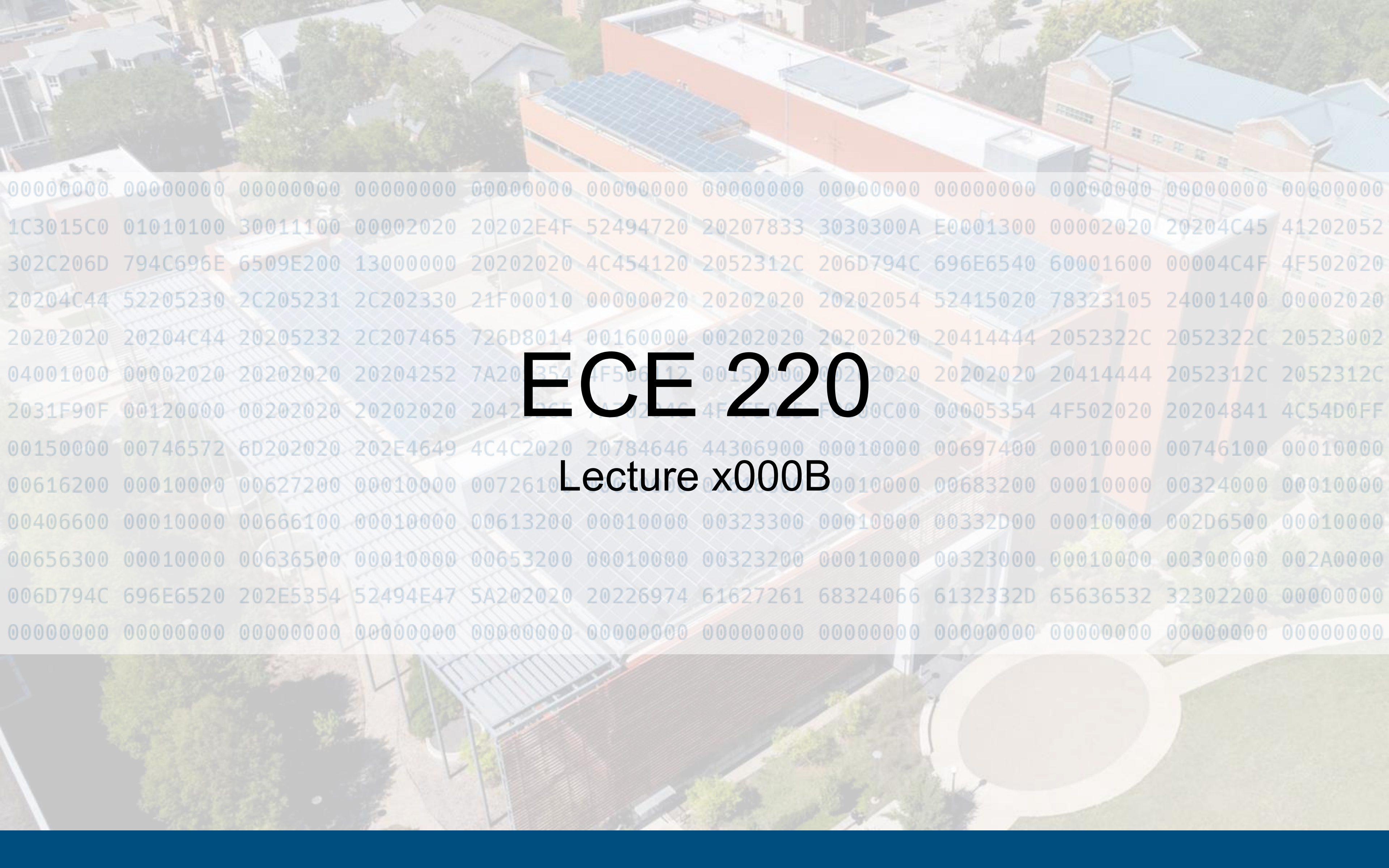




ECE 220

Lecture x000B

Recap

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

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

More 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

One dimensional array

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

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

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

How to calculate offset?

...	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]

More than 2D?

- C allows creating arrays with multiple dimensions.
- Example: Here is a three dimensional array where the first dimension has size `x`, the second dimension has size `y` and last dimension has size `z`.

```
int arr3d[x][y][z];
```

- **Question:** How will `arr3d[4][3][2]` be stored in memory?
 - Hint 1: *Last index varies fastest.*
 - Hint 2: Element `arr3d[x-1][y-1][z-1]` will be bottom most.

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}};` ?

Exercise

Write a C function that given a matrix `mat` of size `nr × nc` and another matrix `tr_mat` of size `nc × nr` 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

Exercise – matrix transpose

Write a C function that given a matrix `mat` of size `nr × nc` and another matrix `tr_mat` of size `nc × nr` 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!

Today: Lesson objectives

- Understand and be able to implement some common array operations
 - **Search:** Linear and binary searches
 - **Sorts:** Insertion, selection, bubble and time permitting quick sorts.

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;  
    }  
    _____;  
}
```

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 – C code

```
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

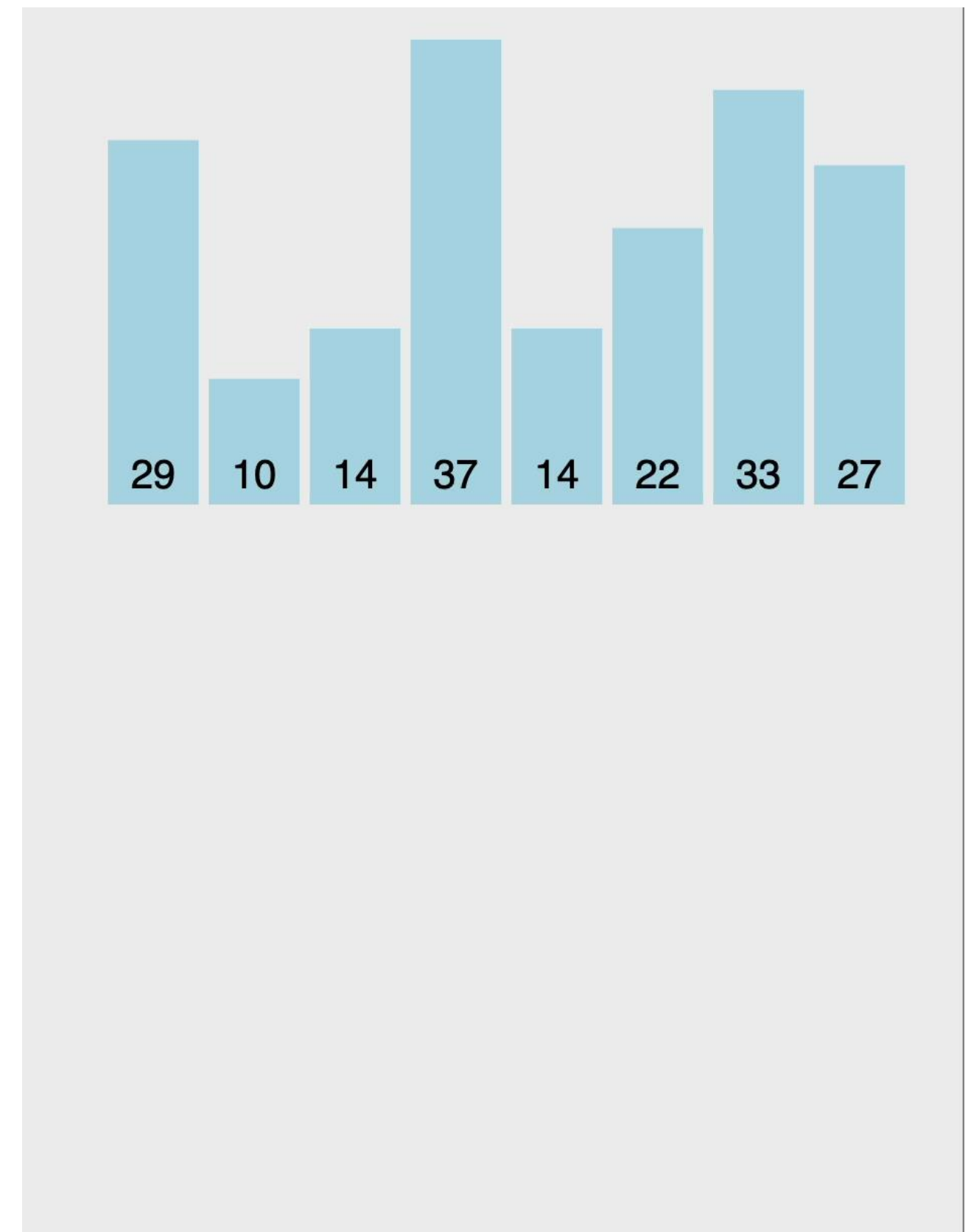
        // Logic to focus search on left or right of mid
        if (key == arr[mid])
            return mid;
        else if (key < arr[mid])
            high = _____;
        else
            low = _____;
    }
    return -1; // Loop exited, element not present.
}
```

Sorting

- 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

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

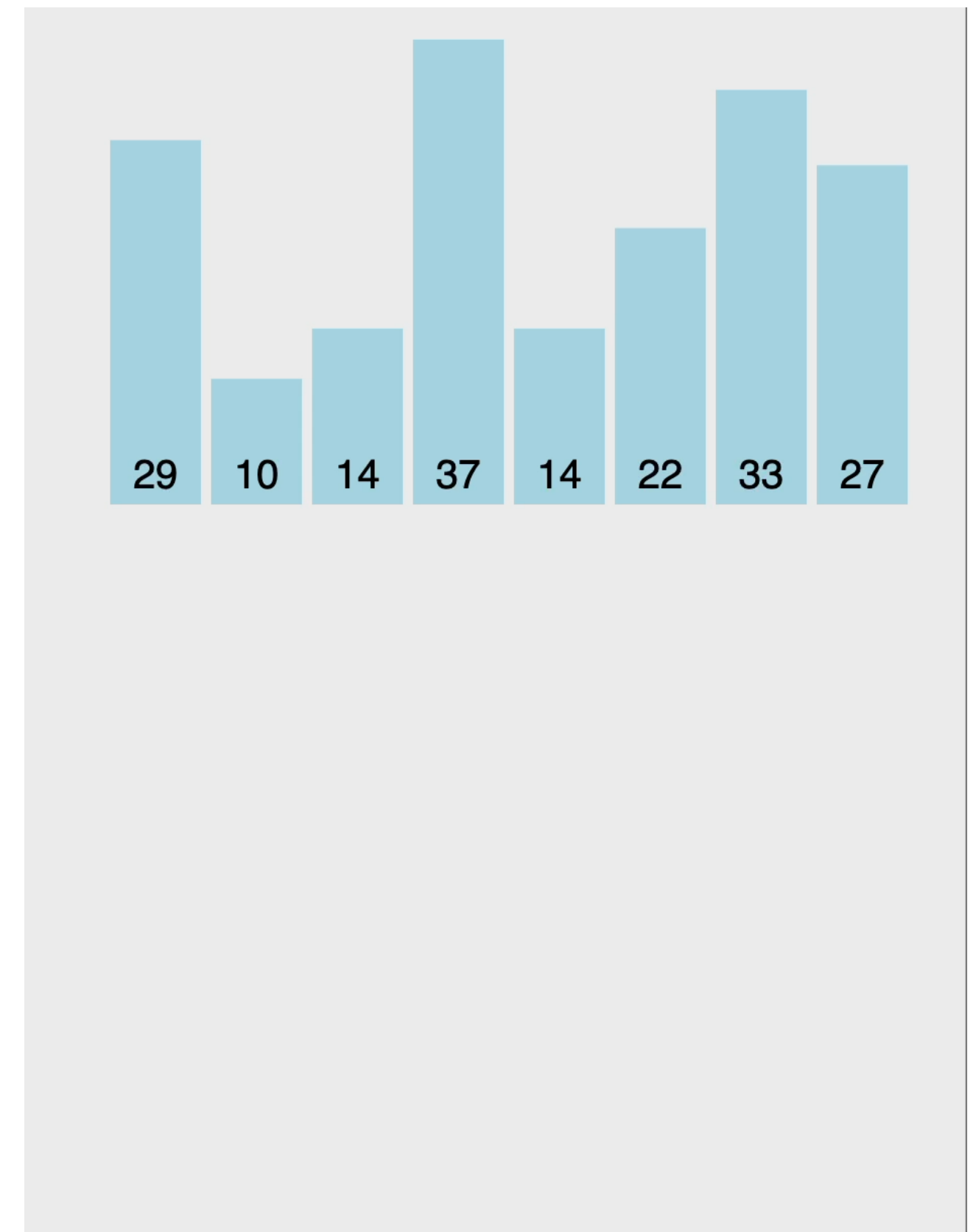


Selection sort – C code

```
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;  
        }  
    }  
}
```

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 – C code

```
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. */

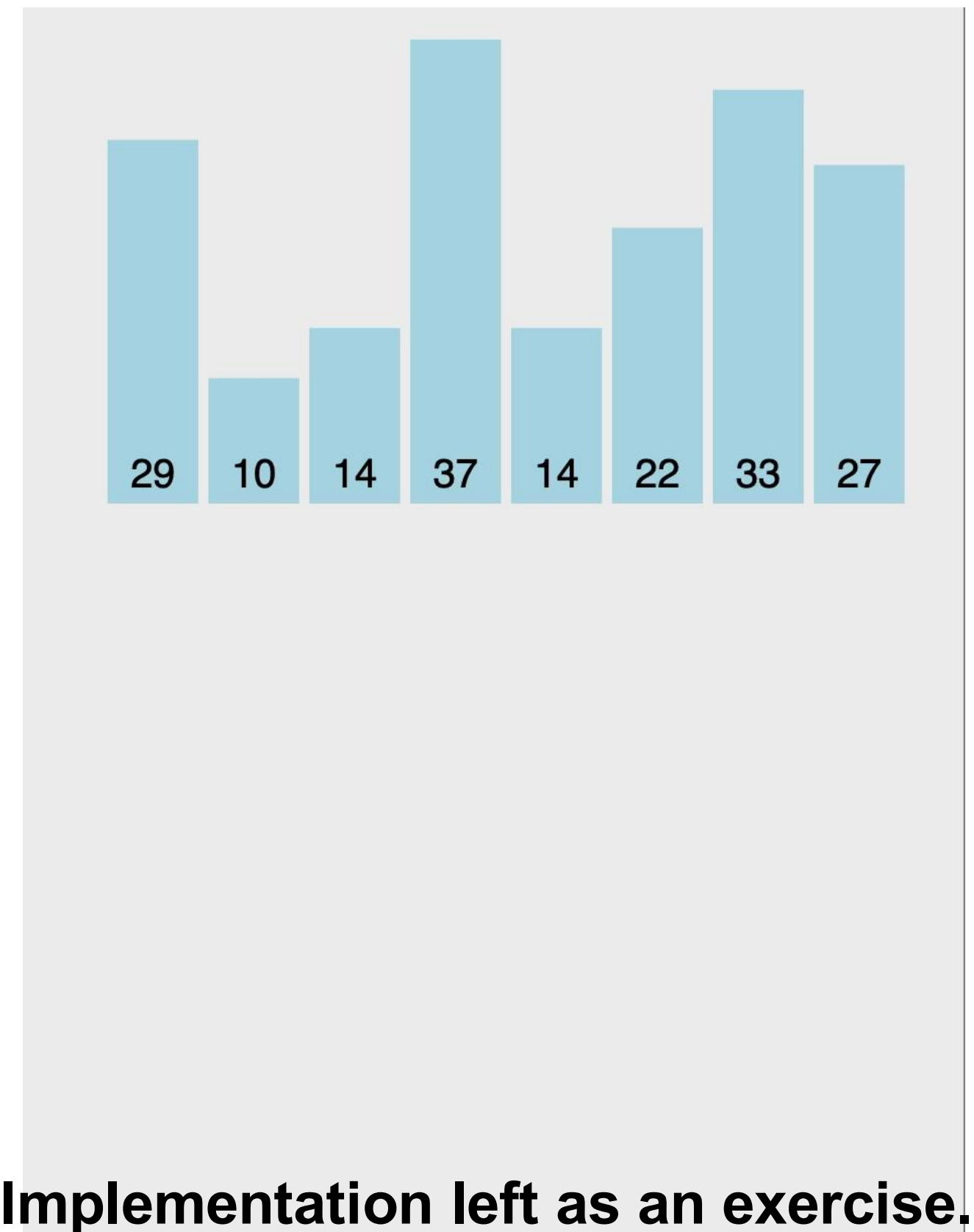
        int current = arr[i];
        int k;

        for (k = i - 1; _____; k--){
            // Move elements one spot over
            _____ = _____;

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

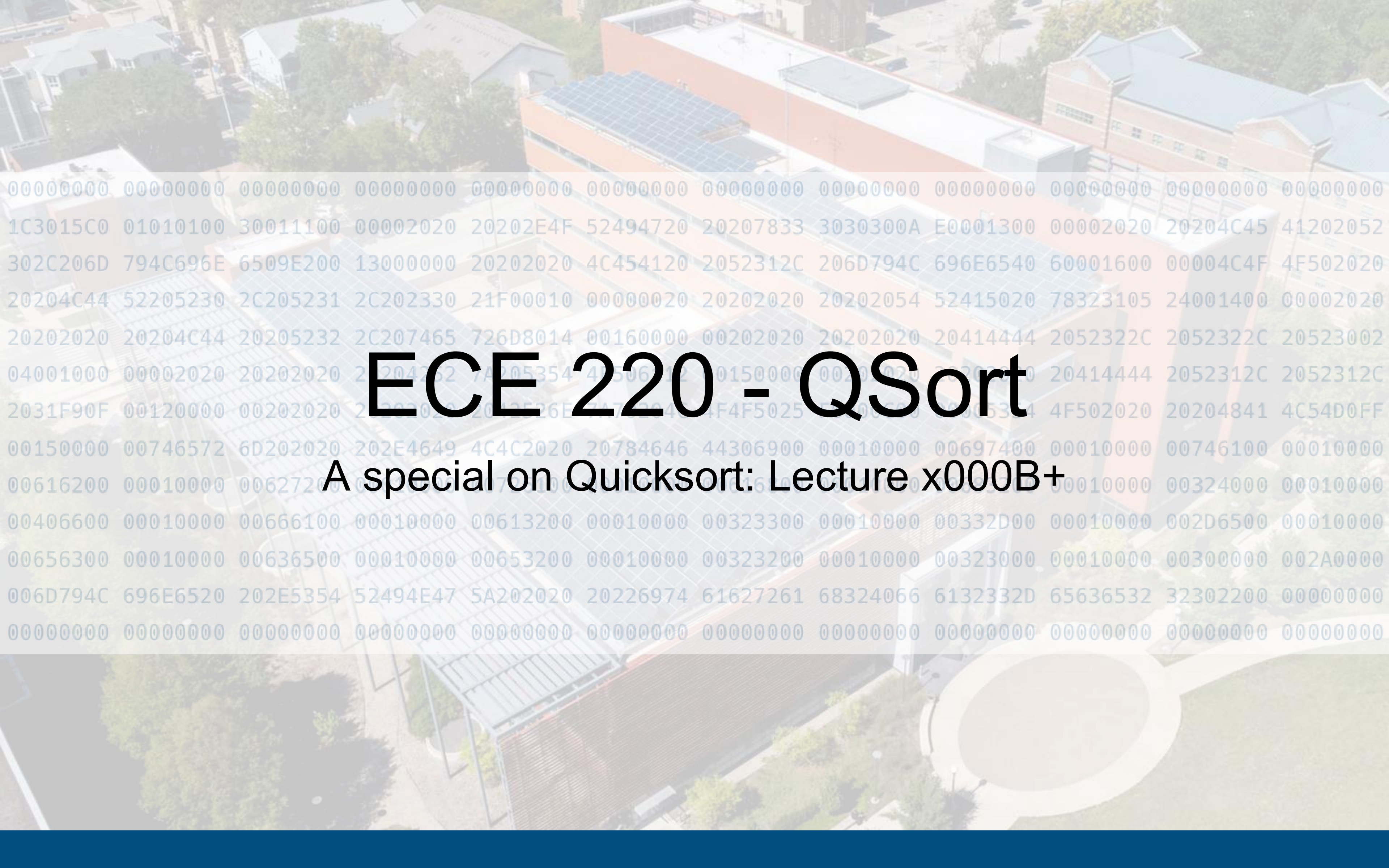
Bubble sort

- 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.



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
- 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 - QSort

A special on Quicksort: Lecture x000B+