Week 7 Part I

Kyle Dewey

Overview

- Code from last time
- Array initialization
- Pointers vs. arrays
- Structs
- `typedef`
- Bubble sort (if time)

Code Wrap-Up

Array Initialization

```
// fine
int arr1[3] = { 1, 2 3 };
...
// compiler issues warning
int arr2[2] = { 4, 5, 6 };
...
// arr3 contains { 1, 0, 0 };
int arr3[3] = { 1 };
```

Pointers vs. Arrays

Pointers vs. Arrays

- Massive point of confusion that even C veterans mess up on
- C allows for pointers to be treated like arrays:

```
char* string = “moo”;
// string is a pointer
string [ 0 ]; // returns ‘m’
```

Pointers vs. Arrays

- However, C also has an explicit array type

```
char* string1 = “moo”; // pointer
char string2[] = “cow”; // array
```

Pointers vs. Arrays

- Pointers can point anywhere, as long as it is of the correct type

```
char character;
char* string1;

string1 = “moo”;
string1 = “cow”;
string1 = &character;
```

Pointers vs. Arrays

- Variables of the array type can **only** point to what they were initialized to

```
char string[] = “foobar”;

string[ 0 ]; // returns ‘f’
string = “foo”; // compiler error
```

Pointers vs. Arrays

- Variables of the array type **must** be initialized to something
  - gcc gives an error
  - ch allows this but crashes if you try to do anything with it

```
char string[]; // compiler error
```

...what?

- Questions:
    - Why introduce a special type that is more restrictive than a pointer?
    - Why can’t they be reassigned?
    - Why is this useful?

Internal Representation

- A pointer is a variable that holds a memory address
- The array type is actually an address in and of itself
    - Effectively a constant

Internal Representation

- Since it acts like a constant, it cannot be reassigned
- When we say:

```
char string[] = “moo”;
printf( “%s”, string );
```

- ...the compiler replaces all occurrences of `string` with the actual memory address where “`moo`” is stored

Internal Representation

- When we say:

```
char* string = “moo”;
printf( “%s”, string );
```

- ...the compiler will first look up what the value of `string` currently is, and pass that value along to `printf` as a memory address
- There is an extra step here

Analogy

- With the array type, it’s like:

```
#define CONSTANT 5
printf( “%i”, CONSTANT );
```

- With the pointer type, it’s like:

```
int x = 5;
printf( “%i”, x );
```

Decay

- Array types can **decay** to a pointer type
- This can be seen with functions:

```
void foo( int* pointer );

int main() {
  int arr[] = { 1, 2, 3 };
  foo( arr ); // legal
}
```

What to Remember

- Pointers can act like arrays, but arrays cannot act like pointers
- When the compiler starts complaining about `*` versus `[]`, this could be why

Structs

Problem

- We want to represent a phone book
- Each entry has:
    - Name
    - Phone number
    - Address

Question

- Which type(s) is/are appropriate for:
  - Name?
  - Phone Number?
  - Address?

Possible Representation

- Use **parallel arrays**
  - Each array holds one kind of item
  - Index N refers to all information for entry #N

```
char** name;
char** address;
int* phoneNumber;
```

Problem

- Poor separation of concerns
- We have to pass around everything related to one person, which is annoying and error prone

```
void printPerson( char* name,
                  char* address,
                  int phone );
```

Another Solution

- Use structures, aka. `struct`s
- Put all data relevant to one entry in one place

```
struct person {
  char* name;
  char* address;
  int phone;
};
```

Structs

```
struct person {
   char* name;
   char* address;
   int phone;
};
```

```
void printPerson( struct person p );
```

Accessing Structs

- Use the dot (.) operator

```
struct person {
  char* name;
  char* address;
  int phone;
};

void printPerson( struct person p ) {
  printf( “Name: %s\n”, p.name );
  printf( “Address: %s\n”, p.address );
  printf( “Phone: %i\n”, p.phone );
}
```

Modifying Structs

- The dot (.) operator can be used along with assignment

```
struct person {
  char* name;
  char* address;
  int phone;
};
struct person p;
p.name = “foo”;
p.address = “123 Fake Street”;
p.phone = 0123456789
```

Initializing Structs

- For a struct definition like so:

```
struct pair {
  int x; int y; };
```

- We can do:

```
struct pair p = { 2, 3 };
struct pair p2 = { .x = 2, .y = 3 };
```

- (Doesn't work in ch, but it does in gcc)

Pointers to Structs

- Structs can also be accessed via pointers
- Can access like so:

```
struct person p;
struct person* pointer = &p;
(*p).name = “foo”;
(*p).address = (*p).name;
(*p).phone = 0123456789
```

Pointers to Structs

- Structs can also be accessed via pointers
- Can also access with the more readable arrow operator

```
struct person p;
struct person* pointer = &p;
p->name = “foo”;
p->address = p->name;
p->phone = 0123456789
```

Struct Semantics

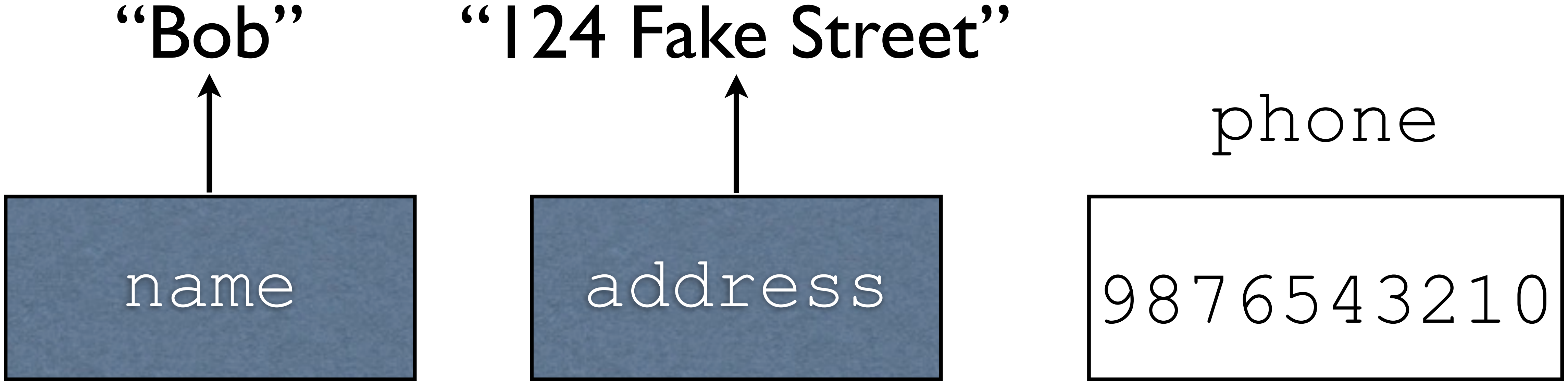
- Consider again:

```
void printPerson( struct person p );
```

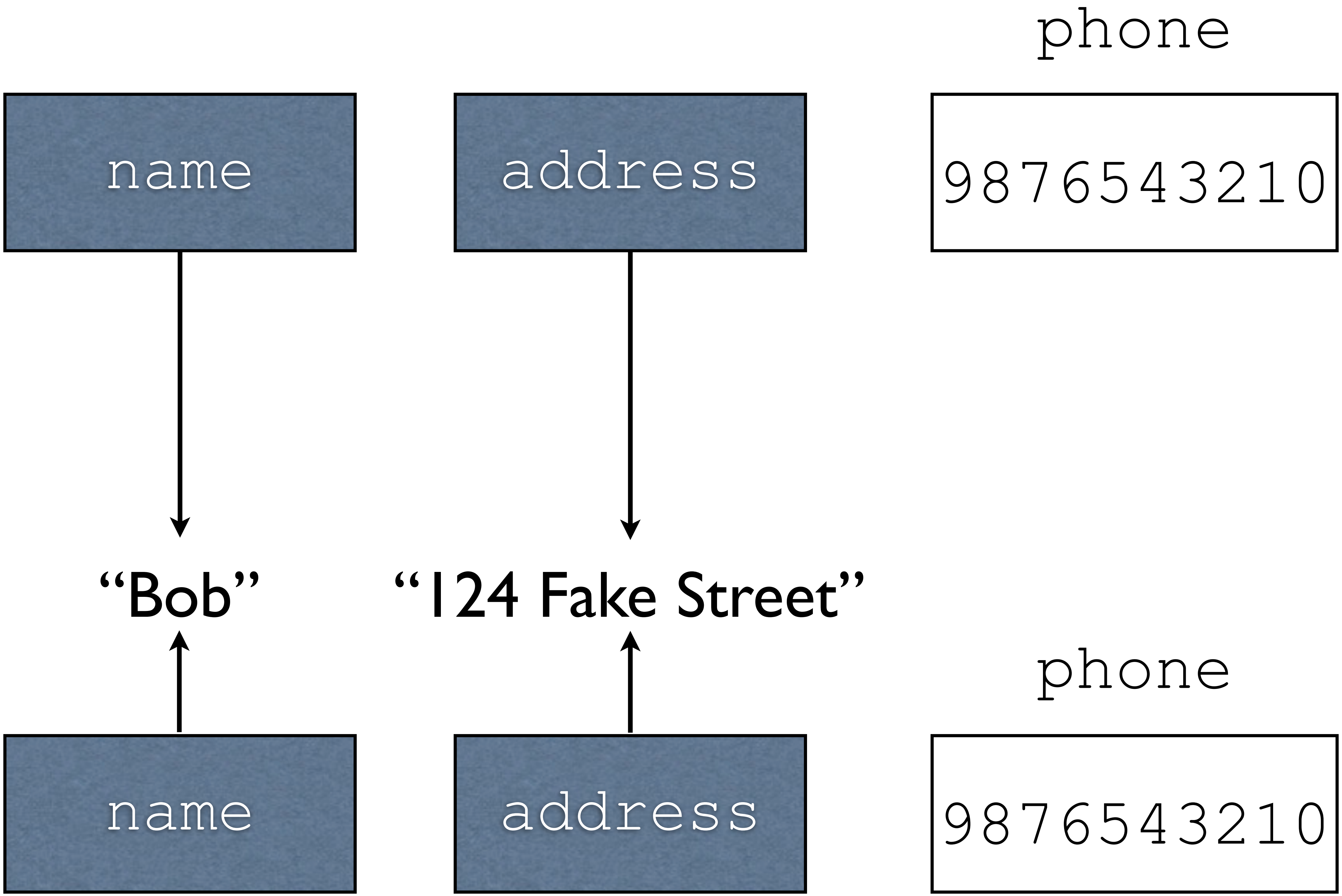
- When structs are passed, the whole thing is copied
- Note that this is a **shallow copy**

Shallow Copy

```
struct person {
   char* name;
   char* address;
   int phone;
};
```



Shallow Copy



Question

```
struct foo {
  int x;
};
void bar( struct foo f ) {
  f.x = 10;
}
int main() {
  struct foo f;
  f.x = 5;
  bar( f );
  // what’s f.x?
  return 0;
}
```

Question

```
struct foo {
  char* x;
};
void bar( struct foo f ) {
  f.x = “moo”;
}
int main() {
  struct foo f;
  f.x = “cow”;
  bar( f );
  // what’s f.x?
  return 0;
}
```

Question

```
struct foo {
  int x;
};
void bar( struct foo* f ) {
  f->x = 10;
}
int main() {
  struct foo f;
  f.x = 5;
  bar( &f );
  // what’s f.x?
  return 0;
}
```

Question

```
struct foo {
  char* x;
};
void bar( struct foo* f ) {
  f->x = “moo”;
}
int main() {
  struct foo f;
  f.x = “cow”;
  bar( &f );
  // what’s f.x?
  return 0;
}
```

Structs and Pointers

- Oftentimes programmers will prefer pointers to structs as opposed to just structs
  - Avoids extra copying
  - **Possibly** appropriate

`typedef`

typedef

- Defines a new type that is an alias for another type

Example

- Before `typedef`...

```
struct foo {
  int x;
};

void bar( struct foo f ) {
  f.x = 10;
}
```

Example

- After `typedef`

```
struct foo {
  int x;
};

typedef struct foo Foo;

void bar( Foo f ) {
  f.x = 10;
}
```

More Examples

```
typedef long double ld;
typedef unsigned long ul;
typedef int SuperAwesome;
```

Uses

- Shorten type names
- A point of abstraction

```
// for one computer
typedef EightBytes int;

// for another computer
typedef EightBytes long;
```

Bubble Sort (If time)

Bubble Sort

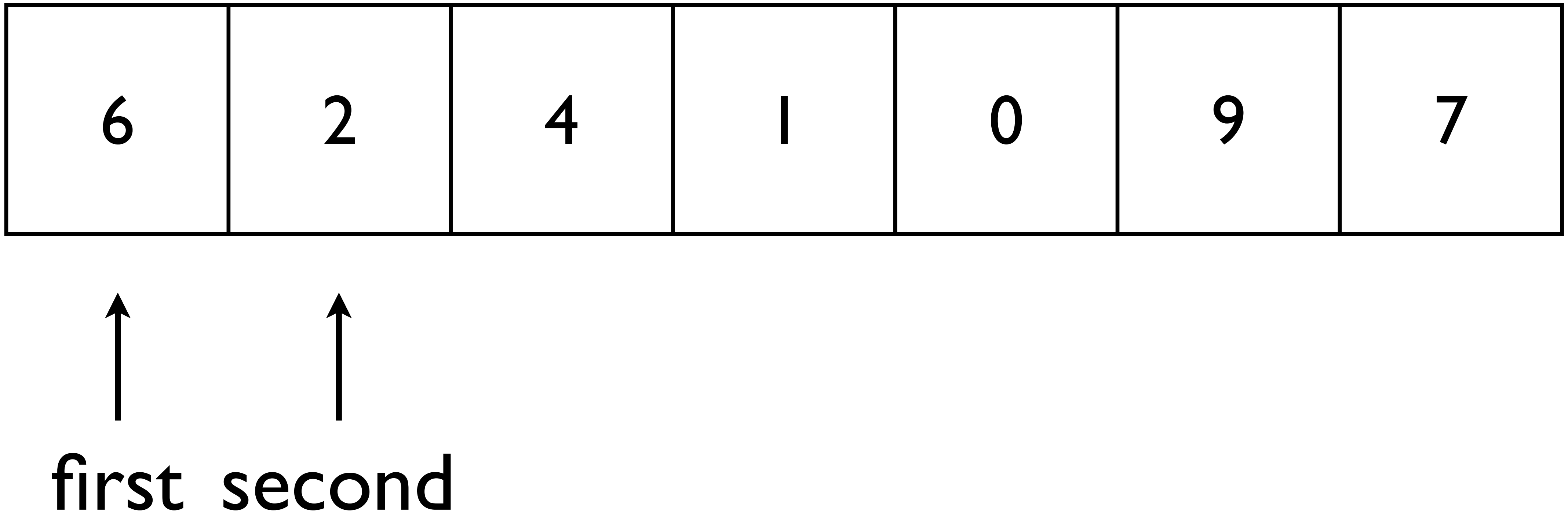
- Another sorting algorithm
- Basic idea:
  - Go through a list of numbers
  - Compare them pairwise
  - If a pair is out of order, swap them
  - Keep doing this until no swaps occur

Example

- We want to sort according to integer <=

6	2	4	1	0	9	7

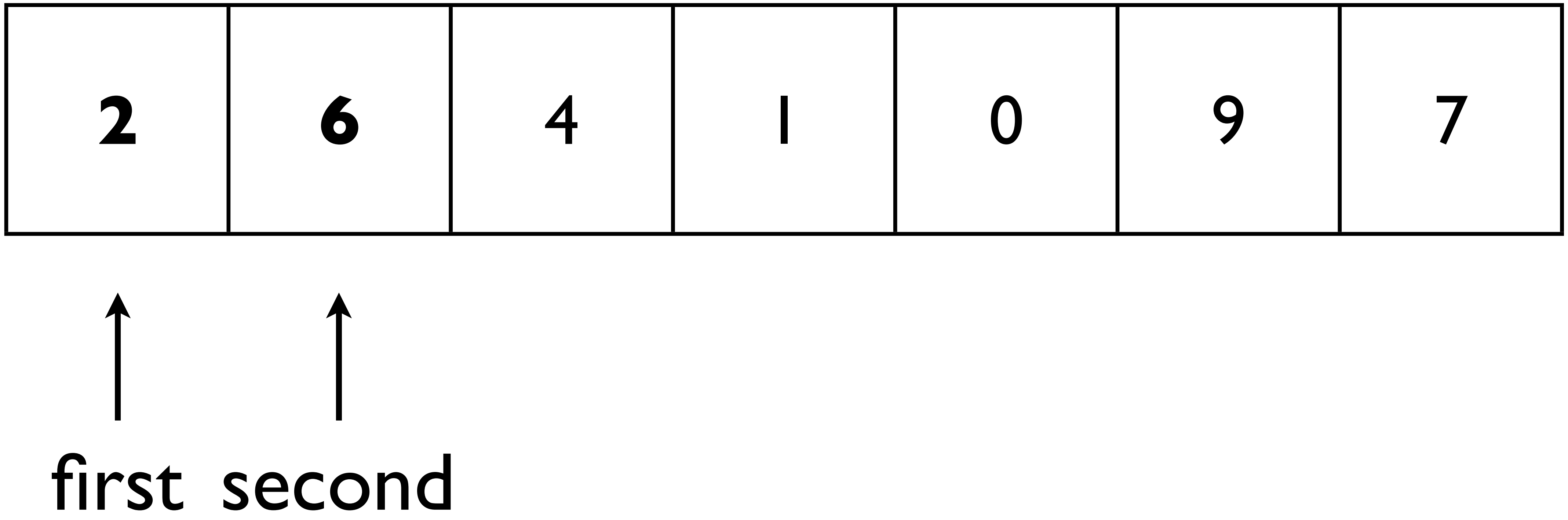
Example



6	2	4	1	0	9	7

Swap occurred?: False

Example

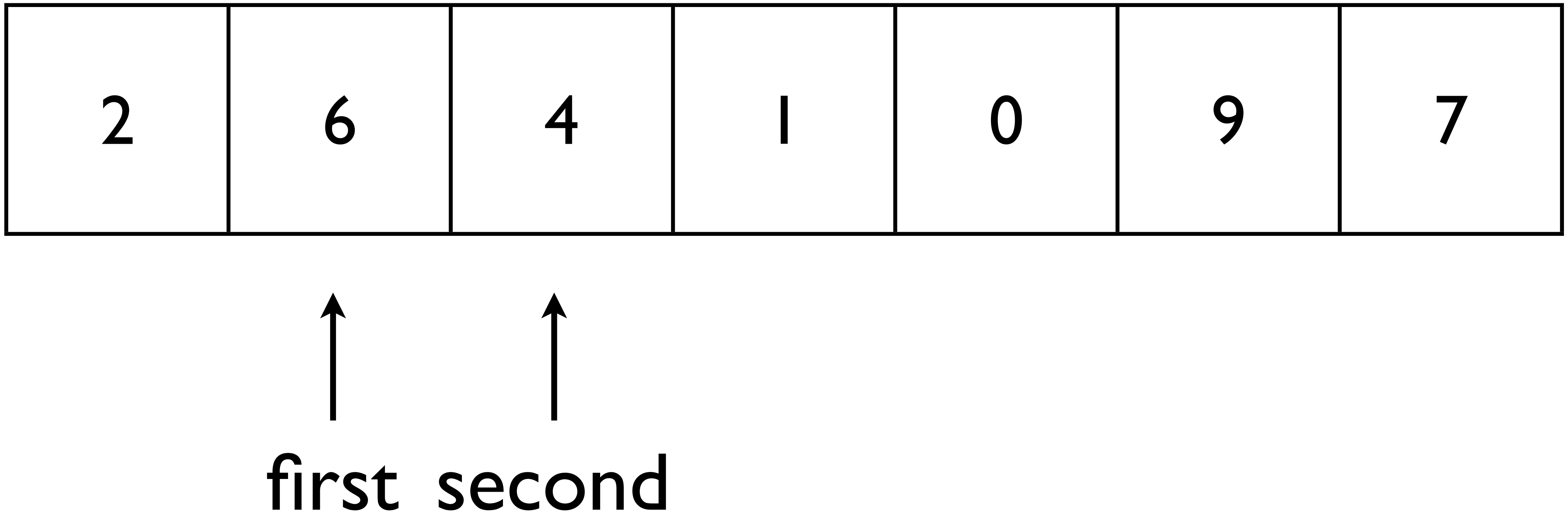


2	**6**	4	1	0	9	7

first second

Swap occurred?: **True**

Example

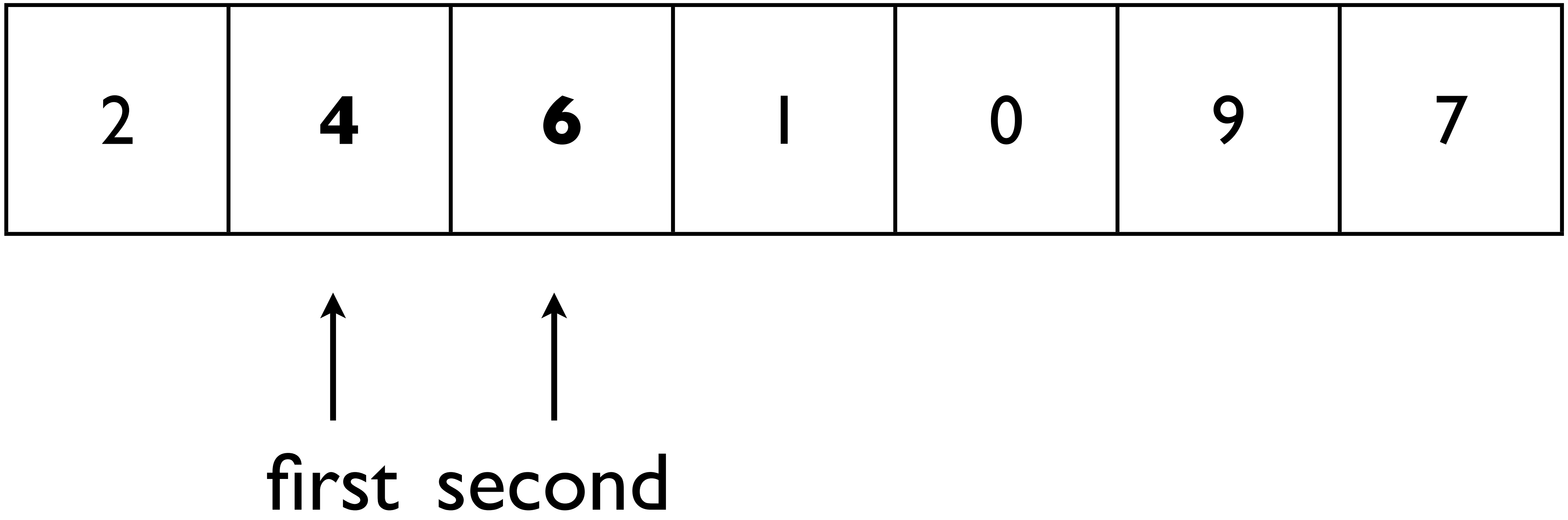


2	6	4	1	0	9	7

first second

Swap occurred?: True

Example

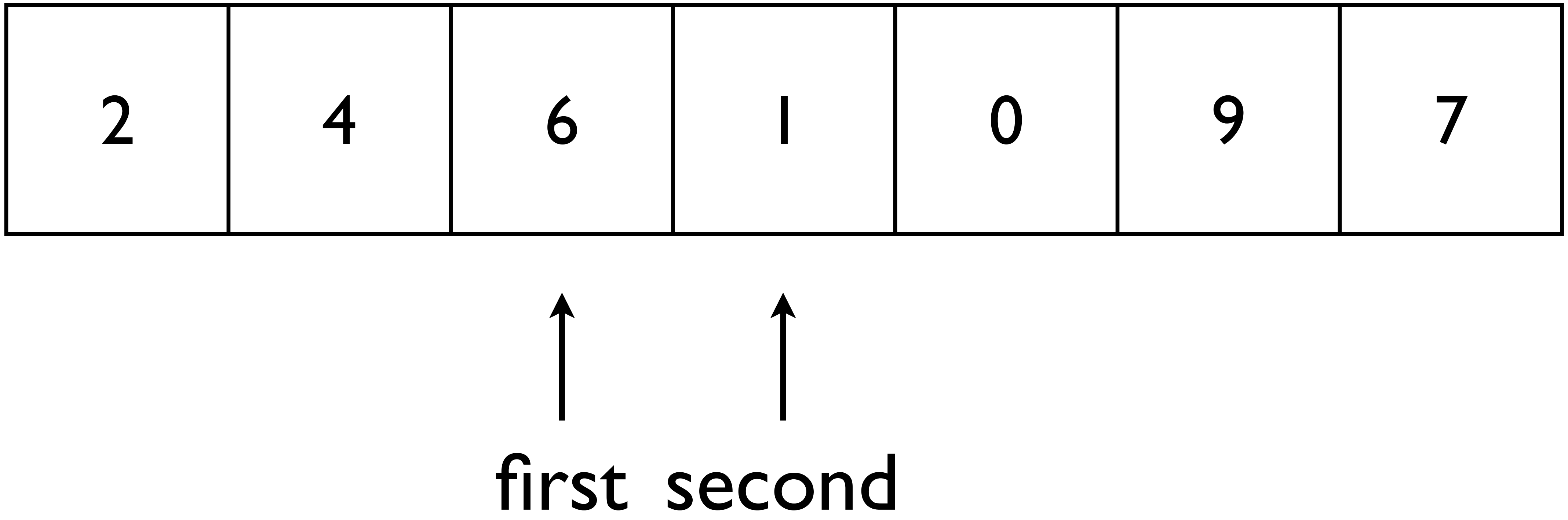


2	**4**	**6**	1	0	9	7

first second

Swap occurred?: True

Example

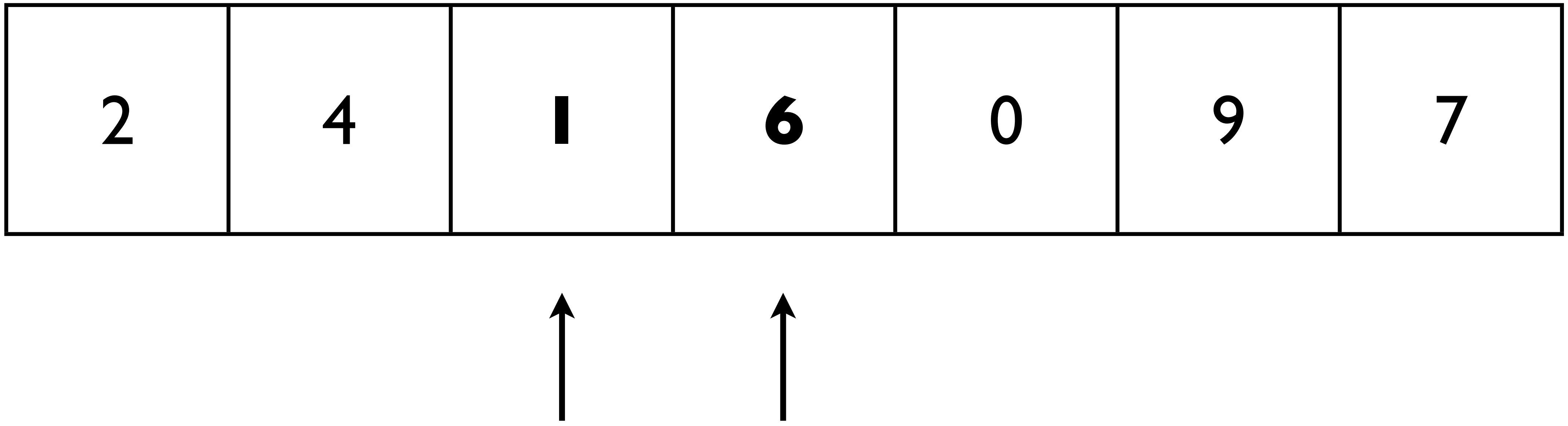


2	4	6	1	0	9	7

first second

Swap occurred?: True

Example

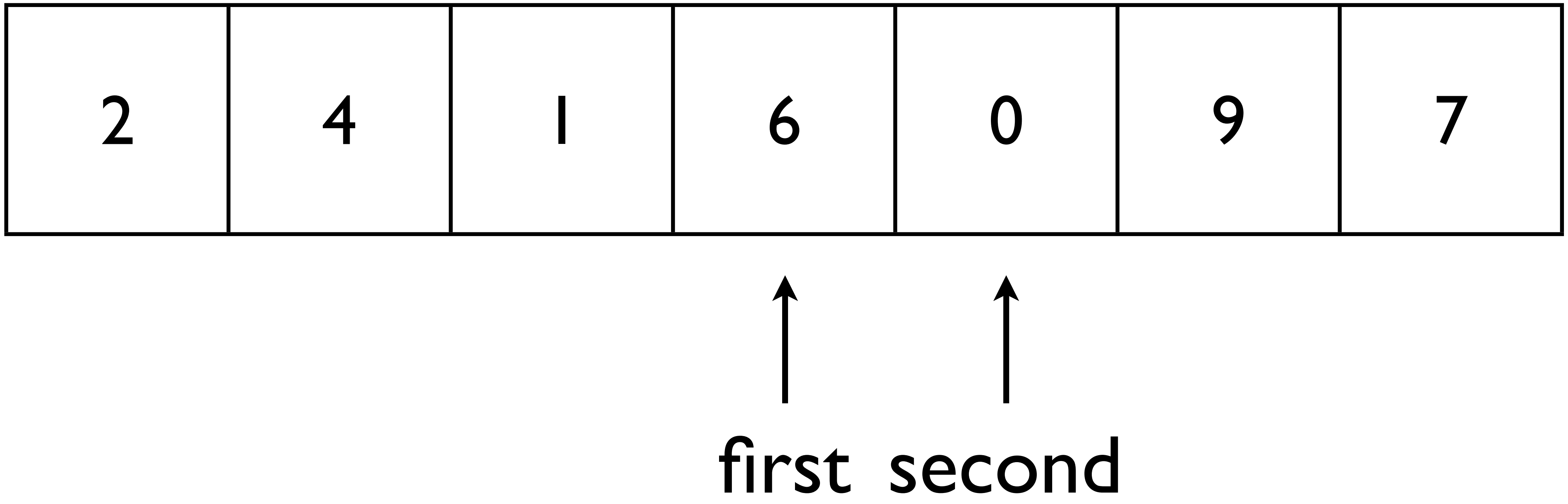


2	4	**1**	**6**	0	9	7

first second

Swap occurred?: True

Example

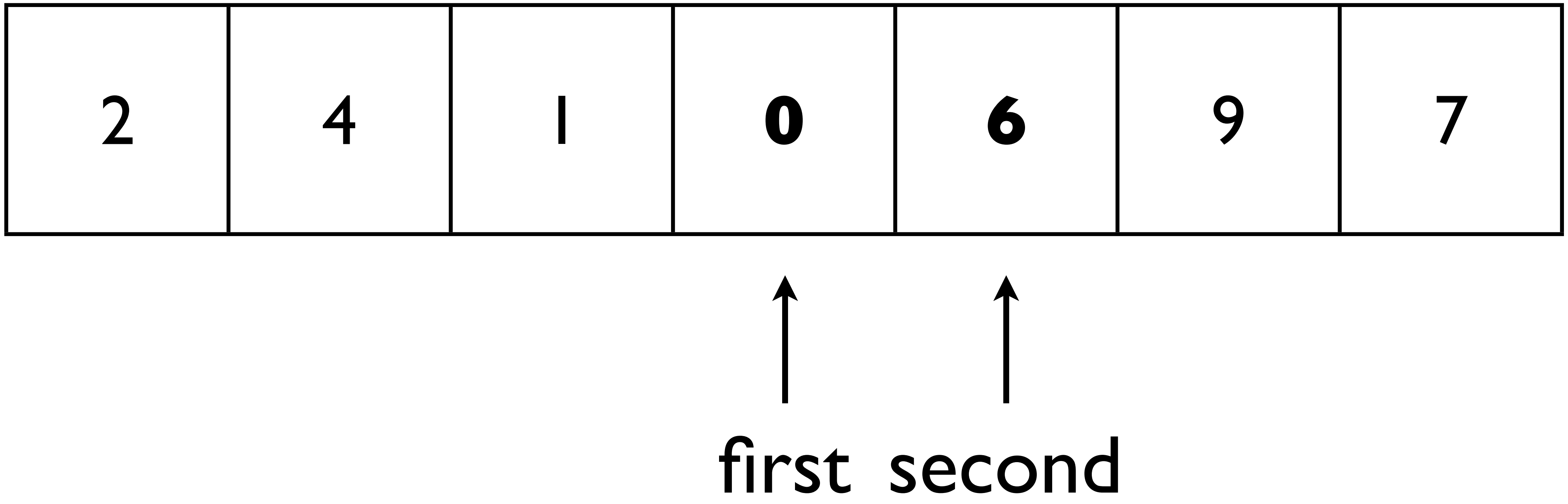


2	4	1	6	0	9	7

first second

Swap occurred?: True

Example

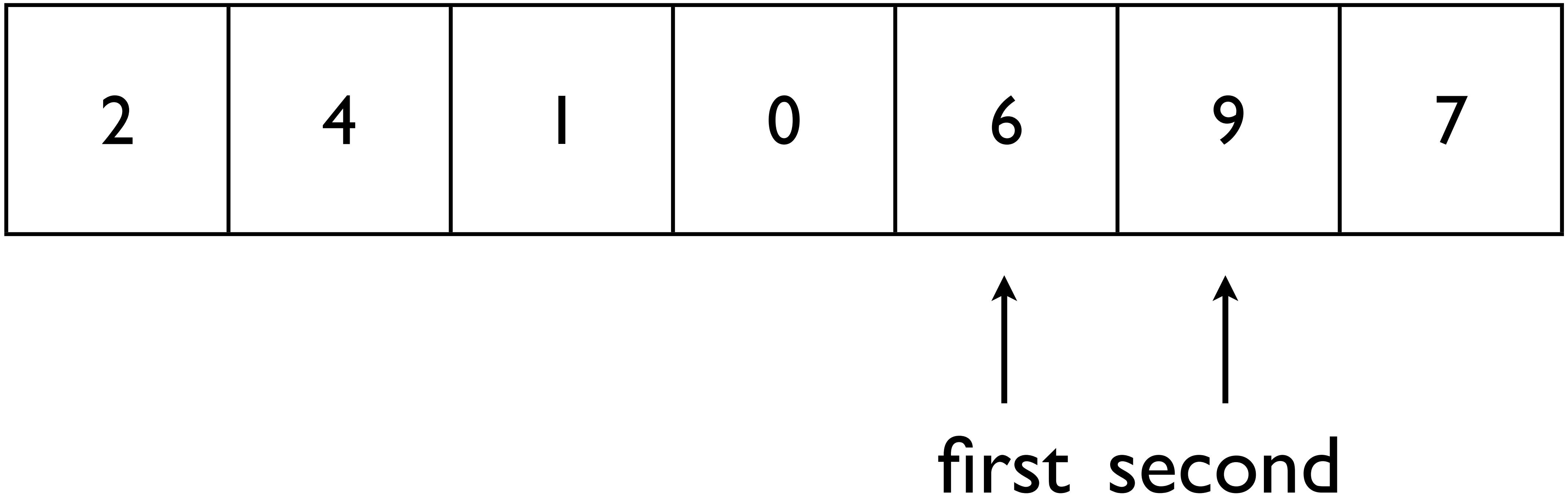


2	4	1	**0**	**6**	9	7

first second

Swap occurred?: True

Example

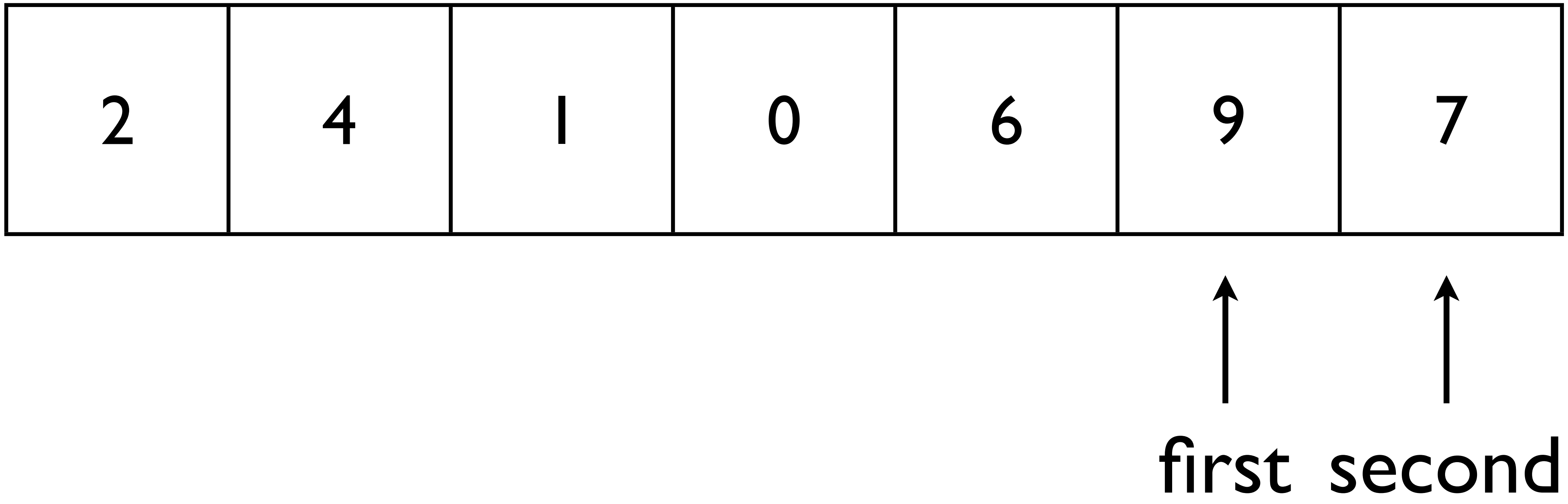


2	4	1	0	6	9	7

first second

Swap occurred?: True

Example

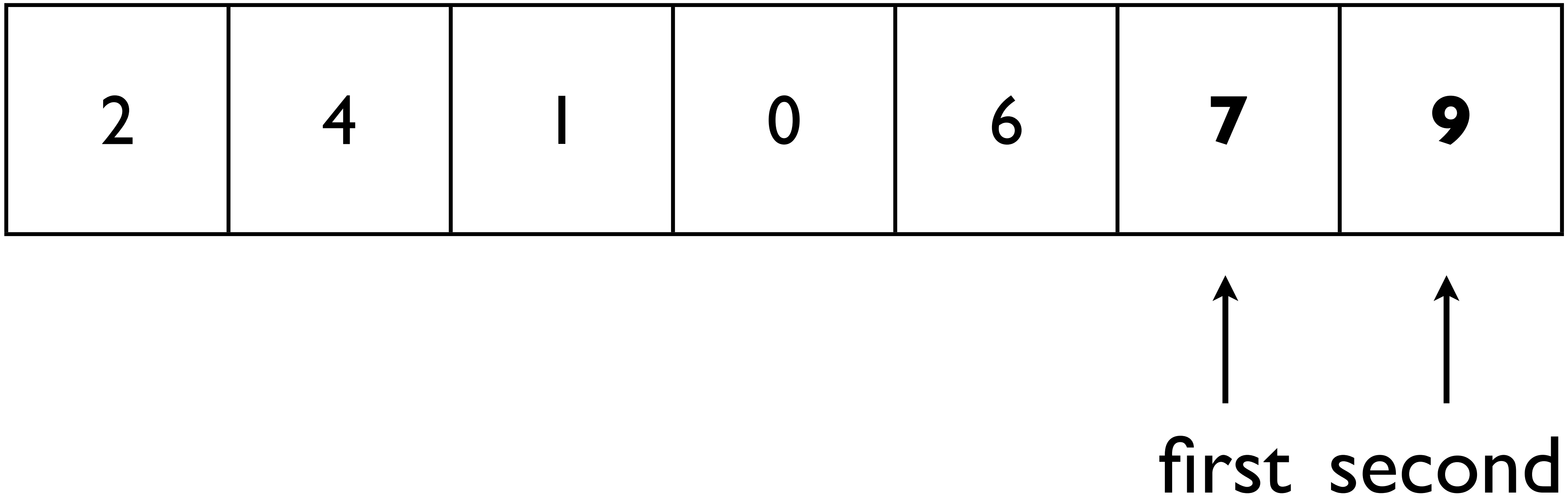


2	4	1	0	6	9	7

first second

Swap occurred?: True

Example

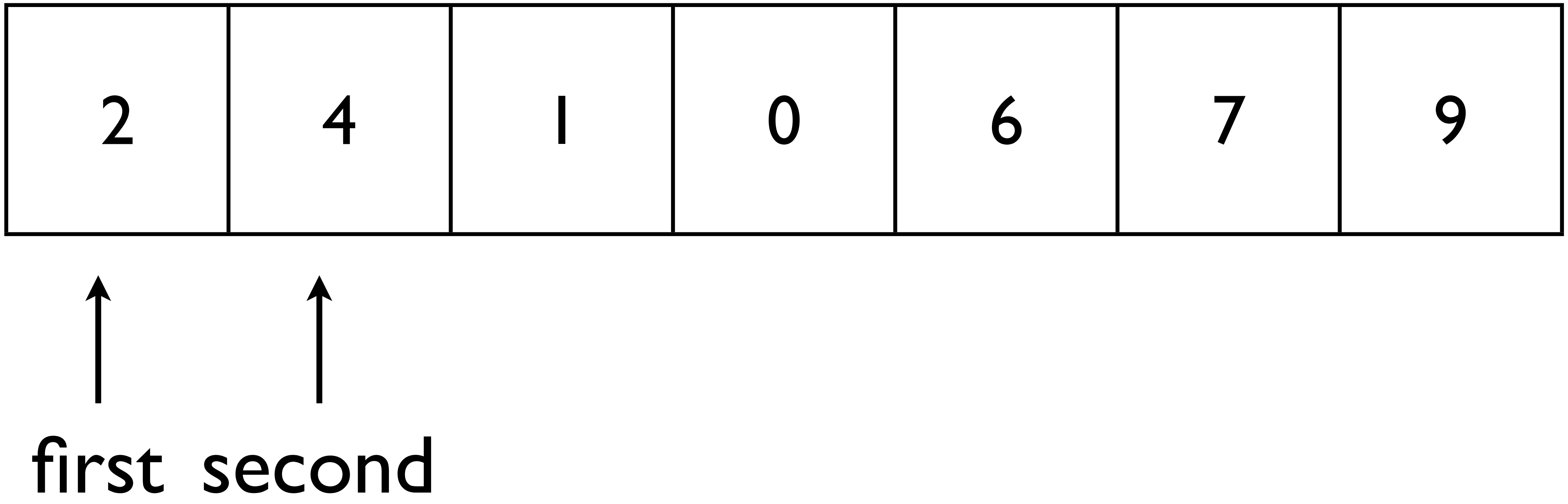


2	4	1	0	6	**7**	**9**

first second

Swap occurred?: True

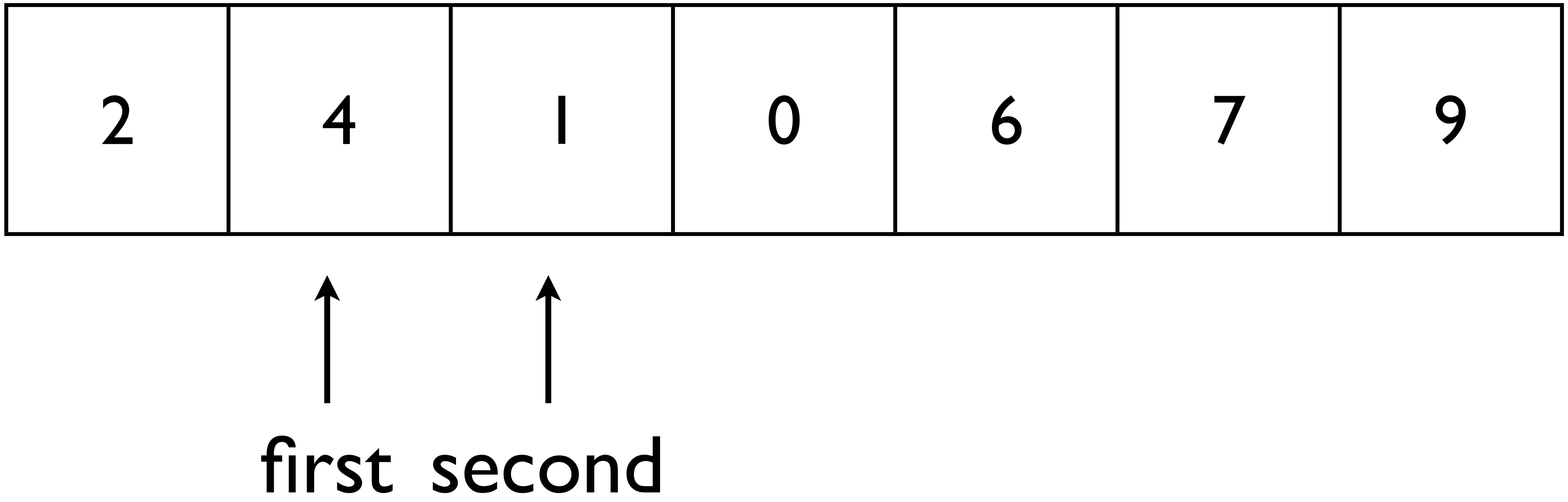
Example



2	4	1	0	6	7	9

Swap occurred?: **False**

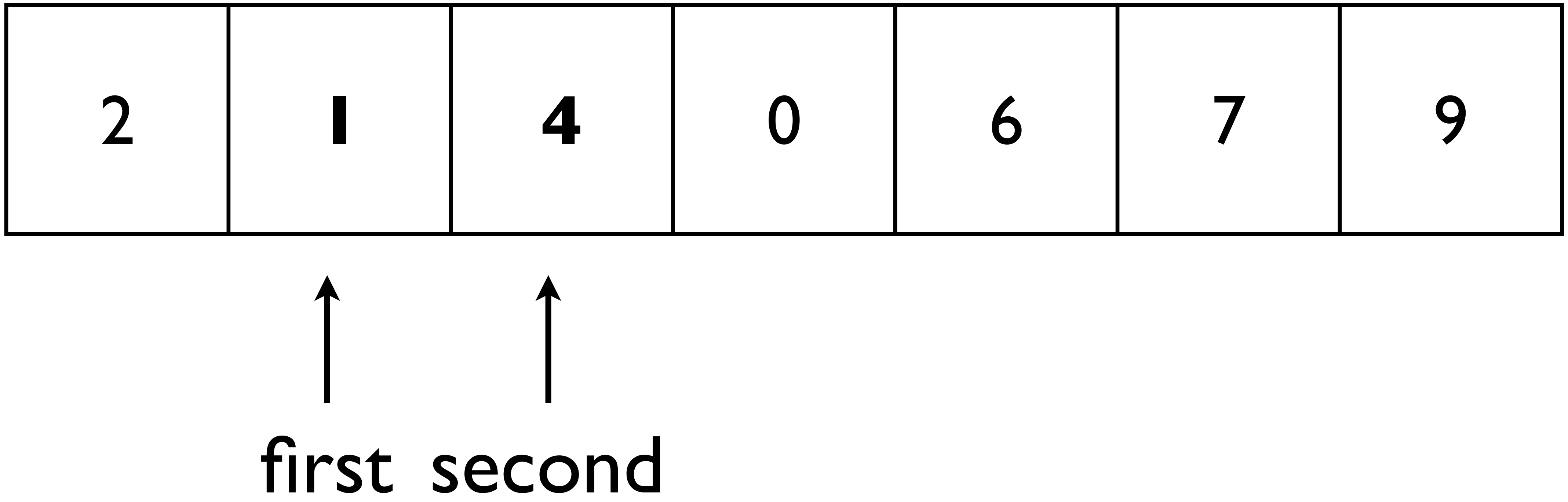
Example



2	4	1	0	6	7	9

Swap occurred?: False

Example

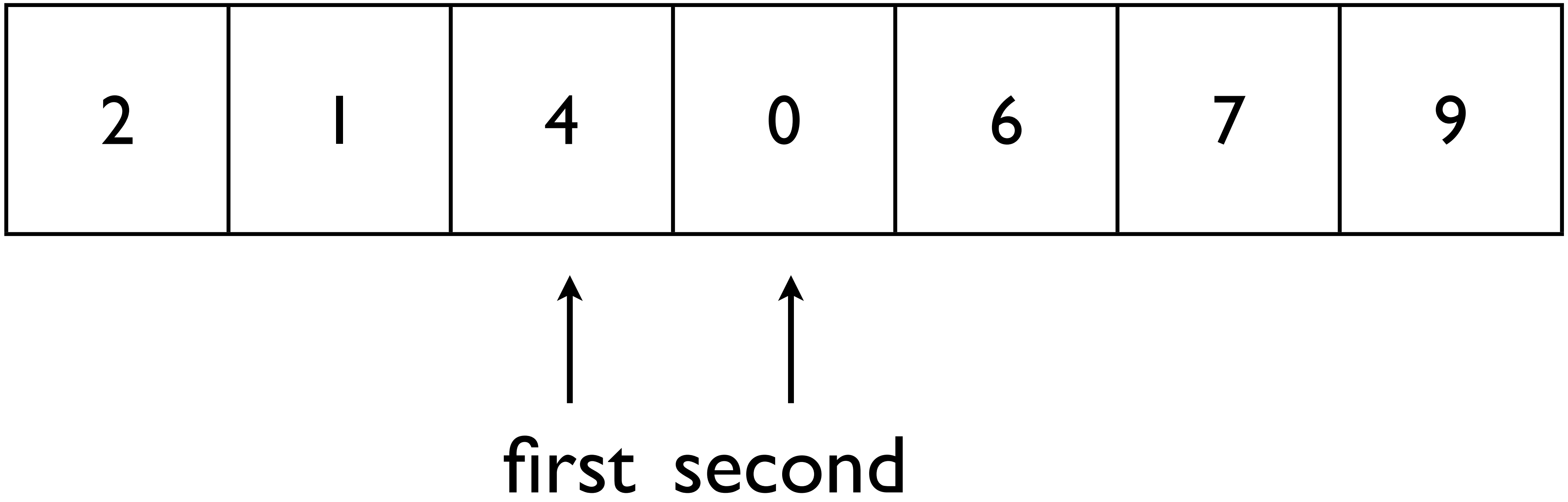


2	**1**	**4**	0	6	7	9

first second

Swap occurred?: **True**

Example

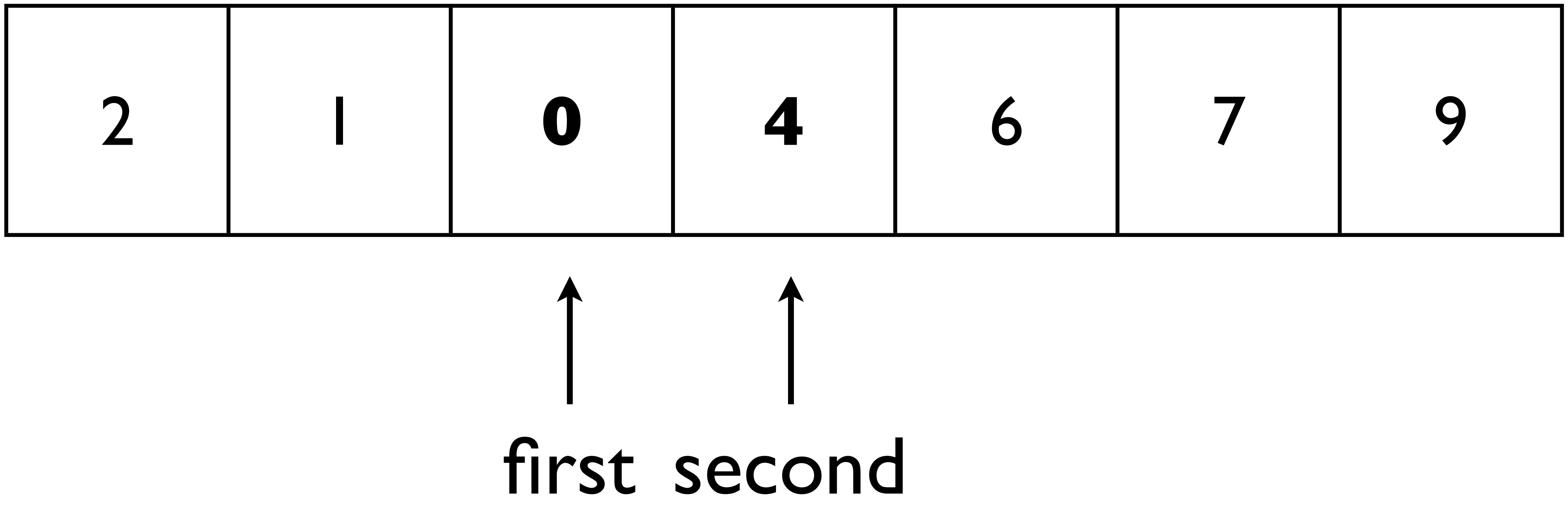


2	1	4	0	6	7	9

first second

Swap occurred?: True

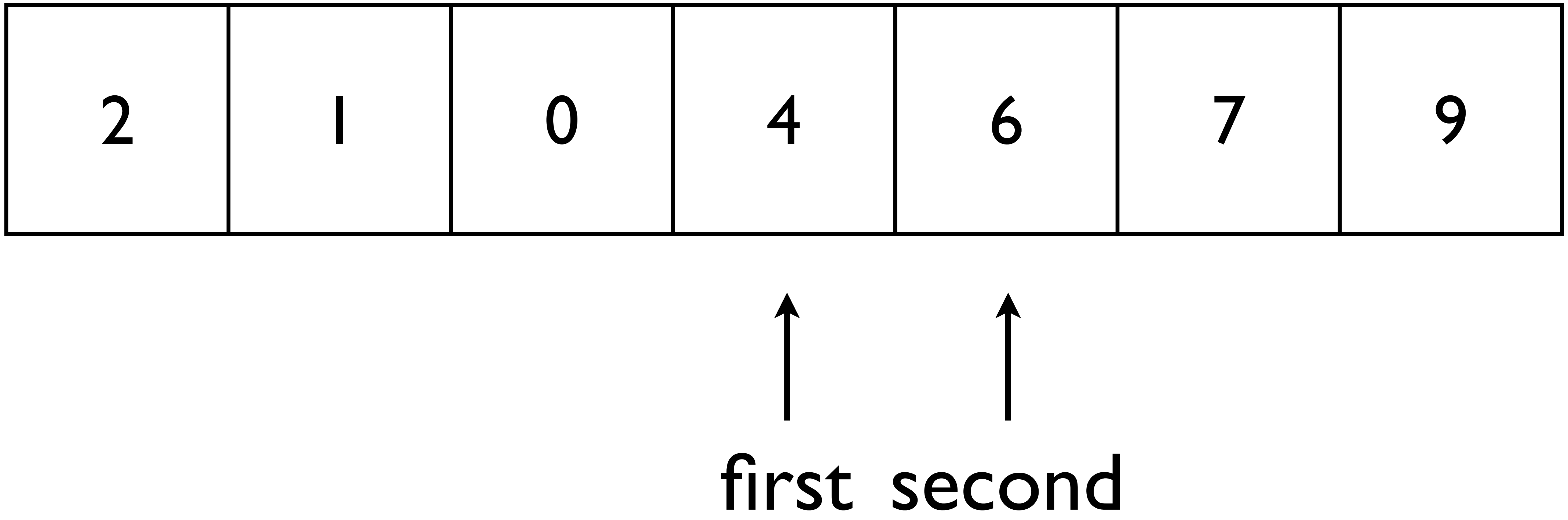
Example



2	1	**0**	**4**	6	7	9

Swap occurred?: True

Example

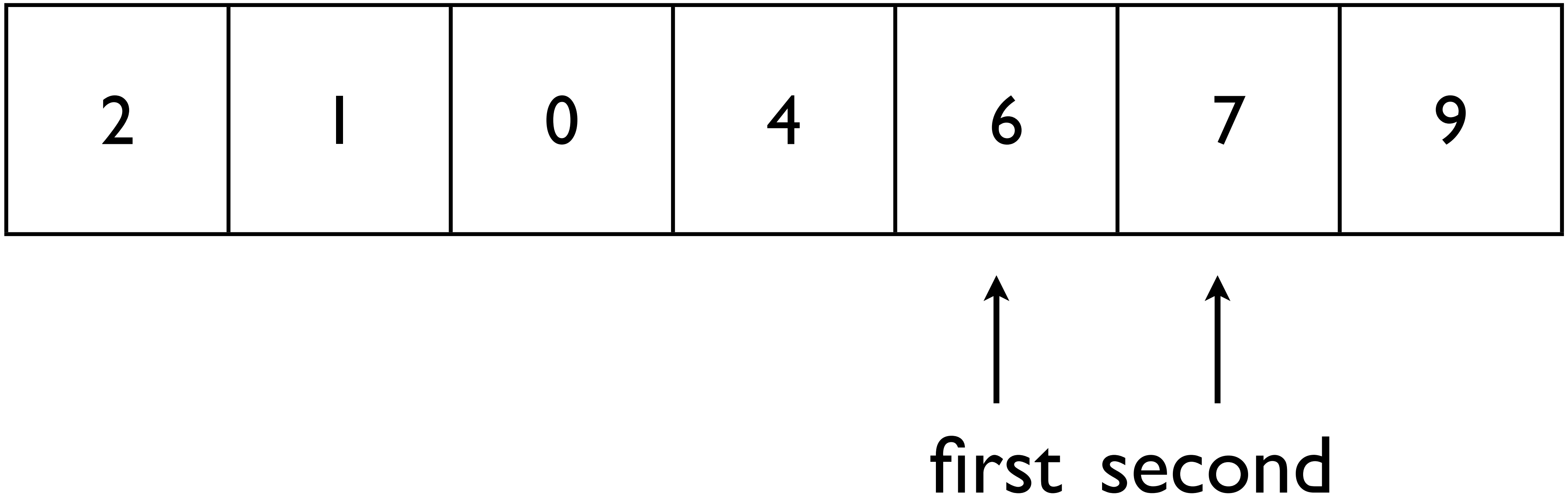


2	1	0	4	6	7	9

first second

Swap occurred?: True

Example

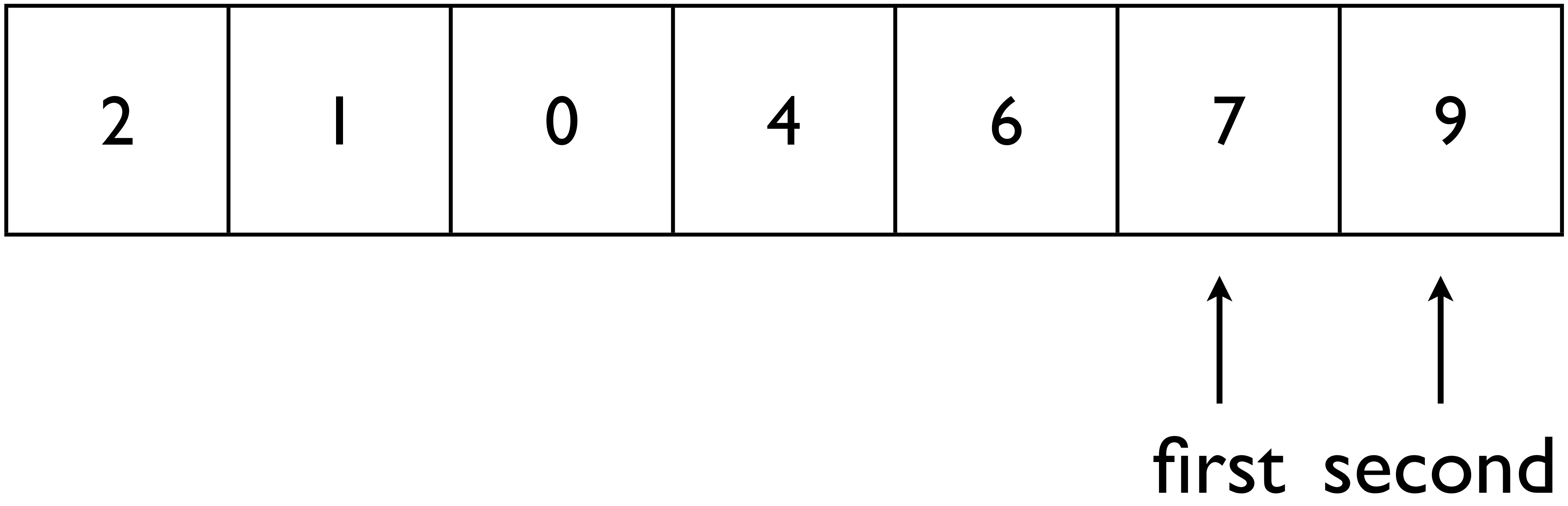


2	1	0	4	6	7	9

first second

Swap occurred?: True

Example

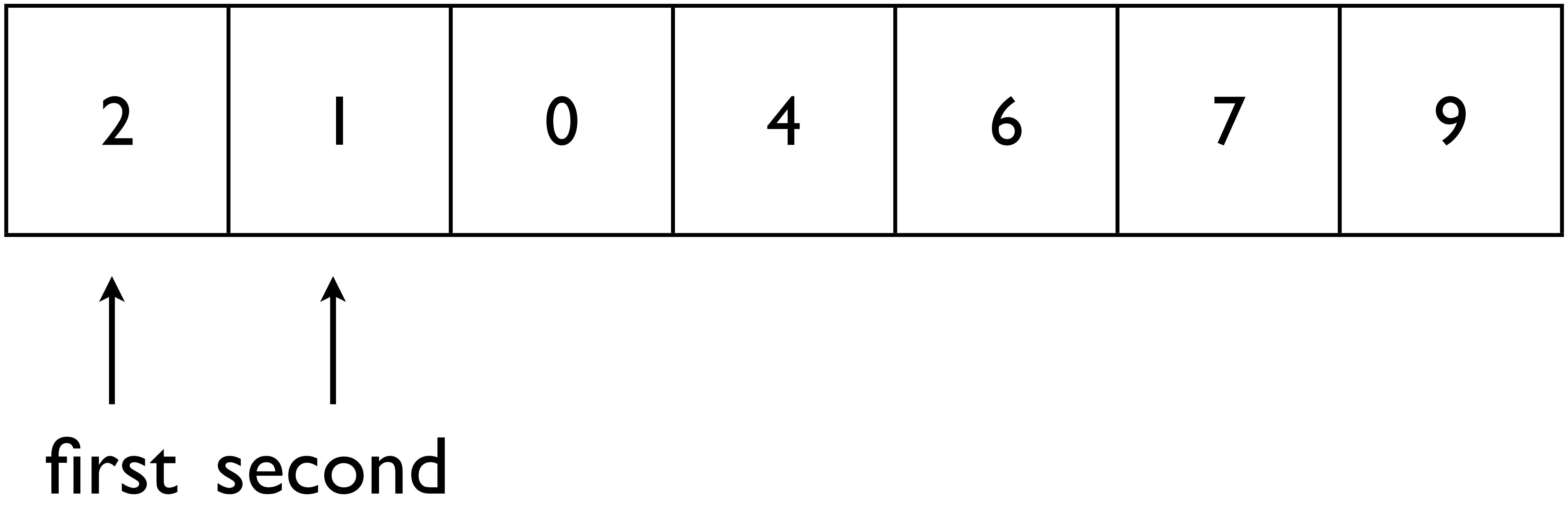


2	1	0	4	6	7	9

first second

Swap occurred?: True

Example

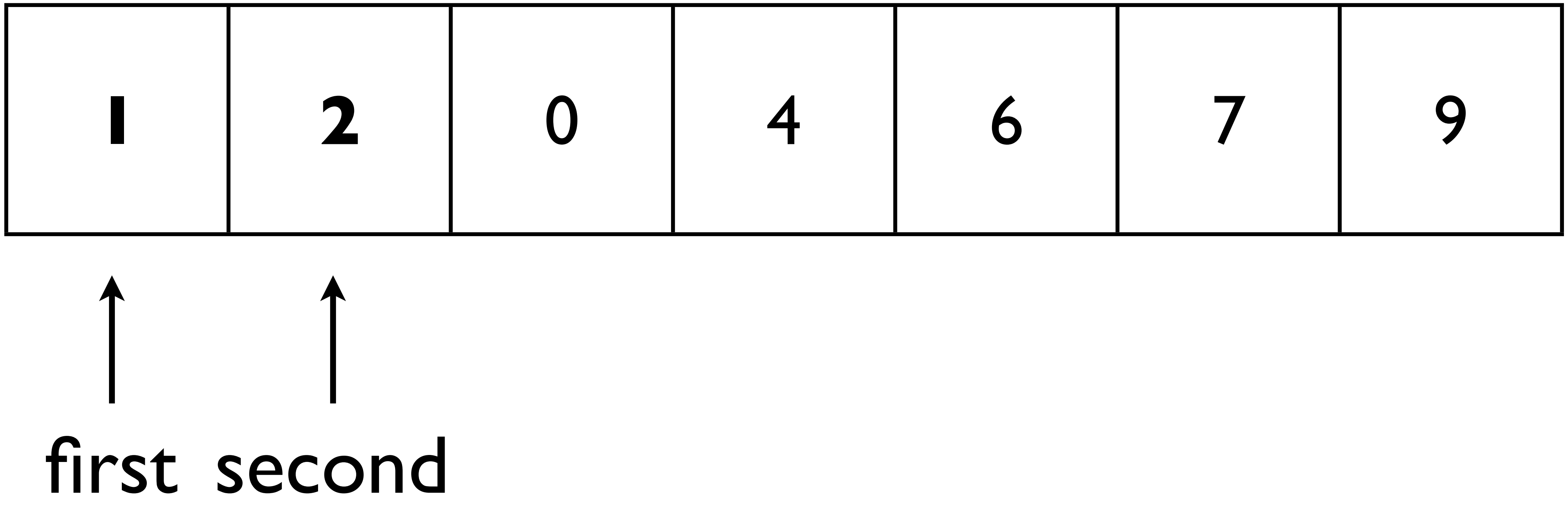


2	1	0	4	6	7	9

first second

Swap occurred?: **False**

Example

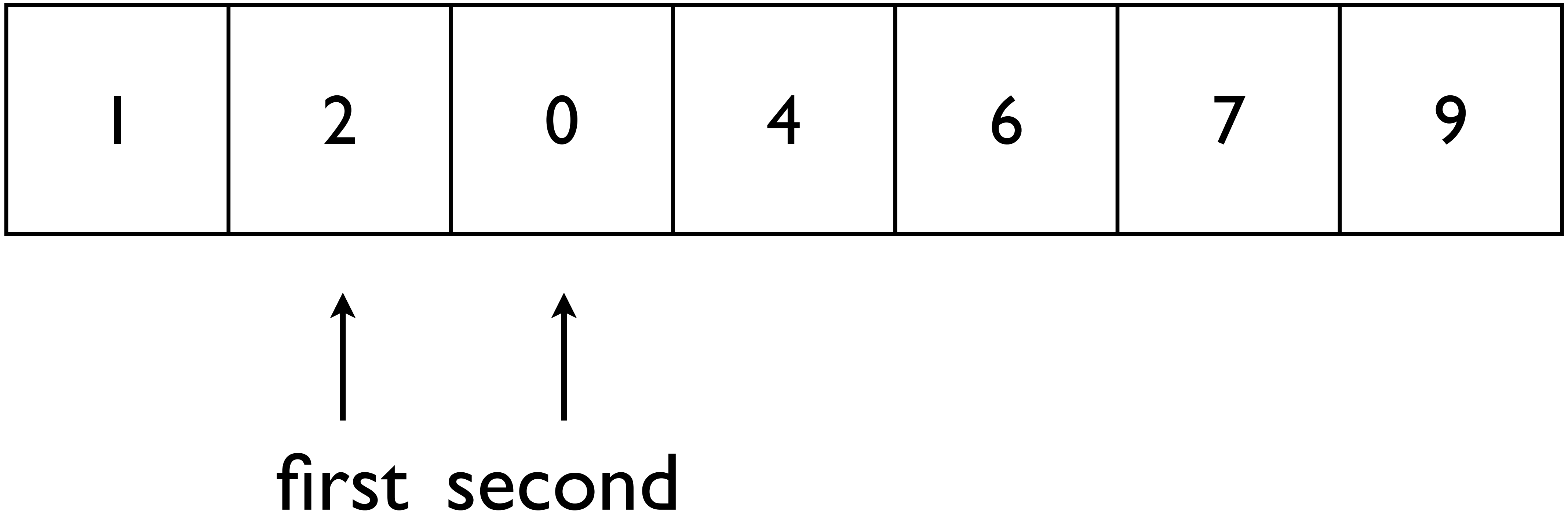


1	**2**	0	4	6	7	9

first second

Swap occurred?: **True**

Example

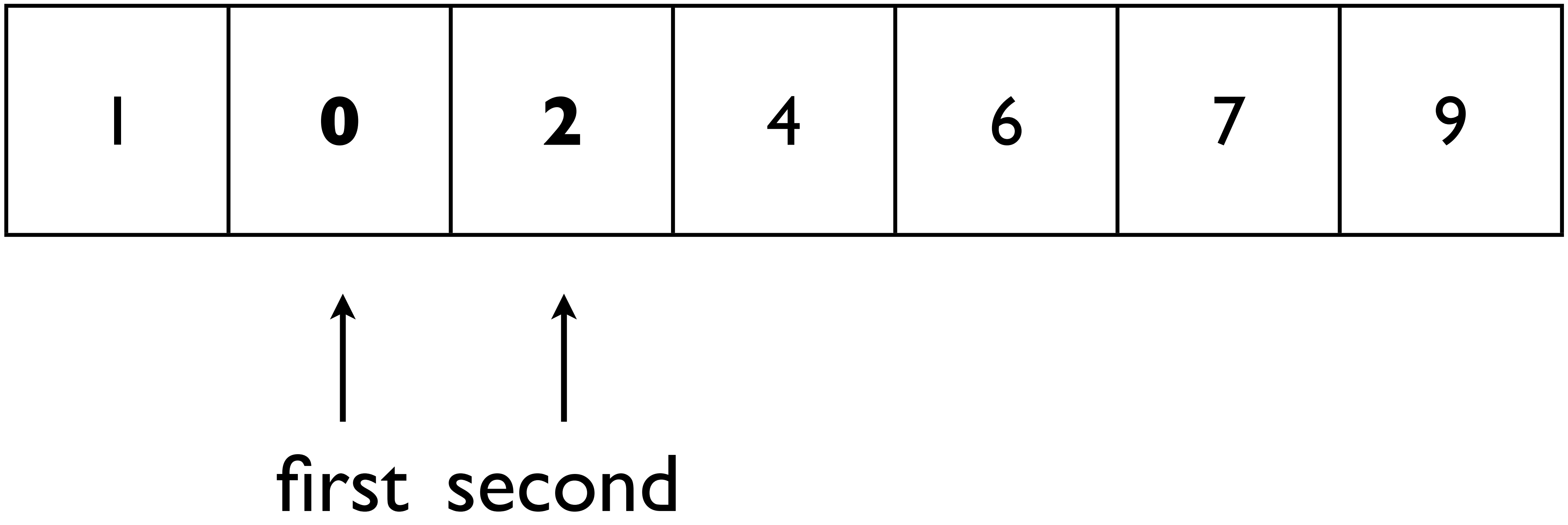


1	2	0	4	6	7	9

first second

Swap occurred?: True

Example

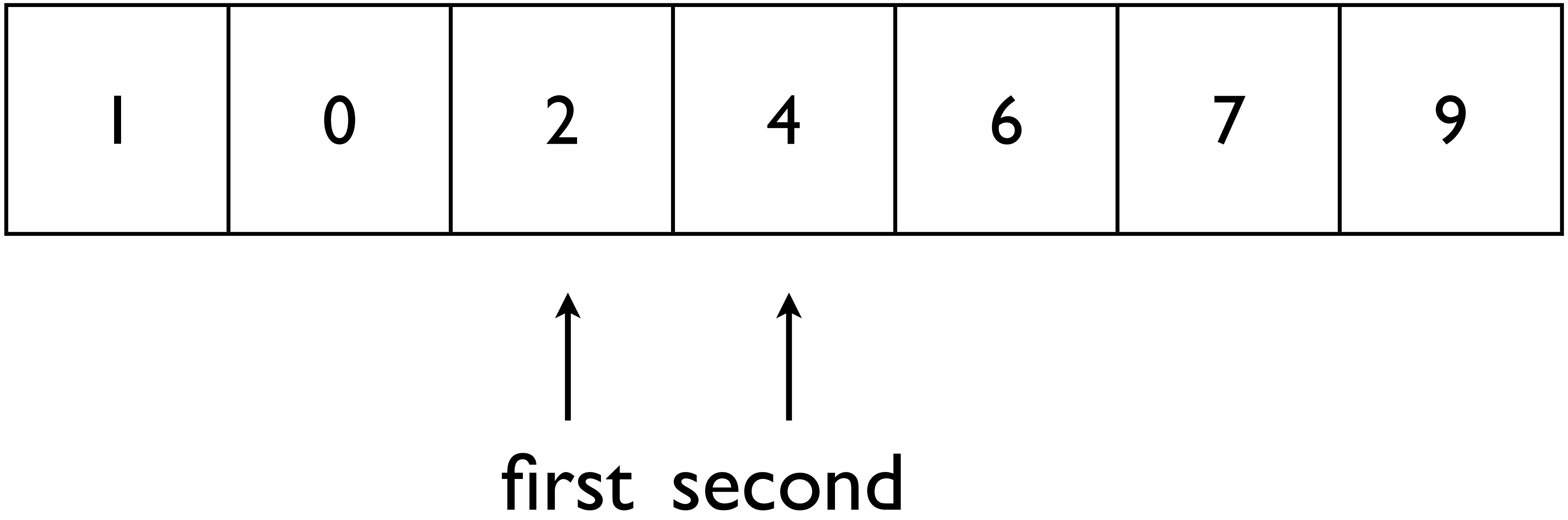


1	**0**	**2**	4	6	7	9

first second

Swap occurred?: True

Example

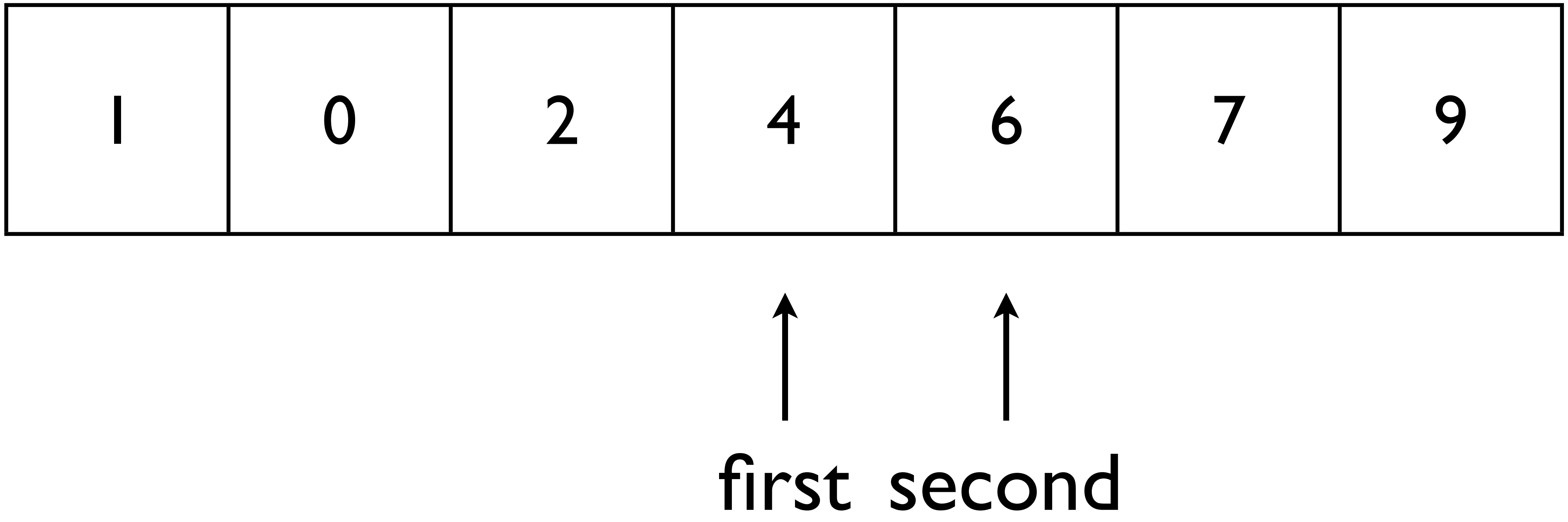


1	0	2	4	6	7	9

first second

Swap occurred?: True

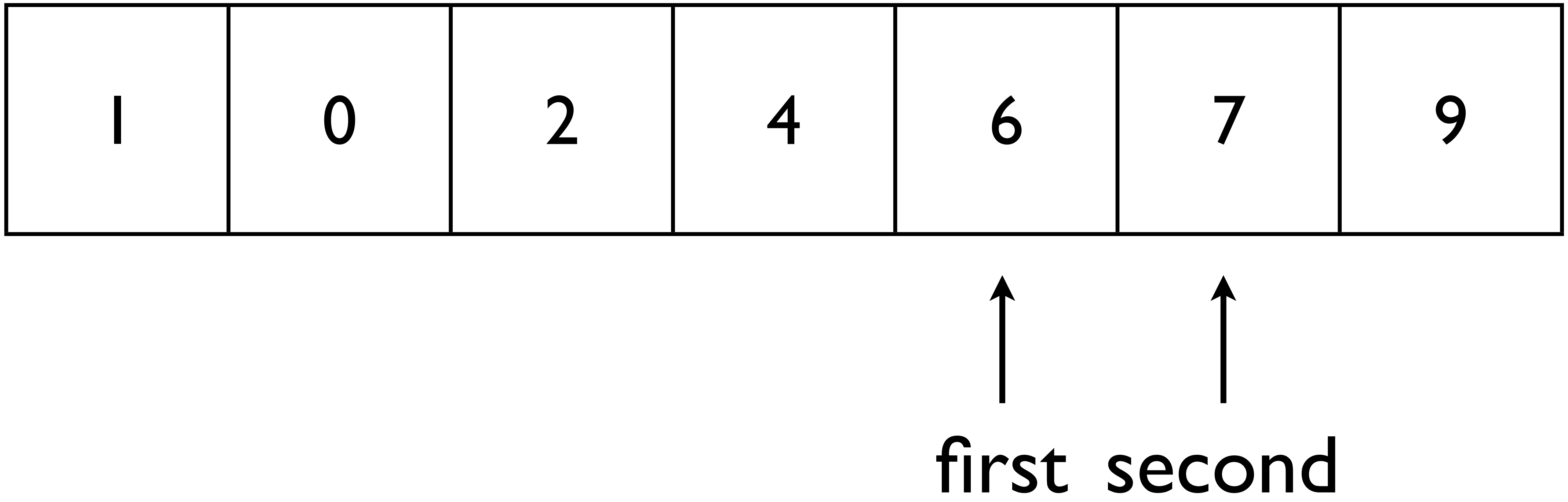
Example



1	0	2	4	6	7	9

Swap occurred?: True

Example

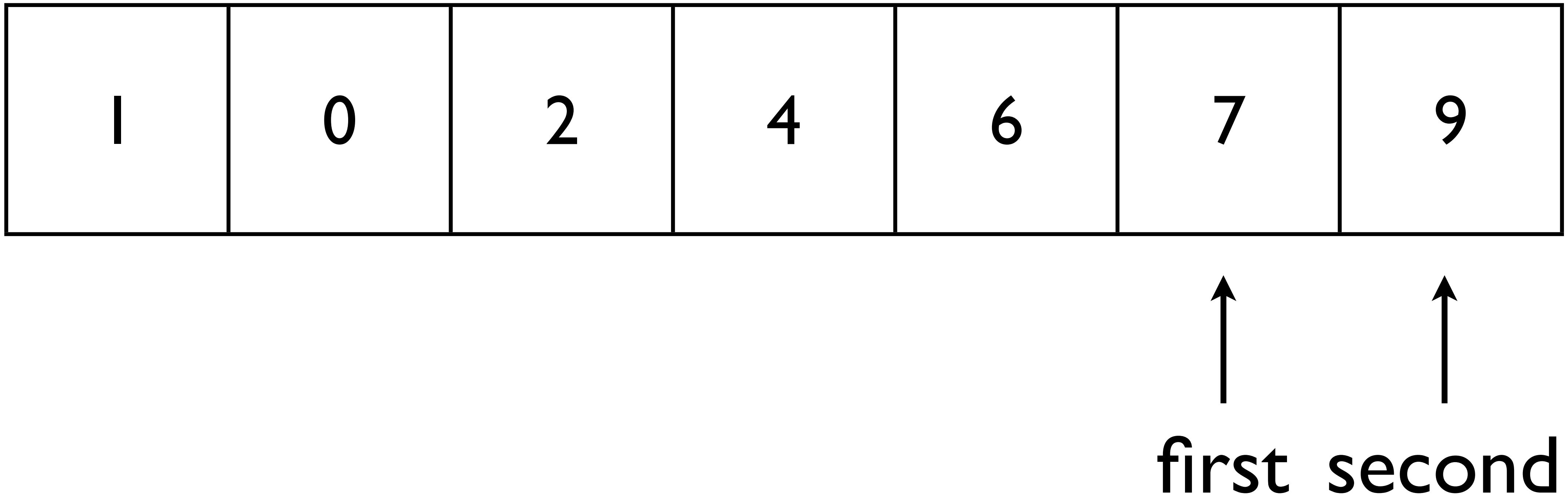


1	0	2	4	6	7	9

first second

Swap occurred?: True

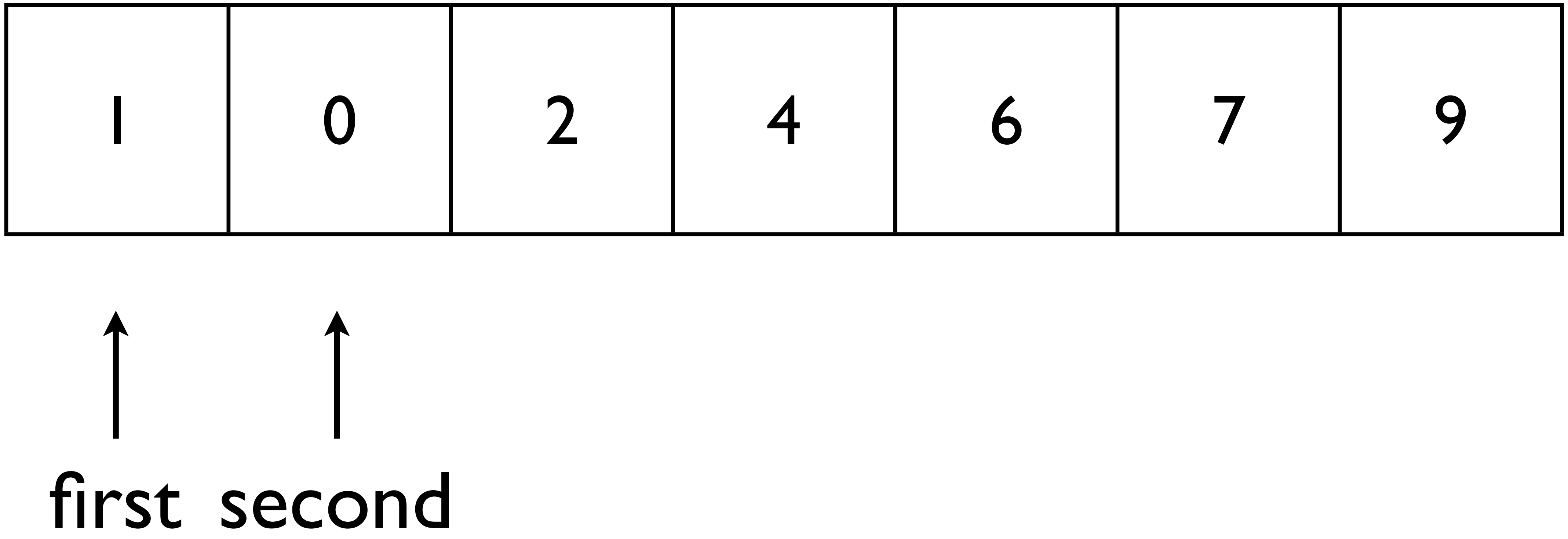
Example



1	0	2	4	6	7	9

Swap occurred?: True

Example



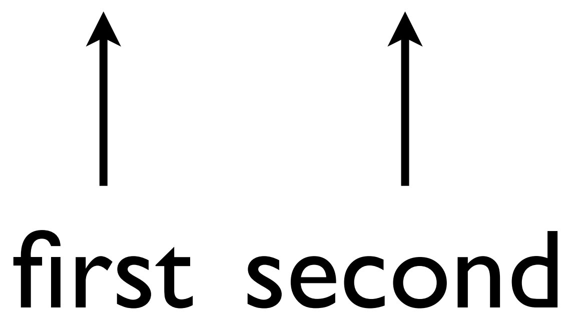
1	0	2	4	6	7	9

first second

Swap occurred?: **False**

Example

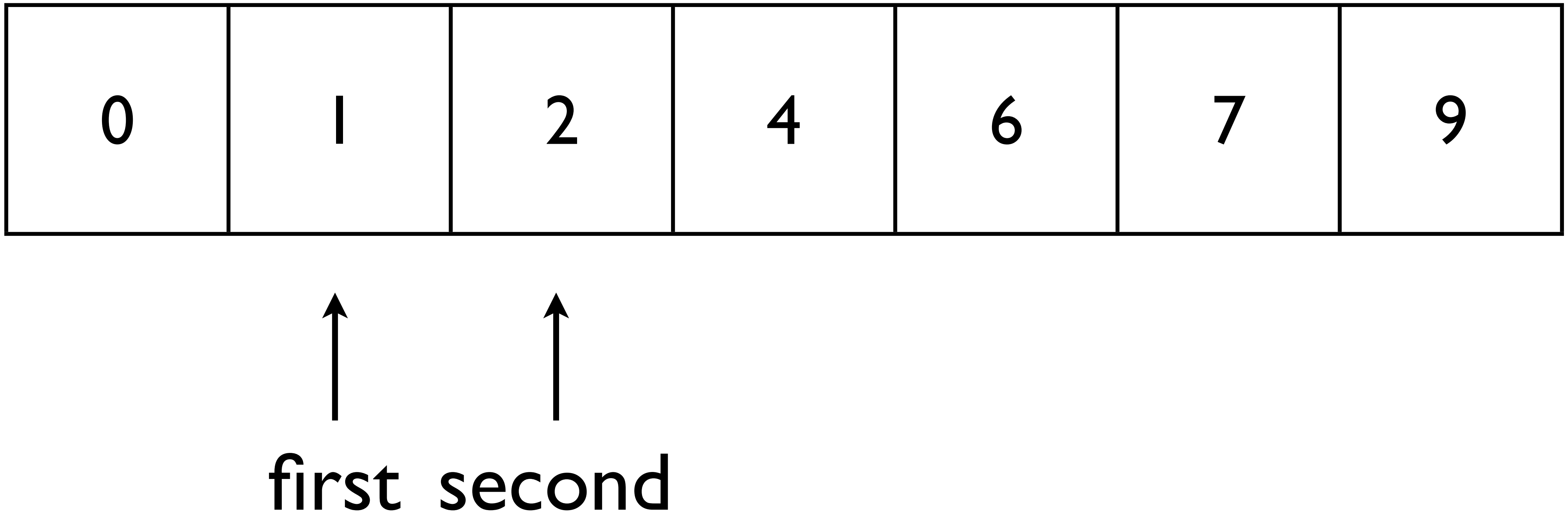
0	**1**	2	4	6	7	9



first second

Swap occurred?:**True**

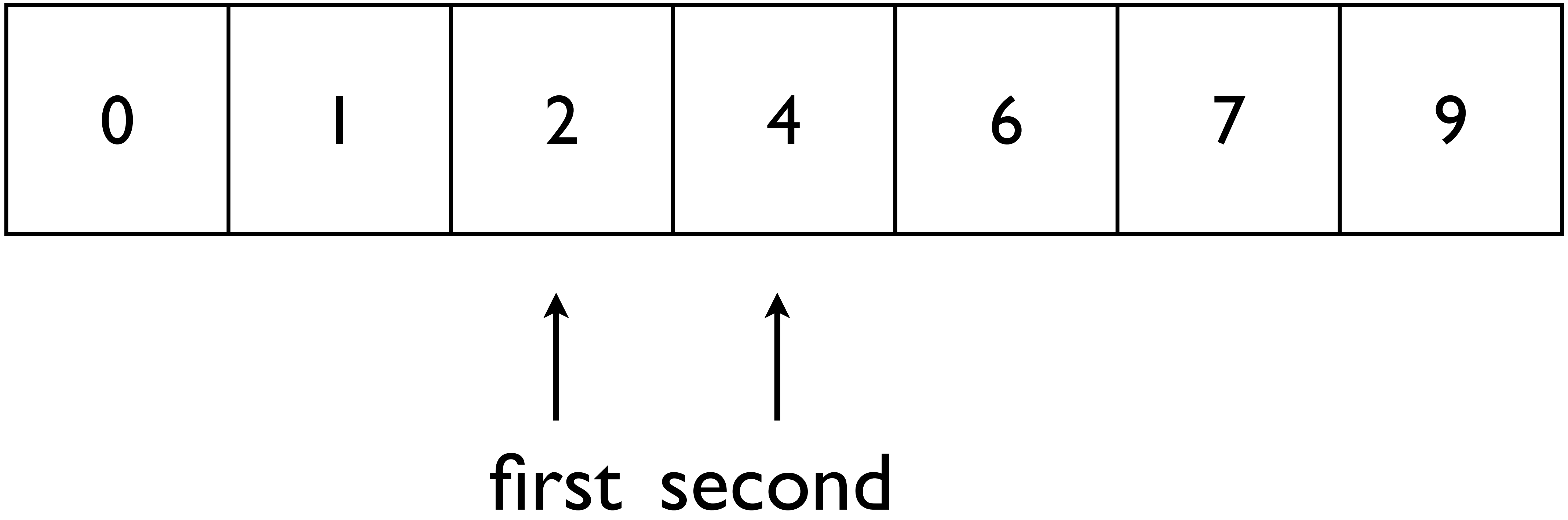
Example



0	1	2	4	6	7	9

Swap occurred?: True

Example

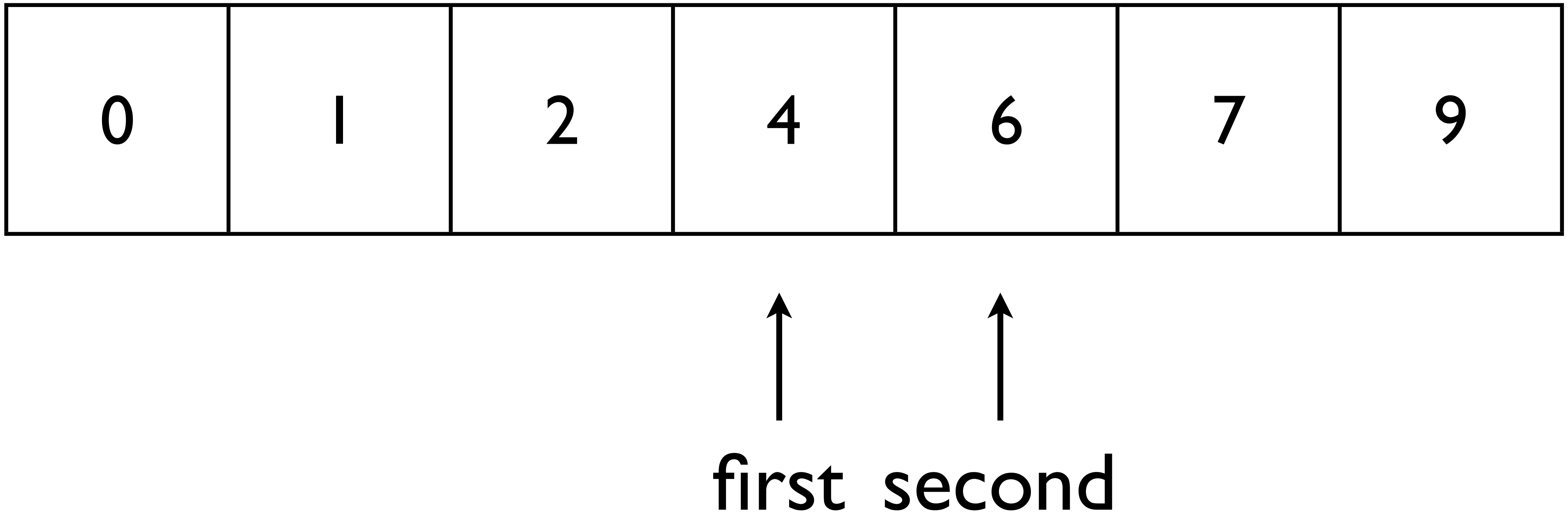


0	1	2	4	6	7	9

first second

Swap occurred?: True

Example

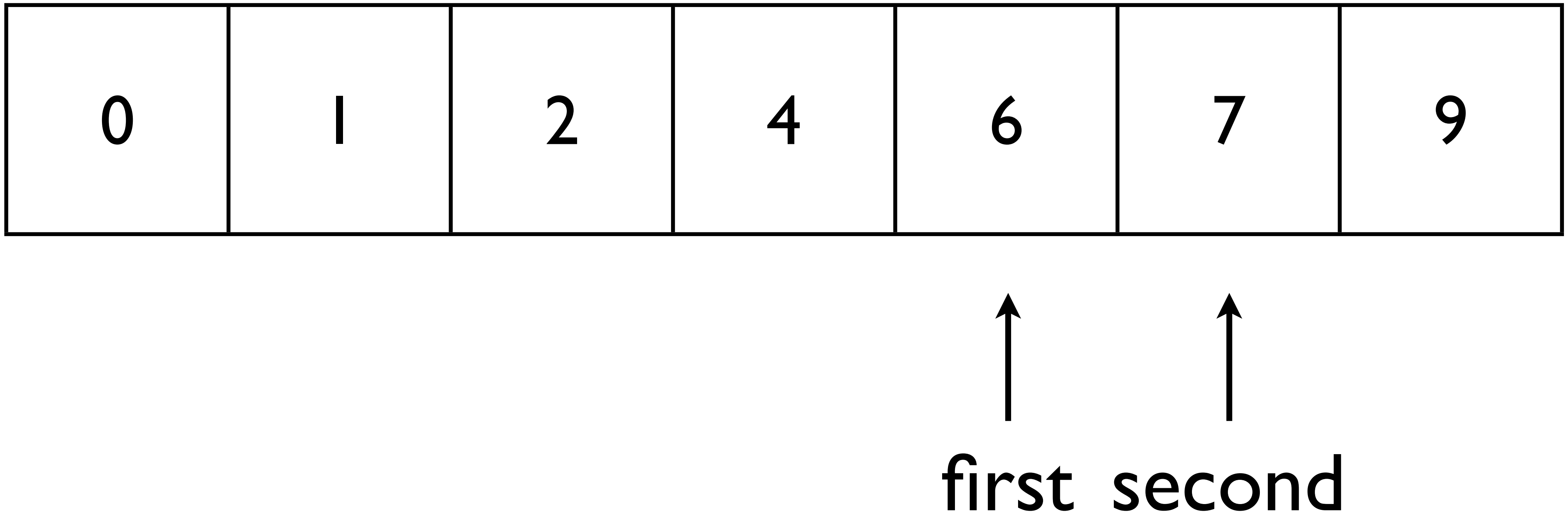


0	1	2	4	6	7	9

first second

Swap occurred?: True

Example

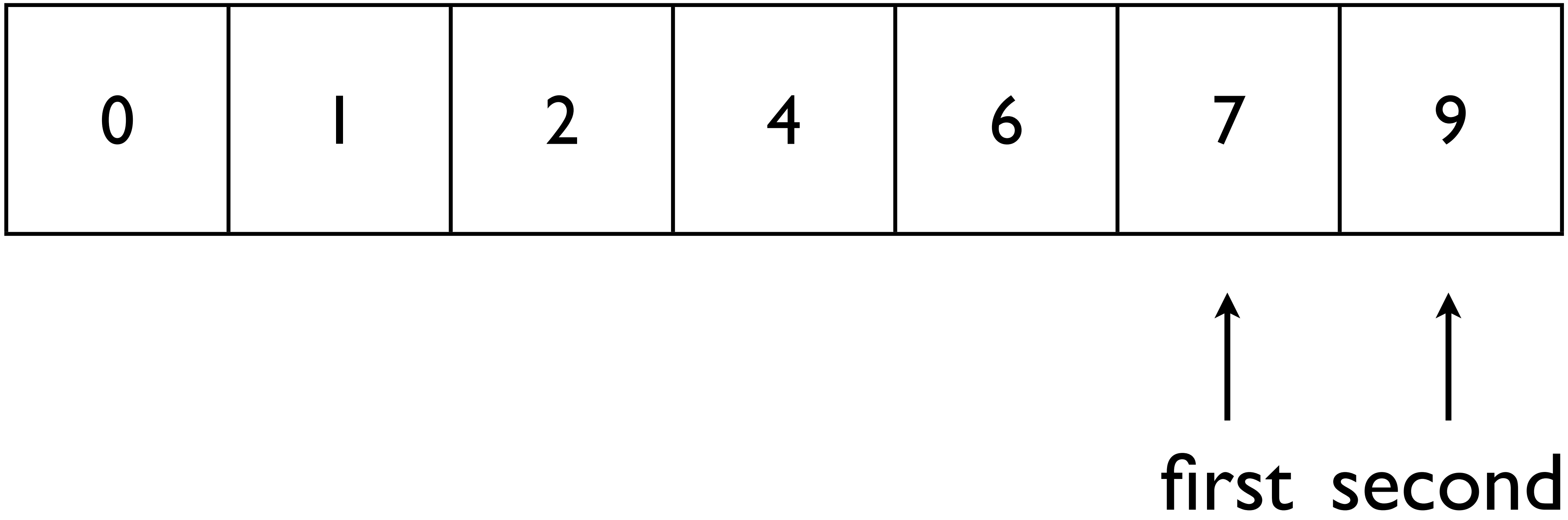


0	1	2	4	6	7	9

first second

Swap occurred?: True

Example

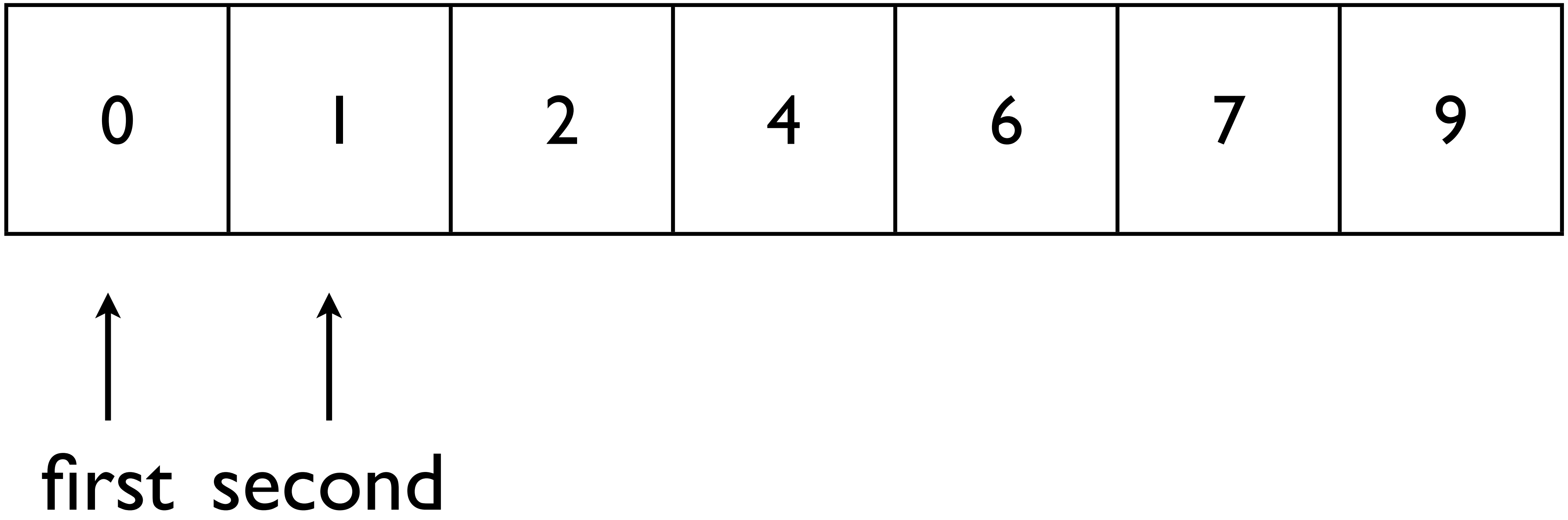


0	1	2	4	6	7	9

first second

Swap occurred?: True

Example

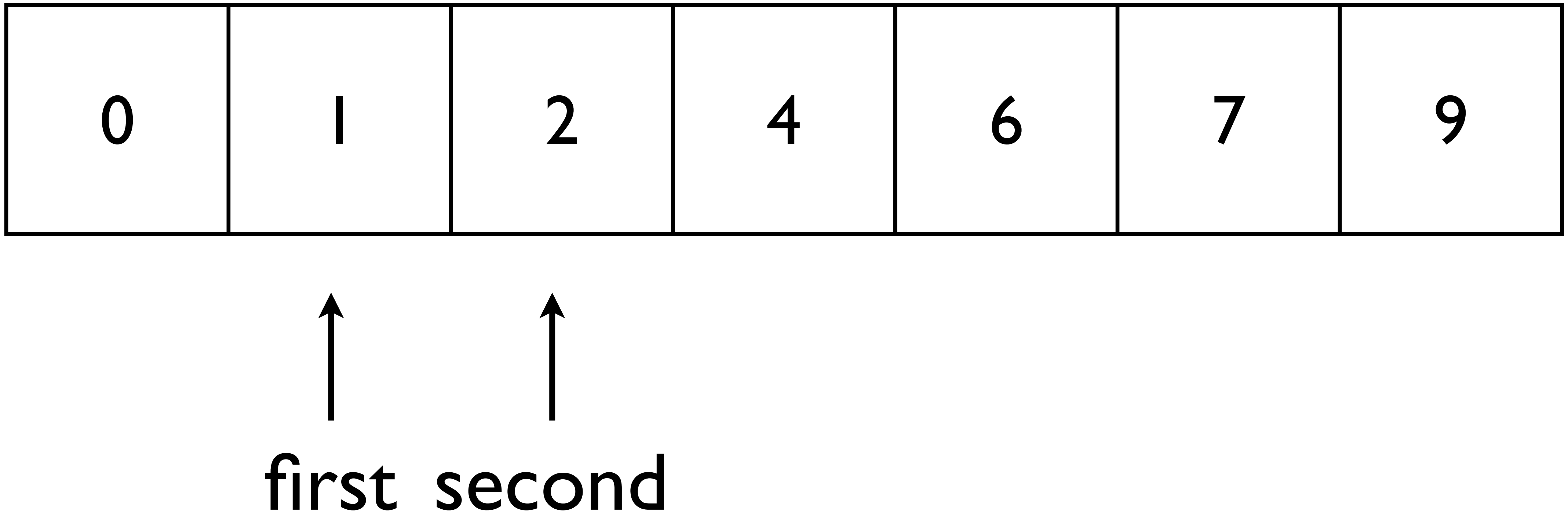


0	1	2	4	6	7	9

first second

Swap occurred?:**False**

Example

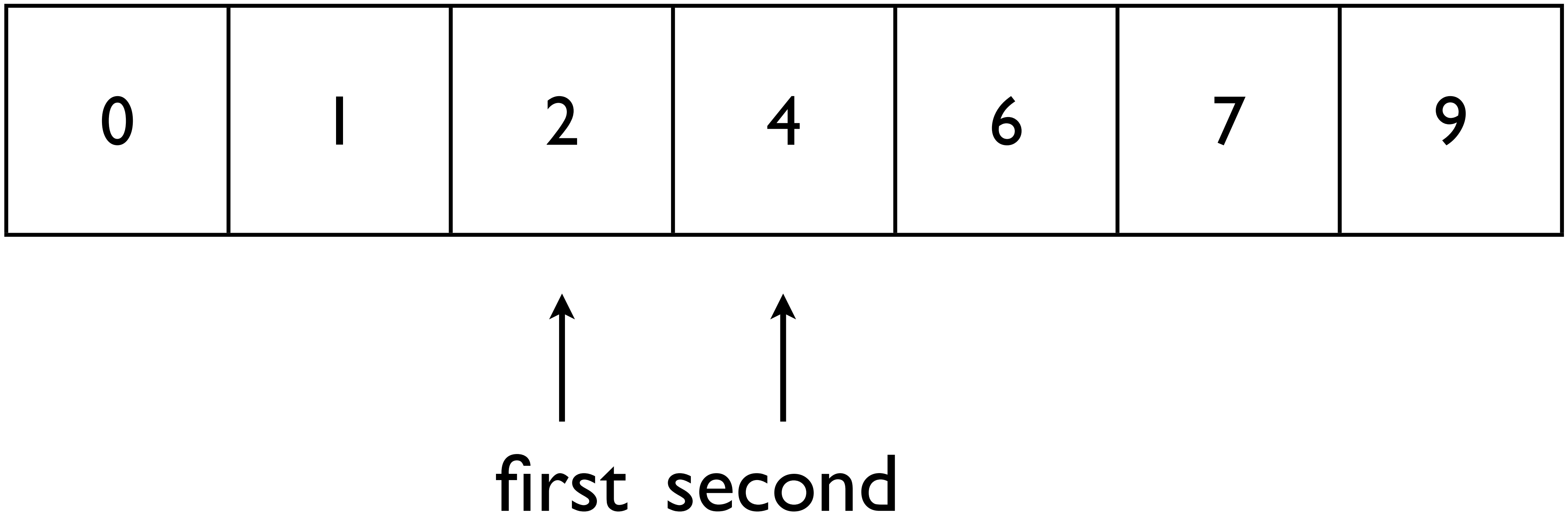


0	1	2	4	6	7	9

first second

Swap occurred?:False

Example

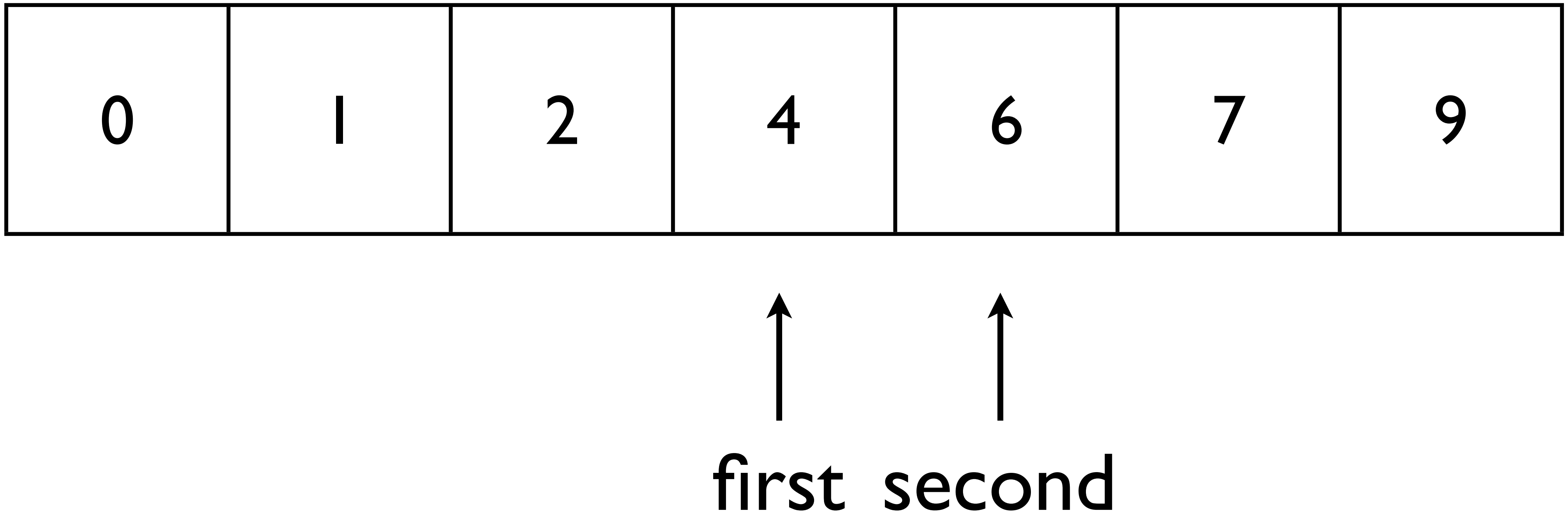


0	1	2	4	6	7	9

first second

Swap occurred?:False

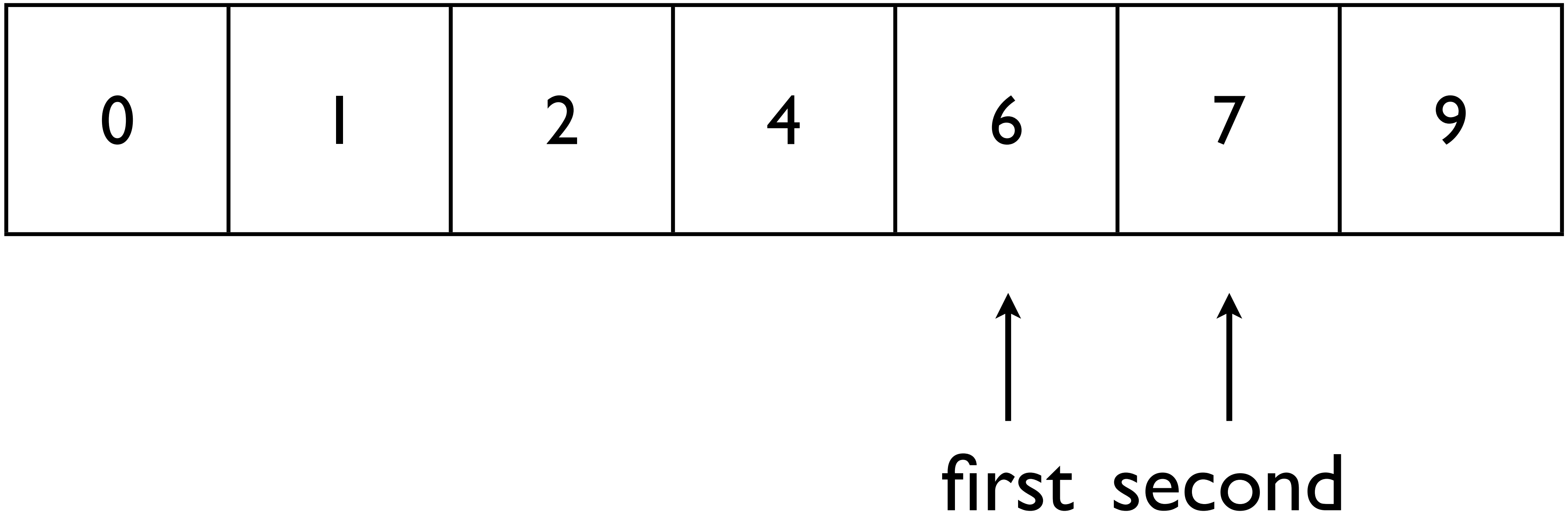
Example



0	1	2	4	6	7	9

Swap occurred?:False

Example

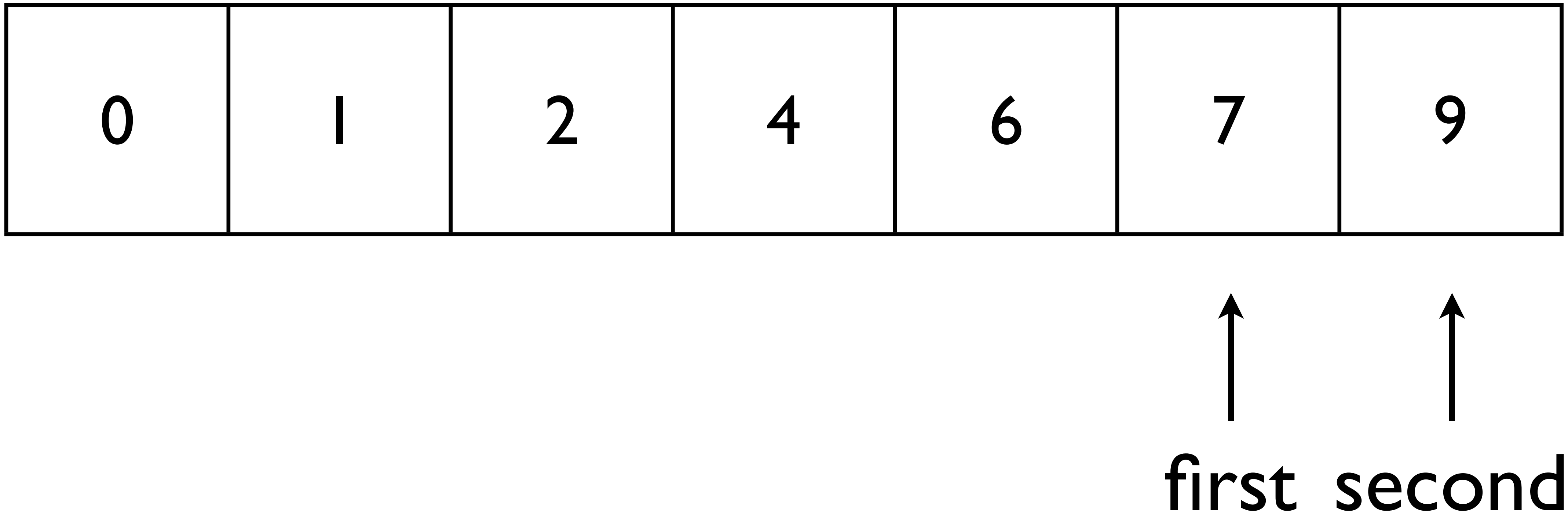


0	1	2	4	6	7	9

first second

Swap occurred?:False

Example



0	1	2	4	6	7	9

Swap occurred?:False

Code