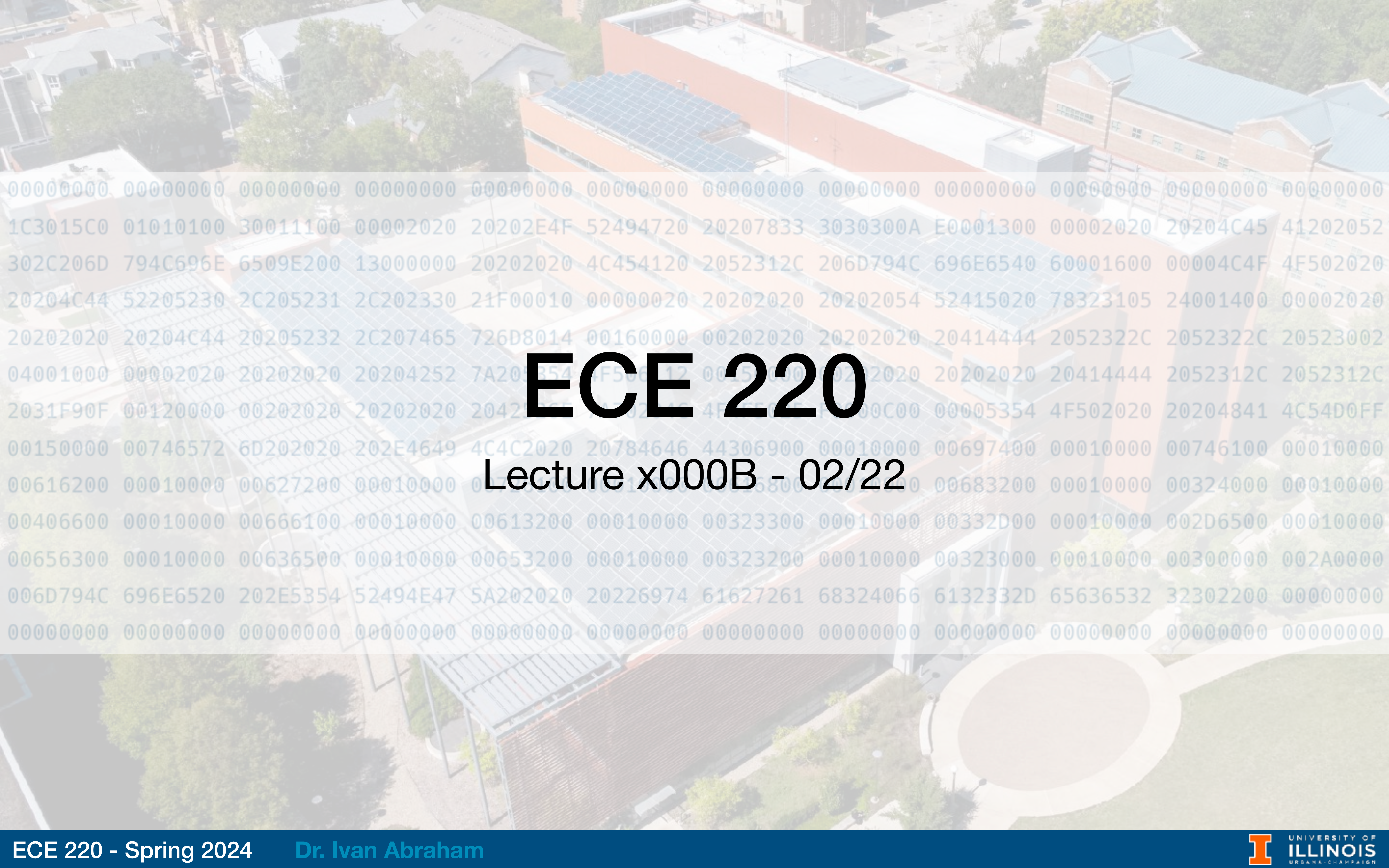

ECE 220
Lecture x000B - 02/22
ECE 220 - Spring 2024
Dr. Ivan Abraham
UNIVERSITY OF ILLINOIS

Recap + reminders

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Recap + reminders

- Midterm:

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Recap + reminders

- Midterm:
    - Regrade deadline for MT1 is midnight of 02/29.

Recap + reminders

- Midterm:
  - Regrade deadline for MT1 is midnight of 02/29.
- Quizzes also next week and week after.

Recap + reminders

- Midterm:
  - Regrade deadline for MT1 is midnight of 02/29.
- Quizzes also next week and week after.
- James Scholar HLCA deadline next week

Recap + reminders

- Midterm:
  - Regrade deadline for MT1 is midnight of 02/29.
- Quizzes also next week and week after.
- James Scholar HLCA deadline next week

- Last time:

Recap + reminders

- Midterm:
  - Regrade deadline for MT1 is midnight of 02/29.
- Quizzes also next week and week after.
- James Scholar HLCA deadline next week
- Last time:
  - Pointer/array duality & pitfalls

Recap + reminders

- Midterm:
  - Regrade deadline for MT1 is midnight of 02/29.
- Quizzes also next week and week after.
- James Scholar HLCA deadline next week

- Last time:
  - Pointer/array duality & pitfalls
  - Strings a.k.a. char arrays and functions (`sscanf`, `fgets`)

Recap + reminders

- Midterm:
  - Regrade deadline for MT1 is midnight of 02/29.
- Quizzes also next week and week after.
- James Scholar HLCA deadline next week

- Last time:
  - Pointer/array duality & pitfalls
  - Strings a.k.a. char arrays and functions (`sscanf`, `fgets`)
  - Multi-dimensional arrays

Multi-dimensional arrays

Multi-dimensional arrays

- C allows for defining *multi-dimensional* arrays (we already saw them with string arrays).

Multi-dimensional arrays

- C allows for defining *multi-dimensional* arrays (we already saw them with string arrays).
- The *dimension* of an array is determined by the minimum number of indices required to access its individual elements.

Multi-dimensional arrays

- C allows for defining *multi-dimensional* arrays (we already saw them with string arrays).
- The *dimension* of an array is determined by the minimum number of indices required to access its individual elements.

One dimensional array

Multi-dimensional arrays

- C allows for defining *multi-dimensional* arrays (we already saw them with string arrays).
- The *dimension* of an array is determined by the minimum number of indices required to access its individual elements.

One dimensional array

0	1	2	3

Two dimensional array

0,0	0,1	0,2	0,3
1,0	1,1	1,2	1,3
2,0	2,1	2,2	2,3

Multi-dimensional arrays

Multi-dimensional arrays

- The syntax for two dimensional arrays is:

Multi-dimensional arrays

- The syntax for two dimensional arrays is:

```
type varname[nr][nc];
```


UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Multi-dimensional arrays

- The syntax for two dimensional arrays is:

```
type varname[nr][nc];
```

where `nr` and `nc` are the number of rows & columns.

Multi-dimensional arrays

- The syntax for two dimensional arrays is:

```
type varname[nr][nc];
```

where `nr` and `nc` are the number of rows & columns.

- Example: `int a[3][4];`

Multi-dimensional arrays

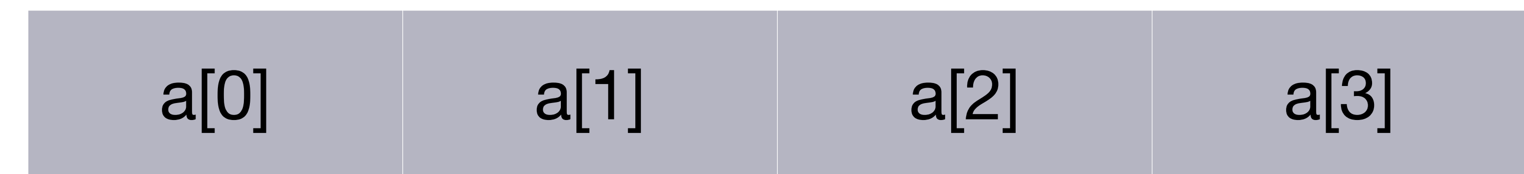
- The syntax for two dimensional arrays is:

```
type varname[nr][nc];
```

where `nr` and `nc` are the number of rows & columns.

- Example: `int a[3][4];`

One dimensional array



Multi-dimensional arrays

- The syntax for two dimensional arrays is:

```
type varname[nr][nc];
```

where `nr` and `nc` are the number of rows & columns.

- Example: `int a[3][4];`

One dimensional array

a[0]	a[1]	a[2]	a[3]

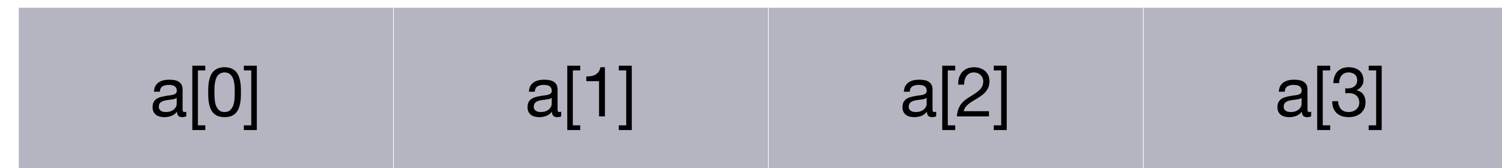
Two dimensional array

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Allocating memory

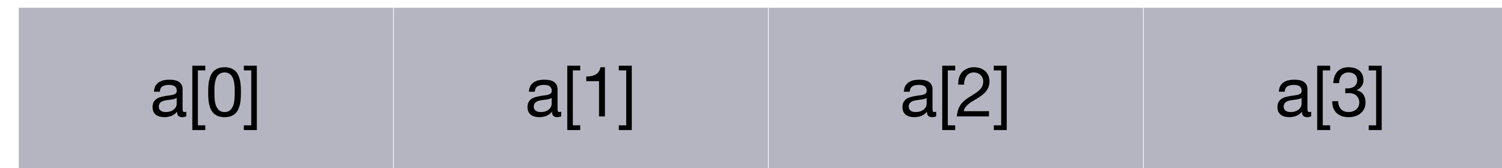
Allocating memory

One dimensional array



Allocating memory

One dimensional array

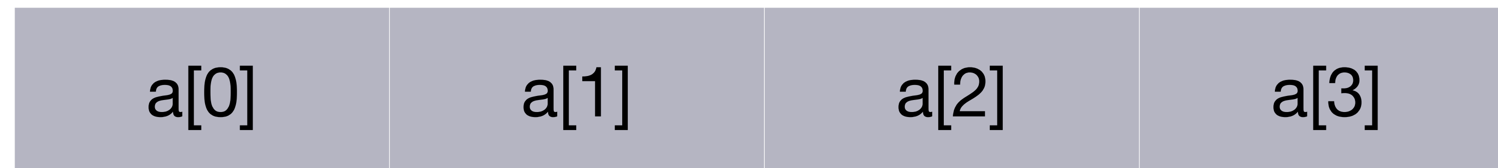


	Offsets
a[0]	0
a[1]	1
a[2]	2
a[3]	3

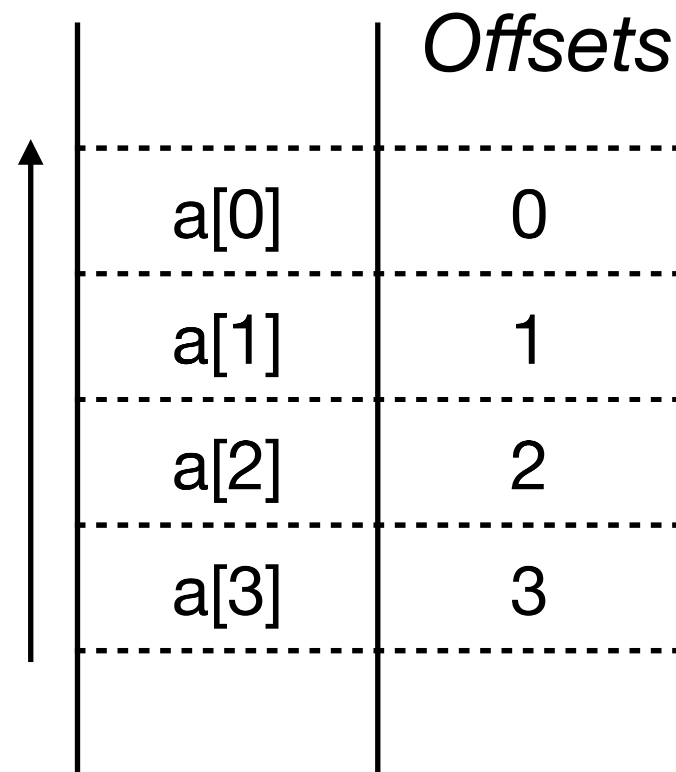
I UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Allocating memory

One dimensional array

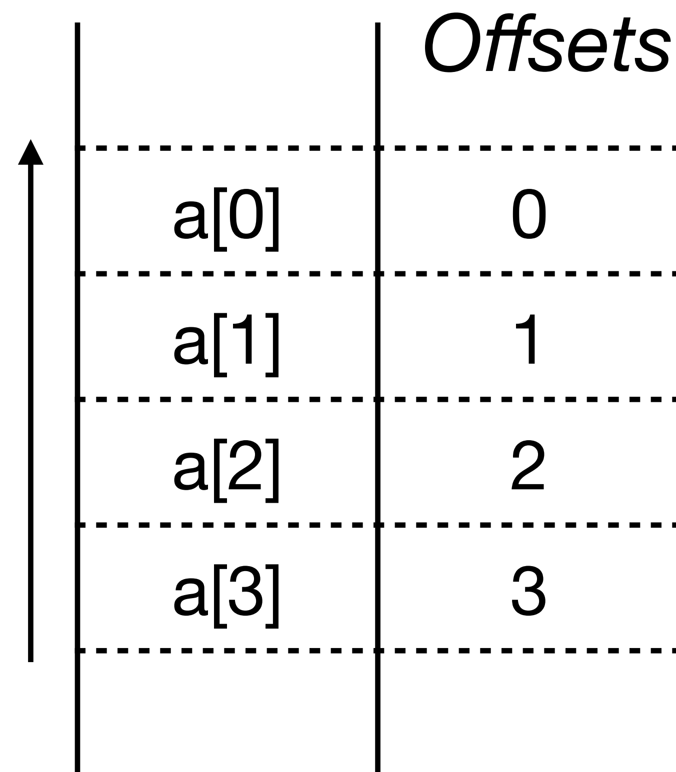


Offsets

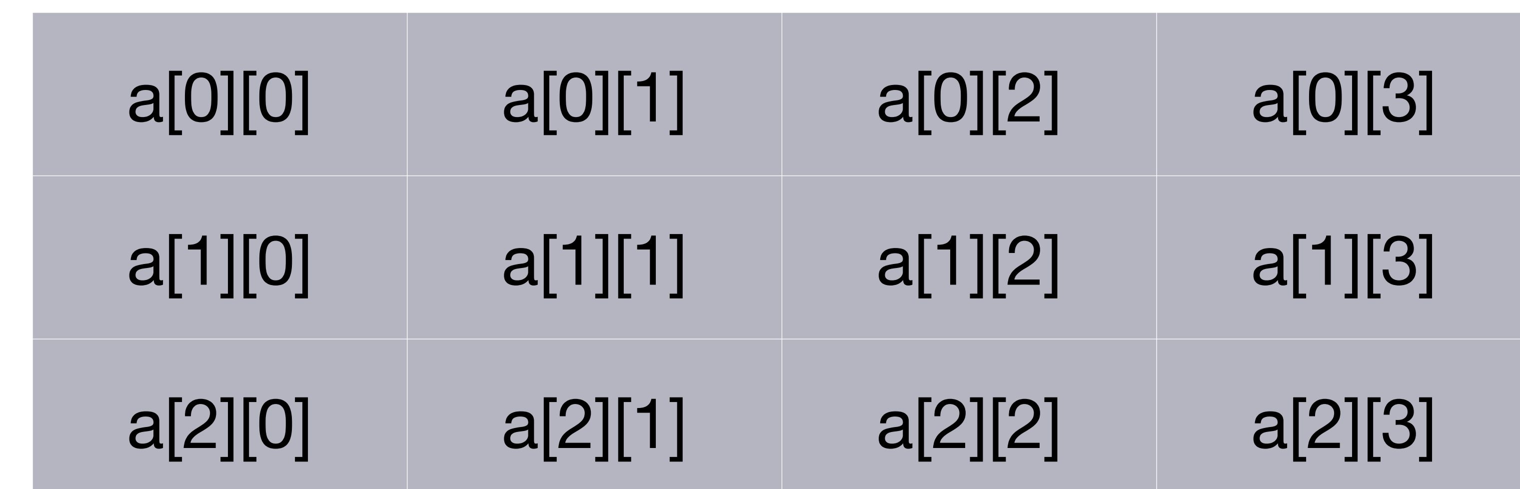


Allocating memory

One dimensional array

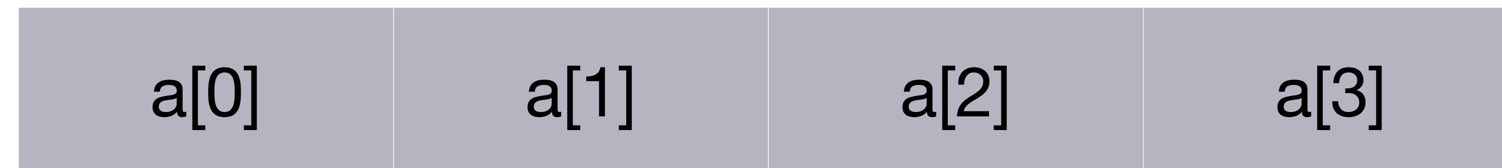


Two dimensional array



Allocating memory

One dimensional array



	Offsets
a[0]	0
a[1]	1
a[2]	2
a[3]	3

...	*Offsets*
a[1][2]	6
a[1][3]	7
a[2][0]	8
a[2][1]	9
a[2][2]	10
a[2][3]	11

Two dimensional array

a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

Allocating memory

One dimensional array

	Offsets
a[0]	0
a[1]	1
a[2]	2
a[3]	3

...	Offsets
a[1][2]	6
a[1][3]	7
a[2][0]	8
a[2][1]	9
a[2][2]	10
a[2][3]	11

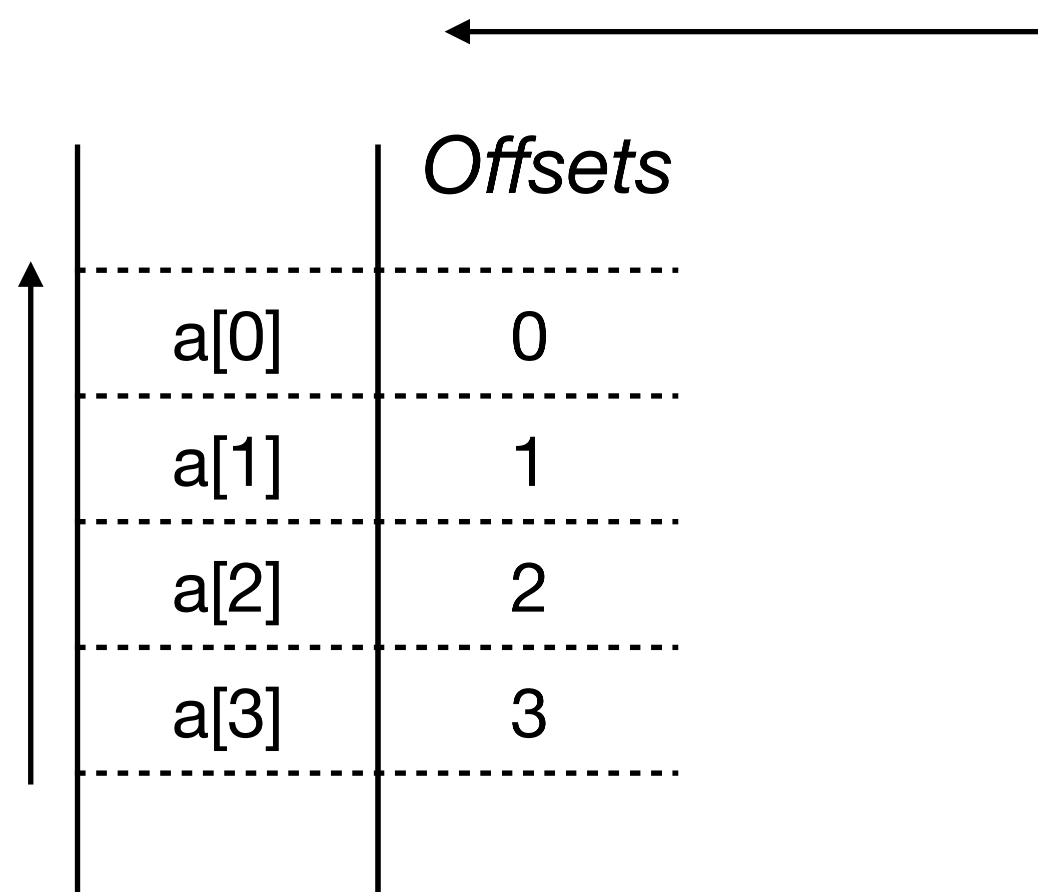
Two dimensional array

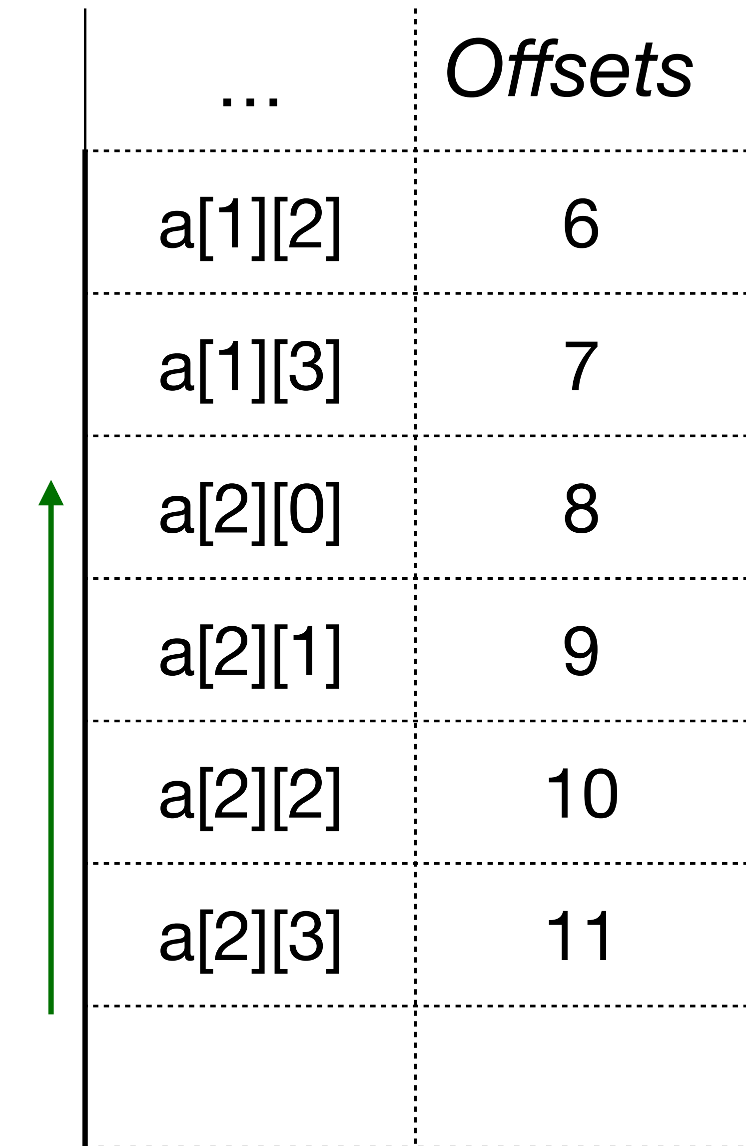
a[0][0]	a[0][1]	a[0][2]	a[0][3]
a[1][0]	a[1][1]	a[1][2]	a[1][3]
a[2][0]	a[2][1]	a[2][2]	a[2][3]

C follows what is called *row-major order*, i.e rows first.

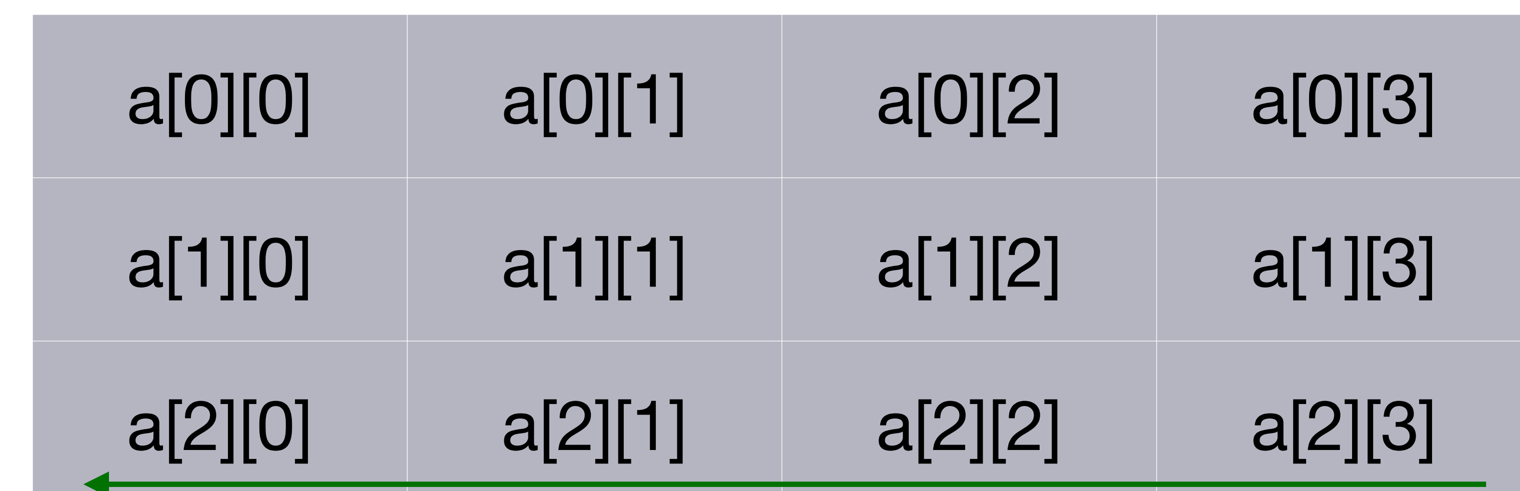
Allocating memory

One dimensional array





Two dimensional array

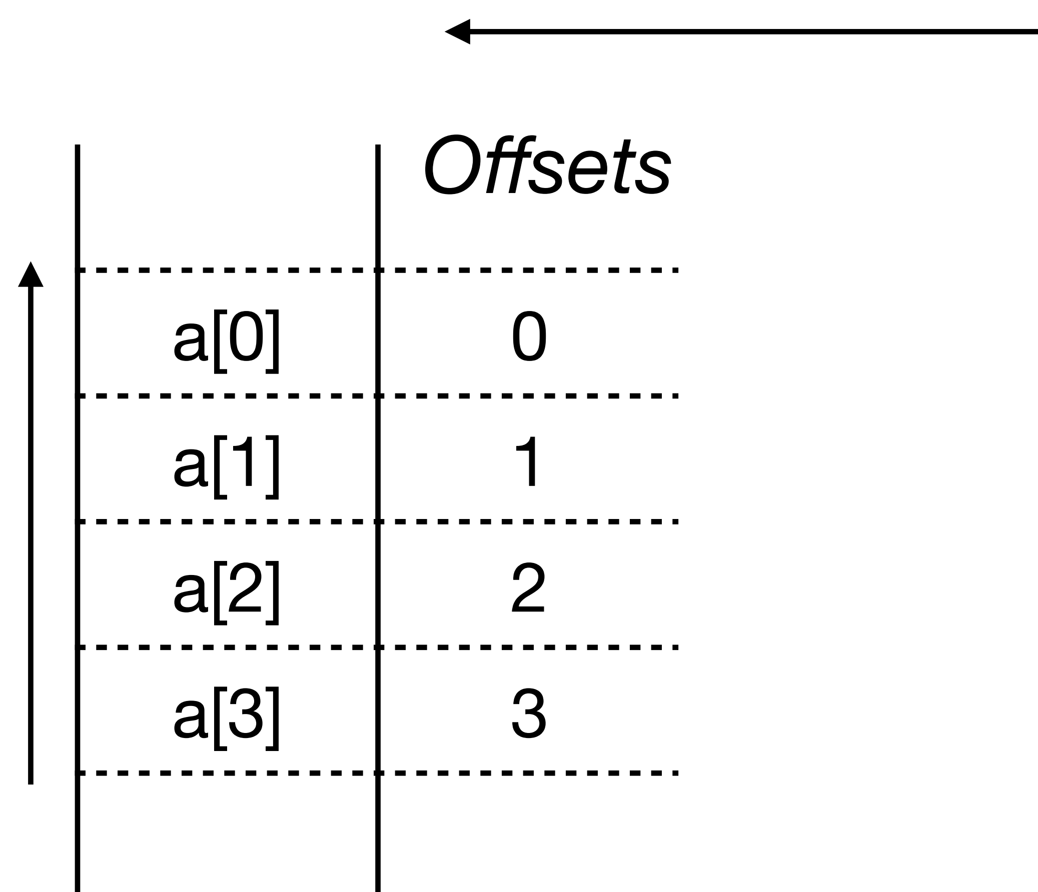


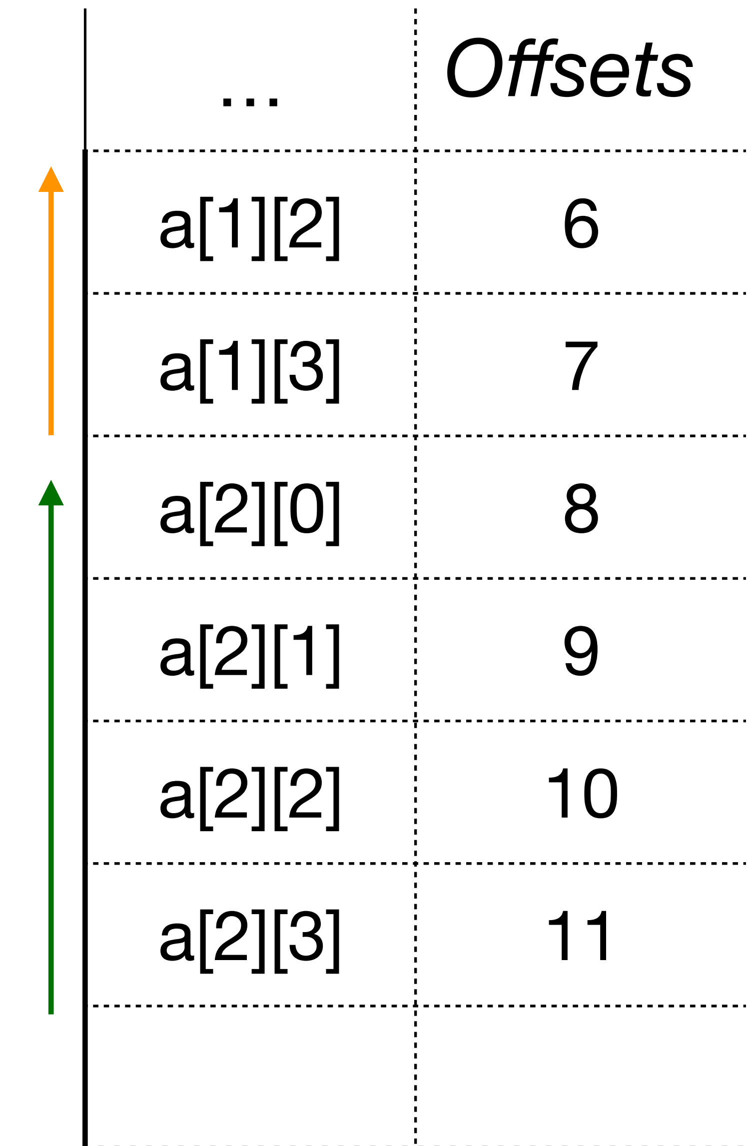
C follows what is called *row-major order*, i.e rows first.

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

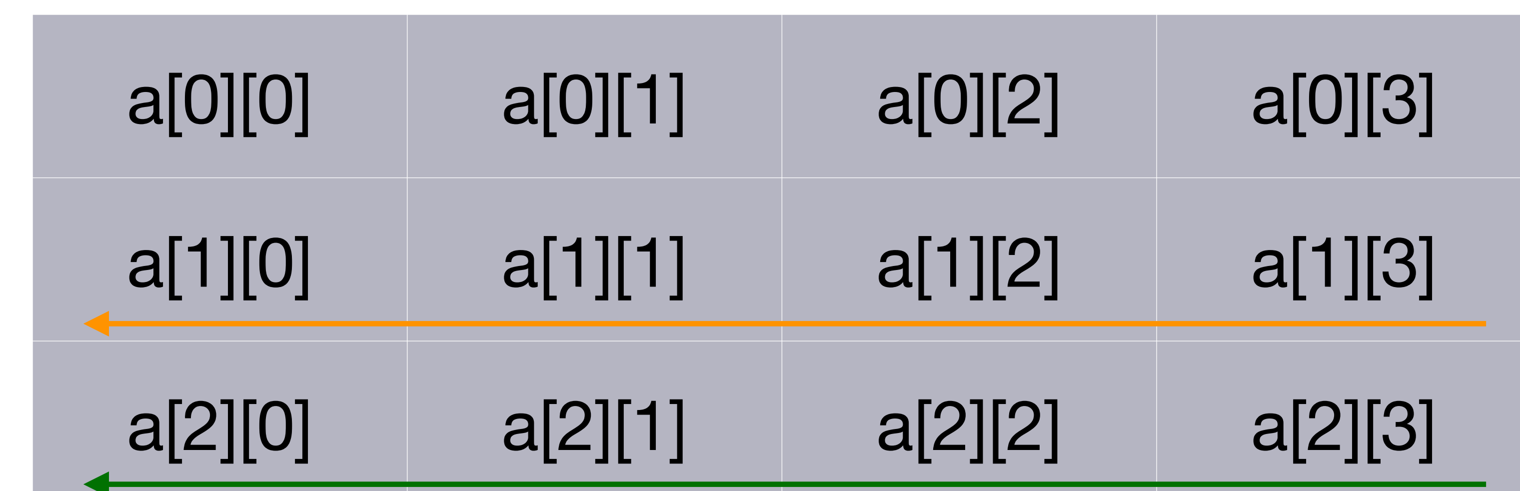
Allocating memory

One dimensional array





Two dimensional array

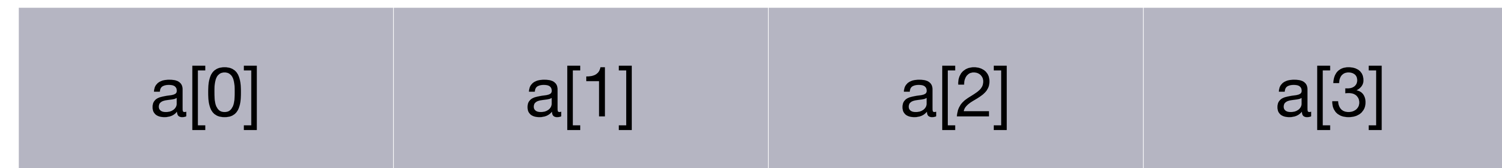


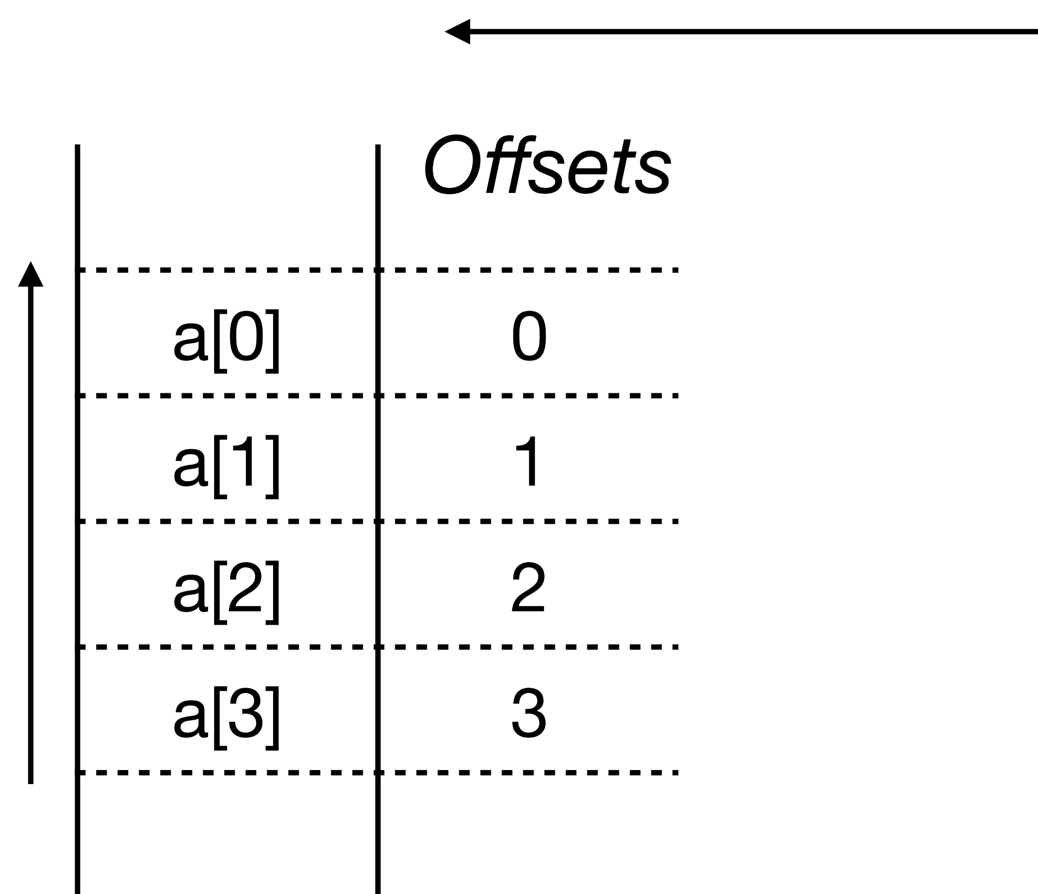
C follows what is called *row-major order*, i.e rows first.

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

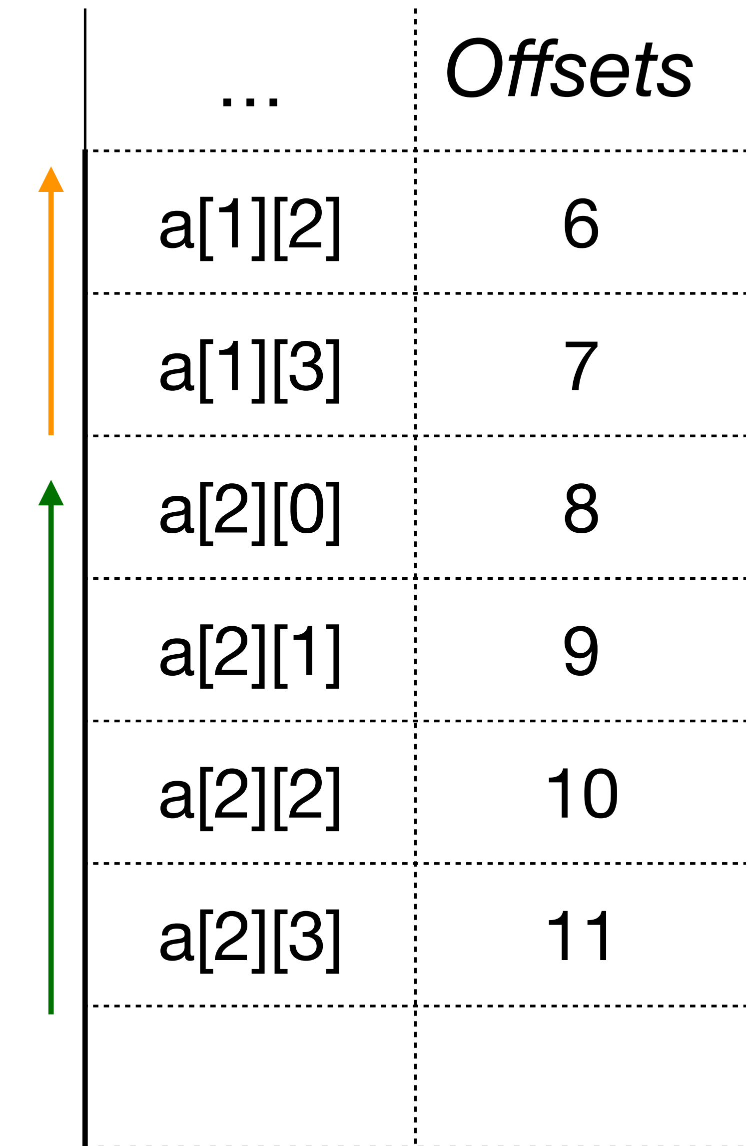
Allocating memory

One dimensional array

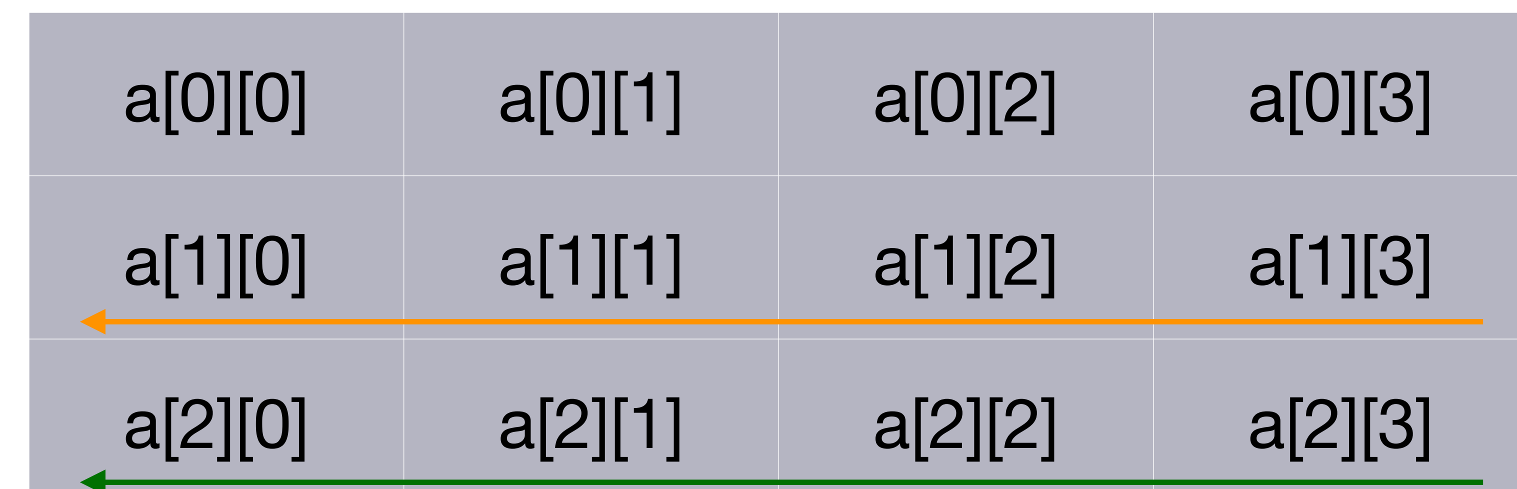




How to calculate offset?



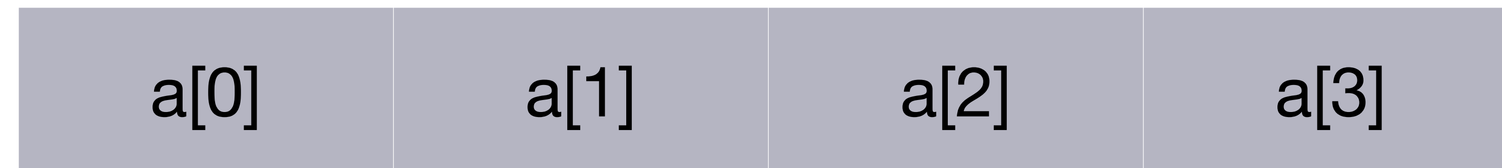
Two dimensional array



C follows what is called *row-major order*, i.e rows first.

Allocating memory

One dimensional array



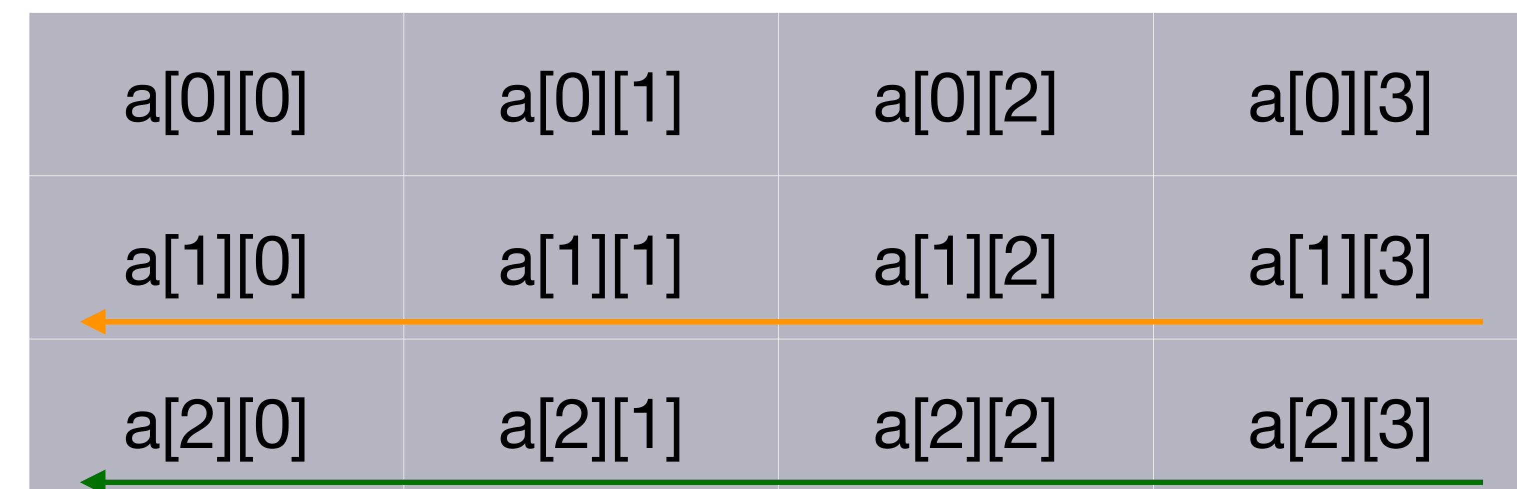
	Offsets
a[0]	0
a[1]	1
a[2]	2
a[3]	3

How to calculate offset?

```
offset = ri * nc + ci
```

...	*Offsets*
a[1][2]	6
a[1][3]	7
a[2][0]	8
a[2][1]	9
a[2][2]	10
a[2][3]	11

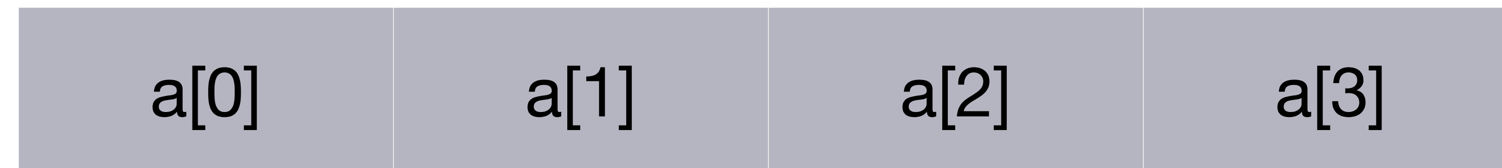
Two dimensional array

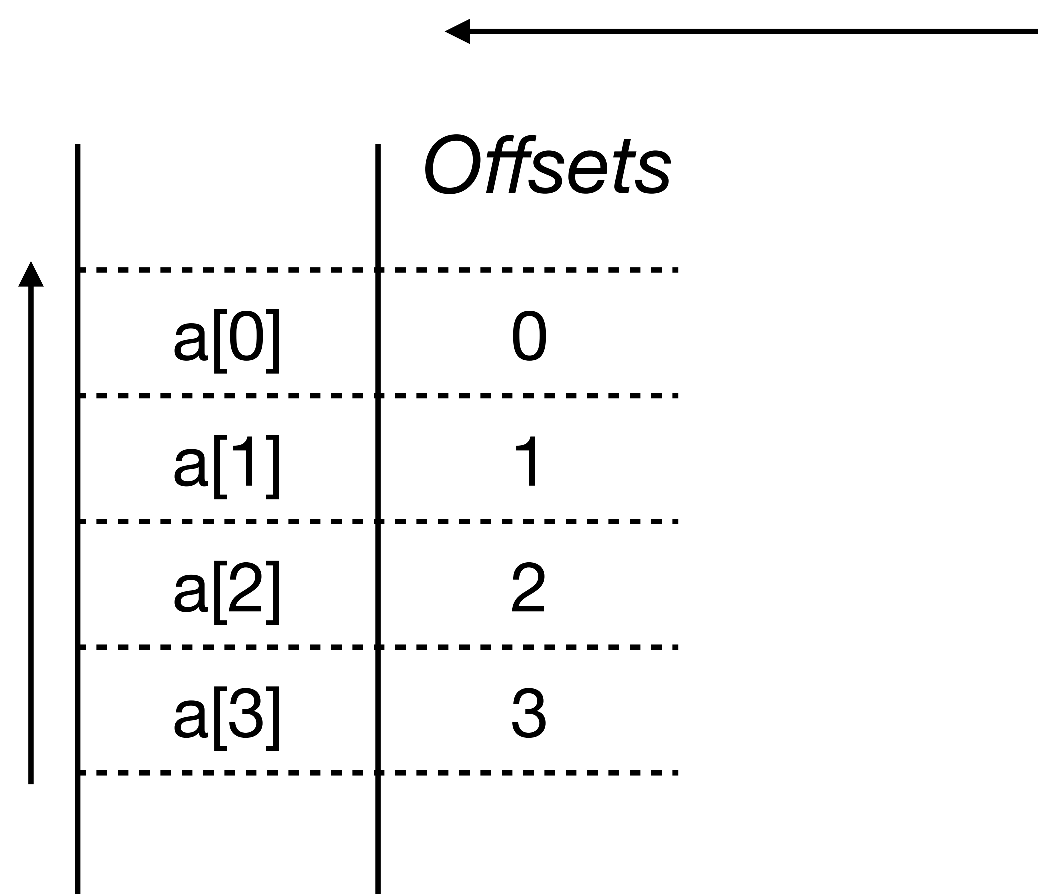


C follows what is called *row-major order*, i.e rows first.

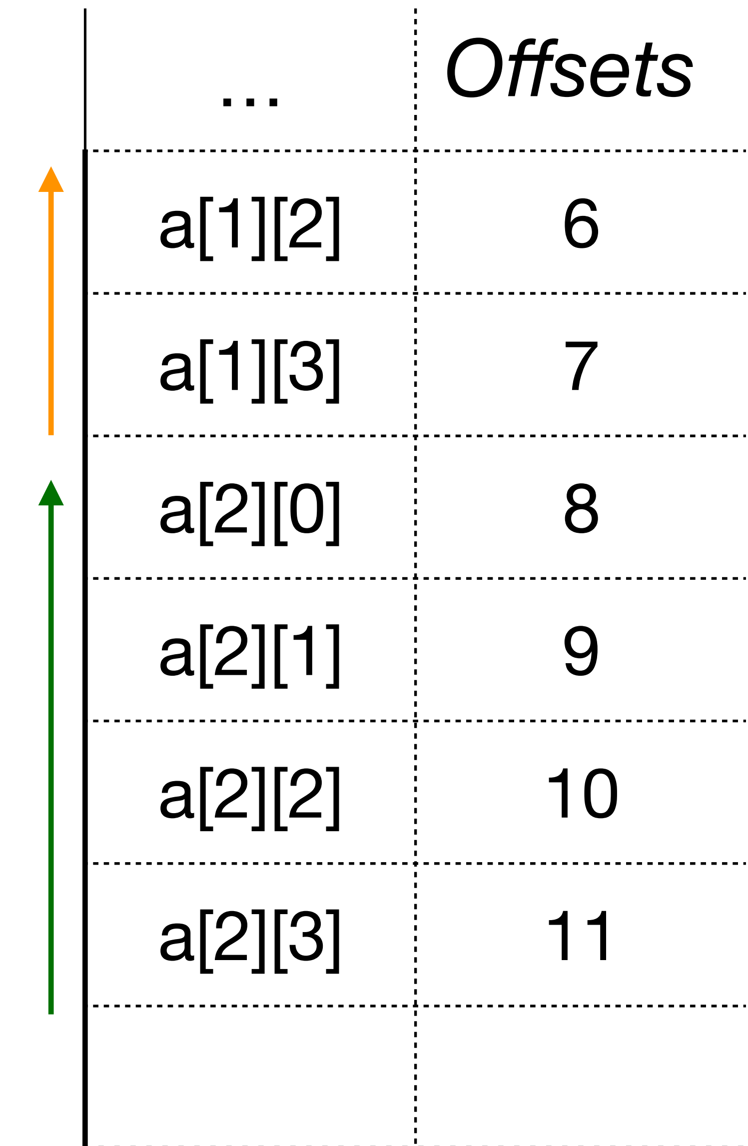
Allocating memory

One dimensional array





How to calculate offset?

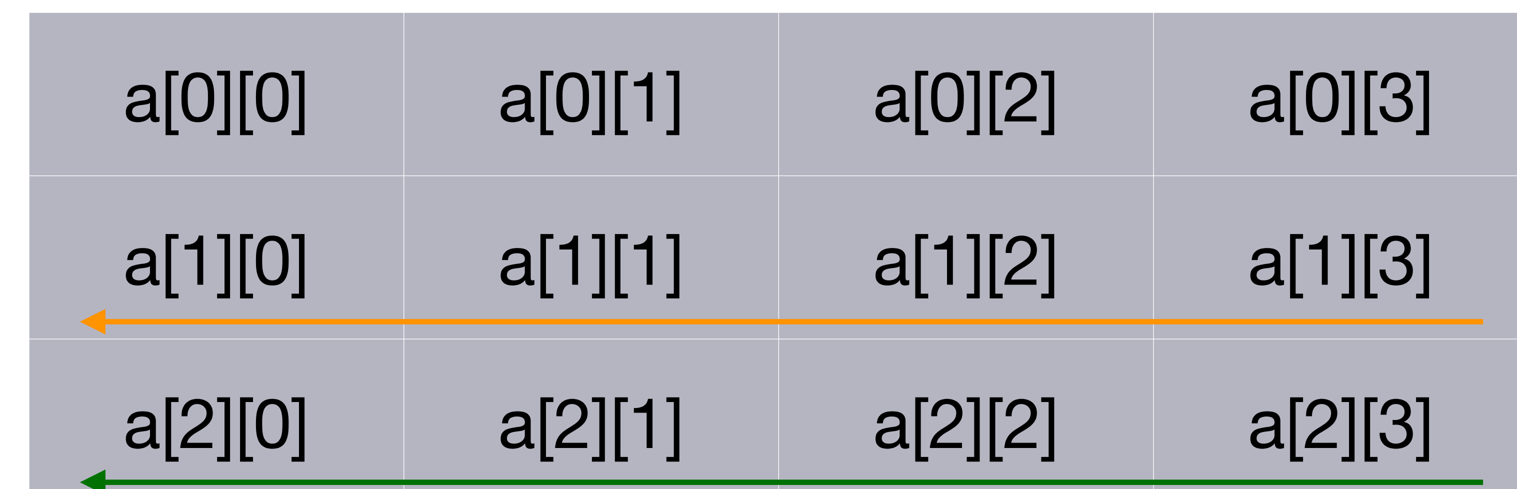


Column index

```
offset = ri * nc + ci
```

Row index

Two dimensional array



C follows what is called *row-major order*, i.e rows first.

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Initializing 2D arrays

- There are multiple ways to initialize a 2D array.

Initializing 2D arrays

- There are multiple ways to initialize a 2D array.
- Here are *four* equivalent ways to initialize a 2×3 array:

Initializing 2D arrays

- There are multiple ways to initialize a 2D array.
- Here are *four* equivalent ways to initialize a 2×3 array:
  - `int a[2][3] = {{1,2,3},{4,5,6}};`

Initializing 2D arrays

- There are multiple ways to initialize a 2D array.
- Here are *four* equivalent ways to initialize a 2×3 array:
    - `int a[2][3] = {{1,2,3},{4,5,6}};`
    - `int a[2][3] = {1,2,3,4,5,6};`

Initializing 2D arrays

- There are multiple ways to initialize a 2D array.
- Here are *four* equivalent ways to initialize a 2×3 array:
  - `int a[2][3] = {{1,2,3},{4,5,6}};`
  - `int a[2][3] = {1,2,3,4,5,6};`
  - `int a[][3] = {{1,2,3},{4,5,6}};`

Initializing 2D arrays

- There are multiple ways to initialize a 2D array.
- Here are *four* equivalent ways to initialize a 2×3 array:
  - `int a[2][3] = {{1,2,3},{4,5,6}};`
  - `int a[2][3] = {1,2,3,4,5,6};`
  - `int a[][3] = {{1,2,3},{4,5,6}};`
  - `int a[][3] = {1,2,3,4,5,6};`

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Initializing 2D arrays

- There are multiple ways to initialize a 2D array.
- Here are *four* equivalent ways to initialize a 2×3 array:
  - `int a[2][3] = {{1,2,3},{4,5,6}};`
  - `int a[2][3] = {1,2,3,4,5,6};`
  - `int a[][3] = {{1,2,3},{4,5,6}};`
  - `int a[][3] = {1,2,3,4,5,6};`
- Why not: `int a[2][] = {{1,2,3},{4,5,6}};` ?

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Exercise 1

- Given a matrix mat stored as a two dimensional array of integers, write a function `exchng_rows` which will exchange the row `r1` with row `r2` function where: `0 <= r1, r2 < NROWS`.

```
#define NROWS 3
#define NCOLS 4

void exchng_rows(int mat[NROWS][NCOLS], int r1, int r2)
```

Exercise 1

- Given a matrix mat stored as a two dimensional array of integers, write a function `exchng_rows` which will exchange the row `r1` with row `r2` function where: `0 <= r1, r2 < NROWS`.

```
#define NROWS 3
#define NCOLS 4

void exchng_rows(int mat[NROWS][NCOLS], int r1, int r2)
```

Dimensions are global symbols

Exercise 1

- Given a matrix mat stored as a two dimensional array of integers, write a function `exchng_rows` which will exchange the row `r1` with row `r2` function where: `0 <= r1, r2 < NROWS`.

```
#define NROWS 3
#define NCOLS 4

void exchng_rows(int mat[NROWS][NCOLS], int r1, int r2)
```

This function signature is well defined.

Dimensions are global symbols

Exercise 1

```
void exchng_rows(int mat[NROWS][NCOLS], int r1, int r2){
  for (int i=0; i<NCOLS; i++){
    int temp = mat[r1][i];
    mat[r1][i] = mat[r2][i];
    mat[r2][i] = temp;
  }
}
```

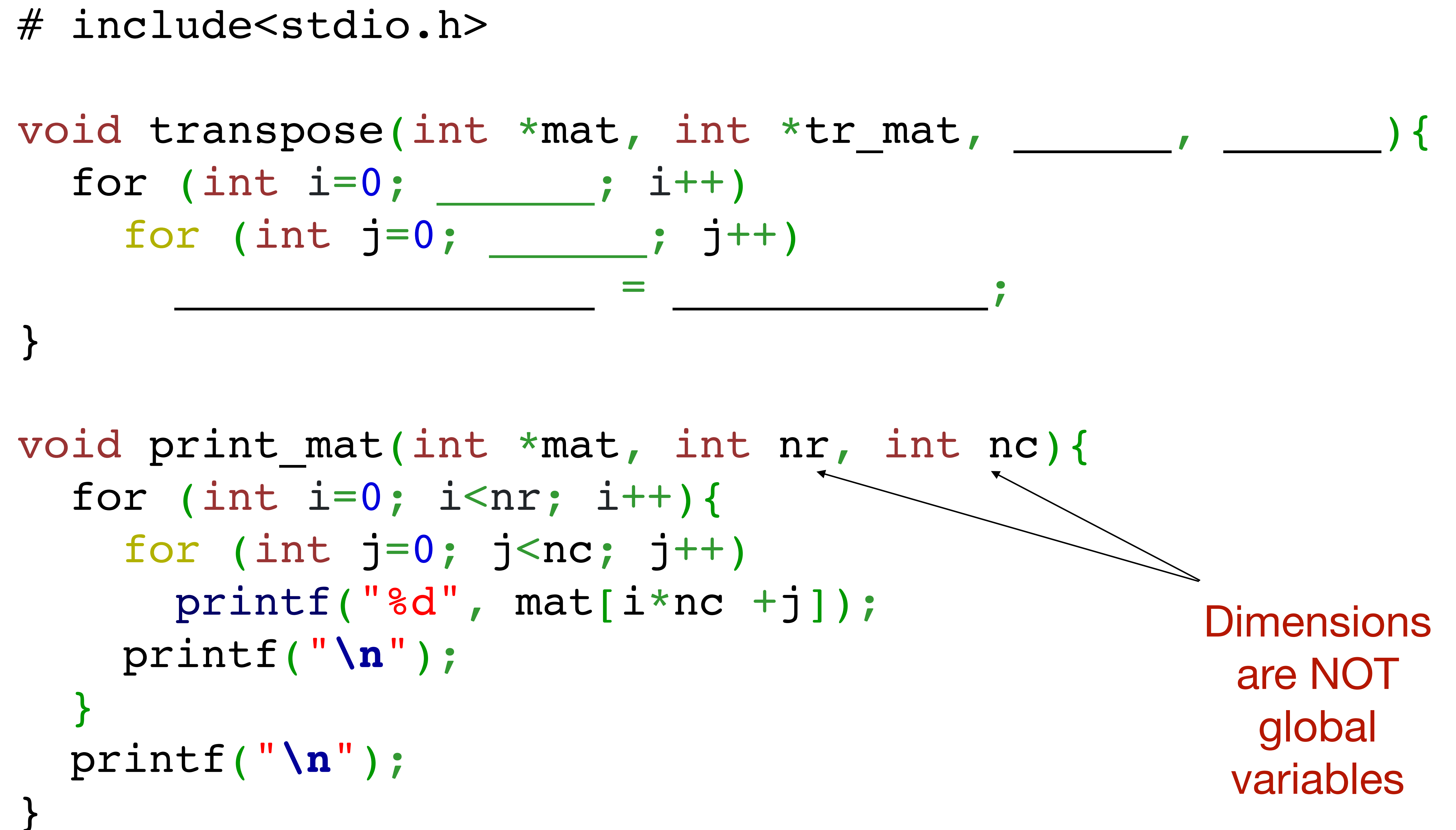
Exercise 2

Write a C function that given a matrix `mat` of size `nr` × `nm` and another matrix `tr_mat` of size `nr` × `nm` copies the *transpose* of `mat` into `tr_mat`.

```
# include<stdio.h>

void transpose(int *mat, int *tr_mat, ______, ______){
  for (int i=0; ______; i++)
    for (int j=0; ______; j++)
      ________________ = ____________;
}

void print_mat(int *mat, int nr, int nc){
  for (int i=0; i<nr; i++){
    for (int j=0; j<nc; j++)
      printf("%d", mat[i*nc +j]);
    printf("\n");
  }
  printf("\n");
}
```

Exercise 2

Write a C function that given a matrix `mat` of size $nr \times nm$ and another matrix `tr_mat` of size $nr \times nm$ copies the *transpose* of `mat` into `tr_mat`.



Exercise 2

Write a C function that given a matrix `mat` of size `nr` × `nm` and another matrix `tr_mat` of size `nr` × `nm` copies the *transpose* of `mat` into `tr_mat`.

```
# include<stdio.h>

void transpose(int *mat, int *tr_mat, ______, ______){
  for (int i=0; ______; i++)
    for (int j=0; ______; j++)
      ________________ = ____________;
}

void print_mat(int *mat, int nr, int nc){
  for (int i=0; i<nr; i++){
    for (int j=0; j<nc; j++)
      printf("%d", mat[i*nc +j]);
    printf("\n");
  }
  printf("\n");
}
```

Matrix is passed in as a pointer.

Dimensions are NOT global variables

Exercise 2

Write a C function that given a matrix `mat` of size `nr` $\times$ `nm` and another matrix `tr_mat` of size `nr` $\times$ `nm` copies the *transpose* of `mat` into `tr_mat`.

```
# include<stdio.h>

void transpose(int *mat, int *tr_mat, ______, ______){
  for (int i=0; ______; i++)
    for (int j=0; ______; j++)
      ________________ = ____________;
}

void print_mat(int *mat, int nr, int nc){
  for (int i=0; i<nr; i++){
    for (int j=0; j<nc; j++)
      printf("%d", mat[i*nc +j]);
    printf("\n");
  }
  printf("\n");
}
```

2D shape information is lost!

Matrix is passed in as a pointer.

Dimensions are NOT global variables

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Exercise 2

Write a C function that given a matrix `mat` of size `nr` × `nm` and another matrix `tr_mat` of size `nr` × `nm` copies the *transpose* of `mat` into `tr_mat`.

```
# include<stdio.h>

void transpose(int *mat, int *tr_mat, ______, ______){
  for (int i=0; ______; i++)
    for (int j=0; ______; j++)
      ________________ = ____________;
}

void print_mat(int *mat, int nr, int nc){
  for (int i=0; i<nr; i++){
    for (int j=0; j<nc; j++)
      printf("%d", mat[i*nc +j]);
    printf("\n");
  }
  printf("\n");
}
```

Lets fill in the blanks!

Problem solving: searching

- Searching whether an element is in a list very common operation
- We explore two approaches for 1-D arrays:

Problem solving: searching

- Searching whether an element is in a list very common operation
- We explore two approaches for 1-D arrays:
  - Linear search

Problem solving: searching

- Searching whether an element is in a list very common operation
- We explore two approaches for 1-D arrays:
  - Linear search
  - Binary search

Linear search

- This is as vanilla as a search gets.
- Go through the list from beginning to end until a match is found:
  - Search item is often called *key.*
  - Animation

Linear search - implementation

```
int linear_search(int list[], int n, int key){
    for (int i = 0; i < n; i++){

    if (________________)
        return i;
    }
    ______________;
}
```


I UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Binary search

Binary search

- In linear search if *key* happens to be last item in list (of size n) then we make n comparisons - denoted $O(n)$ for time complexity.

Binary search

- In linear search if *key* happens to be last item in list (of size n) then we make n comparisons - denoted $O(n)$ for time complexity.
  - *However*, if the list is sorted then we can use this to our advantage.

Binary search

- In linear search if *key* happens to be last item in list (of size n) then we make n comparisons - denoted $O(n)$ for time complexity.
    - *However*, if the list is sorted then we can use this to our advantage.
    - Compare given `key` to middle element `mid`.

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Binary search

- In linear search if *key* happens to be last item in list (of size n) then we make n comparisons - denoted $O(n)$ for time complexity.
  - *However*, if the list is sorted then we can use this to our advantage.
  - Compare given `key` to middle element `mid`.
    - If `key > mid` focus search on right half

Binary search

- In linear search if *key* happens to be last item in list (of size n) then we make n comparisons - denoted $O(n)$ for time complexity.
  - *However*, if the list is sorted then we can use this to our advantage.
  - Compare given `key` to middle element `mid`.
    - If `key > mid` focus search on right half
    - If `key < mid` focus search on left half

Binary search

- In linear search if *key* happens to be last item in list (of size n) then we make n comparisons - denoted $O(n)$ for time complexity.
    - *However*, if the list is sorted then we can use this to our advantage.
    - Compare given `key` to middle element `mid`.
        - If `key > mid` focus search on right half
        - If `key < mid` focus search on left half
        - If `key == mid` then done

Binary search

- In linear search if *key* happens to be last item in list (of size n) then we make n comparisons - denoted $O(n)$ for time complexity.
    - *However*, if the list is sorted then we can use this to our advantage.
    - Compare given `key` to middle element `mid`.
        - If `key > mid` focus search on right half
        - If `key < mid` focus search on left half
        - If `key == mid` then done
- Animation

Binary search

```
int binary(int arr[], int n, int key){
  int low = 0;      // Left pointer
  int high = _____; // Right pointer

  while (high >= low){
    int mid = (__________) / 2; // Pick middle element

    // Logic to focus search on left or right of mid
    if (key == arr[mid])
      return mid;
    else if (key < arr[mid])
      high = _______;
    else
      low = ________;
  }
  return -1; // Loop exited, element not present.
}
```

Sorting

Sorting

- Why sort lists or arrays?

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Sorting

- Why sort lists or arrays?
    - We saw one reason

Sorting

- Why sort lists or arrays?
  - We saw one reason
    - Searching

Sorting

- Why sort lists or arrays?
  - We saw one reason
    - Searching
  - Other reasons?

Sorting

- Why sort lists or arrays?
  - We saw one reason
    - Searching
  - Other reasons?
    - Assigning students by UIN to exam rooms.

Sorting

- Why sort lists or arrays?
  - We saw one reason
    - Searching
  - Other reasons?
    - Assigning students by UIN to exam rooms.
    - Etc.

Sorting

- Why sort lists or arrays?
  - We saw one reason
    - Searching
  - Other reasons?
    - Assigning students by UIN to exam rooms.
    - Etc.
- Finding efficient algorithms for sorting is highly researched problem.

Sorting

- Why sort lists or arrays?
  - We saw one reason
    - Searching
  - Other reasons?
    - Assigning students by UIN to exam rooms.
    - Etc.

- Finding efficient algorithms for sorting is highly researched problem.

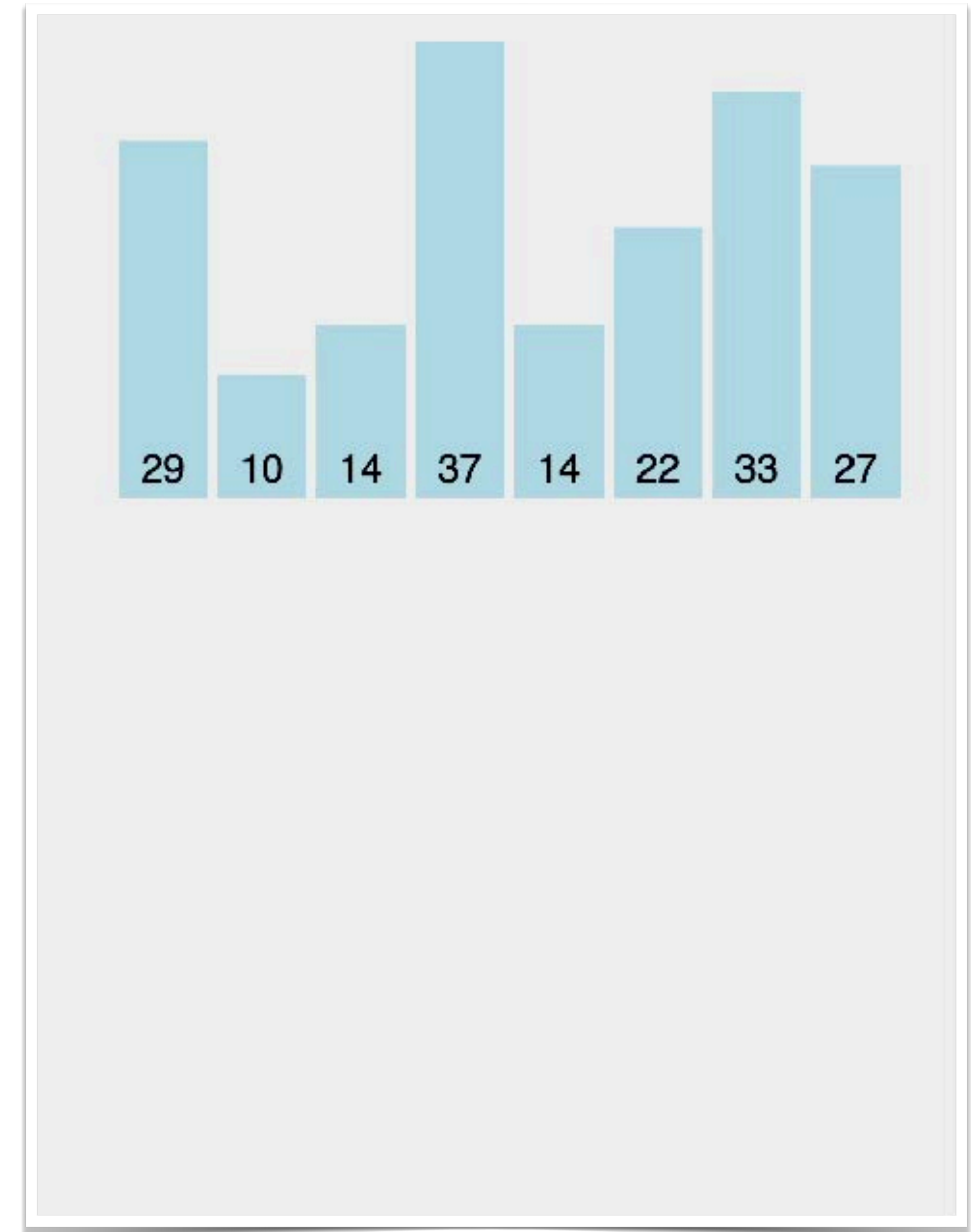
- Many flavors exist: bubble sort, selection sort, insertion sort, quick sort, etc.

Sorting

- Why sort lists or arrays?
  - We saw one reason
    - Searching
  - Other reasons?
    - Assigning students by UIN to exam rooms.
    - Etc.

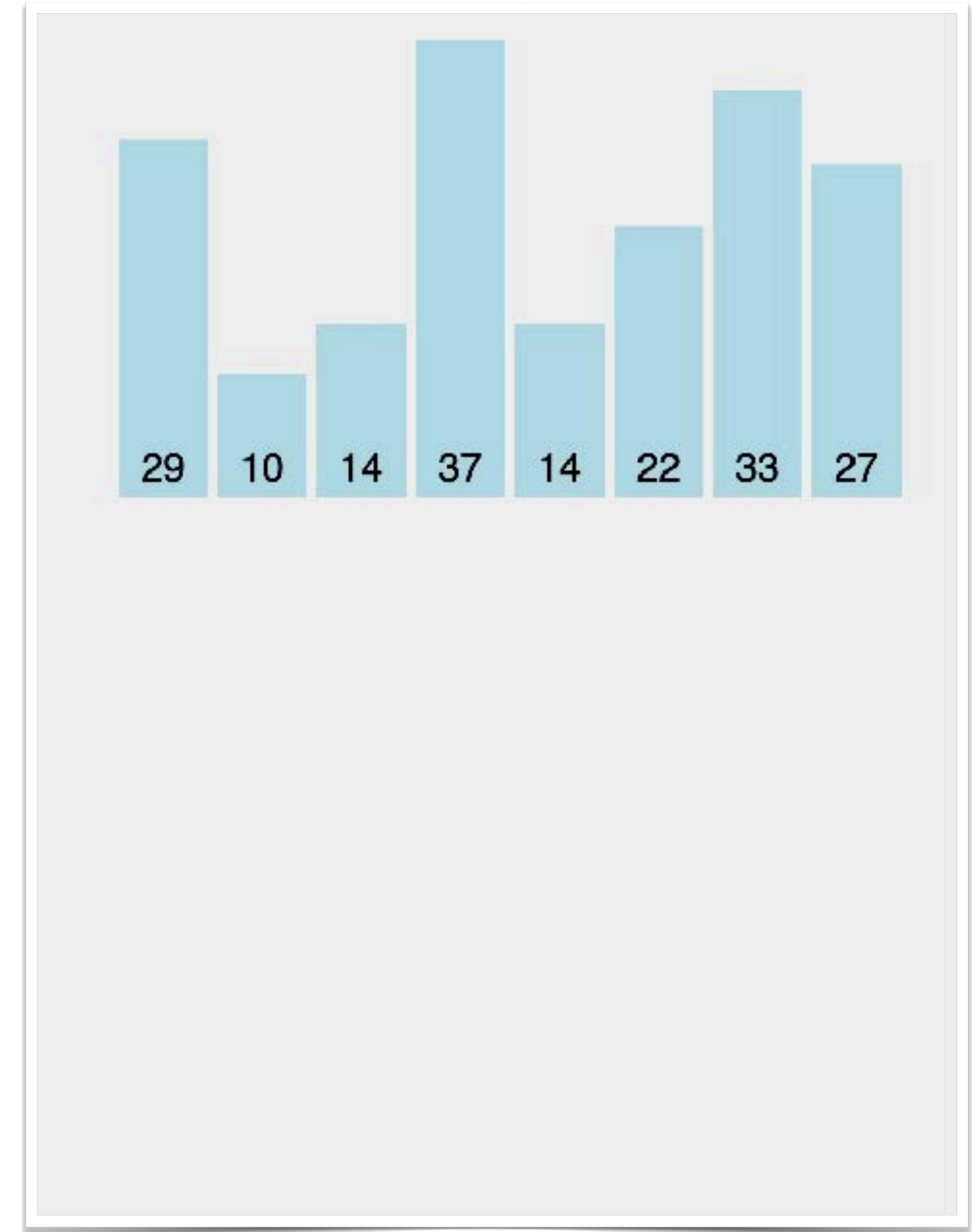
- Finding efficient algorithms for sorting is highly researched problem.
- Many flavors exist: bubble sort, selection sort, insertion sort, quick sort, etc.
- Knowing some of them off the top of your head … probably required for technical interview.

Selection sort



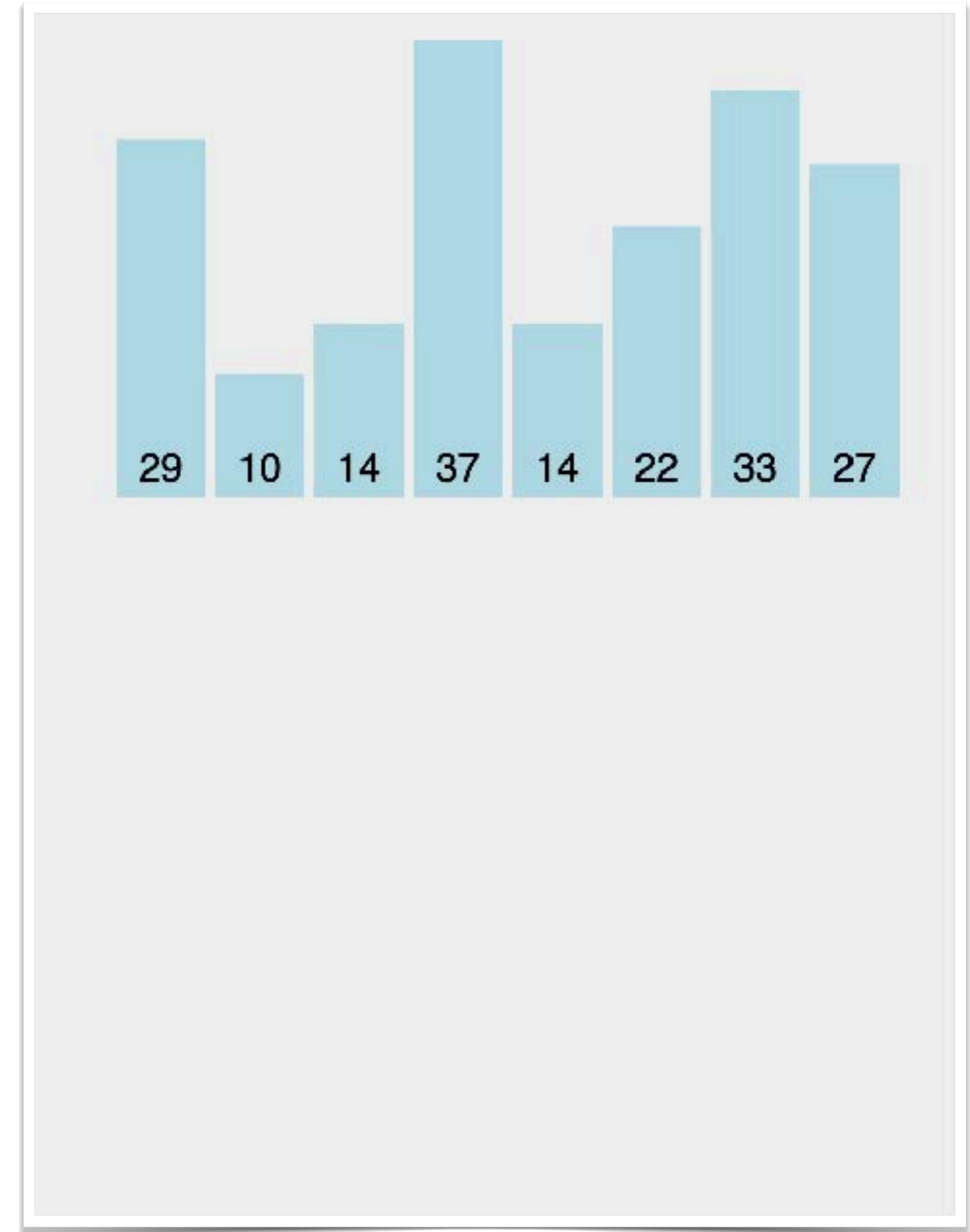
Selection sort

- Conceptually one of the simplest algorithms.



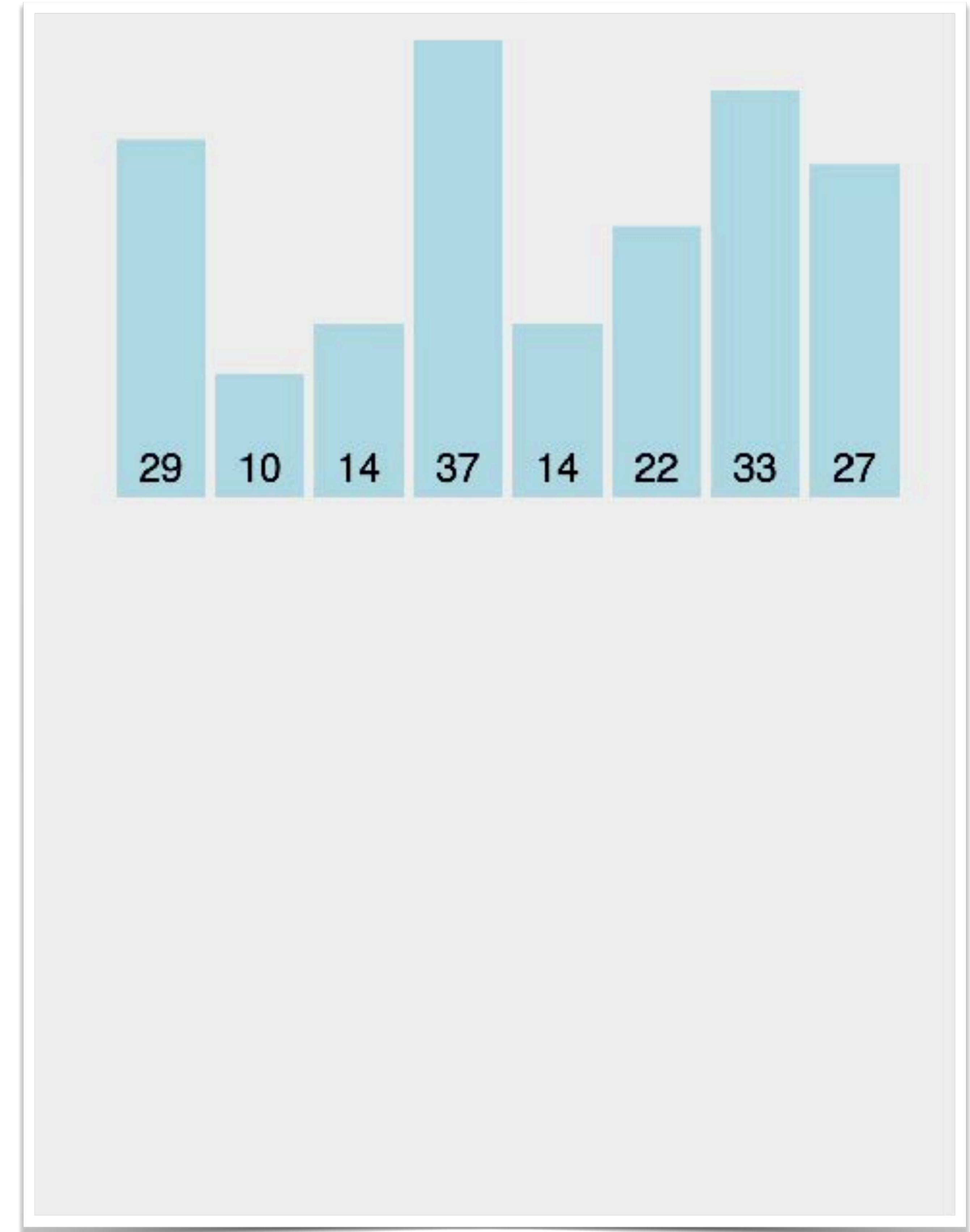
Selection sort

- Conceptually one of the simplest algorithms.
- Starting from one end of array, make N passes.



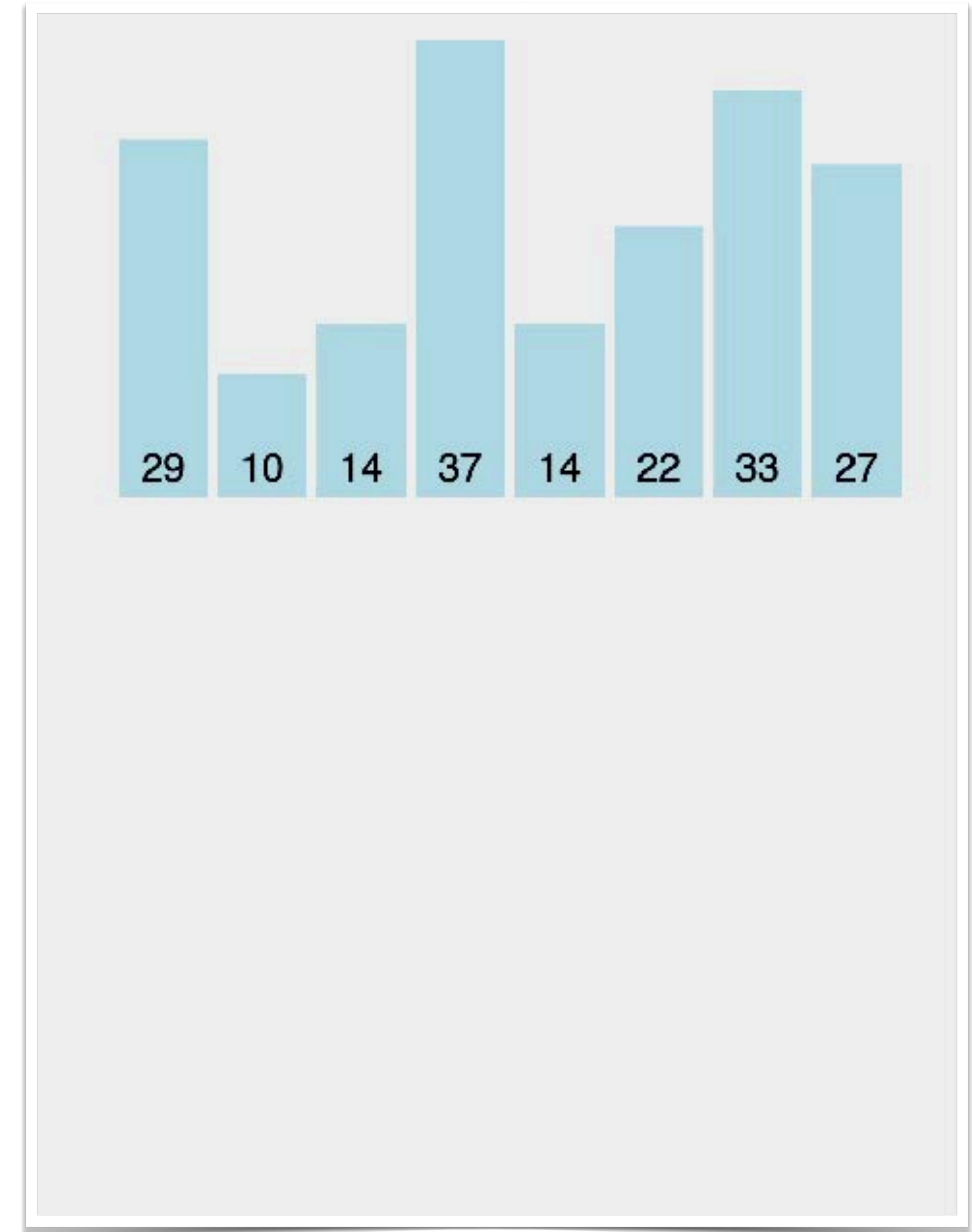
Selection sort

- Conceptually one of the simplest algorithms.
- Starting from one end of array, make N passes.
    - In Nth pass, find Nth smallest item and bring it to the Nth spot with a swap.



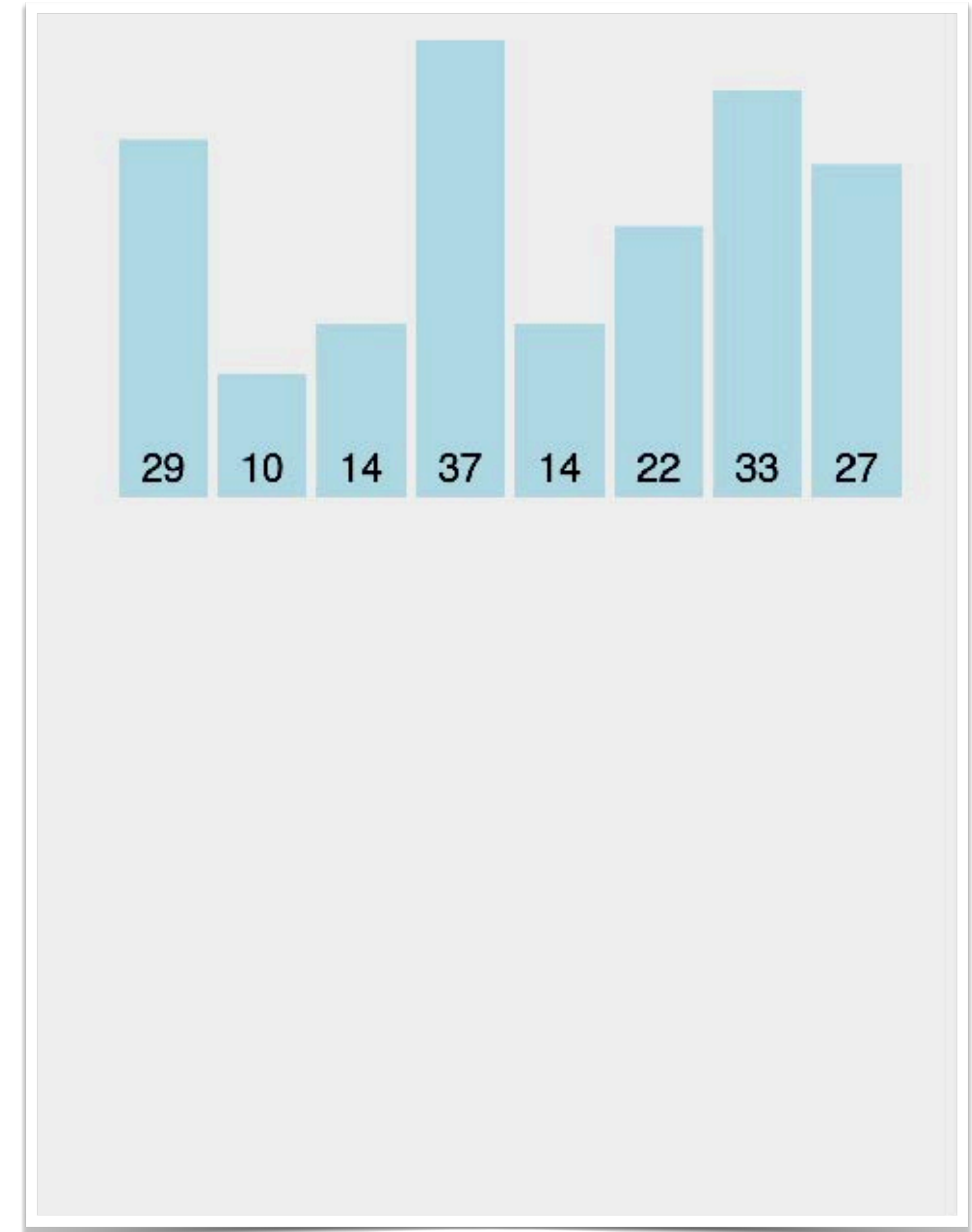
Selection sort

- Conceptually one of the simplest algorithms.
- Starting from one end of array, make N passes.
  - In Nth pass, find Nth smallest item and bring it to the Nth spot with a swap.
  - After N passes, array is sorted.



Selection sort

- Conceptually one of the simplest algorithms.
- Starting from one end of array, make N passes.
  - In Nth pass, find Nth smallest item and bring it to the Nth spot with a swap.
  - After N passes, array is sorted.



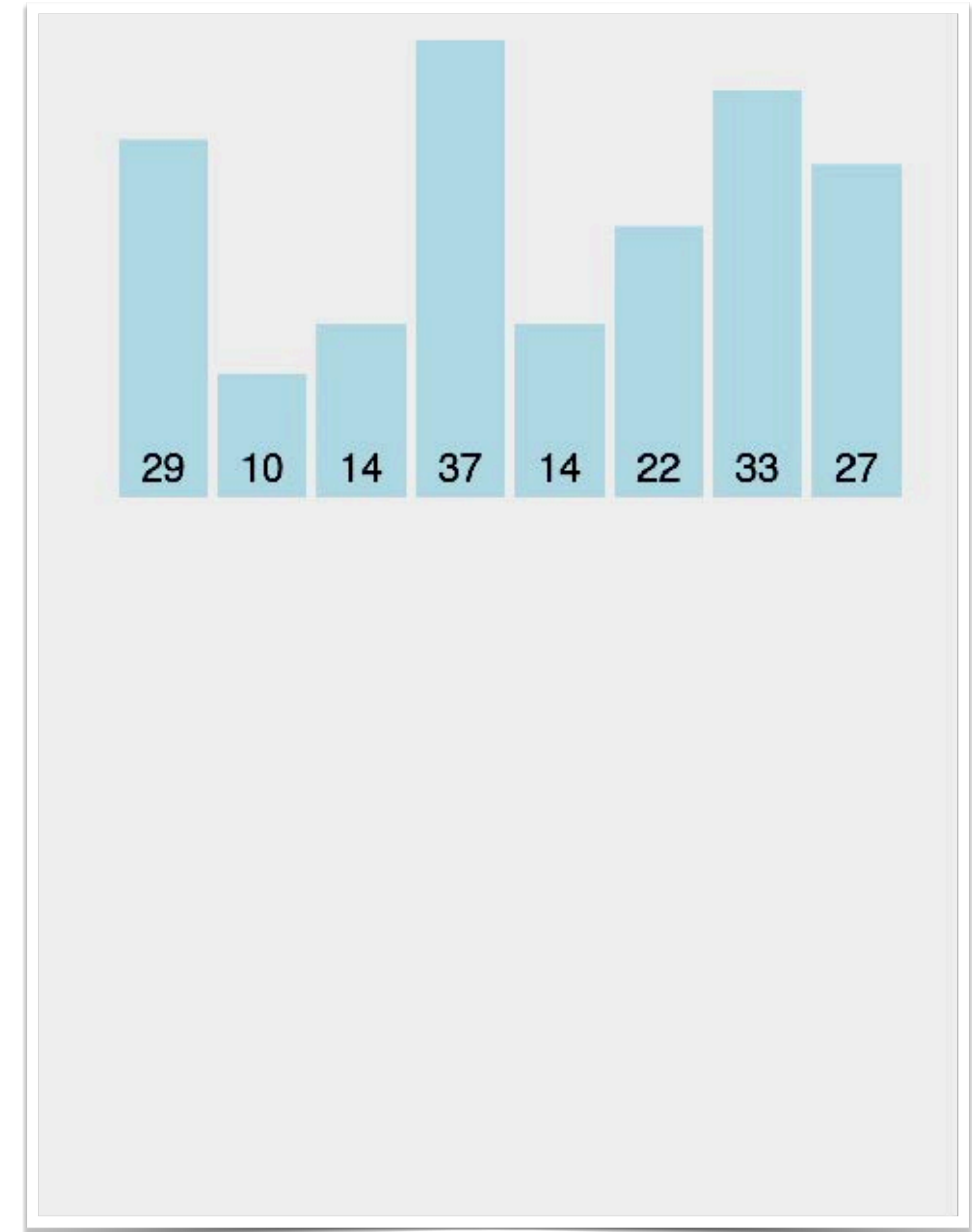
Selection sort

```
void selection_sort(int arr[], int n){
  for (int i = 0; _________; i++){

    int min_idx = i; // Initialize min to first item

    // Find the minimum in the  sublist: list[i..arraySize-1]
    for (int j = i + 1; j < n; j++)
      if (_____________________)
        min_idx = j;

    // swap list[i] with list[currentMinIndex] if necessary;
    if (min_idx != i){
      _______ = ____________;
      arr[min_idx] = arr[i];
      arr[i] = min;
    }
  }
}
```


UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

https://visualgo.net/en/sorting

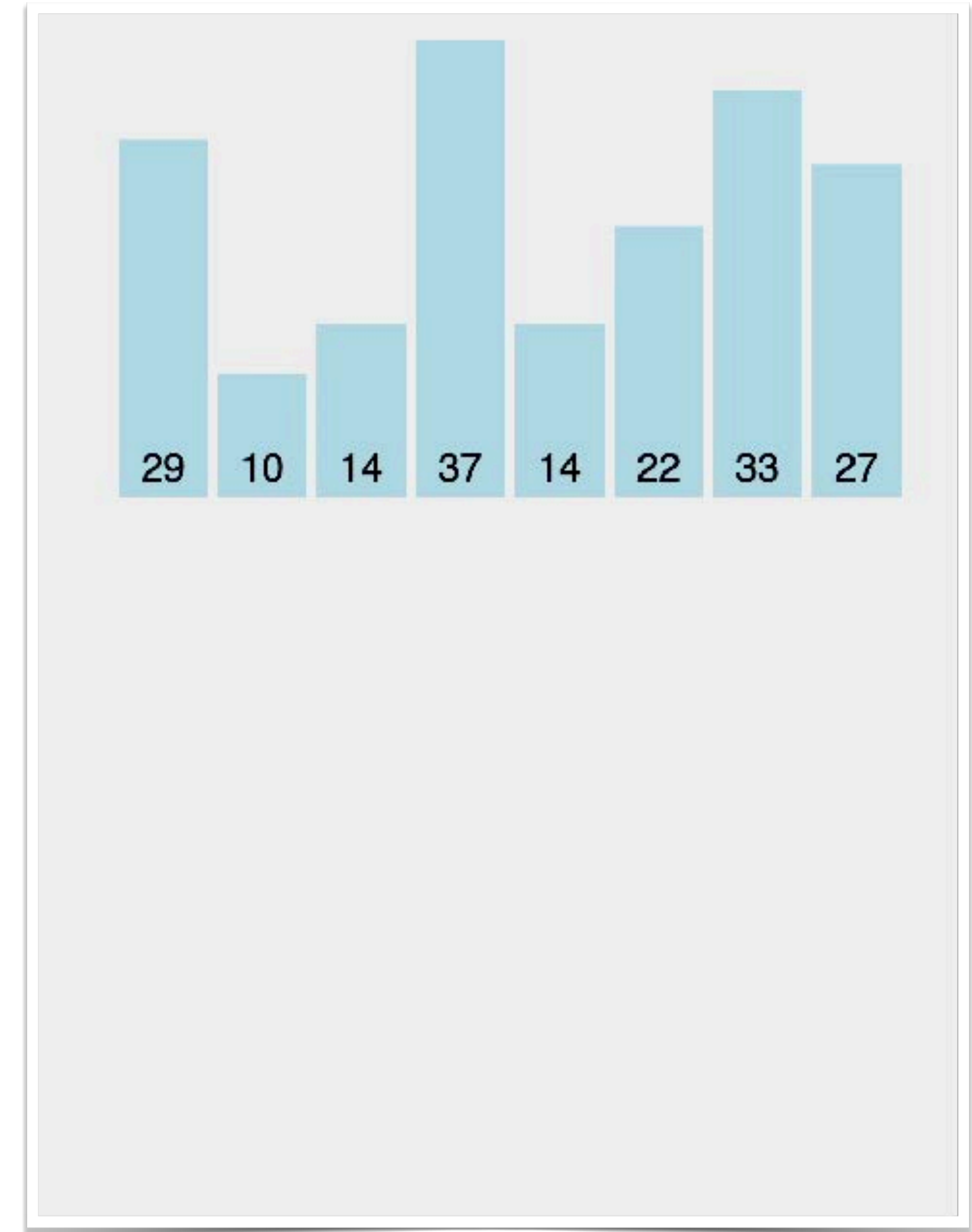
Insertion sort



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

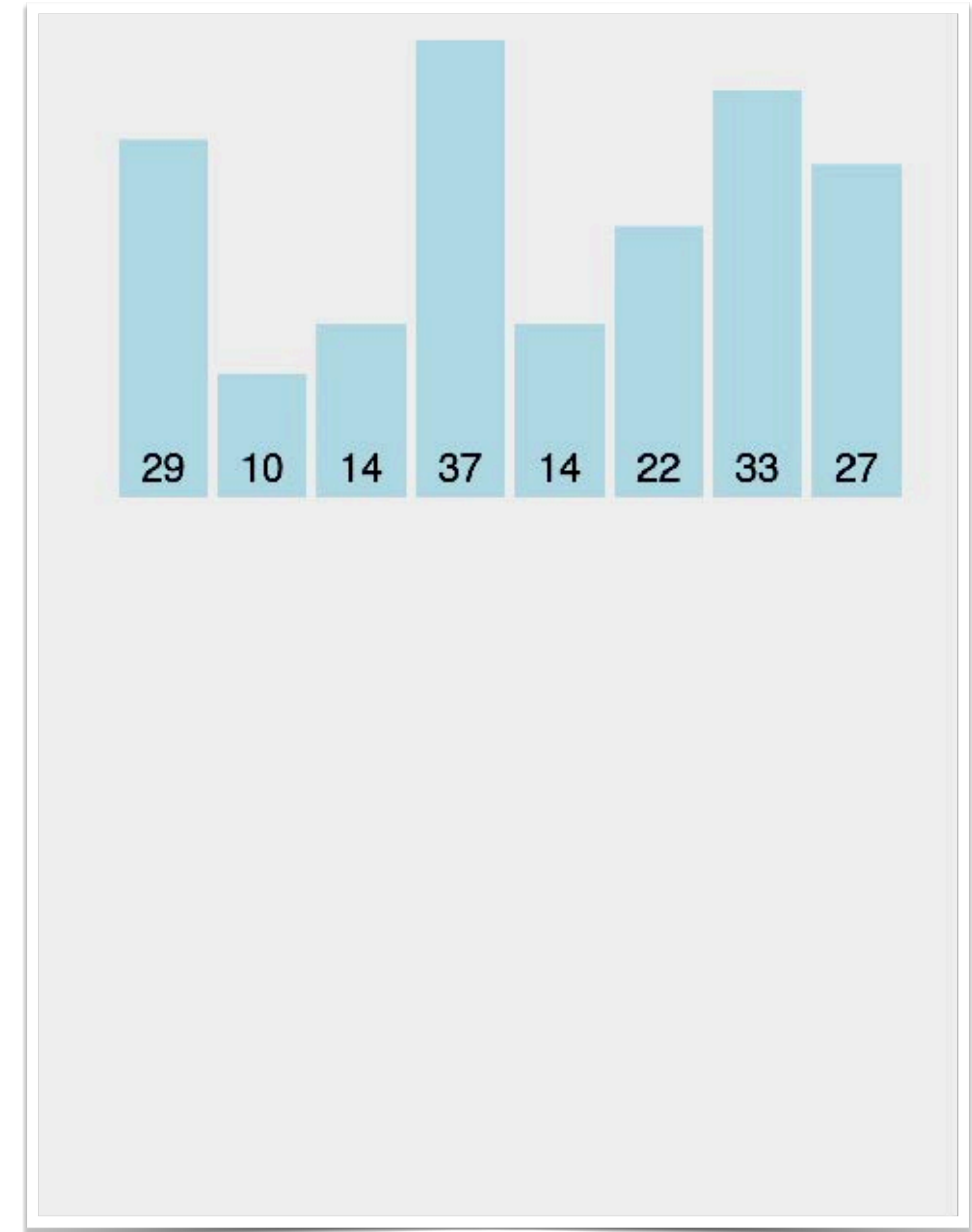
Insertion sort

- Conceptually think of sorting a handful of cards.



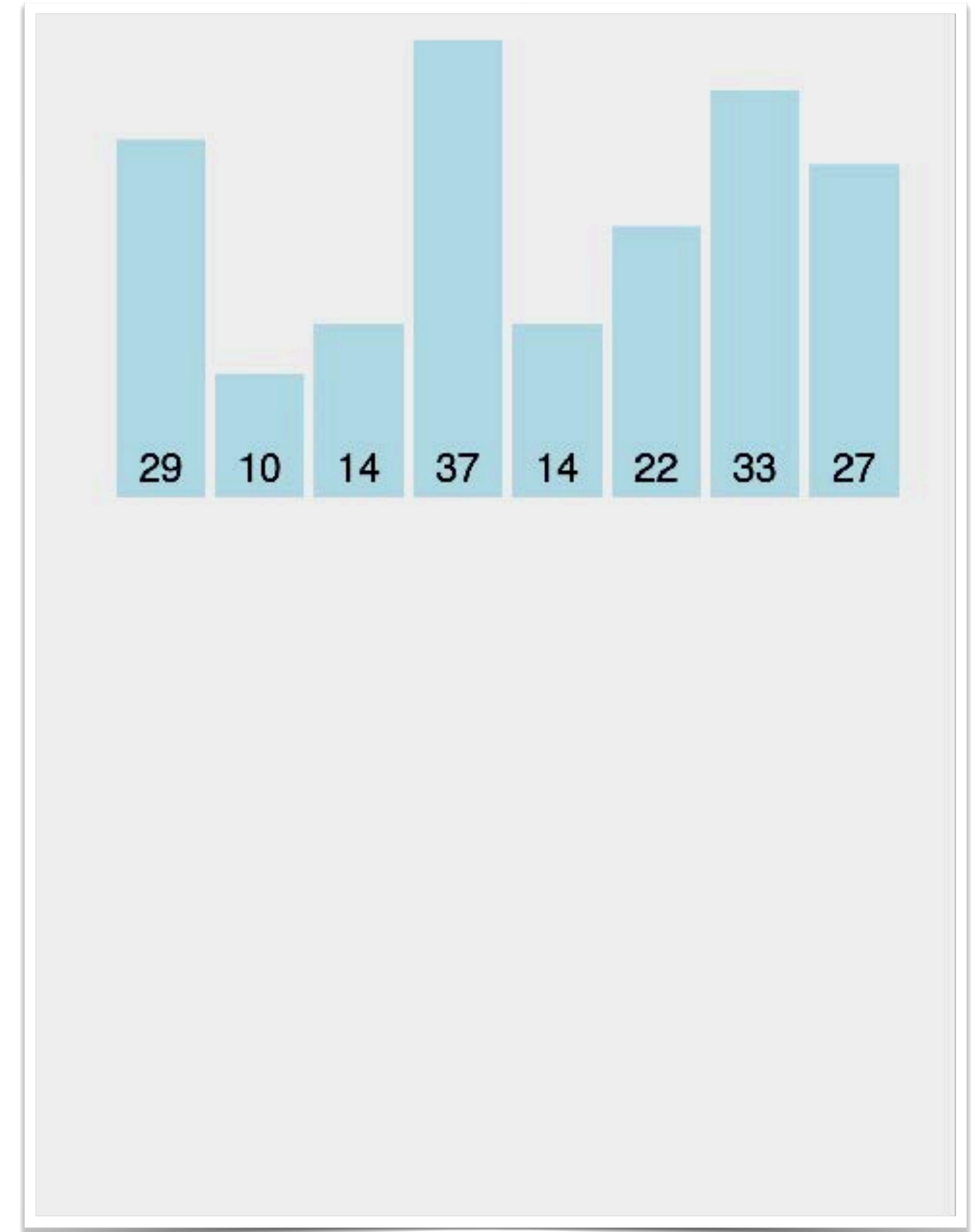
Insertion sort

- Conceptually think of sorting a handful of cards.
- Start from one end of array, assume leftmost element sorted.



Insertion sort

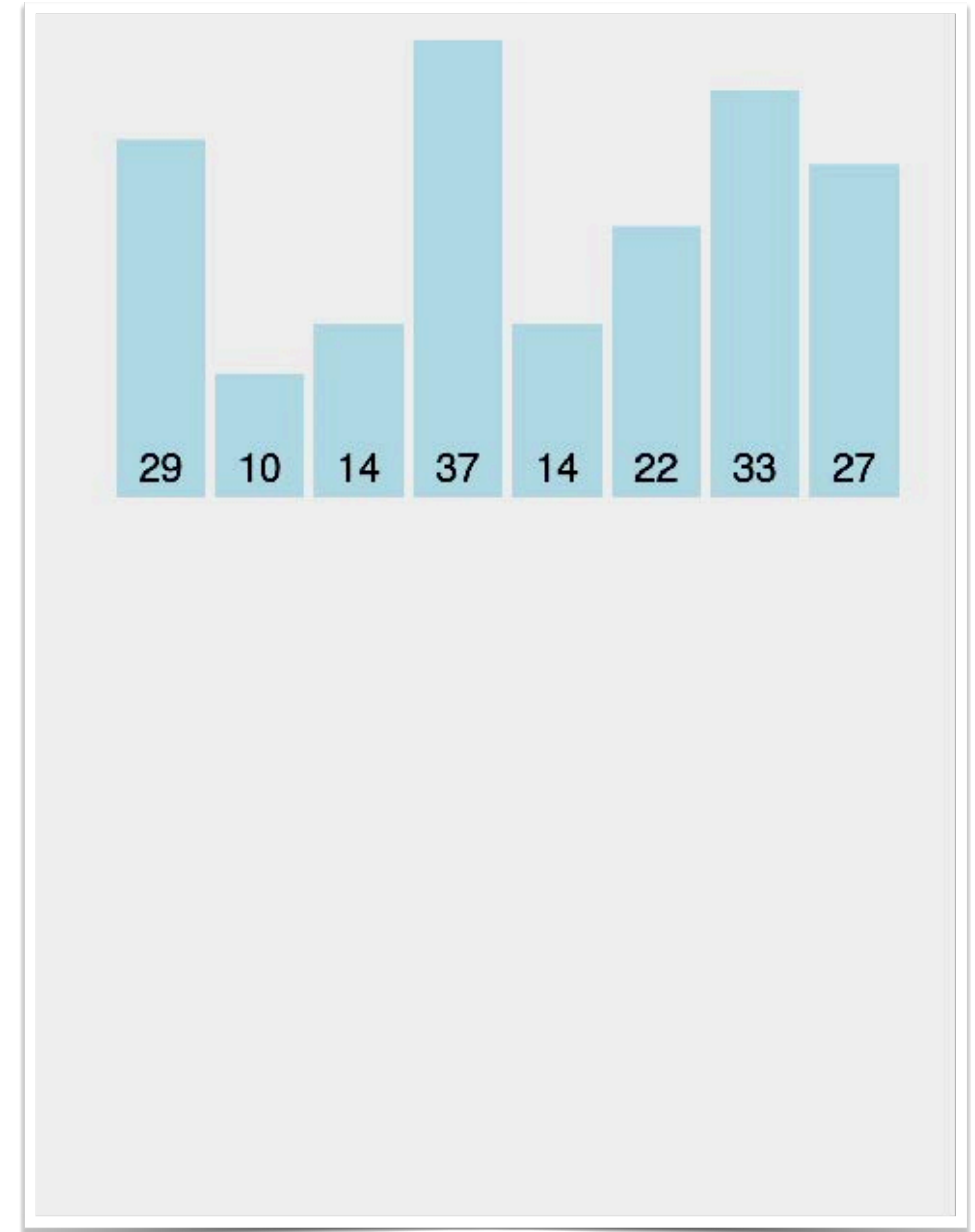
- Conceptually think of sorting a handful of cards.
- Start from one end of array, assume leftmost element sorted.
  - Pick the next card and insert it into the right place in the sorted array; moving elements if needed.



UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

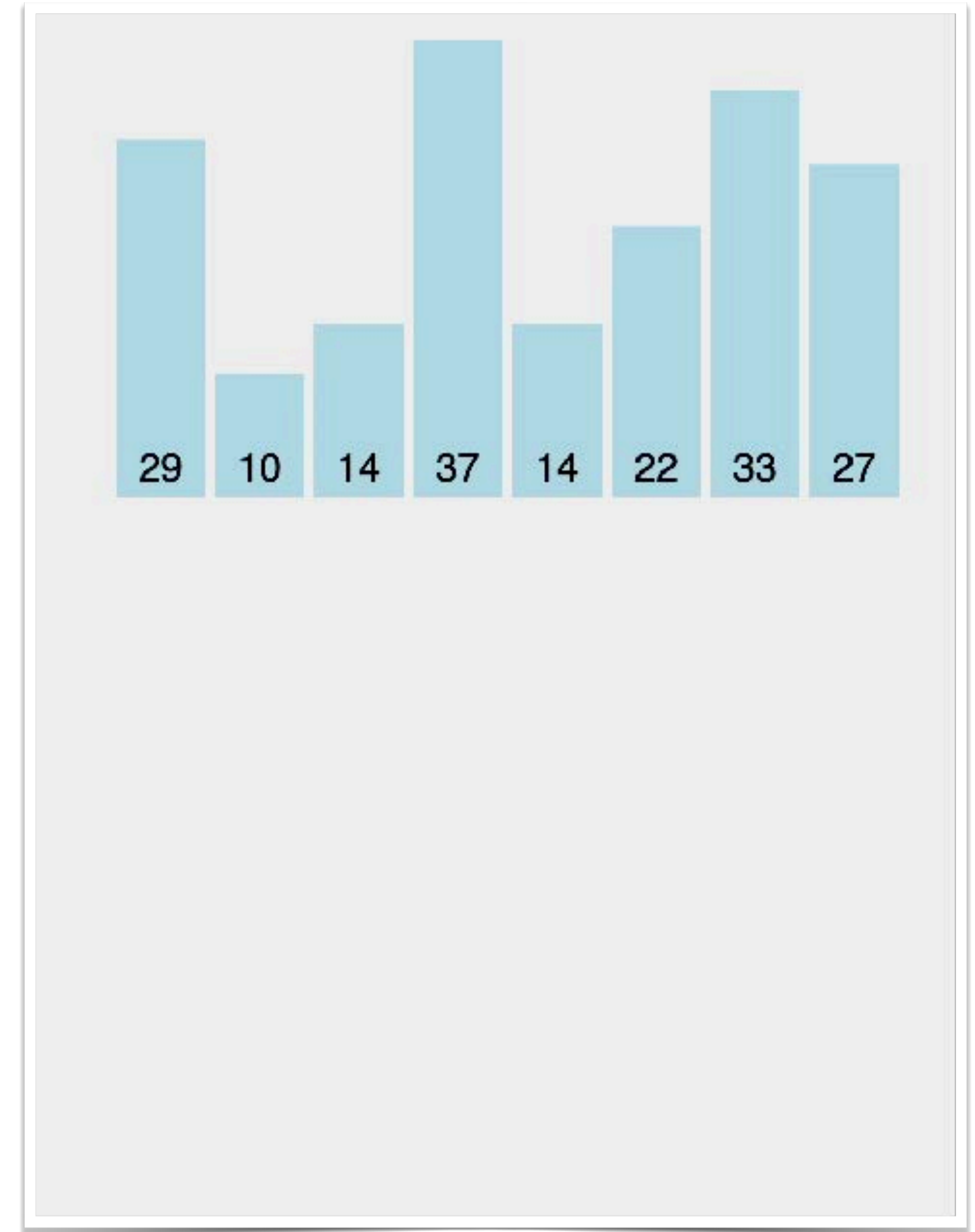
Insertion sort

- Conceptually think of sorting a handful of cards.
- Start from one end of array, assume leftmost element sorted.
  - Pick the next card and insert it into the right place in the sorted array; moving elements if needed.
  - After a single pass, array is sorted.



Insertion sort

- Conceptually think of sorting a handful of cards.
- Start from one end of array, assume leftmost element sorted.
    - Pick the next card and insert it into the right place in the sorted array; moving elements if needed.
    - After a single pass, array is sorted.

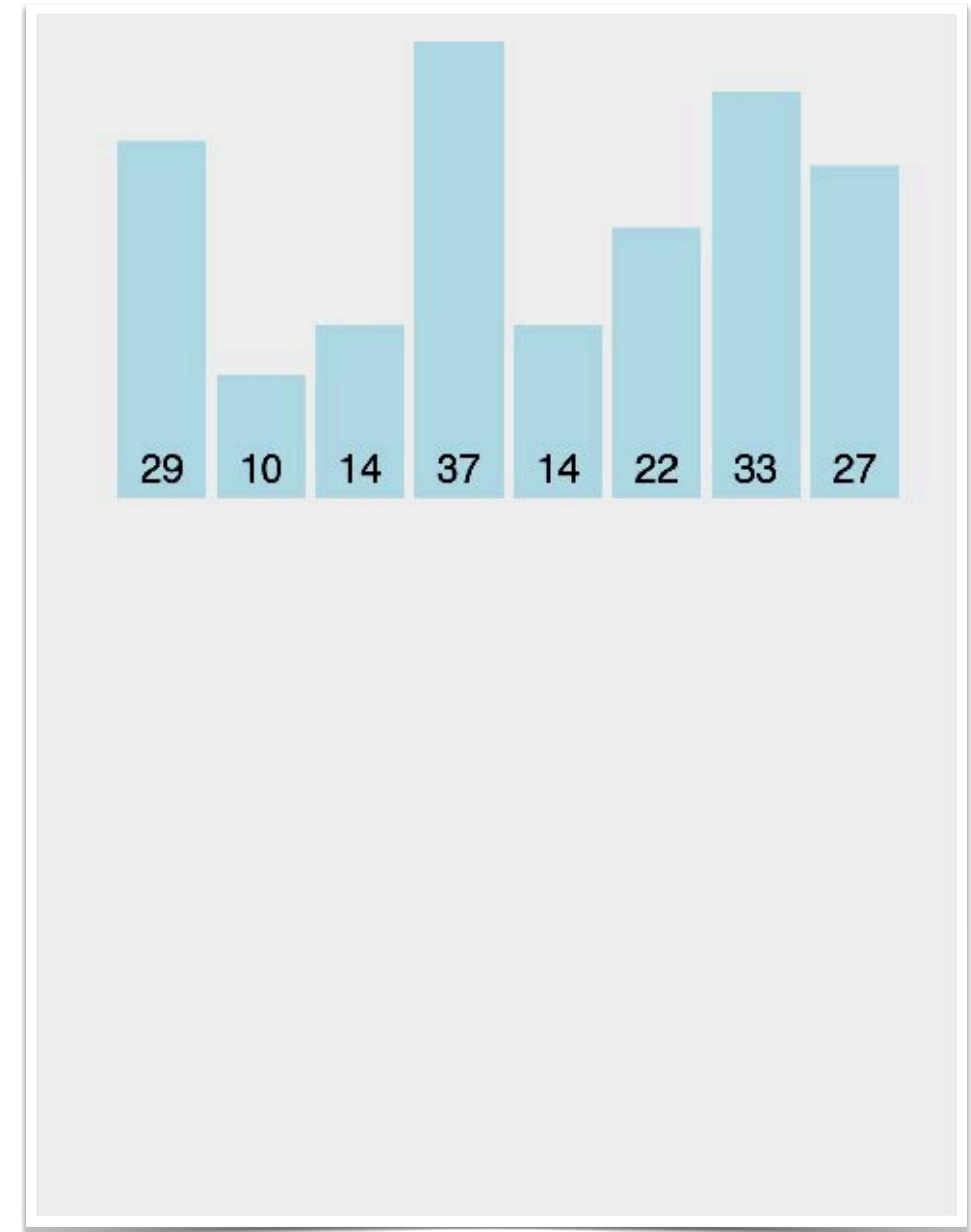


Insertion sort

```
void insertion_sort(int arr[], int n){
   for (int i = 1; i < n; i++){

      /* Insert list[i] into a sorted sublist list[0..i-1] so that
         list[0..i] is sorted. */

      int current = arr[i];
      int k;

      for (k = i - 1; __________________________; k--)
       // Move elements one spot over
        _________________ = _________________;

      // Insert the current element into list[k+1]
      arr[k + 1] = current;
   }
 }
```

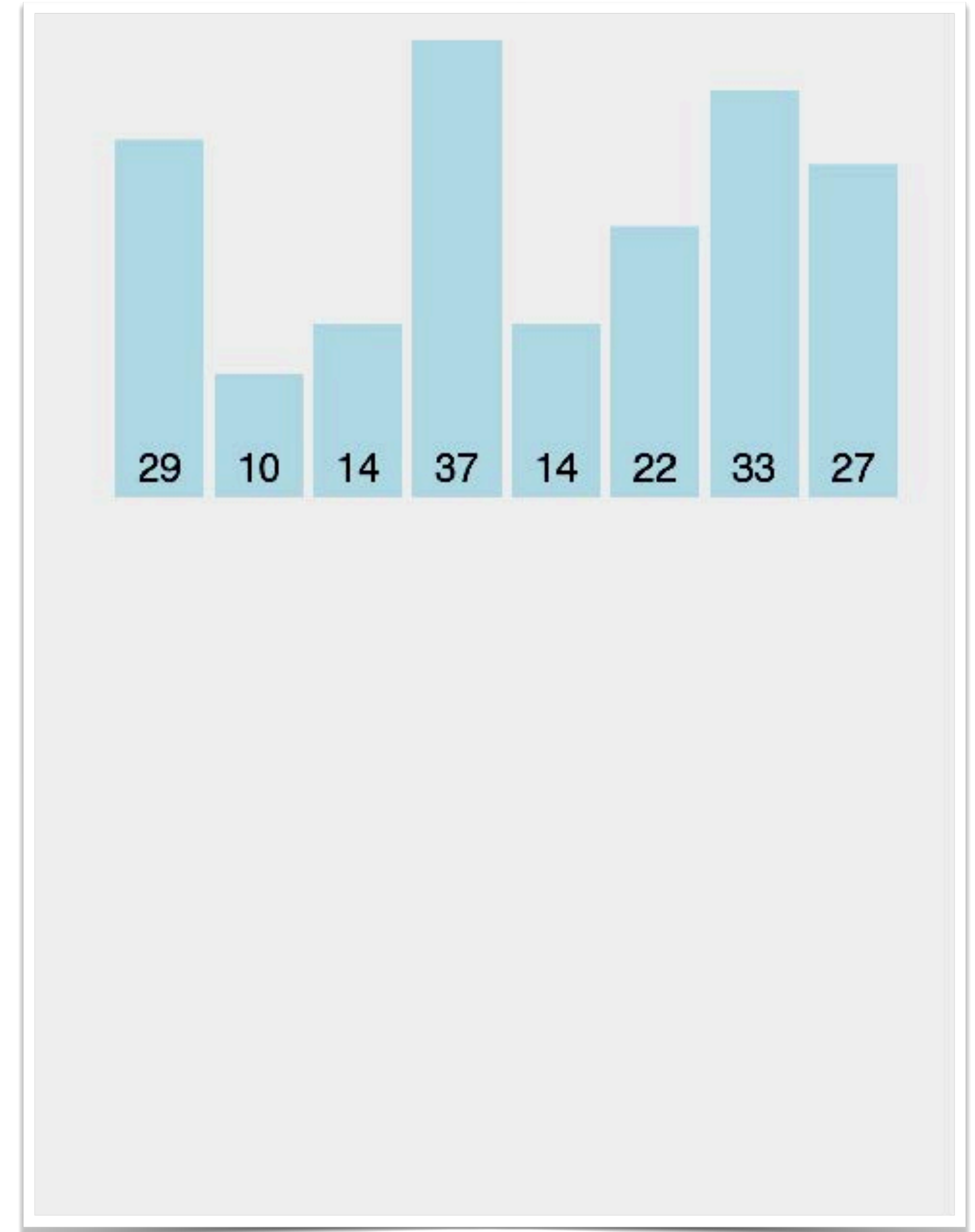

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Bubble sort



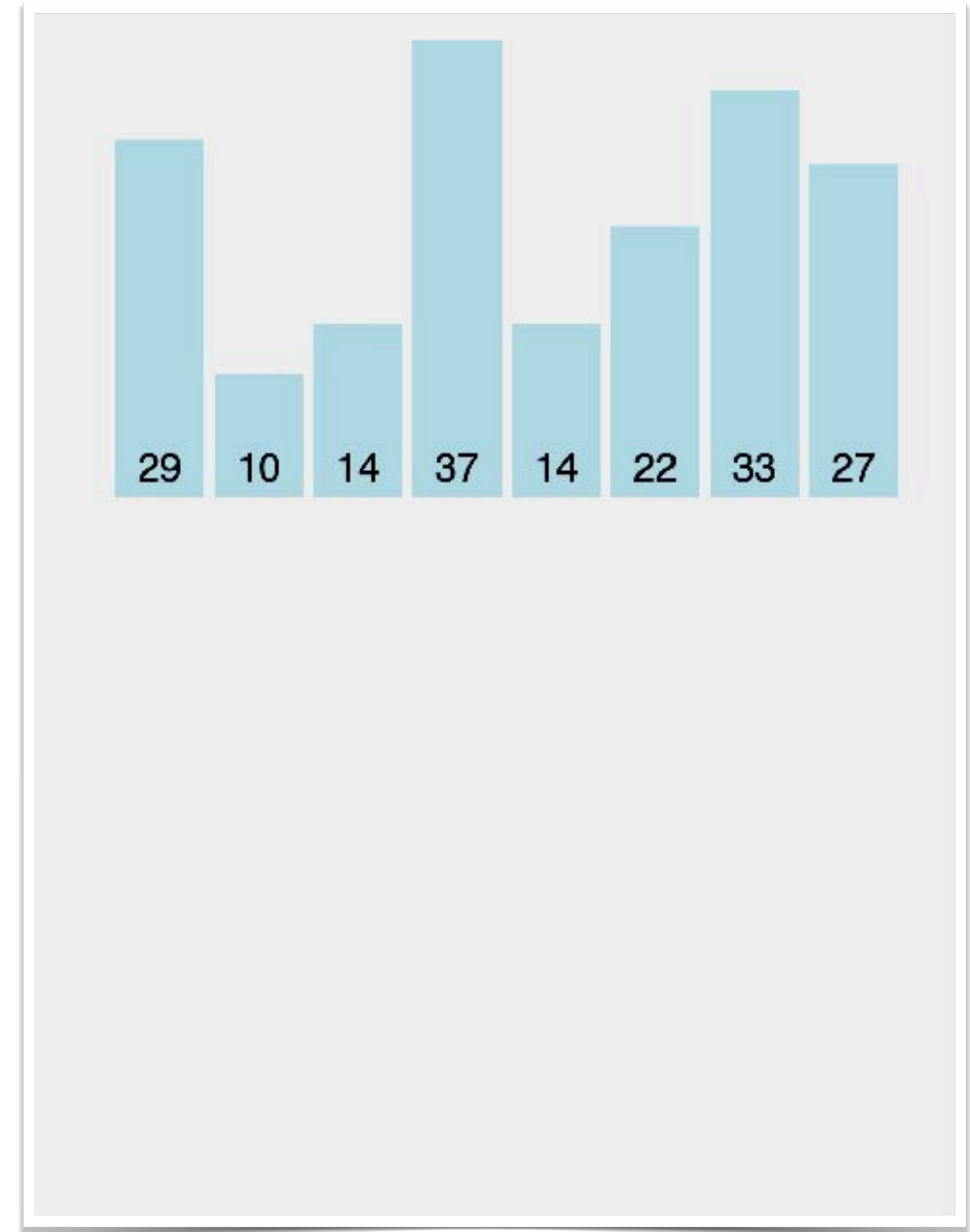
Bubble sort

- One of the more naive sort algorithms with poor performance.



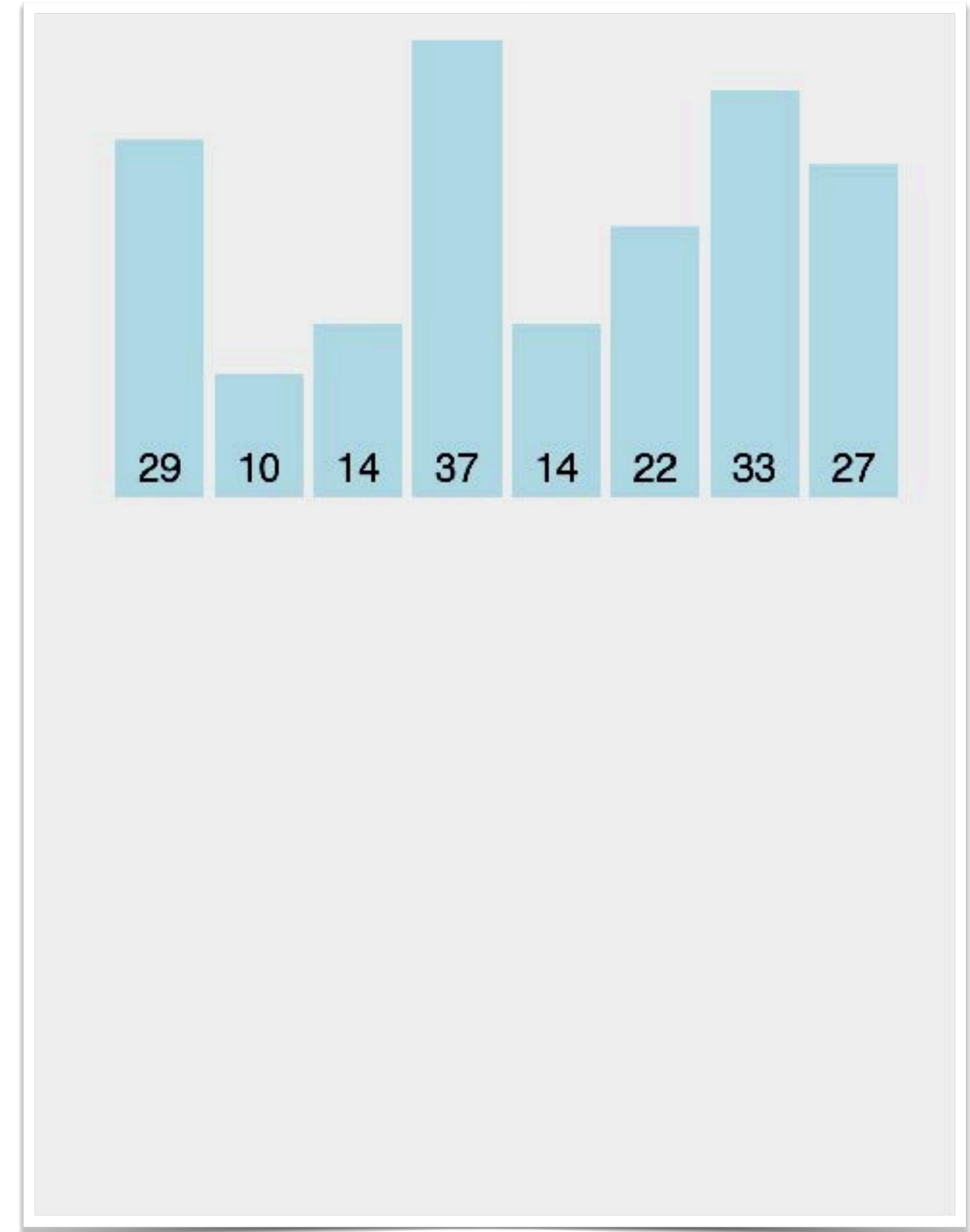
Bubble sort

- One of the more naive sort algorithms with poor performance.
- Iteratively make passes over the array



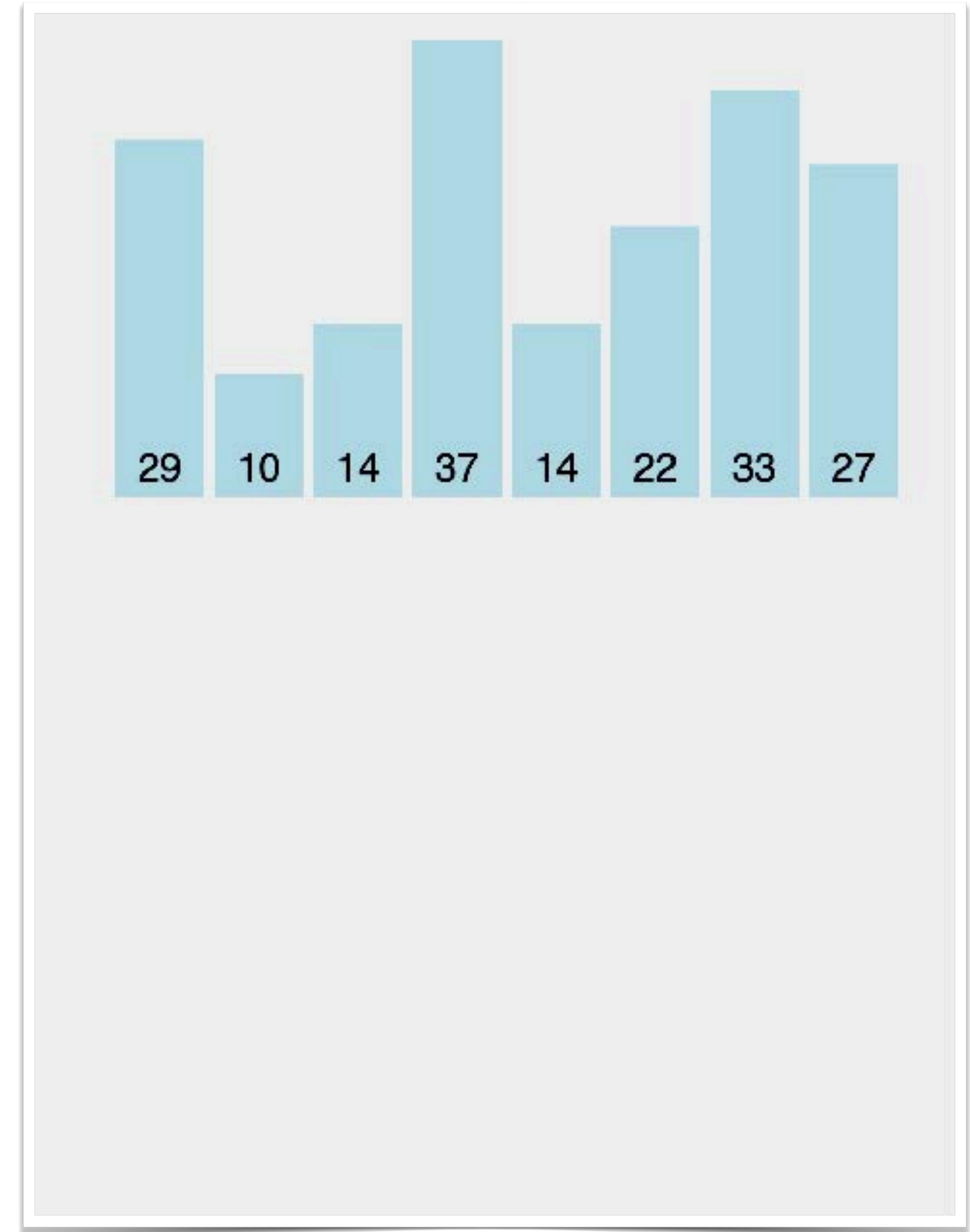
Bubble sort

- One of the more naive sort algorithms with poor performance.
- Iteratively make passes over the array
  - Comparing adjacent pairs & swapping if not in order until …



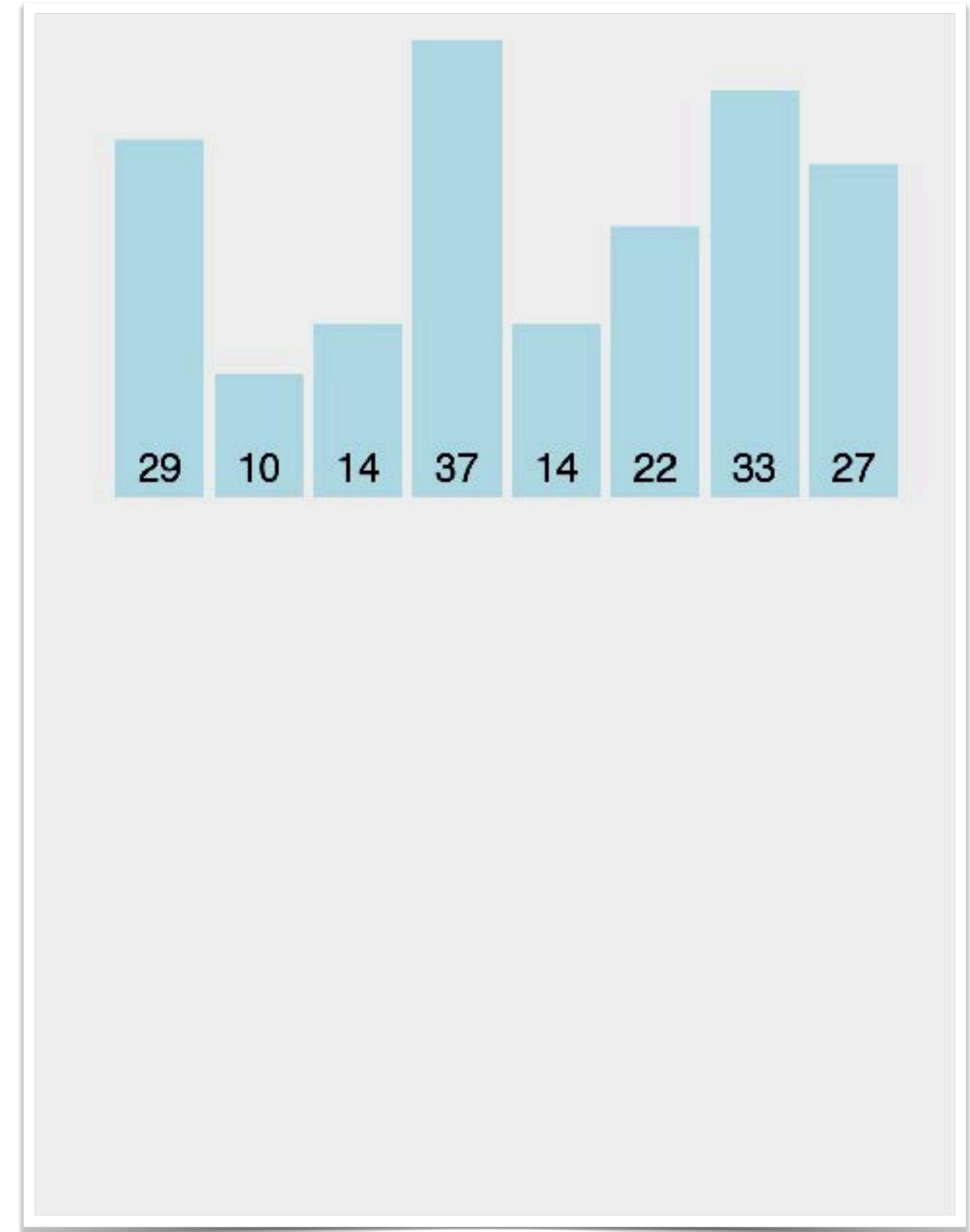
Bubble sort

- One of the more naive sort algorithms with poor performance.
- Iteratively make passes over the array
    - Comparing adjacent pairs & swapping if not in order until …
    - No more swaps are made.



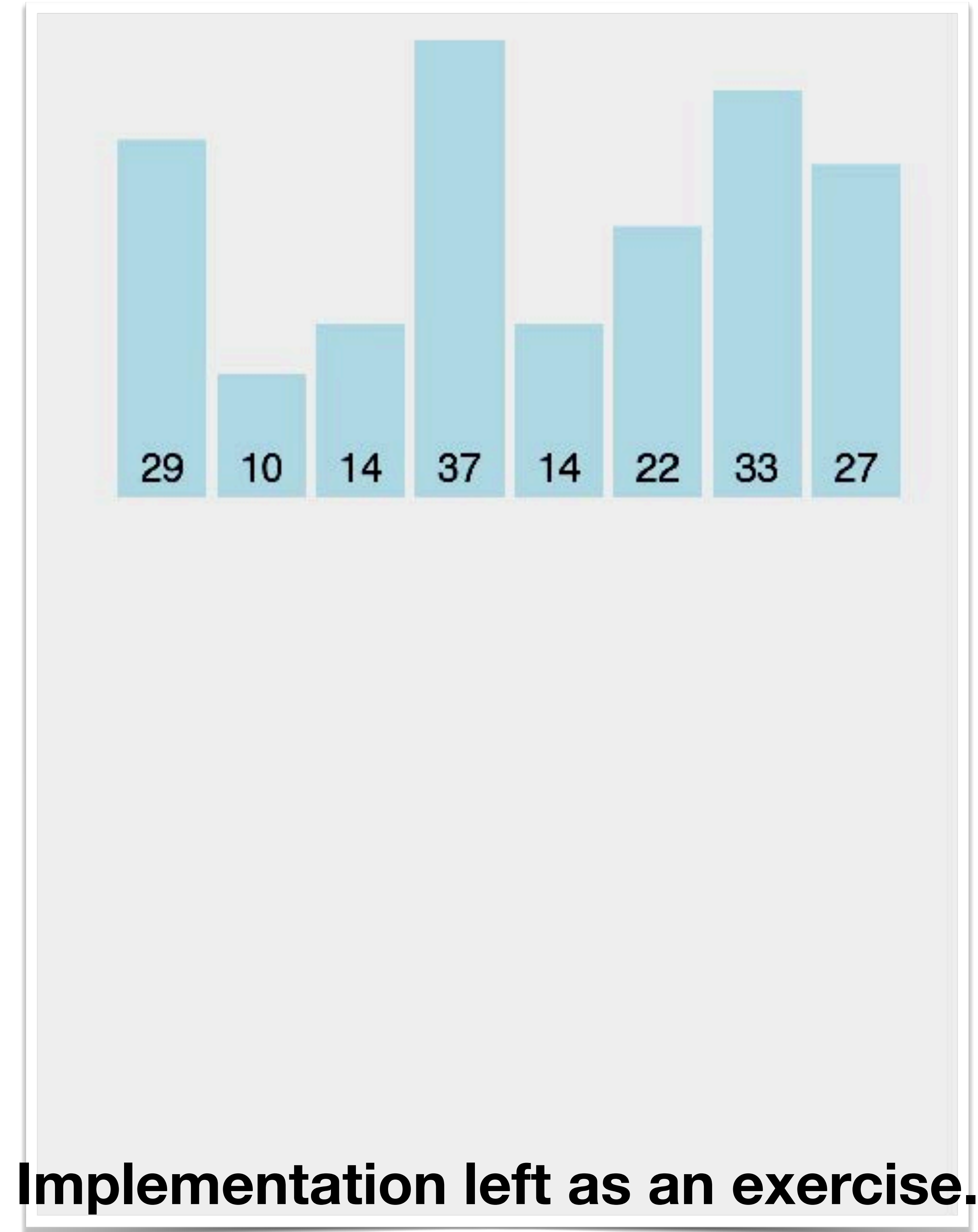
Bubble sort

- One of the more naive sort algorithms with poor performance.
- Iteratively make passes over the array
    - Comparing adjacent pairs & swapping if not in order until …
    - No more swaps are made.



Bubble sort

- One of the more naive sort algorithms with poor performance.
- Iteratively make passes over the array
    - Comparing adjacent pairs & swapping if not in order until …
    - No more swaps are made.



Implementation left as an exercise.

Quick sort

Quick sort

- One of the more faster sorting algorithms.

Quick sort

- One of the more faster sorting algorithms.
- Key idea: choose a pivot element; then …

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Quick sort

- One of the more faster sorting algorithms.
- Key idea: choose a pivot element; then …
    - Move all elements greater than pivot to right of it and smaller than pivot to left of it.

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Quick sort

- One of the more faster sorting algorithms.
- Key idea: choose a pivot element; then …
  - Move all elements greater than pivot to right of it and smaller than pivot to left of it.
  - Subdivide & repeat (recursive)

Quick sort

- One of the more faster sorting algorithms.
- Key idea: choose a pivot element; then …
    - Move all elements greater than pivot to right of it and smaller than pivot to left of it.
    - Subdivide & repeat (recursive)
- Many varieties exist; this course cannot cover them all.

Quick sort

- One of the more faster sorting algorithms.
- Key idea: choose a pivot element; then …
    - Move all elements greater than pivot to right of it and smaller than pivot to left of it.
    - Subdivide & repeat (recursive)
- Many varieties exist; this course cannot cover them all.
    - How to pick pivot?

Quick sort

- One of the more faster sorting algorithms.
- Key idea: choose a pivot element; then …
  - Move all elements greater than pivot to right of it and smaller than pivot to left of it.
  - Subdivide & repeat (recursive)
- Many varieties exist; this course cannot cover them all.
  - How to pick pivot?
    - First, last, mid, random, etc.

Quick sort

- One of the more faster sorting algorithms.
- Key idea: choose a pivot element; then …
  - Move all elements greater than pivot to right of it and smaller than pivot to left of it.
  - Subdivide & repeat (recursive)
- Many varieties exist; this course cannot cover them all.
  - How to pick pivot?
    - First, last, mid, random, etc.
  - Recursive vs. iterative.

Quick sort

- One of the more faster sorting algorithms.
- Key idea: choose a pivot element; then …
  - Move all elements greater than pivot to right of it and smaller than pivot to left of it.
  - Subdivide & repeat (recursive)

- Many varieties exist; this course cannot cover them all.
  - How to pick pivot?
    - First, last, mid, random, etc.
  - Recursive vs. iterative.
- Main point: understand one variety and understand it well.

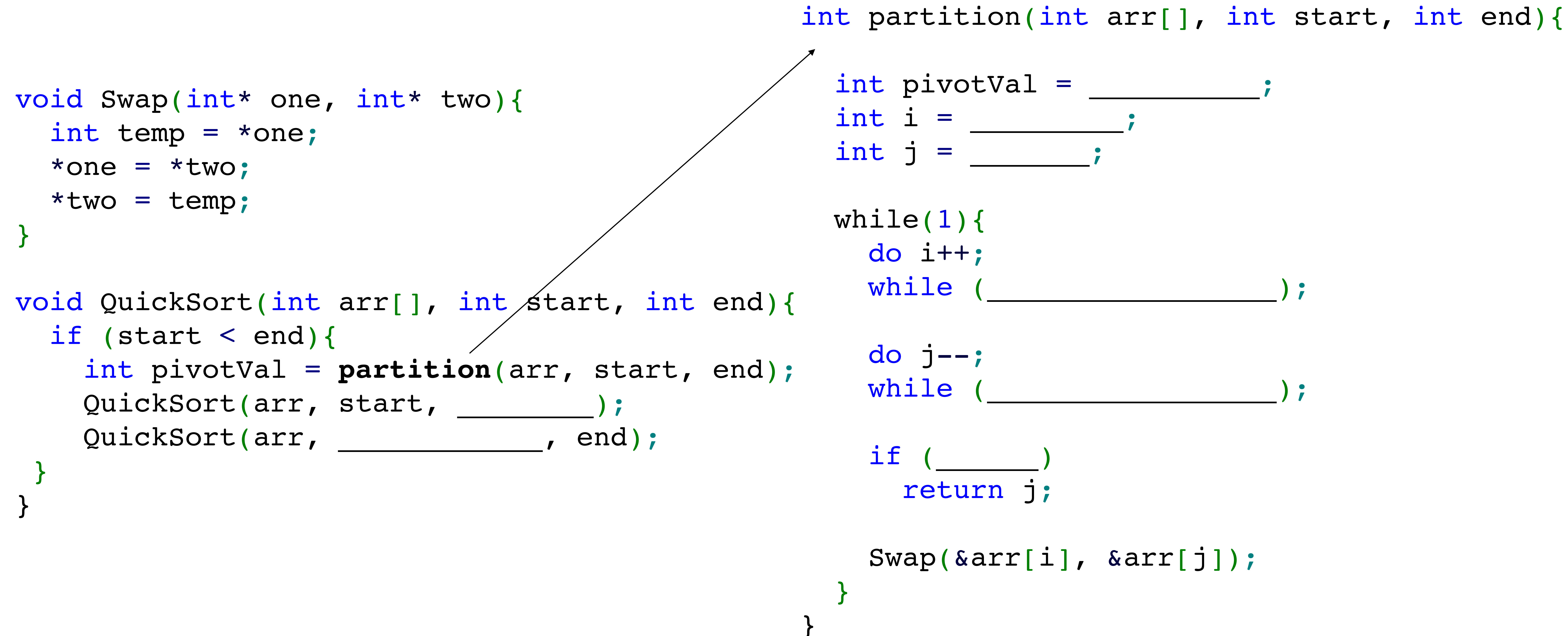
ECE 220

A special on Quicksort: Lecture x000B+

PDF-Link

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN

Implementation



Check Gitlab for reference material: https://gitlab.engr.illinois.edu/itabrah2/ece220-sp24

UNIVERSITY OF ILLINOIS URBANA-CHAMPAIGN