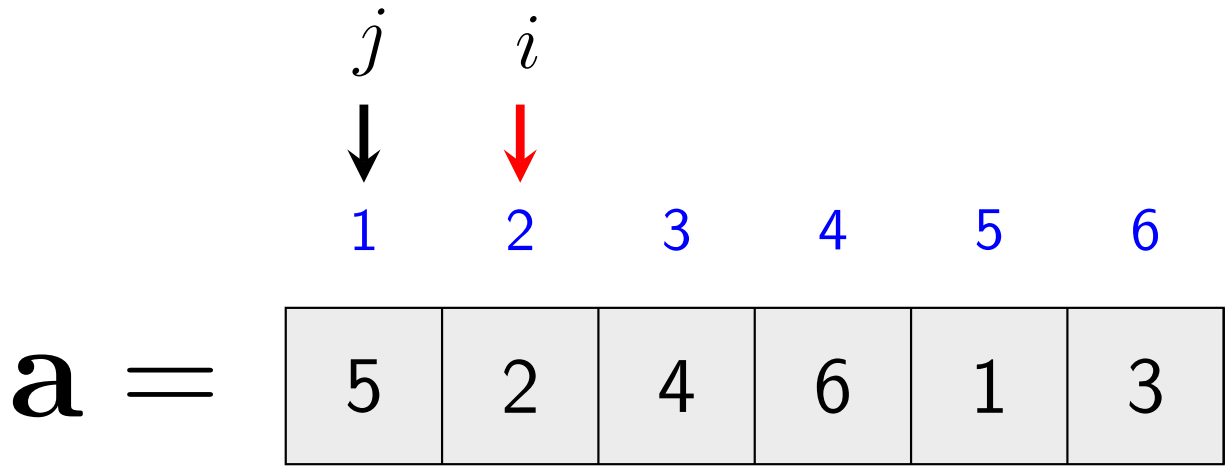


Selection Sort



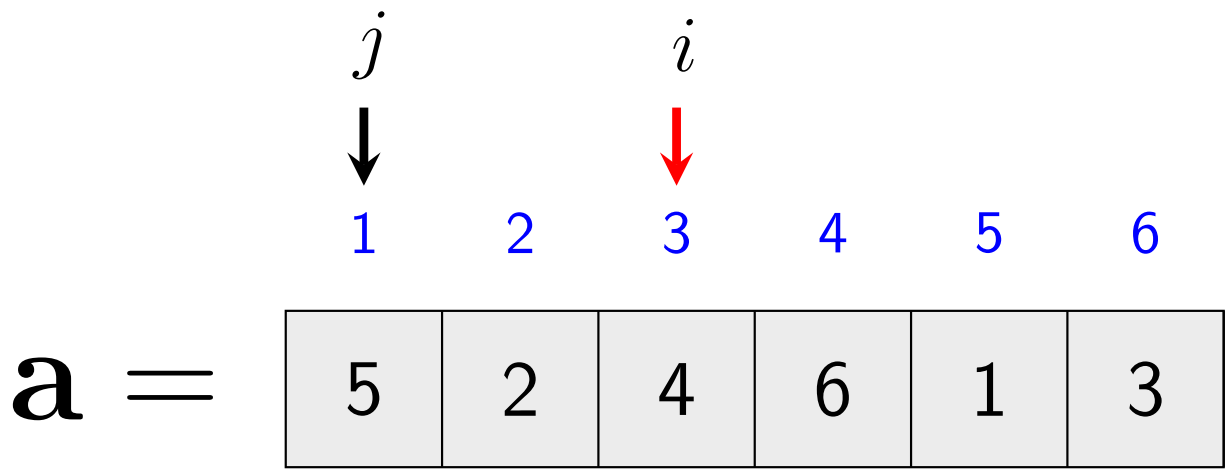
$$min = 1$$

Selection Sort



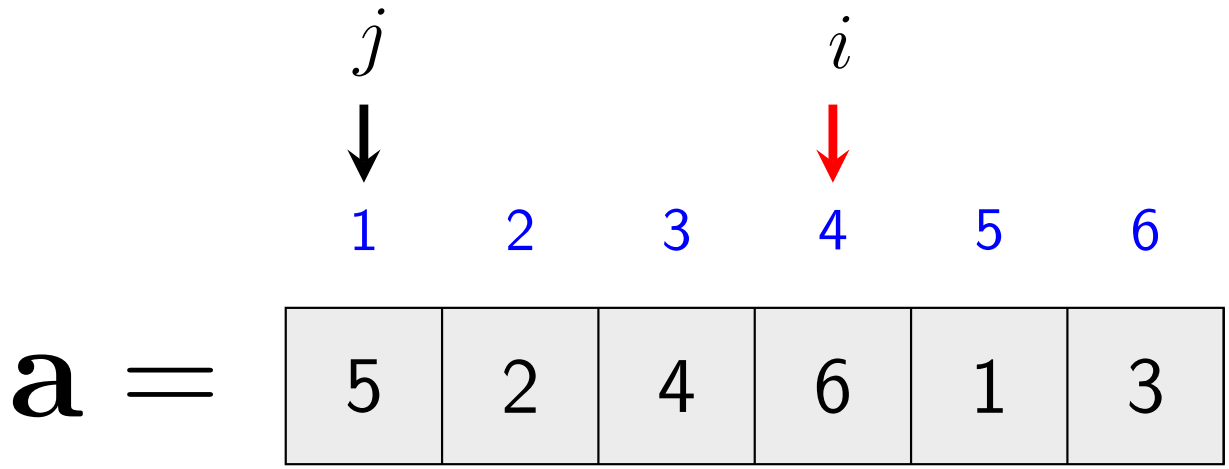
$$min = 2$$

Selection Sort



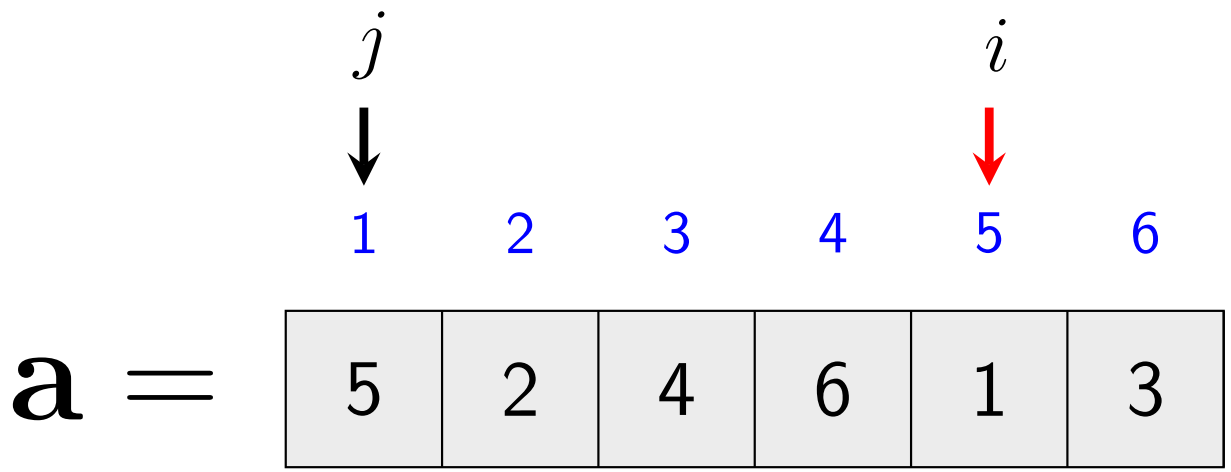
$$min = 2$$

Selection Sort



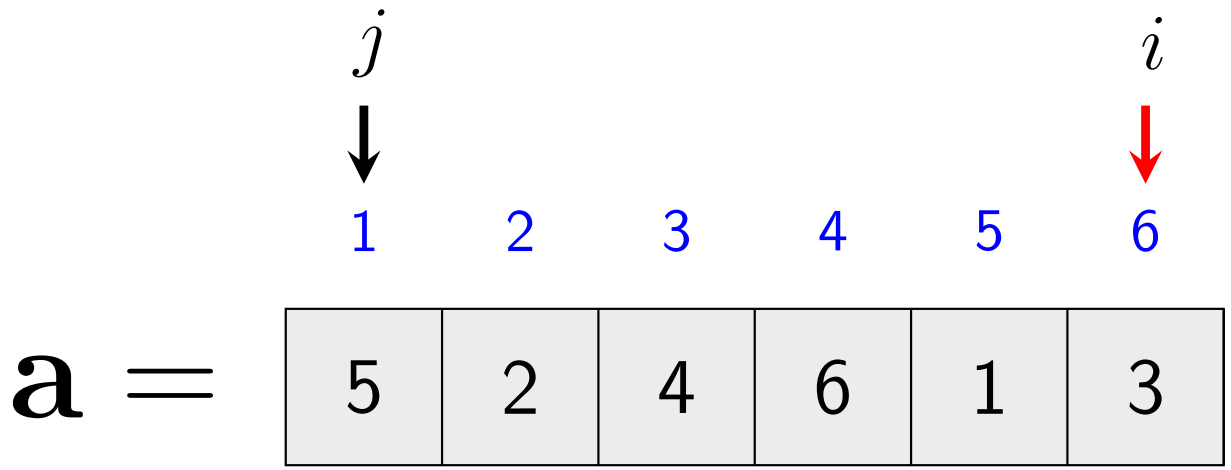
$$min = 2$$

Selection Sort



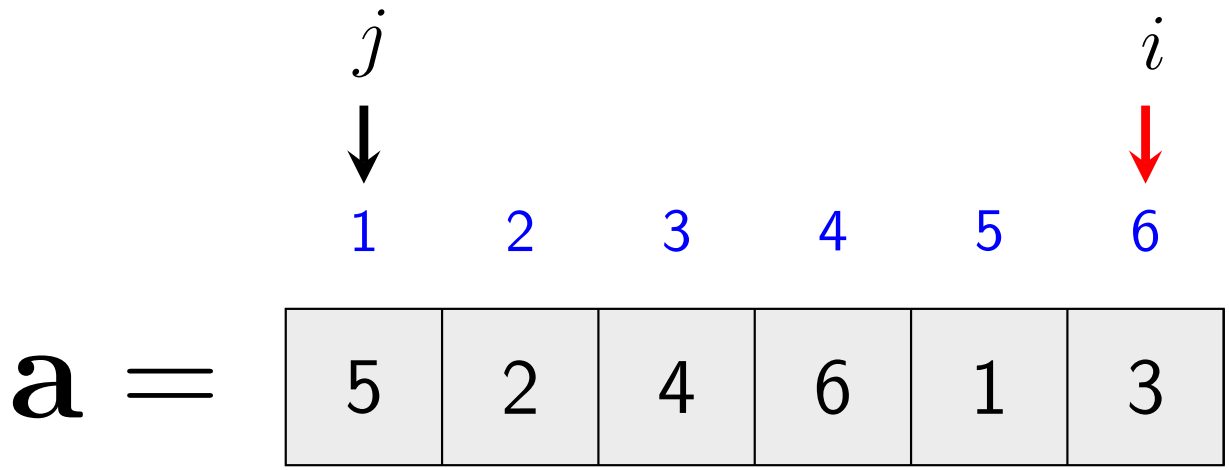
$$min = 5$$

Selection Sort



$$min = 5$$

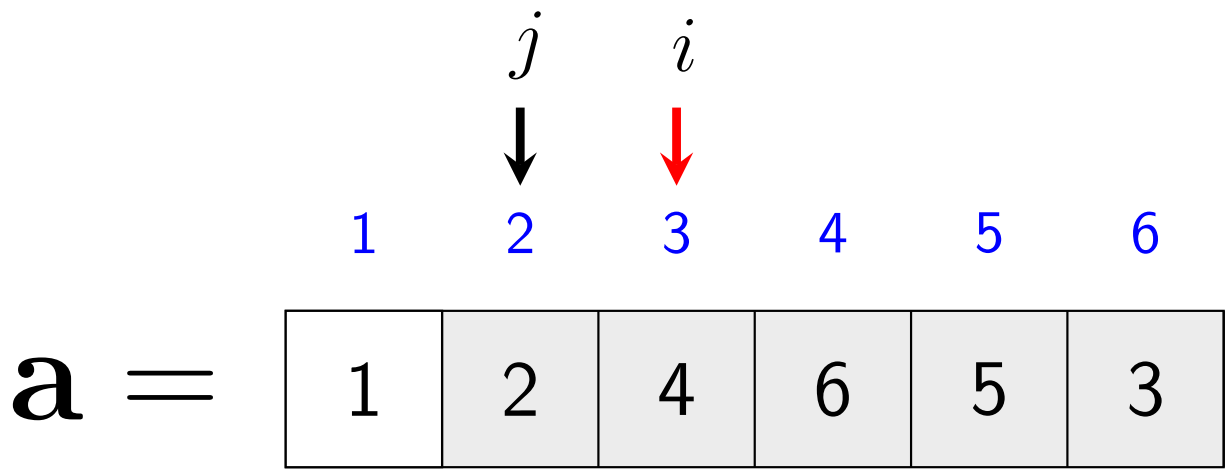
Selection Sort



$min = 5$

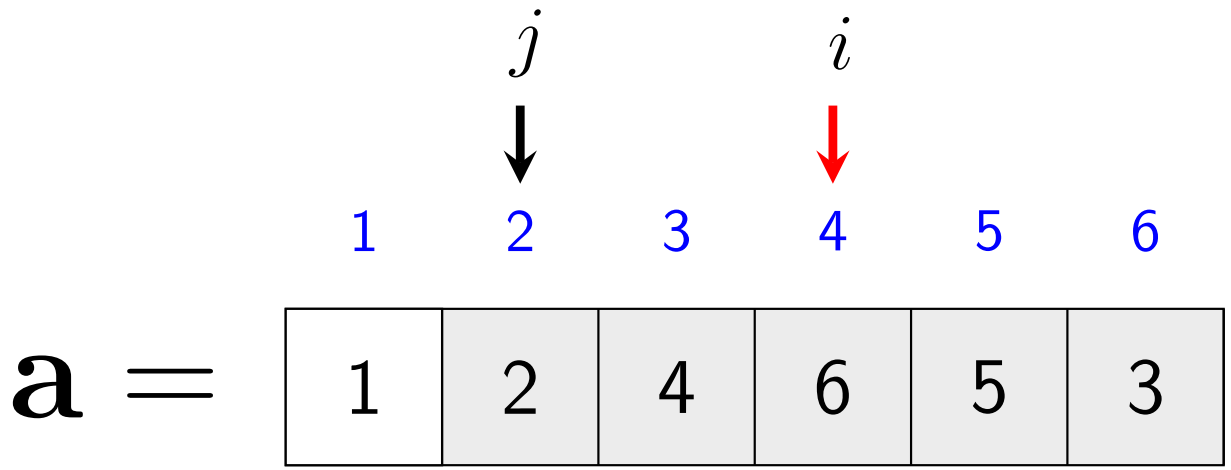
swap a_j and a_{min}

Selection Sort



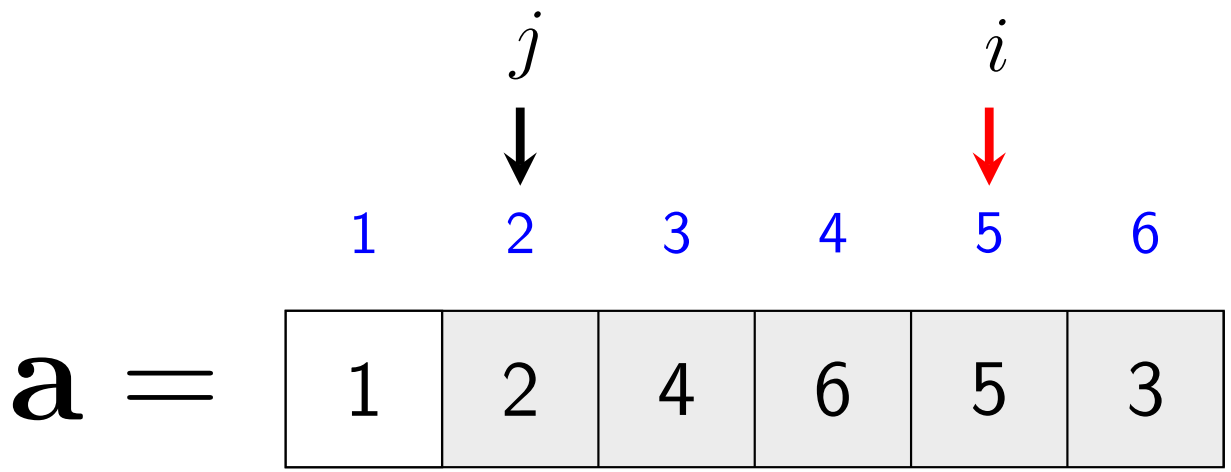
$$min = 2$$

Selection Sort



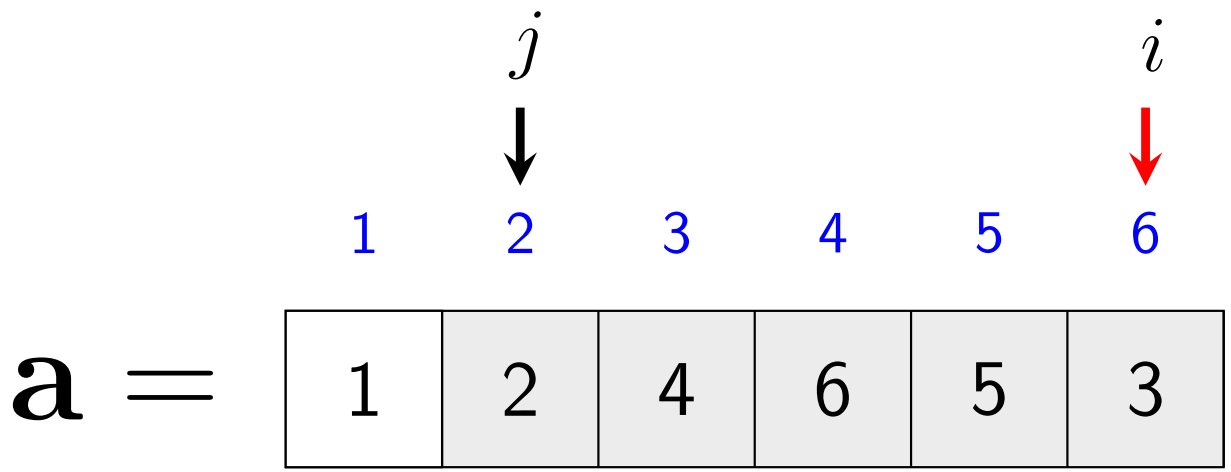
$$min = 2$$

Selection Sort



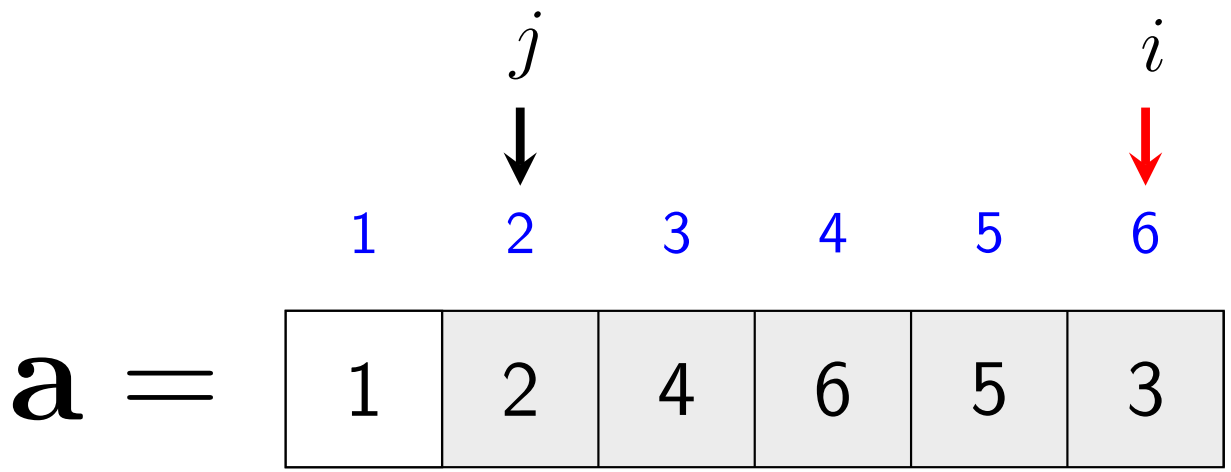
$$min = 2$$

Selection Sort



$$min = 2$$

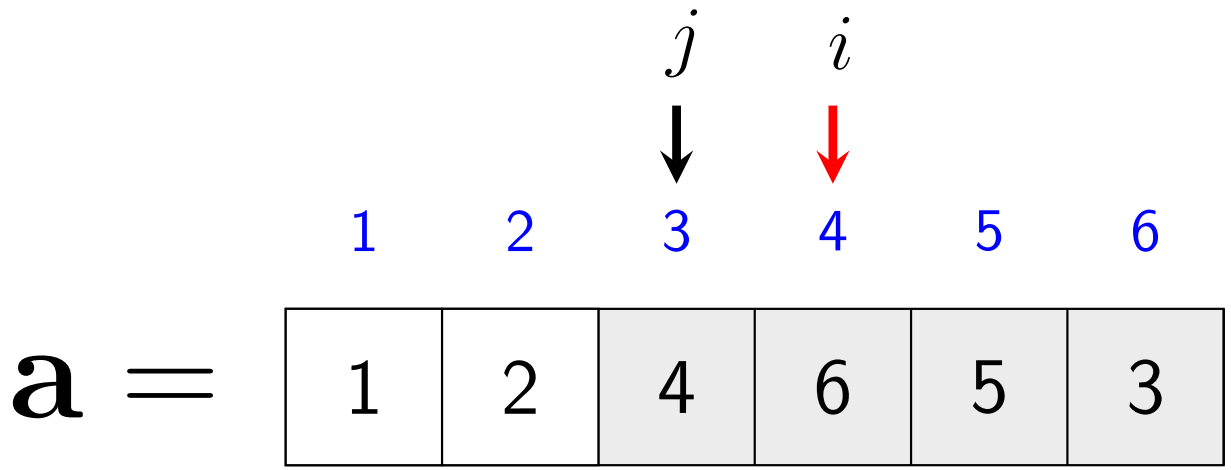
Selection Sort



$min = 2$

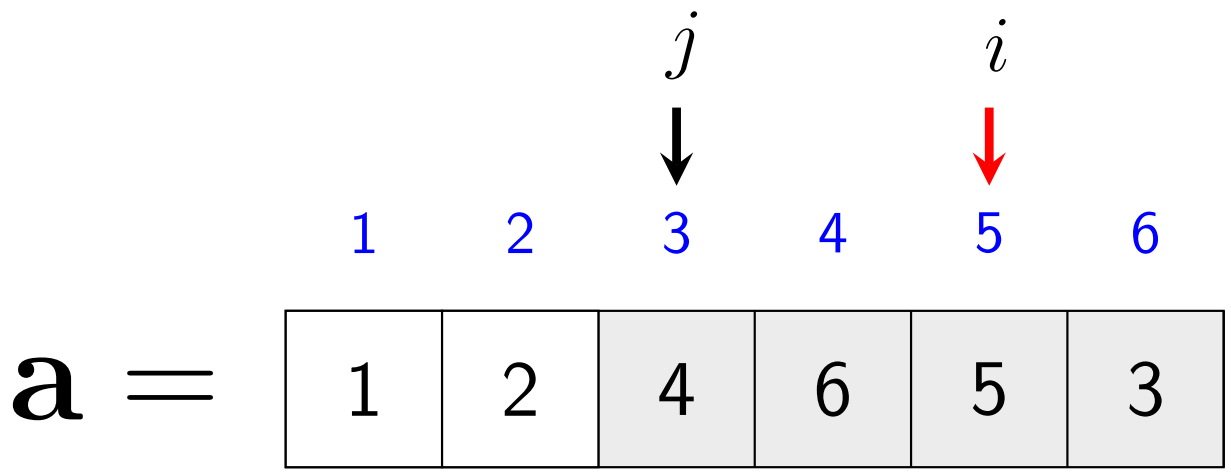
swap a_j and a_{min}

Selection Sort



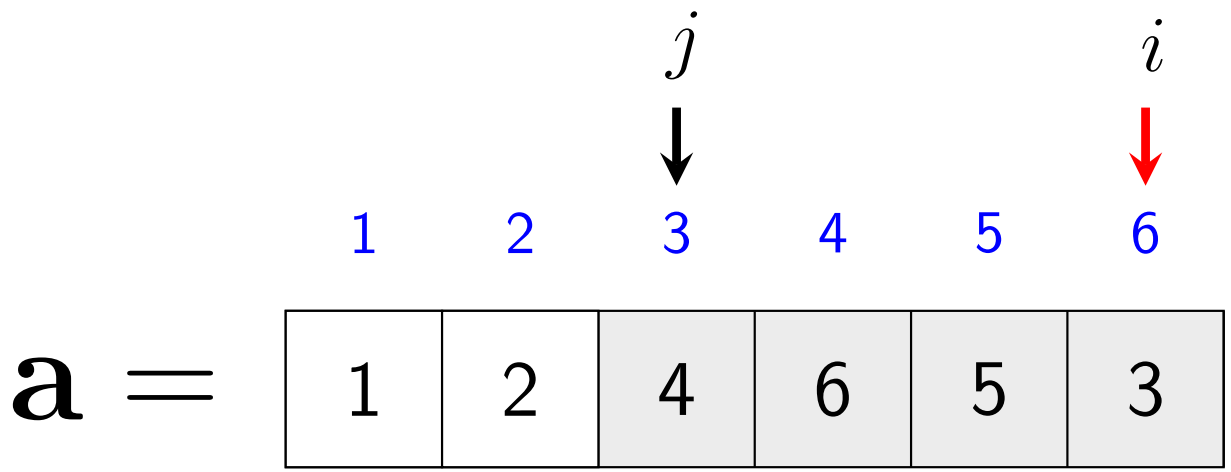
$$min = 3$$

Selection Sort



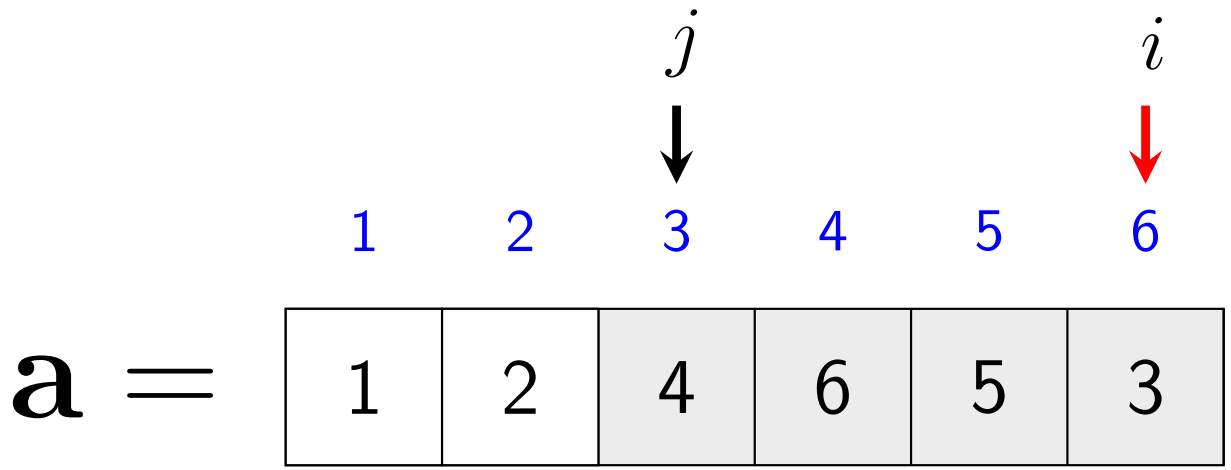
$$min = 3$$

Selection Sort



$$min = 6$$

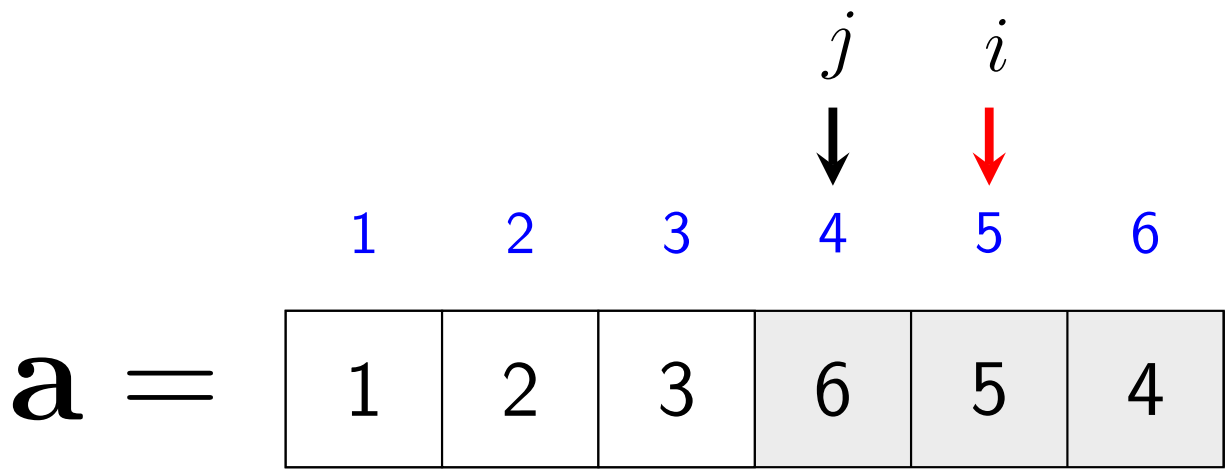
Selection Sort



$min = 6$

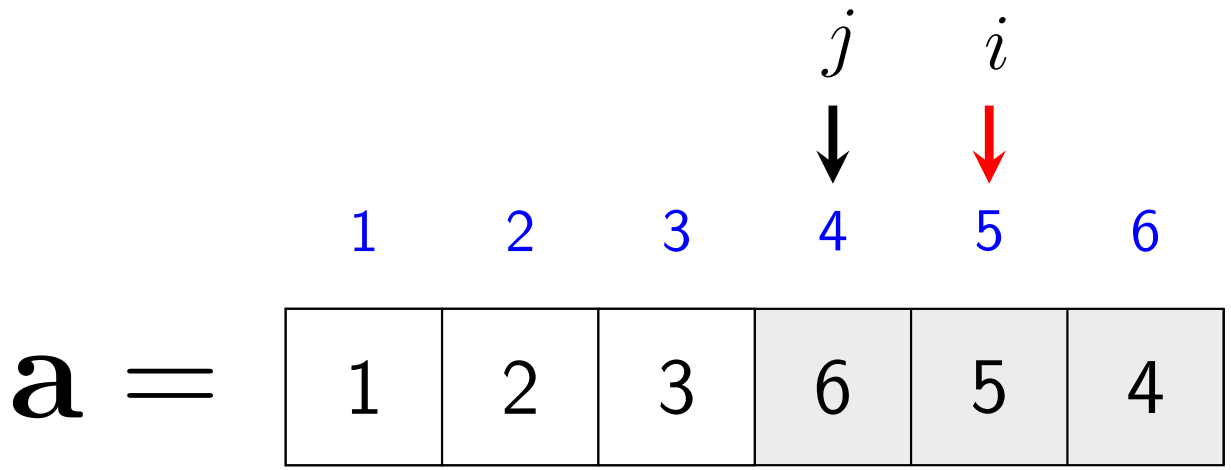
swap a_j and a_{min}

Selection Sort



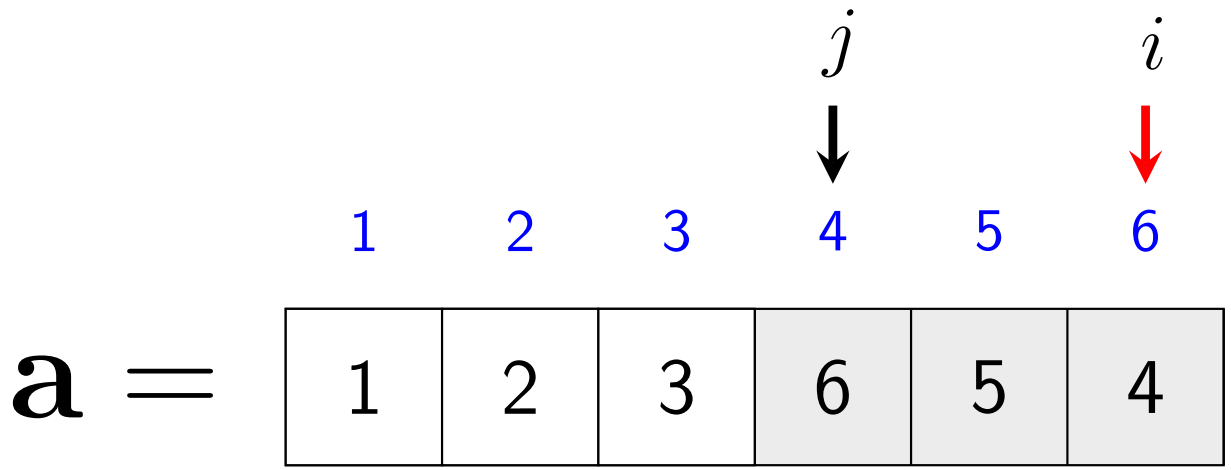
$$min = 4$$

Selection Sort



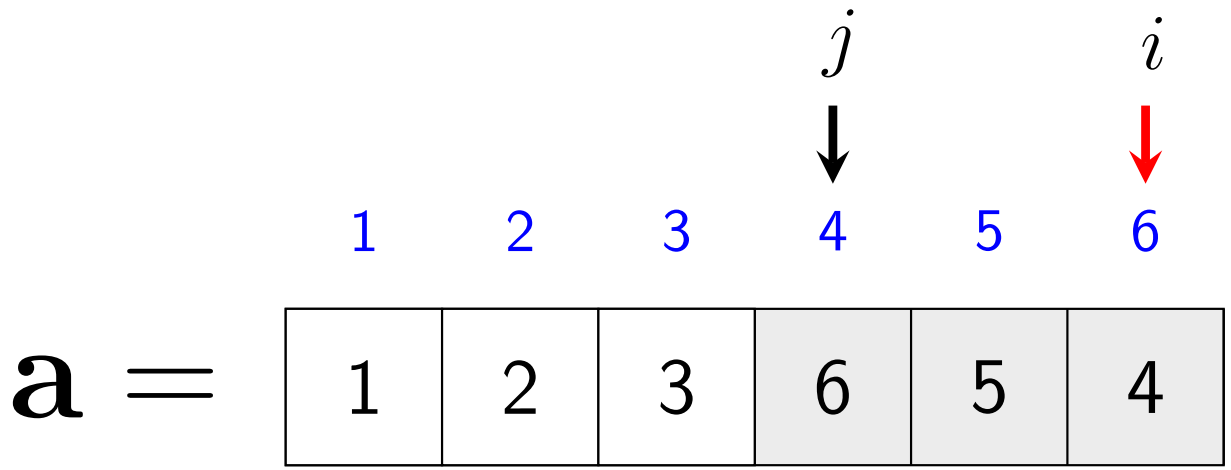
$$min = 5$$

Selection Sort



$$min = 6$$

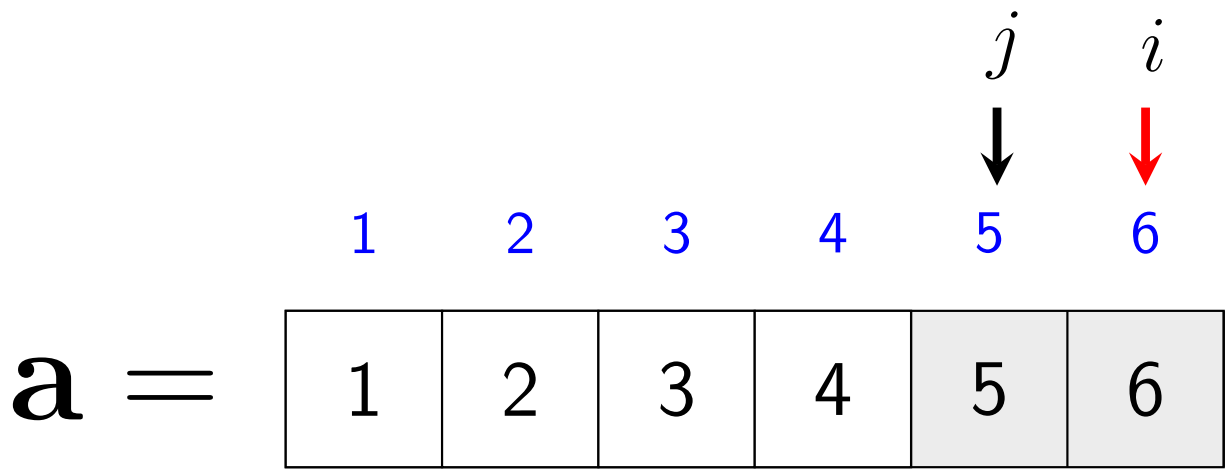
Selection Sort



$min = 6$

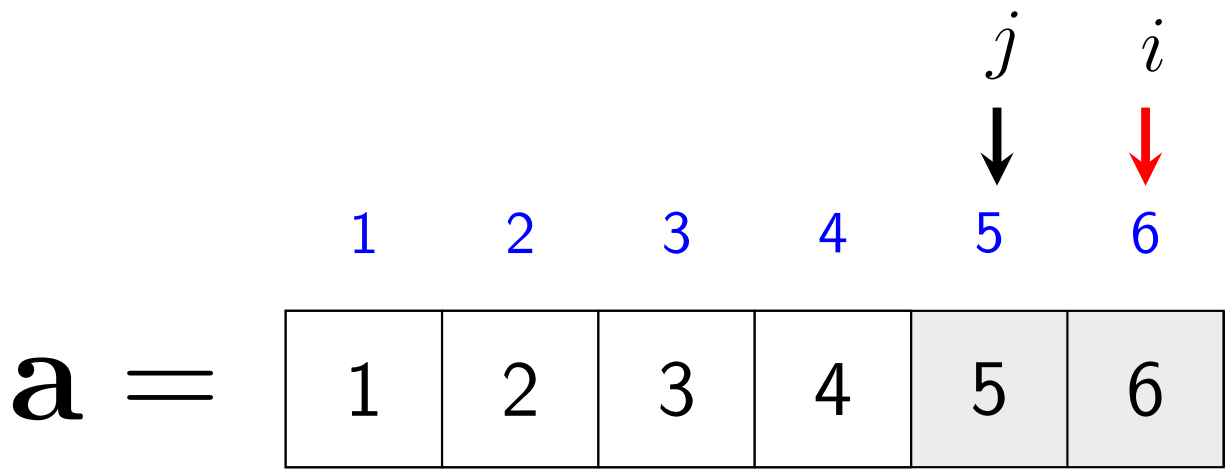
swap a_j and a_{min}

Selection Sort



$$min = 5$$

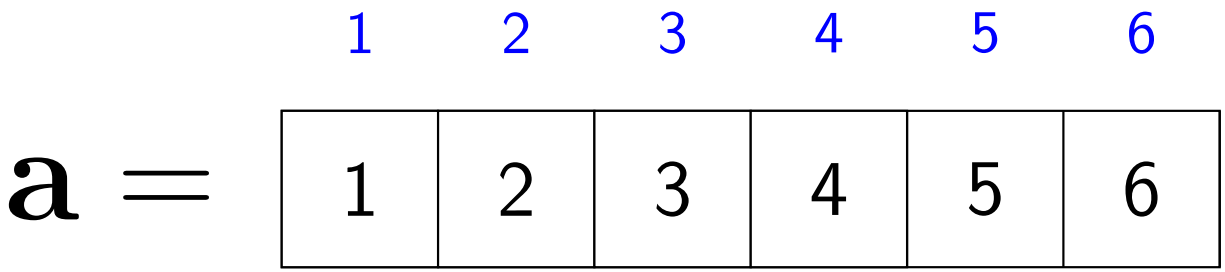
Selection Sort



$min = 5$

swap a_j and a_{min}

Selection Sort



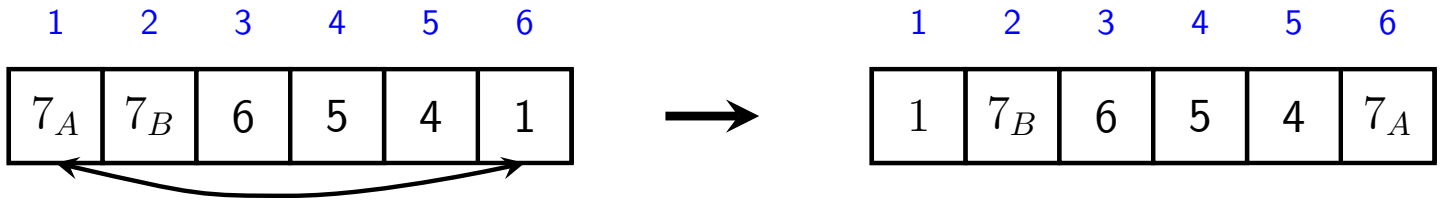
Selection Sort

Algorithm 7 Selection Sort

```
1: procedure SelectionSort(a)
2:     for j ← 1 to a.length−1 do
3:         min ← j
4:         for i ← j + 1 to a.length do
5:             if a_i < a_min then
6:                 min ← i
7:         swap a_j and a_min
8:     return a                                  ▷ Sorted sequence
```

Properties of Selection Sort

- Selection Sort is **in-place**.
- It is **not stable**!



- Selection Sort **always** requires $\frac{n(n-2)}{2}$ comparisons $= \Theta(n^2)$.
- So it has the **same** worst case time complexity as Insertion Sort, but has **inferior** average case and best case complexity.

  (The average case is of the same Θ, but the actual number of comparisons is twice that of Insertion Sort.)
- It only performs at most $n - 1$ swaps (fewer than Insertion Sort); this makes it attractive when physical swapping is expensive.

Summary

- Algorithms: use of pseudocode.
- Binary Search has time complexity $O(\log_2(n))$ on *sorted* lists.
- Simple Sorts, such as Insertion and Selection Sort, have $O(n^2)$ time complexity (number of comparisons).

Further Reading:

1. Chapter 2 in Sedgewick and Wayne **Algorithms (4th Edition)**.
2. Chapter 2 in Cormen (CLRS) **Introduction to Algorithms**.

Acknowledgements: Partly based on earlier COMP 1201 slides by Drs Adam Prugel-Bennet, Long Tran-Thanh and Baharak Rastegari, University of Southampton.