CS 106 INTRODUCTION TO DATA STRUCTURES

SPRING 2020

PROF. SARA MATHIESON

HAVERFORD COLLEGE

ADMIN

- **Welcome prospective students!**

- **Lab 6 due TODAY**
- **Lab 7 posted TODAY (due next Thursday)**

- **Video on (if your internet will accommodate that)**

- **Let me know if you will be missing class or lab**

- **Lab 3 graded** − note many did not finish. I will down-weight Lab 3 given a positive trajectory

Remind me to RECORD

REVISED TA/OFFICE HOURS

Sunday 7-9pm (Juvia)

Monday 8-midnight (Steve)

Tuesday 11:30-12:30pm (Lizzie)

Tuesday 4:30-6pm (Sara)

Wednesday 8-9pm (Steve)

Today/Tomorrow

Thursday 11:30-12:30pm (Lizzie)

Thursday 9-11pm (Will)

Friday 8-10pm (Gareth)

Saturday 4-6pm (Will)

Saturday 8-10pm (Gareth)

APR 14 OUTLINE

- **Finish hash tables**
  - Applications: document classification, spellcheck
  - Other probing strategies
- **Deduplication**
- **Sorting**
  - Quicksort
  - Radix sort

APR 14 OUTLINE

- **Finish hash tables**
  - Applications: document classification, spellcheck
  - Other probing strategies
- **Deduplication**
- **Sorting**
  - Quicksort
  - Radix sort

GROUP EXERCISE (LAST TIME)

Think of the high-level algorithm first, then pseudocode. Assume you can easily read in words from the user or from a document.

1) **Word frequencies**: use a hash table to count the frequencies of each word in a document. This is very useful for document classification, which is (in part) how Google search works!

2) **Spellchecker**: assume you have a HashSet of "correctly" spelled words. Given a word from the user, how could you tell if it is spelled correctly? How could you go about suggesting close alternatives if it's misspelled?

DOCUMENT CLASSIFICATION

Goal: count word frequencies in a document using a dictionary

```
String[] document = {"two","cup","flour","one","cup",
        "butter","one","cup","sugar"};
```

DOCUMENT CLASSIFICATION

Goal: count word frequencies in a document using a dictionary

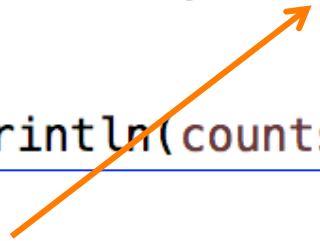
```
String[] document = {"two","cup","flour","one","cup",
        "butter","one","cup","sugar"};

HashMap<String, Integer> counts = new HashMap<String, Integer>();
```

Key is a String (word)
Value is an Integer (count)

DOCUMENT CLASSIFICATION

Goal: count word frequencies in a document using a dictionary

```
String[] document = {"two","cup","flour","one","cup",
        "butter","one","cup","sugar"};

HashMap<String, Integer> counts = new HashMap<String, Integer>();

for (String word : document) {




    counts.put(word, counts.get(word) + 1);
}

System.out.println(counts);
```



"put" is like our "insert"
Increment the previous count by 1

Will this work?
Press "yes" or "no" button!

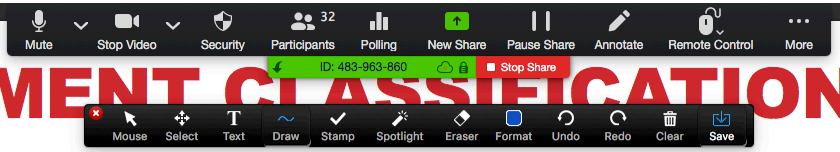
DOCUMENT CLASSIFICATION

Goal: count word frequencies in a document using a dictionary

```
String[] document = {"two","cup","flour","one","cup",
        "butter","one","cup","sugar"};

HashMap<String, Integer> counts = new HashMap<String, Integer>();

for (String word : document) {

    if (!counts.containsKey(word)) {
        counts.put(word, 0);
    }
    counts.put(word, counts.get(word) + 1);
}

System.out.println(counts);
```

"get" will throw an error if word is not in the dictionary
Need to initialize new words (not the only way)

ANOTHER WAY: IF/ELSE



DOCUMENT CLASSIFICATION

Goal: count word frequencies in a document using a dictionary

```
String[] document = {"two","cup","flour","one","cup",
        "butter","one","cup","sugar"};

HashMap<String, Integer> counts = new HashMap<String, Integer>();

for (String word : document) {

    if (!counts.containsKey(word)) {
        counts.put(word, 0);
    } else {
        counts.put(word, counts.get(word) + 1);
    }
}

System.out.println(counts);
```

"get" will throw an error if word is not in the dictionary
Need to initialize new words (not the only way)

DOCUMENT CLASSIFICATION

Goal: count word frequencies in a document using a dictionary

```
String[] document = {"two","cup","flour","one","cup",
        "butter","one","cup","sugar"};

HashMap<String, Integer> counts = new HashMap<String, Integer>();

for (String word : document) {

    if (!counts.containsKey(word)) {
        counts.put(word, 0);
    }
    counts.put(word, counts.get(word) + 1);
}

System.out.println(counts);
```

Output: `{butter=1, flour=1, one=2, two=1, sugar=1, cup=3}`

Example classification: cooking/recipe

DOCUMENT CLASSIFICATION

Goal: count word frequencies in a document using a dictionary

```
String[] document = {"two","cup","flour","one","cup",
        "butter","one","cup","sugar"};

HashMap<String, Integer> counts = new HashMap<String, Integer>();

for (String word : document) {

    if (!counts.containsKey(word)) {
        counts.put(word, 0);
    }
    counts.put(word, counts.get(word) + 1);
}

System.out.println(counts);
```

Output: `{butter=1, flour=1, one=2, two=1, sugar=1, cup=3}`

Example classification: cooking/recipe
Other ideas: omit frequent words, focus on nouns/verbs

SPELLCHECKER

Goal: suggest alternative words the user might have meant

```
// true words
HashSet<String> dictionary = new HashSet<String>();
dictionary.add("butter");
dictionary.add("flour");
dictionary.add("sugar");

String myWord = "flouy";
```

SPELLCHECKER

Goal: suggest alternative words the user might have meant

```
// true words
HashSet<String> dictionary = new HashSet<String>();
dictionary.add("butter");
dictionary.add("flour");
dictionary.add("sugar");

String myWord = "flouy";
```

One idea: go through all substitutions and see if they are in the dictionary

alouy
blouy
clouy
dlouy
elouy
…
floup
flouq
flour ****

APR 14 OUTLINE

- **Finish hash tables**
  - Applications: document classification, spellcheck
  - Other probing strategies
- **Deduplication**
- **Sorting**
  - Quicksort
  - Radix sort

PROBING DISTANCE

Given a hash value $h(x)$, linear probing generates $h(x)$, $h(x)+1$, $h(x)+2$, ...

Primary clustering – the bigger the cluster gets, the faster it grows

Quadratic probing – $h(x)$, $h(x)+1$, $h(x)+4$, $h(x)+9$, ...

Quadratic probing leads to secondary clustering, more subtle, not as dramatic, but still systematic

DOUBLE HASHING

Interval between probes is fixed but computed by a second hash function

Use a secondary hash function $d(k)$ to handle collisions by placing an item in the first available cell of the series
$i+jd(k)\%N, 0\leq j\leq N-1$

$d(k)\neq 0$

N must be prime

$d(k)=q-k\%q, q<N, q$ **is prime**

Extra info: great talk on dictionaries in Python!

https://www.youtube.com/watch?v=npw4s1QTmPg

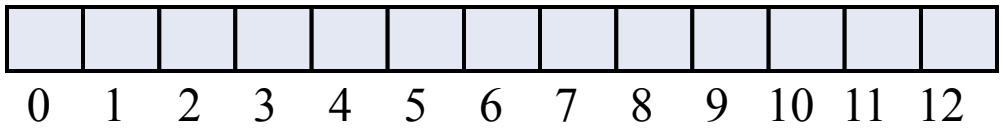
DOUBLE HASHING EXAMPLE

Double hashing:

- N = 13
- h(k) = k % 13
- d(k) = 7 – k % 7

Insert: 18, 41, 22, 44, 59, 32, 31, 73

k	$h(k)$	$d(k)$	Probe Indices		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		



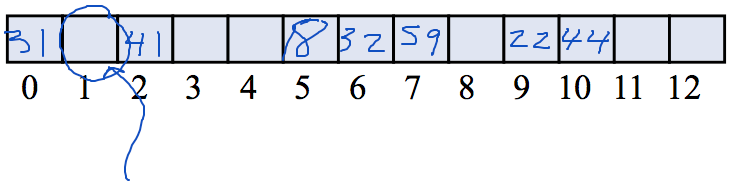
DOUBLE HASHING EXAMPLE

Double hashing:

- N = 13
- h(k) = k % 13
- d(k) = 7 − k % 7

Insert: 18, 41, 22, 44, 59, 32, 31, 73

k	h(k)	d(k)	Probe Indices
18	5	3	5
41	2	1	2
22	9	6	9
44	5	5	5, 10
59	7	4	7
32	6	3	6
31	5	4	5, 9, 0
73	8	4	8



Q: What if there is one spot left but you never find it with this strategy?
A: Make sure moduli (i.e. 13 and 7) are relatively prime (and just generally both prime), but otherwise this could definitely be an issue!

Q: Do you record the index where you put each item?
A: Unfortunately no − this is not part of what is stored in the hash table, so we have to follow the same probing procedure to "lookup" which is why it is so important that we have few collisions and a good strategy for dealing with them when they do occur.

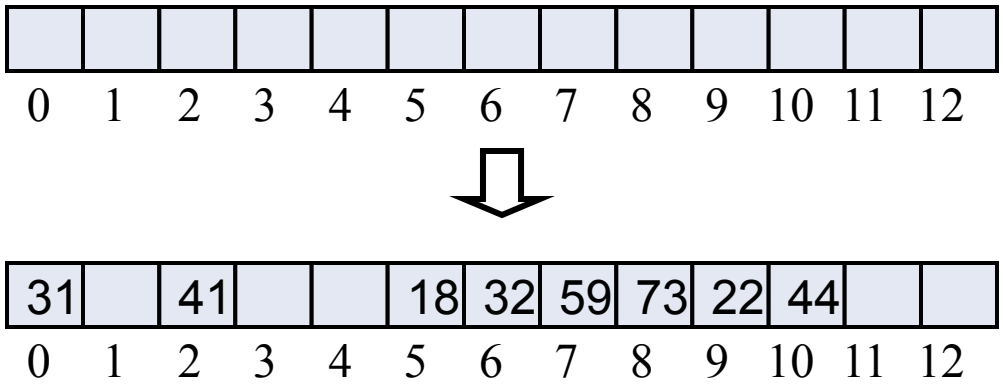
DOUBLE HASHING EXAMPLE

Double hashing:

- N = 13
- h(k) = k % 13
- d(k) = 7 – k % 7

Insert: 18, 41, 22, 44, 59, 32, 31, 73

k	$h(k)$	$d(k)$	Probe Indices
18	5	3	5
41	2	1	2
22	9	6	9
44	5	5	5, 10
59	7	4	7
32	6	3	6
31	5	4	5, 9, 0
73	8	4	8



PERFORMANCE ANALYSIS

In the worst case, searches, insertions and removals take $O(n)$ time

- when all the keys collide

The load factor $\alpha = n/N$ affects the performance of a hash table

Expected time of all operations is $O(1)$ provided α is not close to 100%

PERFORMANCE ANALYSIS

In the worst case, searches, insertions and removals take $O(n)$ time

- when all the keys collide

The load factor $\alpha = n/N$ affects the performance of a hash table

$\alpha = \frac{n}{N}$

n = # keys

N = len of array

Expected time of all operations is $O(1)$ provided α is not close to 100%

$\alpha < .75$ okay

$\alpha < .5$ ☆

HASHTABLE SIZE

Should be a prime

twice the size of max number of keys

or 1.3 times if n is very large

1/1.333 = 75% load factor

Keep track of load factor and expand (rehash) the hash table when necessary

APR 14 OUTLINE

- **Finish hash tables**
  - Applications: document classification, spellcheck
  - Other probing strategies

- **Deduplication**

- **Sorting**
  - Quicksort
  - Radix sort

DATA DEDUPLICATION

Data sets can contain multiple entries that are the same even when they shouldn't.

When this happens it:

takes up extra space

causes algorithms operating on the data to take more time

potentially causes data errors, when one copy of an entry is modified and the other isn't

DATA DEDUPLICATION

Data sets can contain multiple entries that are the same even when they shouldn't.

Goals:

1. **determine programmatically when two entries are duplicates**
2. **identify and remove duplicate entries from the data set**

DATA DEDUPLICATION

Data sets can contain multiple entries that are the same even when they shouldn't.

Goals:

1. **determine programmatically when two entries are duplicates**
2. **identify and remove duplicate entries from the data set**

Lab 7 application: voter registration

DETERMINING EQUALITY

Good deduplication requires a function that can accurately check equality between two entries.

Ideally the equality function:

returns True if two entries are the same

is robust to small errors (e.g., spelling mistakes)

returns False if any part of the entry indicates the entries are actually different

This is necessarily customized per data set!

IDENTIFYING DUPLICATES

There are a number of ways to identify duplicates in a data set:

1. compare all pairs
2. use a dictionary
3. sort the items

Which is the right method to use? Determine this by:

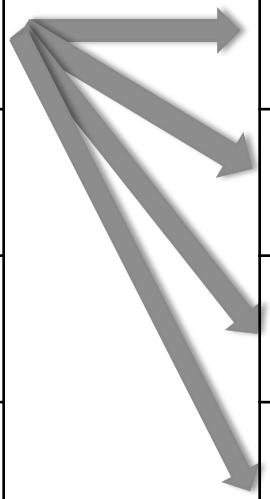
1. timing the different methods
2. formal complexity

You'll determine complexity via timing in Lab 7.

IDENTIFYING DUPLICATES – ALL PAIRS

We can identify duplicates by comparing each item in the data set to each other item in the data set.

Class	Class Name	Semester
CMSC H106B	Introduction to Data Structures	Spring 2020
CMSC H106B	Introduction to Data Structures	Spring 2020
CMSC H106B	Introduction to Data Structures	Spring 2019
CMSC H105A	Introduction to Computer Science	Fall 2016



Class	Class Name	Semester
CMSC H106B	Introduction to Data Structures	Spring 2020
CMSC H106B	Introduction to Data Structures	Spring 2020
CMSC H106B	Introduction to Data Structures	Spring 2019
CMSC H105A	Introduction to Computer Science	Fall 2016

When, based on our method, two items are found to be equal, and they aren't the same item in memory, a duplicate was found.

IDENTIFYING DUPLICATES – DICTIONARIES

We can identify duplicates by putting each item into a dictionary using a key that is unique *only* for each item that is unique. We could count the number of occurrences as the value or check for presence/absence.

Class	Class Name	Semester
CMSC H106B	Introduction to Data Structures	Spring 2020
CMSC H106B	Introduction to Data Structures	Spring 2020
CMSC H106B	Introduction to Data Structures	Spring 2019
CMSC H105A	Introduction to Computer Science	Fall 2016

Key	Value
CMSC H106B	3
CMSC H105A	1

IDENTIFYING DUPLICATES – DICTIONARIES

We can identify duplicates by putting each item into a dictionary using a key that is unique *only* for each item that is unique. We could count the number of occurrences as the value or check for presence/absence.

HashMap<String, Course>

key value row

Class	Class Name	Semester
CMSC H106B	Introduction to Data Structures	Spring 2020
CMSC H106B	Introduction to Data Structures	Spring 2020
CMSC H106B	Introduction to Data Structures	Spring 2019
CMSC H105A	Introduction to Computer Science	Fall 2016

course # + semester

Key	Value
CMSC H106B	3
CMSC H105A	1

DEDUPLICATION WITH SORTING

How can sorting help us perform deduplication?

If we sort all the items so that duplicate items are next to each other in the list, then we only need to compare neighboring items to see if they're the same!

APR 14 OUTLINE

- **Finish hash tables**
  - Applications: document classification, spellcheck
  - Other probing strategies
- **Deduplication**
- **Sorting**
  - Quicksort
  - Radix sort

HOW IMPORTANT IS SORTING IN COMPUTER SCIENCE? THIS IMPORTANT.

https://www.youtube.com/watch?v=k4RRi_ntQc8

SORTING

Sorting is used as a component of many algorithms, and is useful in its own right.

Some applications:

- Sort a list of names.
- Organize a music library.
- Display Google PageRank results.
- Find the median.
- Identify statistical outliers.
- Find duplicates in a mailing list.
- Supply chain management.
- Book recommendations on Amazon.
- Load balancing on a parallel computer.

COMPARISON METHODS

There are many sorting algorithms! Most rely on a method (or operator) that can order two items. For example, the compareTo method from the Comparable interface.

If an operator isn't built-in, than a good *key* can often be created so that a comparison can be done from the key, or a new comparison function can be created.

BUBBLE SORT

Given a list of items and the ability to compare them:

start at the beginning of the list – compare the first two items and put the “smaller” one first

keep doing this with an item and the one after it in the list until you reach the end of the list

start back at the beginning of the list and repeat

stop when you do a full pass through the list with no repeats

BUBBLE SORT - COMPLEXITY

What is the input size for this problem?

We'll call the number of *items in the list* n.

What's the right unit of work for this problem? I.e., what thing (like multiplications) do we want to count?

We'll count the number of comparisons between two items.

BUBBLE SORT – WORST CASE EXAMPLE

How long does this method of sorting take in terms of this input size *in the worst case*?

Example (swaps indicated by ->)

[5,4,3,2,1] -> [4,5,3,2,1] -> [4,3,5,2,1] -> [4,3,2,5,1] ->

BUBBLE SORT – WORST CASE EXAMPLE

How long does this method of sorting take in terms of this input size *in the worst case*?

Example (swaps indicated by ->)

[5,4,3,2,1] -> [4,5,3,2,1] -> [4,3,5,2,1] -> [4,3,2,5,1] ->

[4,3,2,1,5] -> [3,4,2,1,5] -> [3,2,4,1,5] ->

BUBBLE SORT – WORST CASE EXAMPLE

How long does this method of sorting take in terms of this input size *in the worst case*?

Example (swaps indicated by ->)

[5,4,3,2,1] -> [4,5,3,2,1] -> [4,3,5,2,1] -> [4,3,2,5,1] ->

[4,3,2,1,5] -> [3,4,2,1,5] -> [3,2,4,1,5] ->

[3,2,1,4,5] -> [2,3,1,4,5] ->

BUBBLE SORT – WORST CASE EXAMPLE

How long does this method of sorting take in terms of this input size *in the worst case*?

Example (swaps indicated by ->)

[5,4,3,2,1] -> [4,5,3,2,1] -> [4,3,5,2,1] -> [4,3,2,5,1] ->

[4,3,2,1,5] -> [3,4,2,1,5] -> [3,2,4,1,5] ->

[3,2,1,4,5] -> [2,3,1,4,5] ->

[2,1,3,4,5] ->

BUBBLE SORT – WORST CASE EXAMPLE

How long does this method of sorting take in terms of this input size *in the worst case*?

Example (swaps indicated by ->)

[5,4,3,2,1] -> [4,5,3,2,1] -> [4,3,5,2,1] -> [4,3,2,5,1] ->

[4,3,2,1,5] -> [3,4,2,1,5] -> [3,2,4,1,5] ->

[3,2,1,4,5] -> [2,3,1,4,5] ->

[2,1,3,4,5] ->

[1,2,3,4,5]

BUBBLE SORT – WORST CASE EXAMPLE

How long does this method of sorting take in terms of this input size *in the worst case*?

Example (swaps indicated by ->)

[5,4,3,2,1] -> [4,5,3,2,1] -> [4,3,5,2,1] -> [4,3,2,5,1] ->

[4,3,2,1,5] -> [3,4,2,1,5] -> [3,2,4,1,5] ->

[3,2,1,4,5] -> [2,3,1,4,5] ->

[2,1,3,4,5] ->

[1,2,3,4,5]

How many comparisons were done (in terms of n)?

n (n-1) = $O(n^2)$

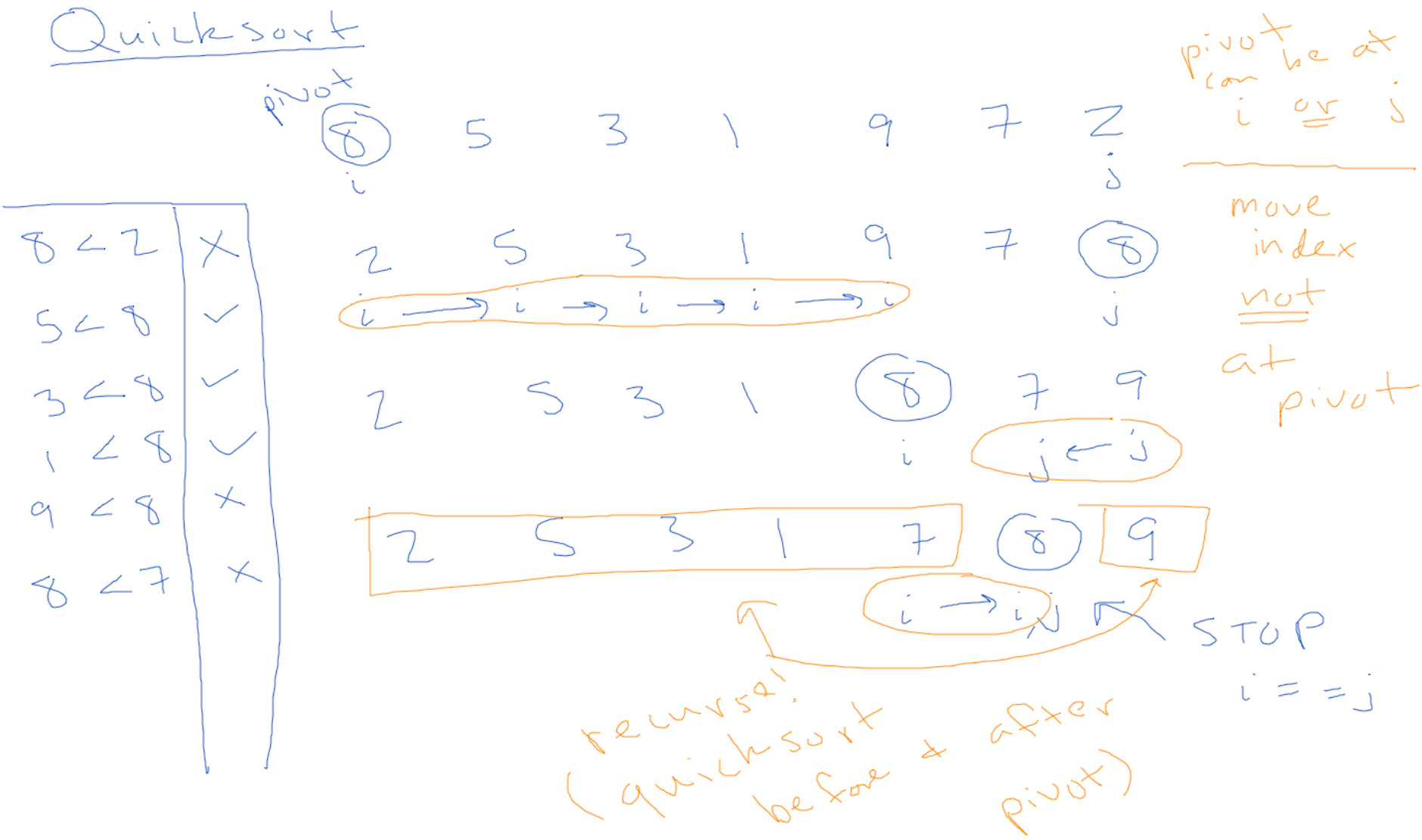
(Number of swaps: (n-1) + (n-2) + (n-3) + … + 1 = n (n – 1) / 2)

WE HAVE ALREADY SEEN SORTING

1) Insertion sort (Lab 3)
2) Heap sort (Lab 6)

This week we'll cover a few other common sorting algorithms

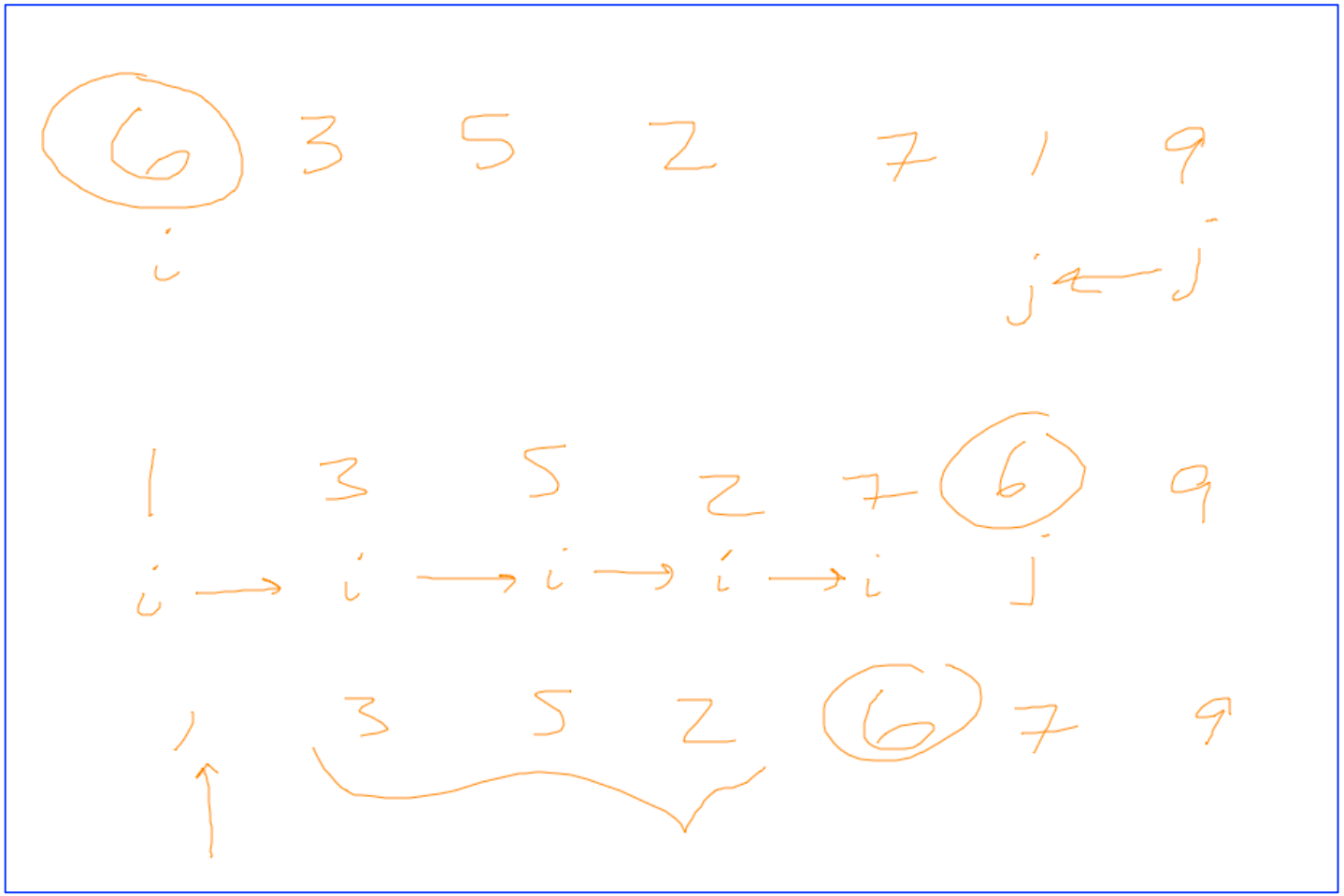
QUICKSORT


Quicksort
pivot
8 5 3 1 9 7 2
i j
pivot can be at i or j
move index not at pivot
8 < 2 X
5 < 8 ✓
3 < 8 ✓
1 < 8 ✓
9 < 8 X
8 < 7 X
2 5 3 1 9 7 8
i → i → i → i → j
2 5 3 1 8 7 9
i j ← j
2 5 3 1 7 8 9
i → i
STOP
i == j
recursion (quicksort before & after pivot)

QUICKSORT (GROUP EXERCISE)

1. Using the first number in each (sub)array as a pivot, quicksort the following array:

$$A = \{6, 3, 5, 2, 7, 1, 9\}$$



QUICKSORT (GROUP EXERCISE)

1. Using the first number in each (sub)array as a pivot, quicksort the following array:

$$A = \{6, 3, 5, 2, 7, 1, 9\}$$

2. Based on the example above, are some pivots better than others?

   Yes! 1 ends up being a bad pivot (in the left recursive call) because it doesn't end up in the middle.

3. Can quicksort be implemented in-place? Why or why not?

   Yes! We only rely on swaps (in-place operation) and keeping track of high and low indices.