CS1010E Lecture #08

Searching & Sorting

Bring order to the world

NUS National University of Singapore
Department of Computer Science
School of Computing

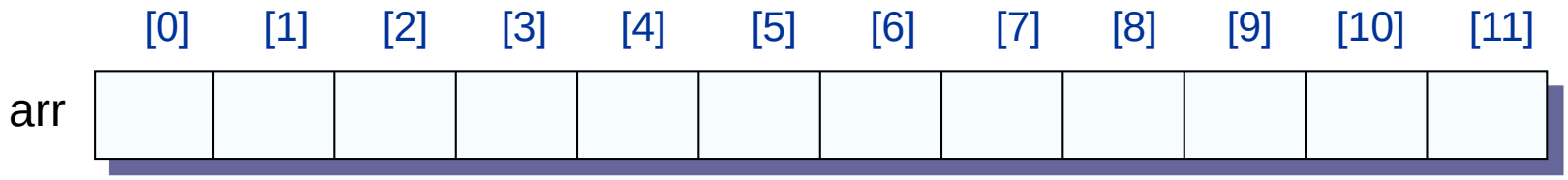
Arrays

PREVIOUS LECTURE

- Array is an indexed collection of homogeneous data.
  - Array index starts from 0.

- Example:

```
double arr[12];
```



  - **arr[0]**, **arr[1]**, **arr[2]**…**arr[11]** are all double variables.

Initializing Array

PREVIOUS LECTURE

- Method one: use a loop

```
int cand[30];
for (i = 0; i < 30; i++) { // init array
  cand[i] = 0;
}
```

- Method two: use initializer
  - Must define and initialize simultaneously

```
int a[3] = {54, 9, 10}; // full initialization
// a[0]=54, a[1]=9, a[2]=10

int b[] = {1, 2, 3}; // skip array size
// size of b is 3 with b[0]=1, b[1]=2, b[2]=3

int c[5] = {17, 3, 10}; // partial initialization
// c[0]=17, c[1]=3, c[2]=10, c[3]=0, c[4]=0
```

Passing Array to Function

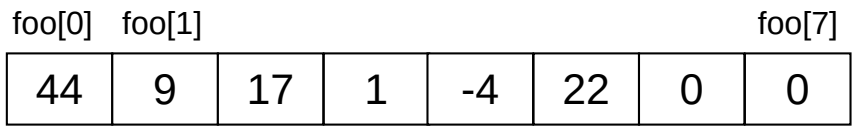
PREVIOUS LECTURE

```
// ...
int main(void) {
  int foo[8] = {44, 9, 17, 1, -4, 22};
  printf("%d\n", sumArray(foo, 4) );
  return 0;
}

int sum_array(int arr[], int size) {
  int i, total = 0;
  for (i = 0; i < size; i++) {
    total += arr[i];
  }
  return total;
}
```

Caller just write array name

Rename given array to **arr**

In main():



In sum_array():

Common Error: Index out of Bound

PREVIOUS LECTURE

- Could result in weird output or even segmentation fault.

```c
#include <stdio.h>

int sum_array(int arr[], int size);

int main(void) {
  int foo[8] = {44, 9, 17, 1, -4, 22};
  printf("sum is %d\n", sum_array(foo, 8));
  printf("sum is %d\n", sum_array(foo, 3));
  return 0;
}

int sum_array(int arr[], int size) {
  int i, total = 0;
  for (i = 0; i < size; i++) {
    total += arr[i];
  }
  return total;
}
```

sum all 8 elements

sum first 3 elements

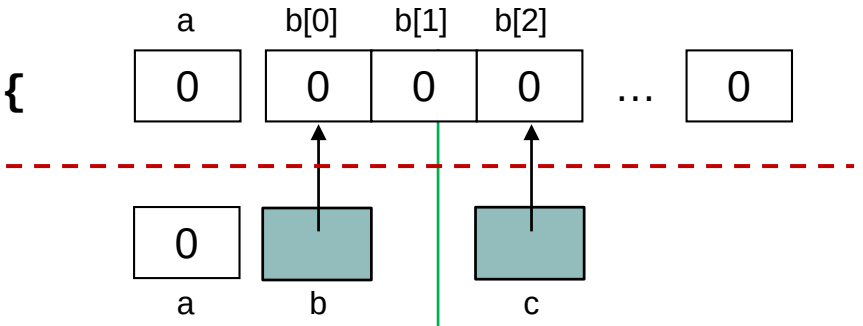
Values must not exceed actual array size!

Quiz

(CS1101C AY2008/09 Semester 2 Exam, Q17)

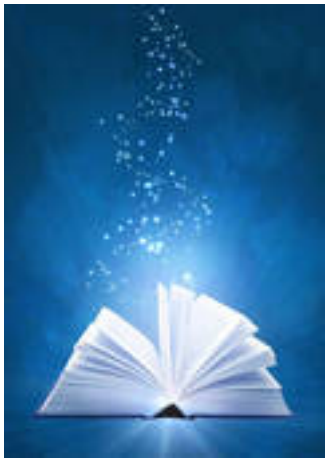
- Write down the output of the following program.

```
#include <stdio.h>
#define N 9
void func(int a, int b[], int *c);

int main(void) {
  int a = 0, b[N] = {0};
  func(a, b, &b[2]);
  printf("%d %d %d %d\n", a, b[0], b[1], b[2]);
  return 0;
}

void func(int a, int b[], int *c) {
  a = 1;
  b[1] = 2;
  *c = 3;
}
```



Learning Objectives

- At the end of this lecture, you should understand:
  - The basic searching and sorting algorithms.
  - How to implement searching and sorting algorithms using arrays.



Introduction

- Today, we will study some simple yet useful classical algorithms which find their place in many CS applications.
  - Searching for some data amid very large collection of data
  - Sorting very large collection of data according to some order

- We will begin with an algorithm (idea), then show how the algorithm is transformed into a C program (implementation).
  - This brings back (reminds you) our very first lecture:

    the importance of beginning with an algorithm

Searching

- Searching is a common task that we carry out without much thought everyday.
  - Searching for a location in a map.
  - Searching for the contact of a particular person.
  - Searching for a nice picture for your project report.
  - etc.

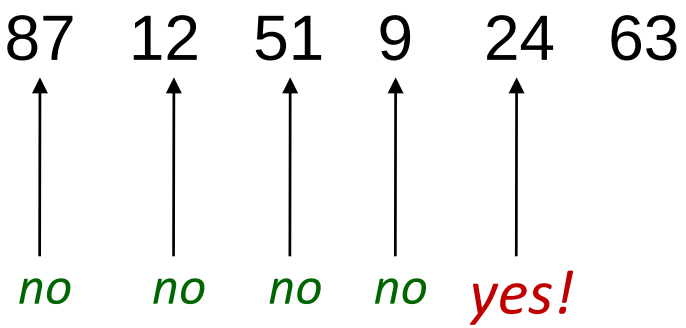
- In this lecture, you will learn how to search for an item (sometimes called a search key) in an array.
  - Problem statement:

    Given an array of data and a search key *X*, return the position of *X* in the array if it exists.

Linear Search : Algorithm

- Also known as Sequential Search.
- Idea: Search the array from one end to the other end in linear progression.
- Example:

Search for key 24 in this array:



Q: What to report if key is not found?

Linear Search : Program

```
// Search for key in arr using linear search.
// Return index if found, or -1 otherwise.
int linear_search(int arr[], int size, int key) {

  int i;
  for (i = 0; i < size; i++) {
    if (key == arr[i]) {
      return i;
    }
  }

  return -1;  // not found
}
```

Linear Search : Performance Analysis

- We use the number of comparisons algorithms make to analyze their performance.
  - The ideal searching algorithm will make the least possible number of comparisons to locate the desired data.
  - This topic is called analysis of algorithms, which will be formally introduced in CS1020.
- For an array with *n* elements:

Q: What is the maximum possible number of comparisons made?

n comparisons

Q: Under what circumstances do we encounter such a worst case?

(a) Key not found
(b) Found at last position

Binary Search

- You are going to witness a radically different approach, one that has become the basis of many well-known algorithms.
- The idea is simple and fantastic, but applied on the searching problem, it has this pre-condition that the array must be sorted beforehand.
- How the data is organized (in this case, sorted) usually affects how we choose/design an algorithm to access them.
- In other words, sometimes (actually, very often) we seek out new way to organize the data so that we can process them more efficiently.

Binary Search : Algorithm

(Pre-condition: list is sorted in ascending order)

- Algorithm
  - Look for the *key* in the middle position of the list.

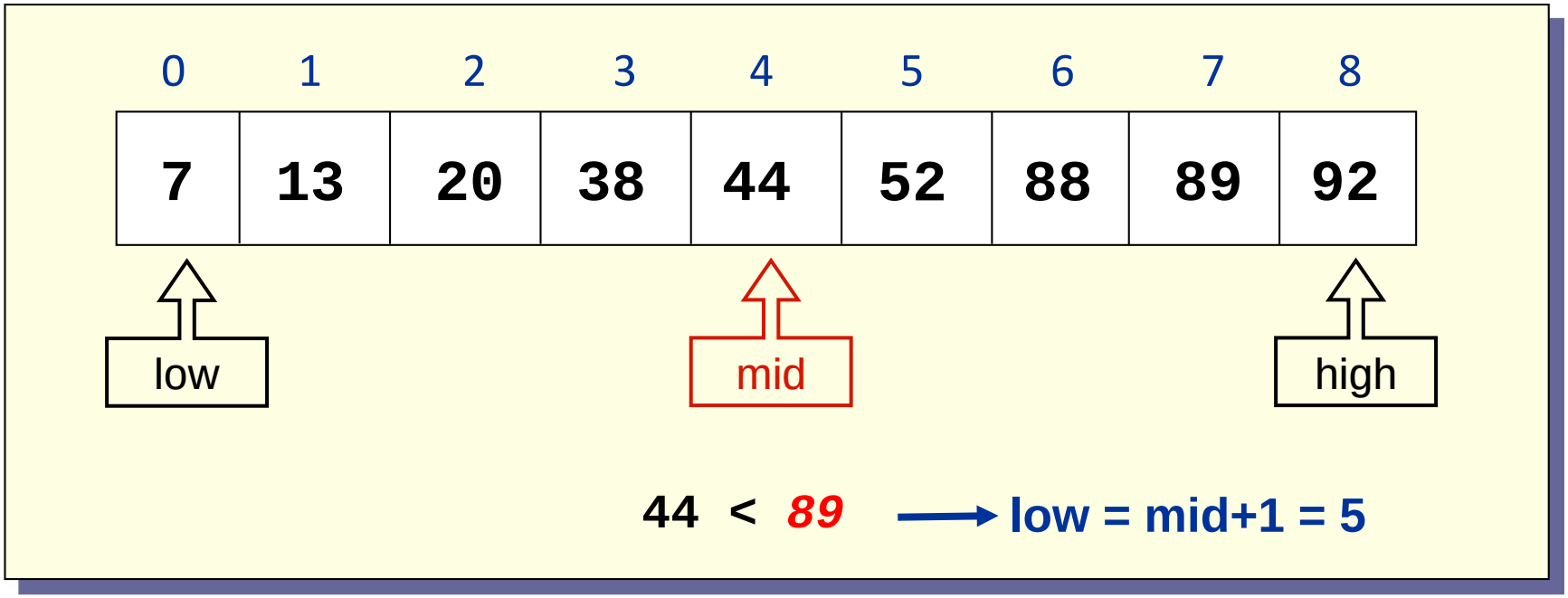
    Either of the following 2 cases happens:
    - If the *key* is **smaller** than the middle element, then "discard" the right half of the list and repeat the process.
    - If the *key* is **greater** than the middle element, then "discard" the left half of the list and repeat the process.
  - Terminating condition: either the *key* is found, or when all elements have been "discarded".

Sequence of Successful Search (1/3)

	low	high	mid
#1	0	8	4

search (*89*)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

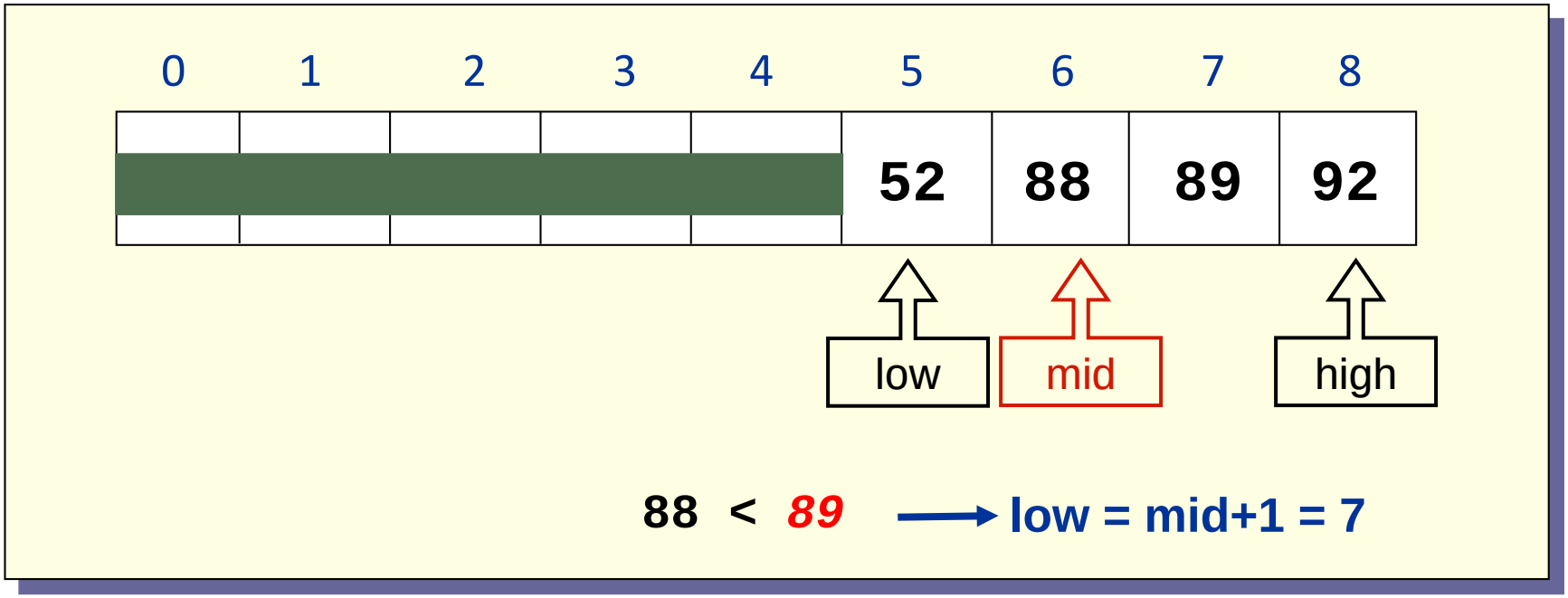


Sequence of Successful Search (2/3)

	low	high	mid
#1	0	8	4
#2	5	8	6

search (*89*)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

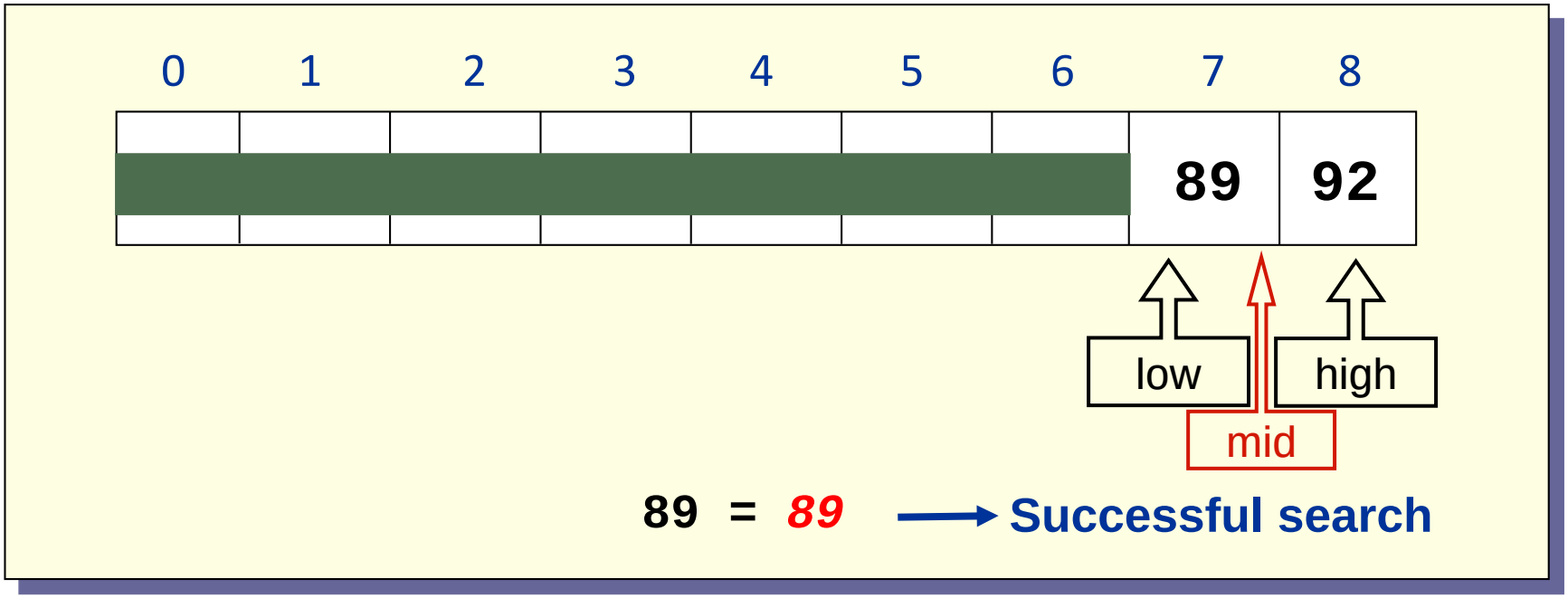


Sequence of Successful Search (3/3)

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	7	8	7

search (*89*)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

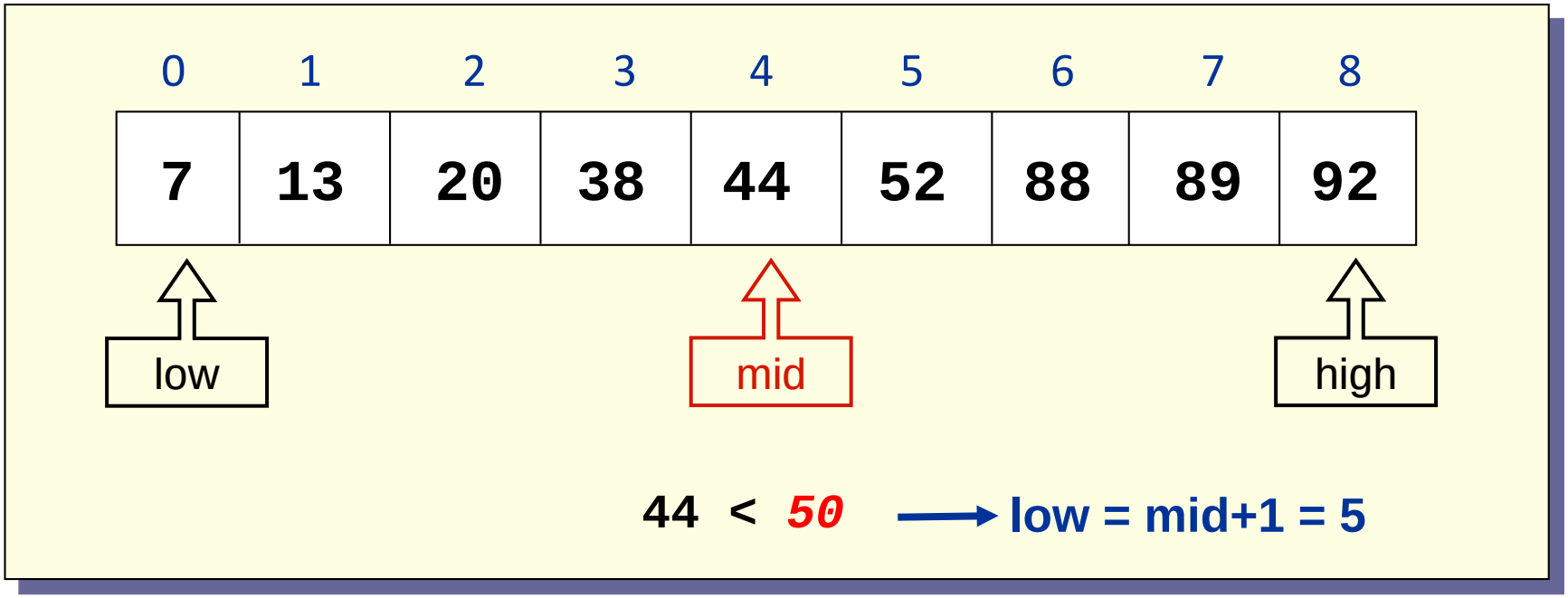


Sequence of Unsuccessful Search (1/4)

	low	high	mid
#1	0	8	4

search (*50*)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

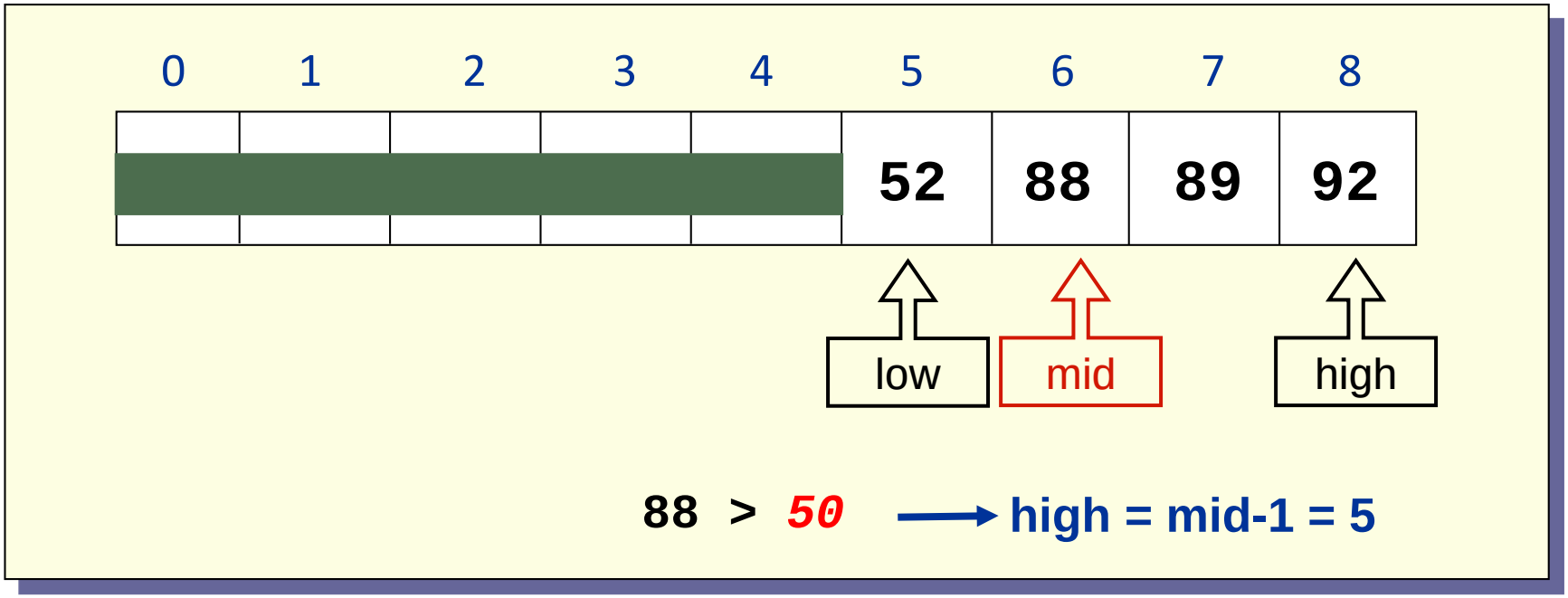


Sequence of Unsuccessful Search (2/4)

	low	high	mid
#1	0	8	4
#2	5	8	6

search (*50*)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

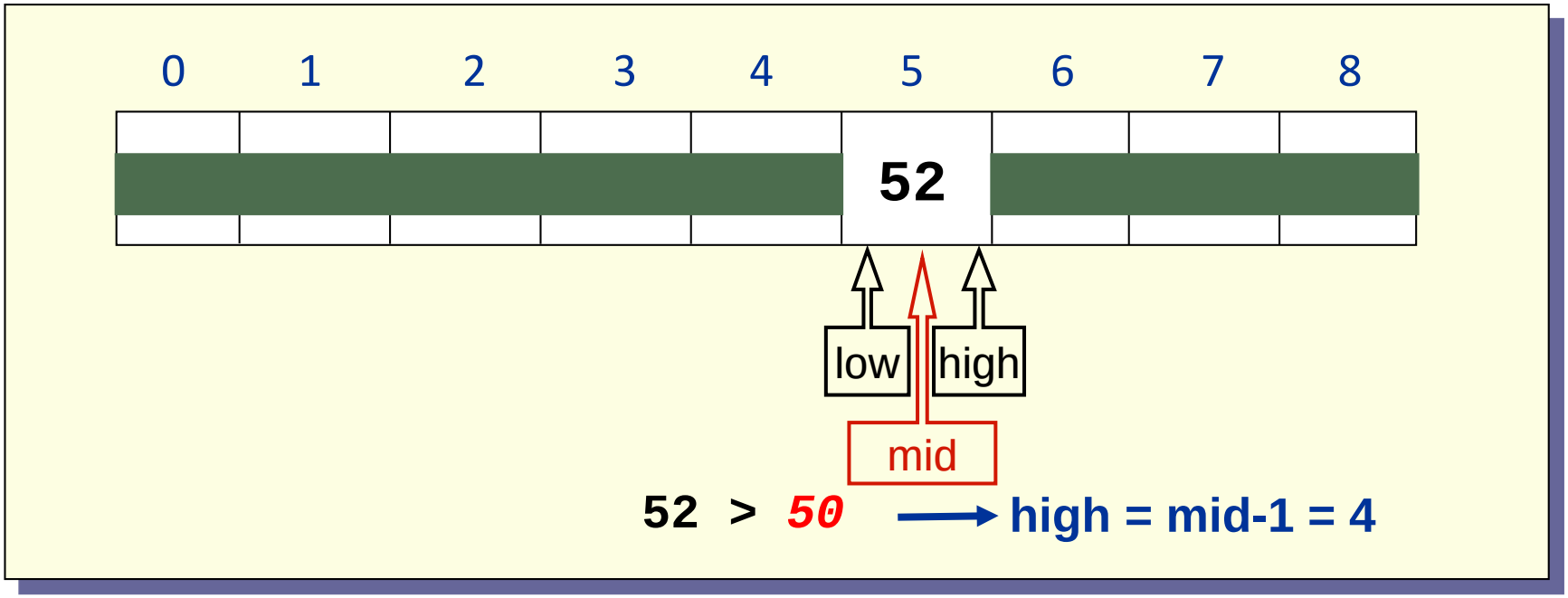


Sequence of Unsuccessful Search (3/4)

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5

search (*50*)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$

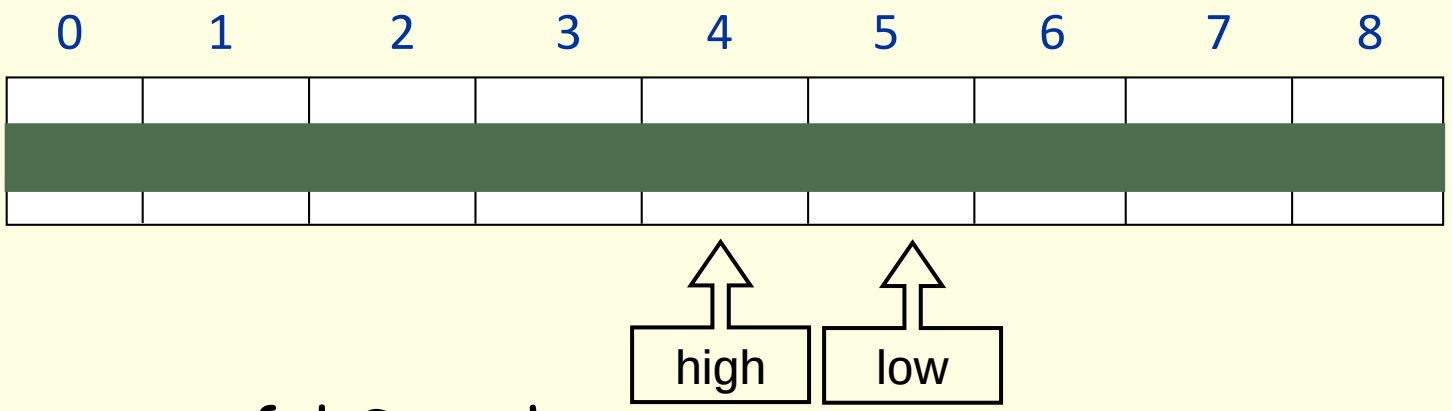


Sequence of Unsuccessful Search (4/4)

	low	high	mid
#1	0	8	4
#2	5	8	6
#3	5	5	5
#4	5	4	

search (*50*)

$$mid = \left\lfloor \frac{low + high}{2} \right\rfloor$$



Unsuccessful Search

`low > high` **no more to search**

Binary Search : Program

```
// Search for key in sorted 'arr' using binary search
// Return index if found; return -1 otherwise
int binary_search(int arr[], int size, int key) {

  int low = 0, high = size-1, mid;

  while (low <= high) {
    mid = (low+high) / 2;
    if (key == arr[mid]) {
      return mid;
    } else if (key < arr[mid]) {
      high = mid - 1;
    } else {
      low = mid + 1;
    }
  }

  return -1;
}
```

Binary Search : Analysis

- In binary search, each step eliminates the problem size (array size) by half.
  - The problem size gets reduced to 1 very quickly (see next slide).
- This is a simple yet powerful strategy, of halving the solution space in each step.
- Such strategy, a special case of divide-and-conquer paradigm, can be naturally implemented using recursion (a topic to be covered later).

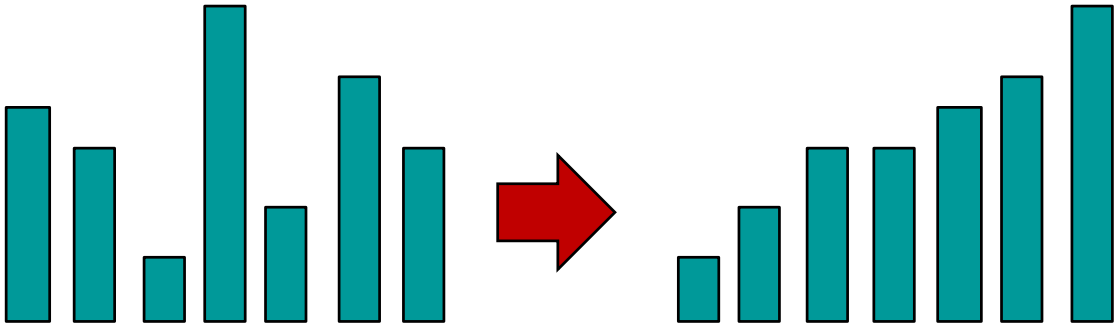
Binary Search : Performance

- Worst-case analysis:

Array size n	Linear Search (n comparisons)	Binary Search ($\log_2 n$ comparisons)
100	100	≈7
1,000	1,000	≈10
10,000	10,000	≈14
100,000	100,000	≈17
1,000,000	1,000,000	≈20
10^9	10^9	≈30

Sorting

- Sorting is an process of arranging items in sequence.
- Sorting is important because once a set of items is sorted, many problems (such as searching) become easy.
  - e.g. determining whether the items in a set are all unique.



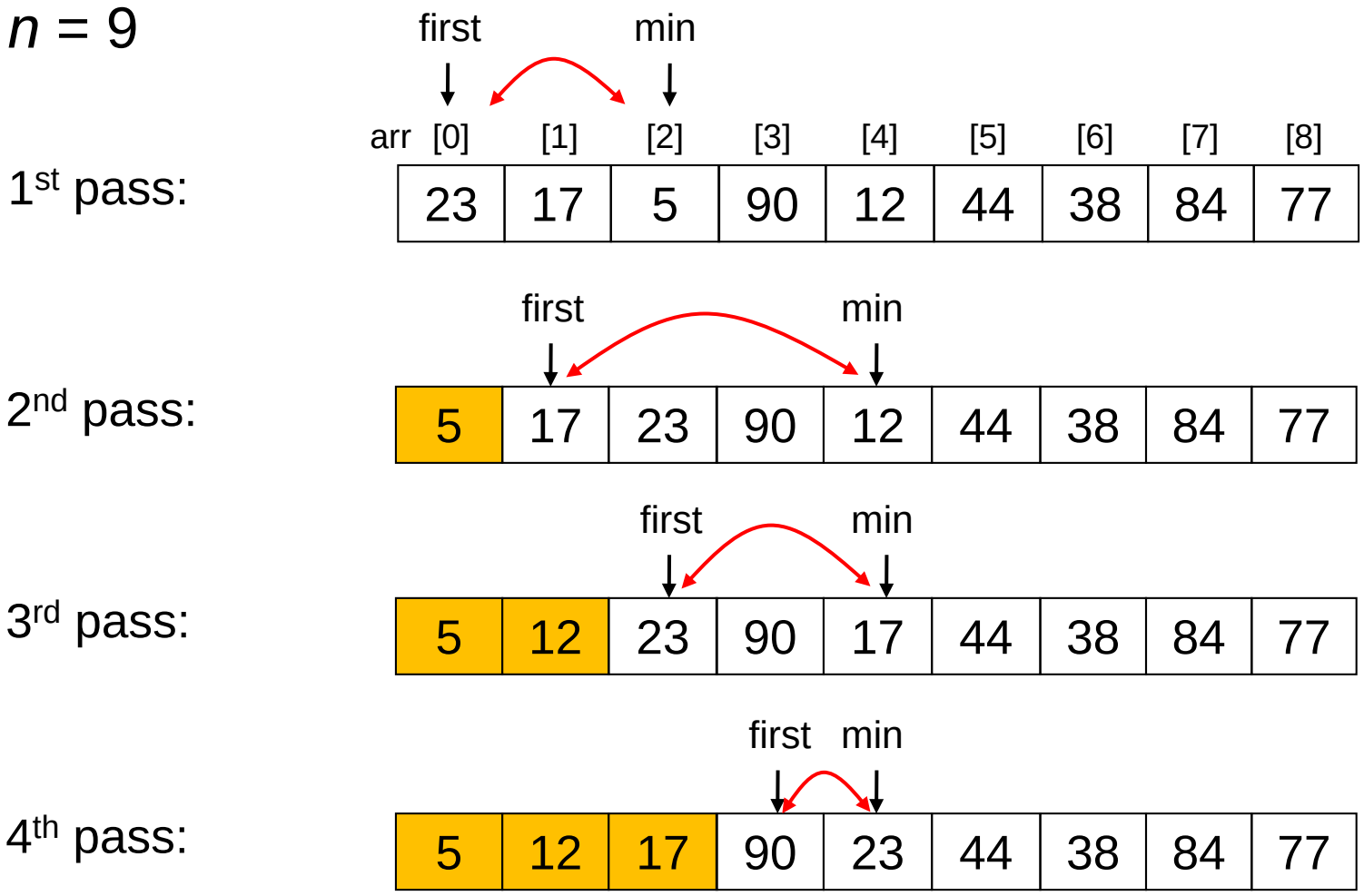
- Problem statement:

  Given an array of data, arrange all data in ascending order.

Selection Sort : Algorithm

- Algorithm
  - Step 1: Find the smallest element in the list.
  - Step 2: Swap this smallest element with the element in the first position. (Now, the smallest element is in the right place.)
  - Step 3: Repeat steps 1 and 2 with the list having one fewer element (i.e. the smallest element just found and placed is “exempted” from further processing).

Selection Sort : Illustration (1/2)



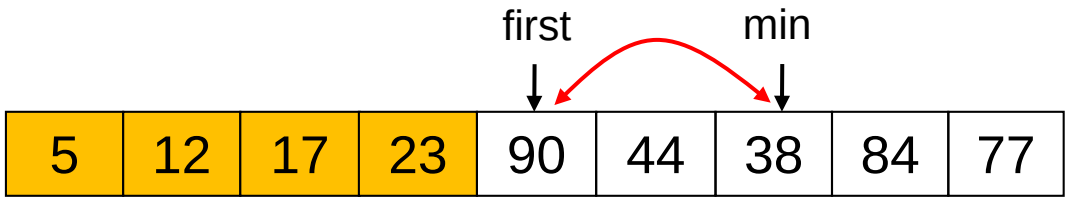
Selection Sort : Illustration (2/2)

Q: How many passes are needed for an array of *n* elements?

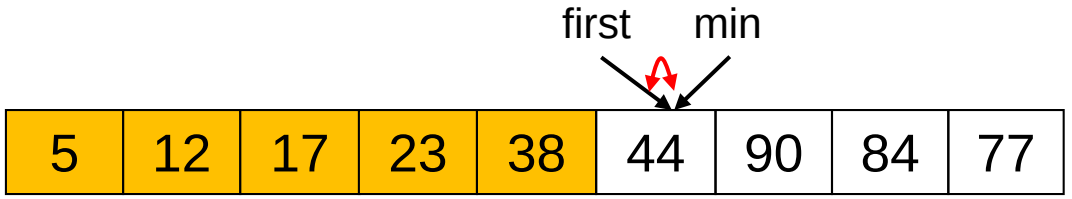
n-1

n = 9

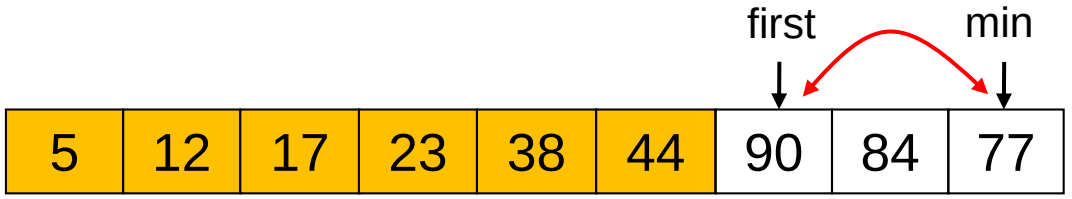
5th pass:



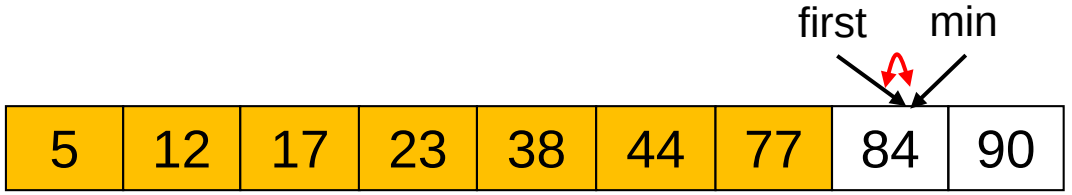
6th pass:



7th pass:



8th pass:



Final array:

Selection Sort : Program

```
// Sort array 'arr' in increasing order
void selection_sort(int arr[], int size) {

  int i, start_idx, min_idx, temp;

  for (start_idx = 0; start_idx < size-1; start_idx++) {
    // find the index of the minimum element
    min_idx = start_idx;
    for (i = start_idx+1; i < size; i++) {
      if (arr[i] < arr[min_idx]) {
        min_idx = i;
      }
    } // end inner for loop

    // swap minimum element with the element at start_idx
    temp = arr[start_idx];
    arr[start_idx] = arr[min_idx];
    arr[min_idx] = temp;
  } // end outer for loop
}
```

Selection Sort : Performance

Non-examinable

- We choose the number of comparisons as our basis of analysis.
- Comparisons of array elements occur in the inner loop, where the minimum element is determined.
- Assuming an array with *n* elements. Table below shows the number of comparisons for each pass.
- The total number of comparisons is calculated in the formula below.
- Such an algorithm is call an n^2 algorithm, or quadratic algorithm, in terms of running time complexity.

Pass	#comparisons
1	$n-1$
2	$n-2$
3	$n-3$
…	…
$n-1$	1

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

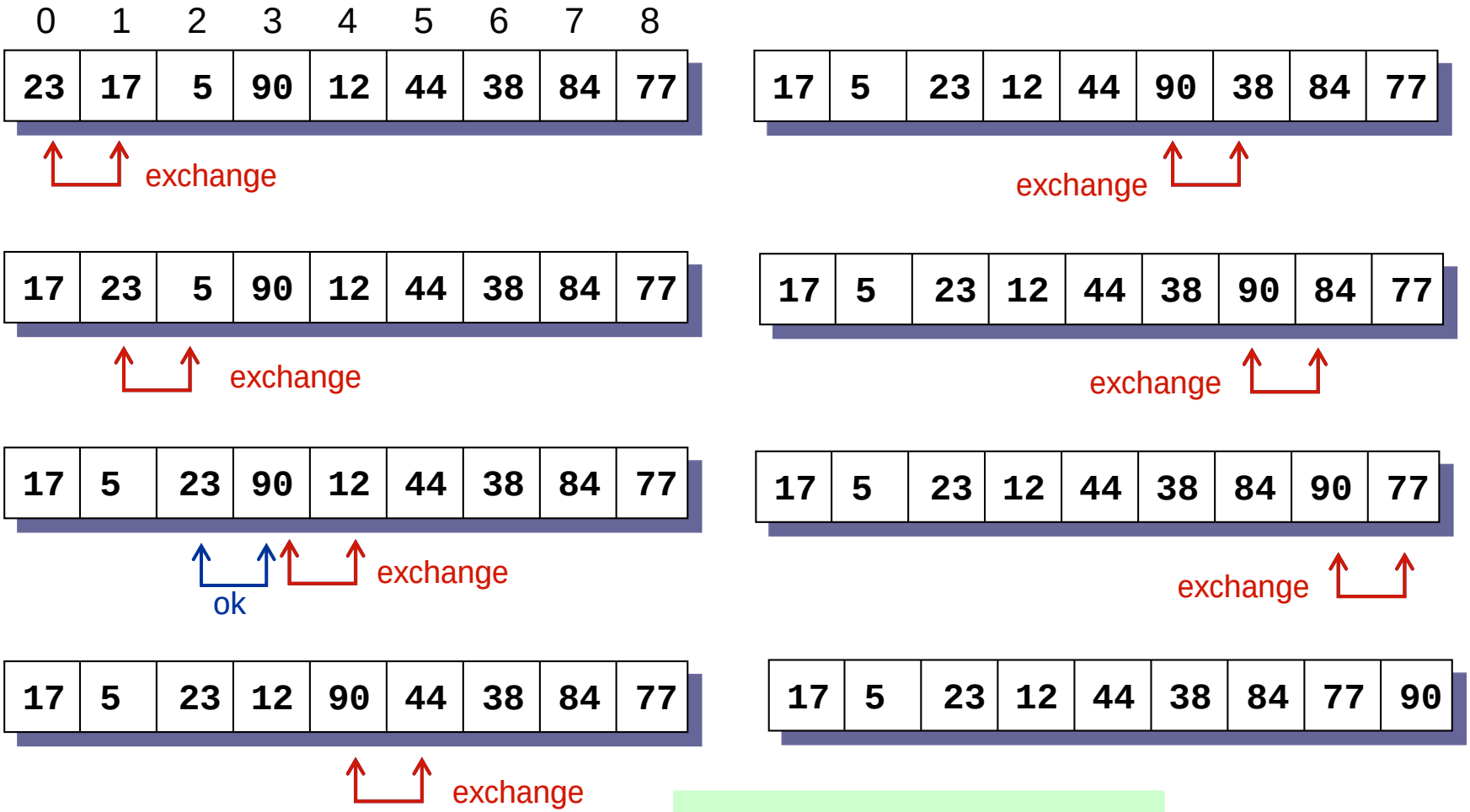
Bubble Sort

- Bubble sort is another classic sorting algorithm. The key idea is to make pairwise comparisons and exchange the positions of the pair if they are out of order.

- A YouTube video clip to simulate bubble sort:
  - http://www.youtube.com/watch?v=lyZQPjUT5B4

- Animated sorting algorithms:
  - http://www.sorting-algorithms.com/
  - http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html

One Pass of Bubble Sort



Q: Is the array sorted?
Q: What did we achieve?

Bubble Sort : Program

```
// Sort array 'arr' in increasing order
void bubble_sort(int arr[], int size) {

  int i, limit, temp;

  for (limit = size-2; limit >= 0; limit--) {

    for (i = 0; i <= limit; i++) { // one pass
      if (arr[i] > arr[i+1]) { // out of order -> swap
        temp = arr[i];
        arr[i] = arr[i+1];
        arr[i+1] = temp;
      }
    } // end inner for loop

  } // end outer for loop
}
```

Bubble Sort : Performance

Non-examinable

- Bubble Sort, like Selection Sort, requires $n - 1$ passes for an array with n elements. Comparisons occur in the inner loop.
- The number of comparisons in each pass is given in the table below and the total number of comparisons is calculated in the formula below.
- Like Selection Sort, Bubble Sort is also an n^2 algorithm, or quadratic algorithm, in terms of running time complexity.

Pass	#comparisons
1	$n - 1$
2	$n - 2$
3	$n - 3$
...	...
$n - 1$	1

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

Bubble Sort : Enhanced Version

- It is possible to enhance Bubble Sort algorithm to reduce the number of passes.
  - Suppose that in a certain pass, no swap is needed. This implies that the array is already sorted, and hence the algorithm may terminate without going on to the next pass.

- We will discuss this enhanced version in the next tutorial.

Quiz

(CS1010J AY2016/17 Semester 1 Exam, Q1.2)

- Given an integer array of 6 elements as shown below,

```
5, 3, 25, 21, 17, 10
```

How does the array look like after the second pass in Bubble sort to sort the elements in ascending order?

More Sorting Algorithms

- We have introduced 2 basic sorting algorithms Bubble Sort and Selection Sort. The third classic sorting algorithms is Insertion Sort.
  - These 3 are the simplest sorting algorithms.
  - However, they are very slow, as their running time complexity is quadratic.

- Faster sorting algorithms exist and are the topics of more advanced programming modules.
  - Merge Sort (CS1020E)
  - Quick Sort (CS1020E)
  - Heap Sort (CS2010)

Homework

- Before attending our next lecture, please attempt Problem Set 3 Exercise #12 Set Containment.
  - We will discuss it in the next lecture.

- This exercise is quite tough.
  - It requires both universal (for all) and existential (there exists) argument, and involves nested loops.
  - You are advised to complete your tutorial questions before attempting it.

Today's Summary

Searching and Sorting

Searching algorithms

- Linear (sequential) search
- Binary search

Sorting algorithms

- Selection sort
- Bubble sort