COMP1002
DATA STRUCTURES AND ALGORITHMS

LECTURE 1: INTRODUCTION, FILES & SORTING

Curtin Computing

Last updated: 30 July 2019

Copyright Warning

Objectives

- Introduce to the Unit
- Revise Arrays and File I/O
- Introduce time complexity
- Describe simple sorting algorithms and related concepts
  - Bubble sort
  - Insertion sort
  - Selection sort
- See the benefits of sorting – e.g. searching a sorted list

Data Structures and Algorithms

- This unit introduces students to fundamental algorithms and data structures used in almost any computer program.
- Basic structures include stacks, queues and linked lists. Advanced structures explored are trees, graphs, hash tables and heaps.
- Algorithms discussed include sorting and recursion. Complexity analysis of these areas is also examined.
- The unit covers general computing structures and algorithms rather than being language-specific.
- Implementation languages are Java and/or Python.

Data Structures and ADTs

- Data structures and abstract data types (ADTs) are concepts that predate object-orientation
- A data structure is a particular way of organising the storage of data in a computer so as to make it efficient and easy to access/add/remove that data
  - *e.g.*, arrays, binary trees, linked lists
- An ADT is a data structure that only defines the operations for manipulating its data, but not how that structure is implemented
  - *e.g.*, stacks, queues
  - Stack can be done with an array OR with a linked list

Objects vs Data Structures/ADTs

- Objects have some similarities with these concepts
  - An object's fields can be considered to be a data structure
  - Objects can be used to implement data structures
- But objects ≠ data structures
  - Data structures can be implemented in any programming paradigm, not just O-O languages
  - Objects are more about the application's specific needs, data structures are more about general-purpose structures that could be useful in any application

Arrays

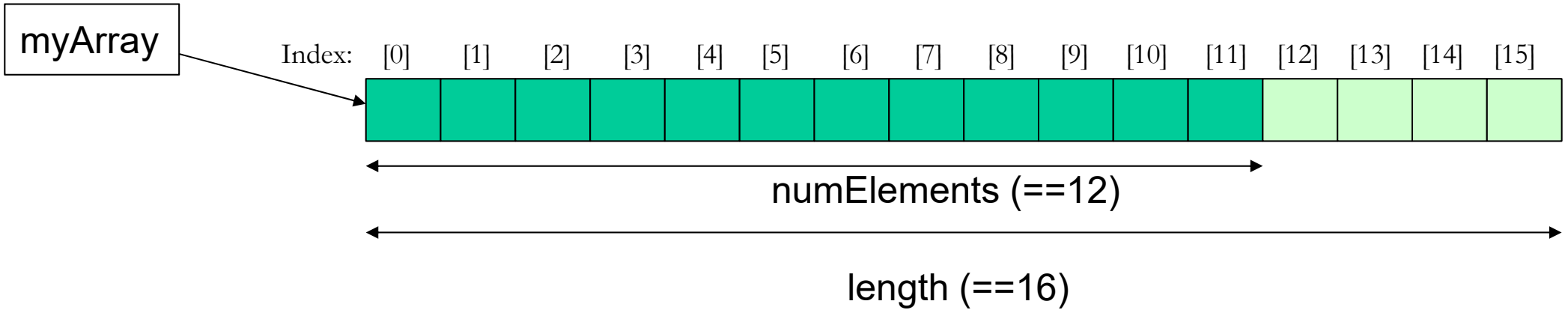
- We often work with *sets* of similar data
  - *e.g.*, how to handle the list of student marks in DSA?
  - double student1Mark, student2Mark, student3Mark, …?
- Arrays are one solution to this problem
- Simplest kind of data structure for storing data sets
  - Arrays are built-in to *all* programming languages (package in Python)
  - Instead of just one element, an array is a variable that contains *many* elements
  - The array variable itself is actually a reference to the first element of the array

Arrays - Properties

- Elements are located sequentially in memory
  - *i.e.*, the array is a *contiguous* block of memory
- All elements must have same data type (*e.g.*, double)
- Arrays can be initialised to any size (within memory limits)
- However, once initialised they cannot be resized
  - Must create a new array and copy over the contents of the old array in order to ‘resize’ an array

Arrays – Accessing Elements

- Once you have allocated an array, you need to be able to work with the elements inside the array
- Elements are accessed via an *index* (or *subscript*)
  - Java, Python and C index from 0 to N-1 (‘zero-based’)
    - In this case, the index is an *offset* from the first element



Arrays In Code (Java)

- Declaring: put '[ ]' on the end of the data type
  - *e.g.*, `double[ ] myArray;`
  - Any data type can be used with arrays, including classes
- Allocating: use new keyword with special [ ] syntax
  - *e.g.*, `myArray = new double[16];`
- Indexing: myArray[index], index must be an int
  - Negative indexes or indexes that are past the end of the array (*i.e.*, >= length) will cause an error during runtime.
  - sample code to output contents of myArray

```
for(int ii = 0; ii < numElements; ii++)
 {    //could use myArray.length
    System.out.println(myArray[ii]);
 }
```

Arrays In Code (Python)

- Declaring: use numpy package:
  - *e.g.*, `import numpy as np`
  - `myArray = np.empty(100, dtype=float)`
  - `myArray = np.zeros(100, dtype=float)`
  - Any data type can be used with arrays, including classes

- Indexing: myArray[index], index must be an int
  - Negative indexes work from the end of the array.
  - Indexes that are past the end of the array (*i.e.*, >= length) will cause an error during runtime.
  - sample code to output contents of myArray

```
for i in range(len(myArray)):
    print(myArray[i])
```

Output (using np.empty)

```
0.0
0.0
2.13132096926e-314
2.13135068287e-314
2.13135178957e-314
```

Arrays Pros and Cons

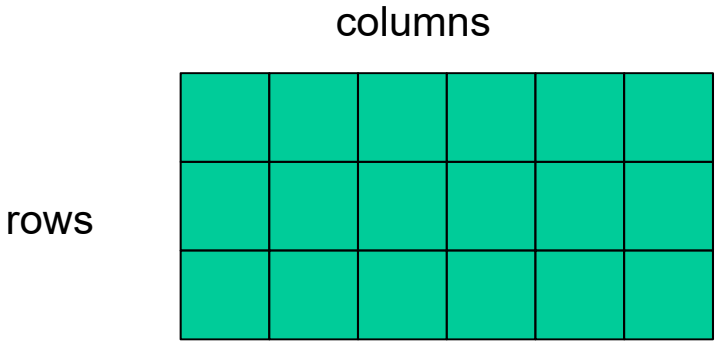
- ☑ Available in *all* programming languages
- ☑ Fast (direct) access to any element in array
  - Indexing is just a small arithmetic operation (arr + idx)
- ☑ No storage overhead – each element exactly fits data
- ☒ Fixed size once allocated – does not grow/shrink
- ☒ Can't insert a new element at the front without shuffling all other elements up by one
  - Same goes for inserting in the middle (sorted) and removing
- ☒ Doesn't track how many elements actually used
  - And when `used < capacity`, you are wasting space

Multi-dimensional Arrays

- So far we've only talked about 1D arrays
  - *i.e.,* a single list of elements
- What about 2D arrays?
  - *e.g.,* a matrix in maths is a 2D structure.
  - *e.g.,* a 3Mpix digital image is a 2048x1536 array (2D)
    - Actually, it's 2048x1536x3 since colour images have three channels: Red, Green, Blue
- Or even higher?
  - No reason why we can't have a 3D, 4D, 5D, N-D array

2D Arrays

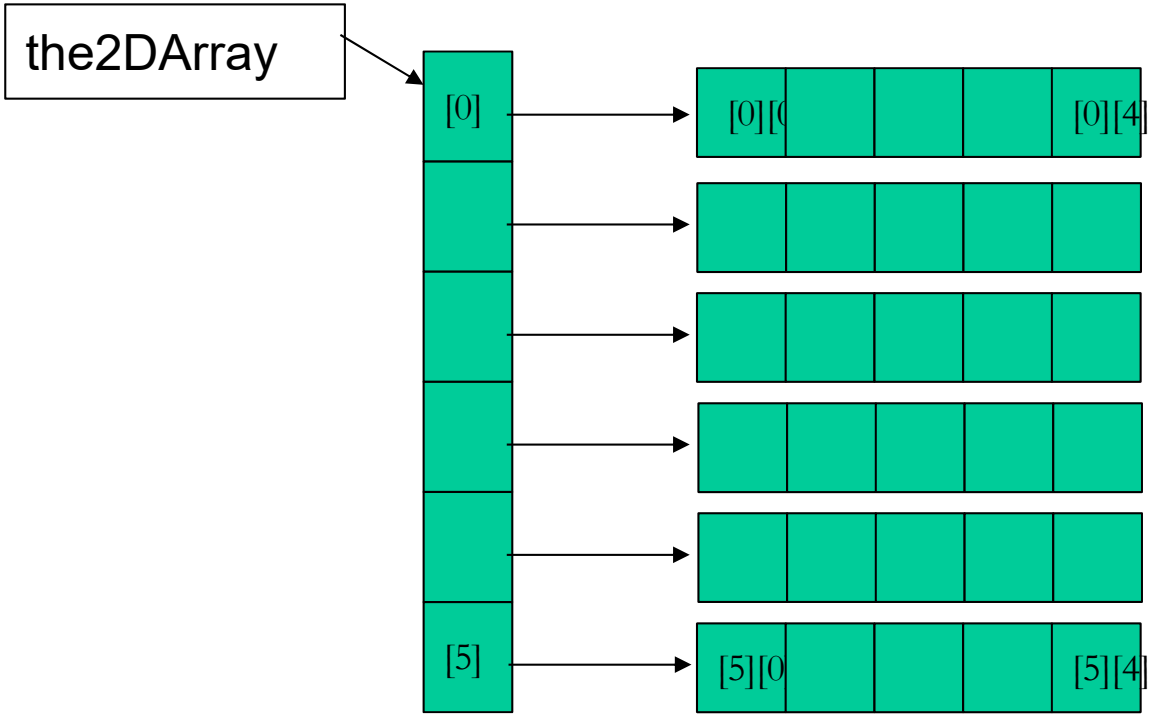
- Declaring 2D arrays is similar to 1D arrays
  - Just add an extra [ ] to the data type:
  - `Java: int [][] a2DArray;`
  - `Python: a2DArray = np.zeros((10,10), dtype=int)`
- Accessing a 2D array is also similar:
  - `a2DArray[row][column]`



  - The above is a 3x6 *matrix* (rows x cols)
  - OR, it is a 6x3 *image* (width x height)
    - depends on order of processing

2D Arrays – What Are They?

- 2D arrays are really an ‘array of arrays’
  - Java and Python hide this detail, but C doesn’t



N-D Arrays

- What works with 2D also works for 3D, 4D, .., N-D
  - Just keep adding extra `[]`'s or `()`'s to the data type
  - It can start to get hard to keep track of it all, but just remember that each array dimension works the same
    - *i.e.,* a 4D array is just a list of 3D arrays, each of which are just a list of 2D arrays, which in turn are just a list of 1D arrays.
    - *e.g.,*

```java
// Java
int[][][] array3D;
int[][] array2D;
int[] array1D;
```

```python
# Python
array3D = np.zeros((10,10,10), dtype=int)
array2D = np.zeros((10,10), dtype=int)
array1D = np.zeros((10), dtype=int)
```

```java
array2D = new int[3][6];  // Allocate a 3x6 array
array1D = array2D[0];     // Elements of array2D are just 1D arrays
```

Passing and Returning Arrays

- Arrays can be passed as a parameter to a method, just like any other variable
  - One difference from primitives: passing the array doesn't *copy* the array, it only passes a *reference* to the array
    - In this way, arrays are more like objects than primitives
  - So if the method changes the passed-in array, it will affect the 'original' array in the calling method
    - Since the two are in fact the exact same array
- You can also return an array from a method

Passing and Returning Arrays (Java)

```
public static void main() {
   int[] anArray = new int[3];
   int[] copyOfAnArray;

   anArray[0] = 1;                          ← Init anArray to some values
   anArray[1] = -16;
   anArray[2] = 5;

   copyOfAnArray = copyIntArray(anArray);   ← Passing arrays just uses the name, not []
}


public static int[] copyIntArray(int[] arrayToCopy) {   ← Note: data type is int[]
   int[] dupArray;                                      ← Array that will hold the copy
   int ii, n;

   n = arrayToCopy.length;
   dupArray = new int[n];
   for (ii = 0; ii < n; ii++) {                         ← do it n times, where n is the size of the array
      dupArray[ii] = arrayToCopy[ii];                   ← 'deep copy' of array elements
   }

   return dupArray;                                     ← Return the array reference
}
```

Passing and Returning Arrays (Python)

```
def copyIntArray(arrayToCopy):

    n = len(arrayToCopy)
    dupArray = np.zeros((n), dtype=int)      ← Array that will hold the copy

    for ii in range(n):                      ← do it n times, where n is the size of the array
        dupArray[ii] = arrayToCopy[ii]       ← 'deep copy' of array elements

    return dupArray                          ← Return the array reference


anArray = np.zeros((3), dtype=int)

anArray[0] = 1                               ← Initialise anArray to some values
anArray[1] = -16
anArray[2] = 5

copyOfAnArray = copyIntArray(anArray)        ← Passing arrays just uses the name, not []

print(copyOfAnArray)
```

Arrays of Objects (Java)

- The datatype of each element of an array is the datatype the array was declared as.

```
int[] anIntArray = new int[3];
  anIntArray[1] IS an int
double[] aDoubleArray = new double[3];
  aDoubleArray[1] IS a double
String[] aStringArray = new String[3];
  aStringArray[1] IS a String
Object[] anObjectArray = new Object[3];
  anObjectArray[1] can be any type of Object
```

- This is used just like any variable of the type:

```
anObjectArray[1].equals(newObject)
```

Arrays of Objects (Python)

- The datatype of each element of an array is the datatype the array was declared as.

```
anIntArray = np.zeros((10,10), dtype=int)
  anIntArray[1] IS an int
aFloatArray = np.zeros((10,10), dtype=float)
  aFloatArray[1] IS a float
anObjectArray = np.zeros((10,10), dtype=object)
  anObjectArray[1] can be a String, or any object
```

- This is used just like any variable of the type:

```
anObject Array[1] = "Wilma"
```

Files for Persistent Data Storage

- RAM is volatile and private to an application, so it's not a good match for the following purposes:
  - Storing application data long-term (between runs)
  - Sharing information between applications
  - Reading in bulk data provided by a user
- In contrast, files stored on disk are (semi)permanent, can be shared between applications and can be manipulated by the user outside the application

File Input/Output

- Unlike RAM, files are effectively an input to and/or an output from the application
  - Hence the term File I/O, for Input/Output
- Three basic steps in file I/O: (applies to any platform)
  - Open the file
  - Read data from file and/or write data to file
  - Close the file

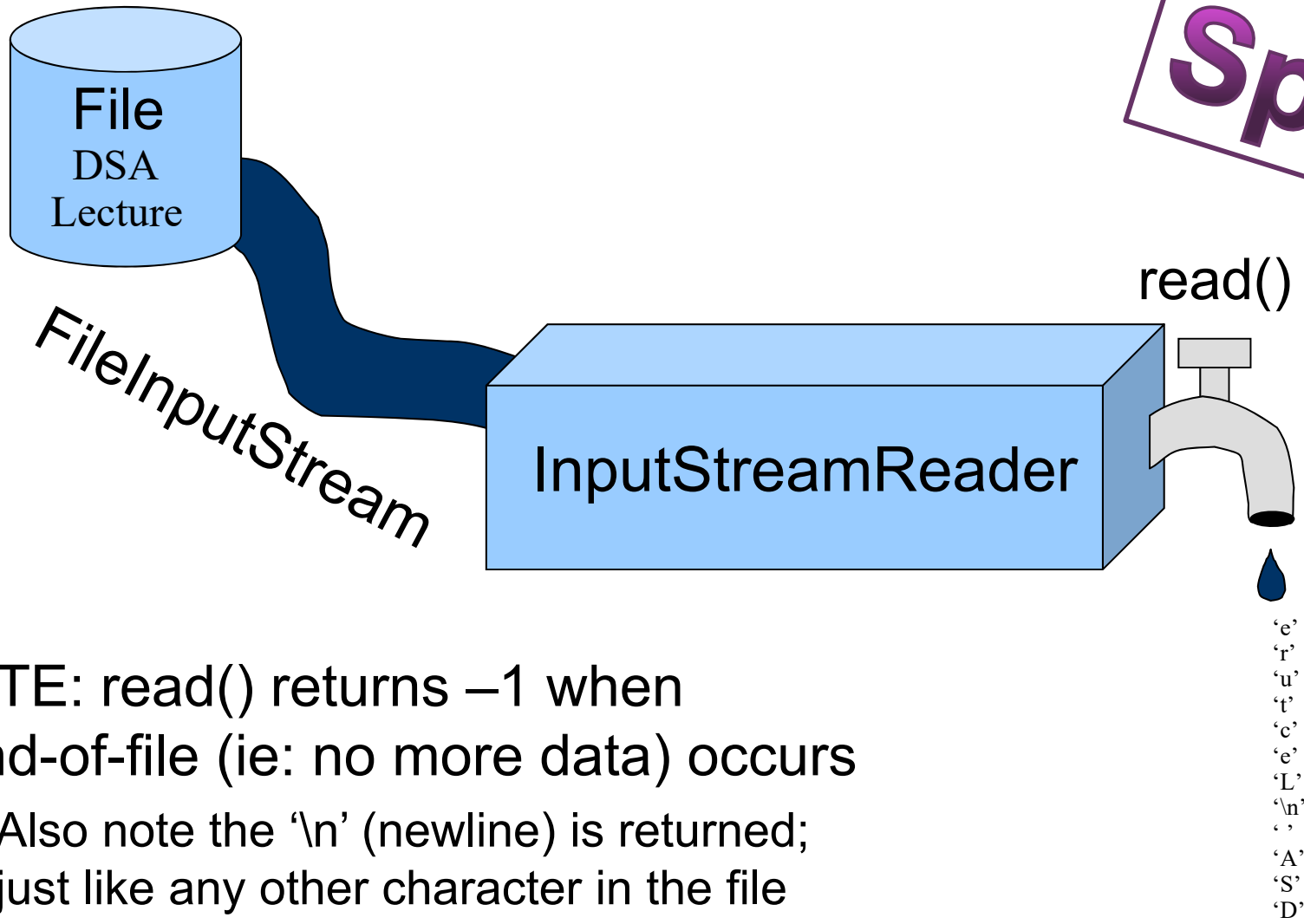
Text vs Binary Data

- Data files generally fall into two broad categories:
  - Files containing text data (relatively unstructured)
  - Files containing binary data (everything else)
- Binary data is highly structured
  - *e.g.,* images, databases, executable files, *etc*.
  - There is a lot of prior knowledge on precisely where information exists in the file and how large blocks are
- Text data is *un*structured, so it is usually impossible to know beforehand how many bytes to read
  - *e.g.,* how many characters in an arbitrary line of text?

Efficient File I/O

- So reading blocks of N bytes from a binary file is OK
  - You will know beforehand how many bytes are needed
- But when reading text, you have to constantly be ready for the end of data since you can't predict it
  - *e.g.,* end of word, end of line, or end of file
- One approach is to read data in a byte at a time, and check each byte for end-of-X (' ', '\n' or -1)
  - The problem is that this will be very slow since hard disks are fastest at reading blocks of data at a time
    - It's like filling a bucket one drop at a time vs opening the tap and letting the water flow freely into the bucket

InputStreamReader



Java Specific

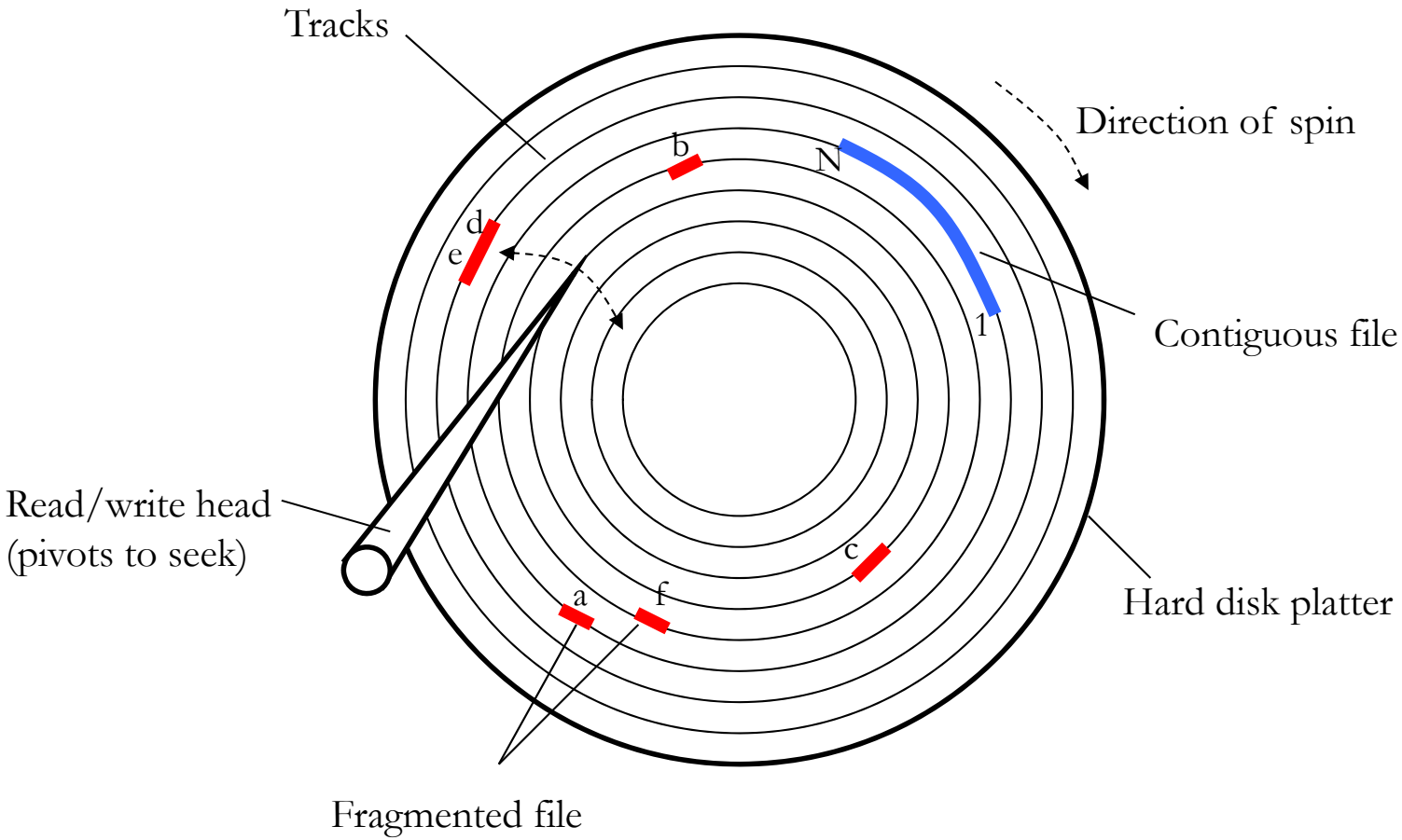
NOTE: read() returns –1 when end-of-file (ie: no more data) occurs

- Also note the '\n' (newline) is returned; just like any other character in the file

How Hard Disks Work

- To understand file access efficiency, you have to understand how a disk drive reads and writes data
  - Applies to CDs/DVDs as well, but not to SSDs.
    - SSD = solid state drive. These behave more like RAM
    - HDD = hard disk drive
- A hard disk drive is a set of circular disks (‘platters’) that can apply and retain magnetism on its surface
  - Data is stored via this magnetism: magnetic = 1, not = 0.
- The magnetism is manipulated by a read-write head
  - An arm with a small magnetometer on its tip

Internals of a Hard Disk



Data Organisation on Hard Disks

- A drive is made up of multiple platters and heads
- Data is organised in concentric circles around the disk, called tracks (or alternatively, cylinders)
  - A cylinder is the set of equivalent tracks across all platters
- Each track is split into pieces called sectors
- The O/S then maps these tracks and sectors to blocks
  - Blocks are the smallest chunk of data that the O/S will read/write to/from disk. Usually 512 bytes or 1Kb
- Nowadays, the HDD itself maps tracks/sectors to an internal config, but that is not seen by you or the O/S

Hard Disk Access Speed

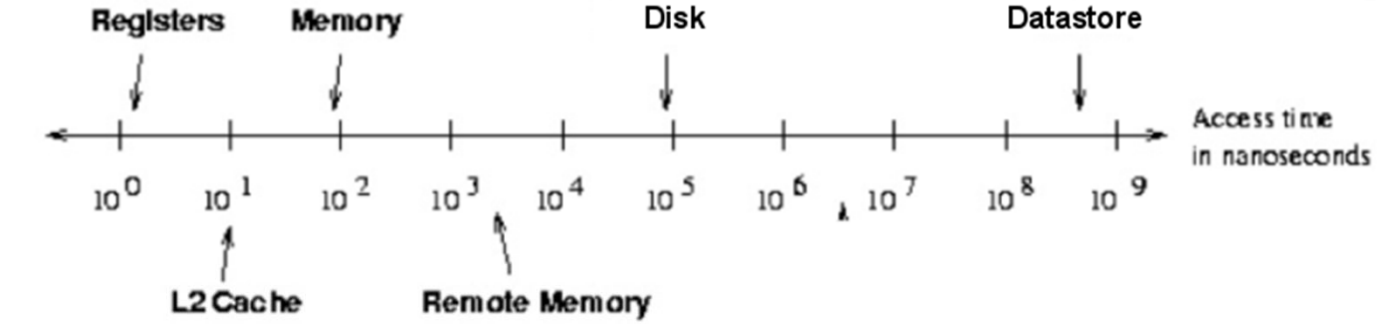
- Accessing data is a two-stage process
  - First the head must be positioned on the track where the data resides (*somewhere* on that track)
  - Then the head must wait until the disk, already spinning, brings the target sector under the head so it can be read
- Usually files are organised so that their data is stored in sequential sectors on the same track(s) (if possible)
- So there are two important elements to HDD speed:
  - Seek time – how long the head takes to get to a track
  - Transfer time – how fast can the data be streamed off the track in sequential order (faster spin = faster reading)

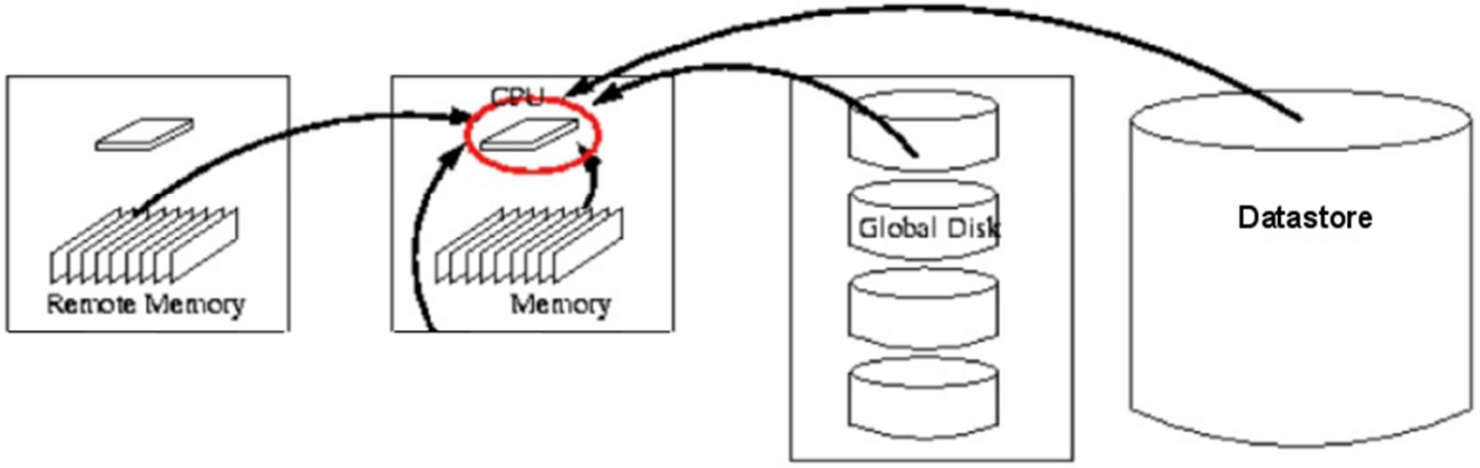
Modern Access Speeds

- Seek time is worst: on average around 7ms
  - Physical limits of motors/actuators
- Transfer speed increases with HDD density (size)
  - So 50Mb/s is common nowadays
- Thus when dealing with files, you want to minimise the amount of seeking and maximise streaming
  - Not totally in your control – the O/S is the one that decides where files go based on what sectors are available
  - But modern O/S's do a good job of locating files contiguously to avoid fragmentation of the files

Data Locality

- Comparing Disk and RAM is only one variable for data movement
- We may be going through a network, to the cloud, to tape archives…





Data Locality

Location	Access time	Access time (cycles)
Register	<1ns	-
L1 cache	1ns	4
L2 cache	4ns	10
L3 cache	15-30ns	40-75
Memory	60ns	150
Solid state disk	50us	130,000
Hard disk	10ms	26,000,000
Tape	10sec	26,000,000,000

Source: https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

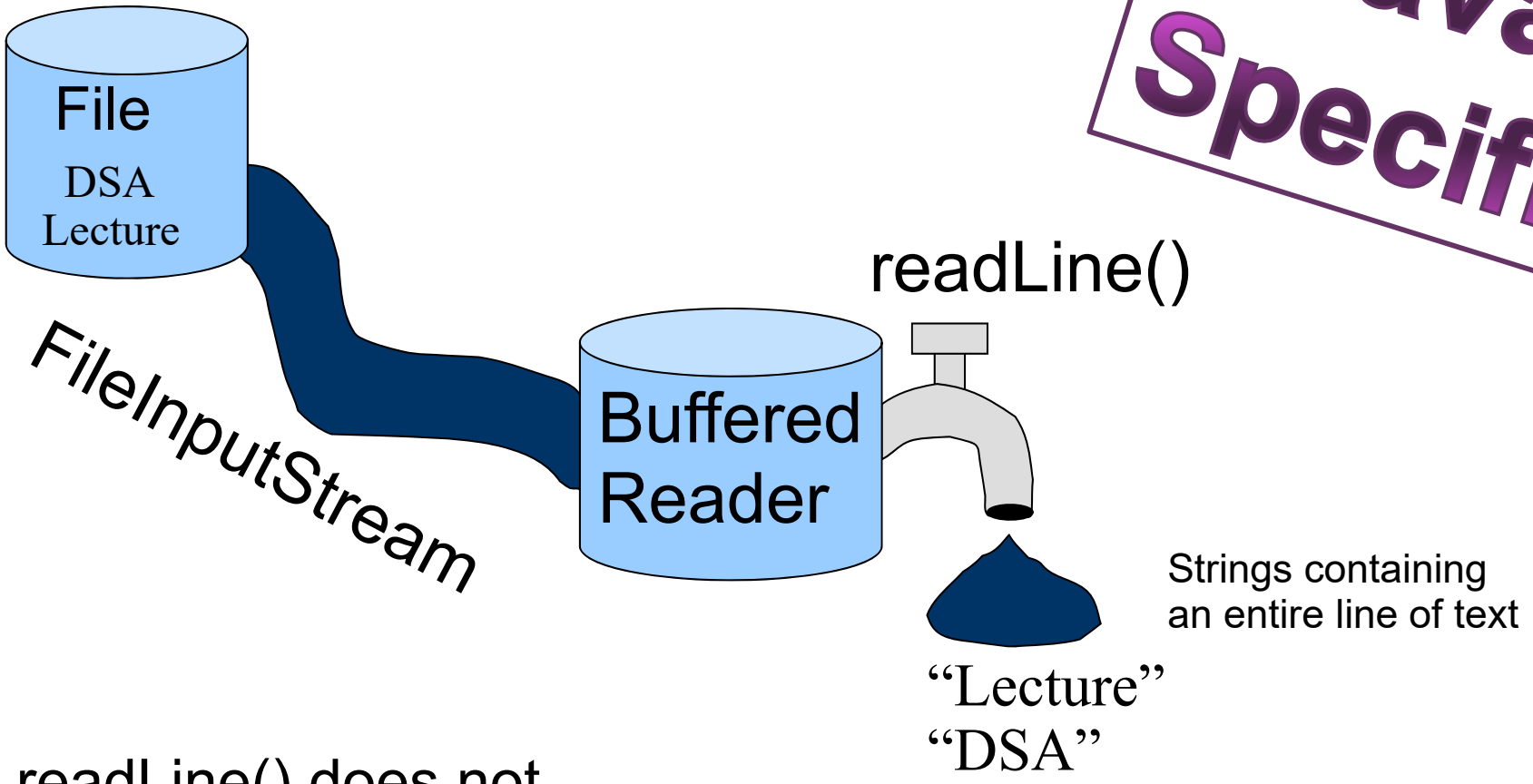
Implications for File I/O Efficiency

- When a programmer reads N bytes from a file, the O/S determines the track/sector of that file and sends a read instruction to the HDD for the N bytes
  - The HDD will then seek and read N bytes in one operation
  - This will (ideally) involve one seek and one stream read
    - (actually, modern HDDs do all sorts of optimisations, but if the file is contiguous it will almost surely be read in one operation)
  - The read data will be sent to RAM by the HDD
  - The O/S will be alerted to the fact that the data is now available and return it to the program

Implications for File I/O Efficiency

- If instead you ask for the N bytes one byte at a time, the O/S will submit N read requests to the HDD
  - And each subsequent read will be sent too 'late' – the next sector will have spun past the head and the HDD will need to wait until the disk spins back around to read it
  - Worse, there may be another process running that asks for data from a different part of the disk
    - This will be interleaved with your N reads
    - It will cause the head to flit back and forth across the disk to access the two different files, racking up the seek time overheads
- So reading a file a single byte at a time will be *sloow*

BufferedReader



Java Specific

NOTE: readLine() does not
include the ‘\n’ in the returned lines
and returns null when end-of-file occurs

File Reading: One Line at a Time

```
METHOD readFileExample IMPORT filename EXPORT NOTHING

theFile ← OPENFILE filename
lineNum ← 0
INPUT line FROM theFile

WHILE NOT (theFile = EOF)                ← EOF = end of file. Detecting this is very language-specific
   lineNum ← lineNum + 1
   processLine(line)                     ← Whatever processing on the line is required

   INPUT line FROM theFile               ← Read the next line
ENDWHILE

CLOSEFILE theFile                        ← Close the file

ENDMETHOD
```

File Reading: One Line at a Time (Java)

```
private void readFileExample(String inFilename) {
   FileInputStream fileStrm = null;
   InputStreamReader rdr;
   BufferedReader bufRdr;
   int lineNum;
   String line;
   try {
      fileStrm = new FileInputStream(inFilename);  ← Open the file
      rdr = new InputStreamReader(fileStrm);                    ← Create a reader to read the stream
      bufRdr = new BufferedReader(rdr);            ← To read the stream one line at a time

      lineNum = 0;
      line = bufRdr.readLine();                                 ← Read the first line
      while (line != null) {                       ← While not end-of-file, process and read lines
         lineNum++;
         processLine(line);                                     ← Whatever processing on the line is required
         line = bufRdr.readLine();                              ← Read the next line
      }
      fileStrm.close();                                         ← Clean up the stream
   }
   catch (IOException e) {                                      ← MUST catch IOExceptions
      if (fileStrm != null) {                   ← Clean up the stream if it was opened
         try { fileStrm.close(); } catch (IOException ex2) { } // We can't do anything more!
      }
      System.out.println("Error in file processing: " + e.getMessage());          ← Or do a throw
   }
}
```

File Reading: One Line at a Time (Python)

```
def readFileExample(filename):
    try:
        fileObj = open(filename)                    ← Open the file

        lineNum = 0
        line = fileObj.readline()                   ← Read the first line

        while line:                                 ← While not end-of-file, process and read lines
            lineNum += lineNum
            processLine(line)
            line = fileObj.readline()

        fileObj.close()                             ← Clean up the stream

    except IOError as e:                            ← MUST be ready for file IO exceptions
        print("Error in file processing: " + str(e))
```

Notes on File Handling

- Make sure you close the file as soon as possible
  - The operating system must track what files are open
    - The O/S remembers where you were in the file, etc
  - The resources available for this tracking are limited
    - Run out and the O/S will terminate your program
  - Thus don't leave files open – clean them up *early*
    - Java doesn't free objects immediately – it waits for the garbage collector, so *always* explicitly close() a file once finished with it

Notes on File Handling

- You *must* handle IOException
  - IOExceptions must be caught
    - Checked Exception
    - The compiler will complain otherwise
    - This also forces us to do the try..catch around close()
  - Note that you can add a throw IOException clause to the method to 'avoid' having to catch IOExceptions
    - It just means that now the *calling* method must catch them

(Python doesn't **require** the handling of IOError, but you should put exception handling into all code that uses files)

Parsing Text Files

- When dealing with text, it's often necessary to take it apart and organize it ready for processing/storage
- This is called parsing – to determine and extract the structure of a piece of text
  - The word originally comes from syntax analysis of written language
- Examples of where parsing is needed:
  - Natural language processing (*e.g.,* spelling/grammar checks)
  - Building Web search indexes
  - Compilers: must parse code to detect stmts and variables

Tokenizing

- Tokenizing is the first step in parsing: the process of breaking up a stream of text into basic elements
  - We'll use US spelling (with a 'z') to avoid confusion
  - These elements are called tokens, and what they are depends on what the application is parsing
    - *e.g.,* Single words, entire lines, equation terms, *etc.*
  - Tokens are broken up by searching for character(s) that *delimit* the boundary of a token
    - *e.g.,* lines are separated by a '\n' newline character
    - *e.g.,* words are separated by spaces, commas and periods
    - *e.g.,* equation operands are separated by operators +/*-

Tokenizing in Java and Python

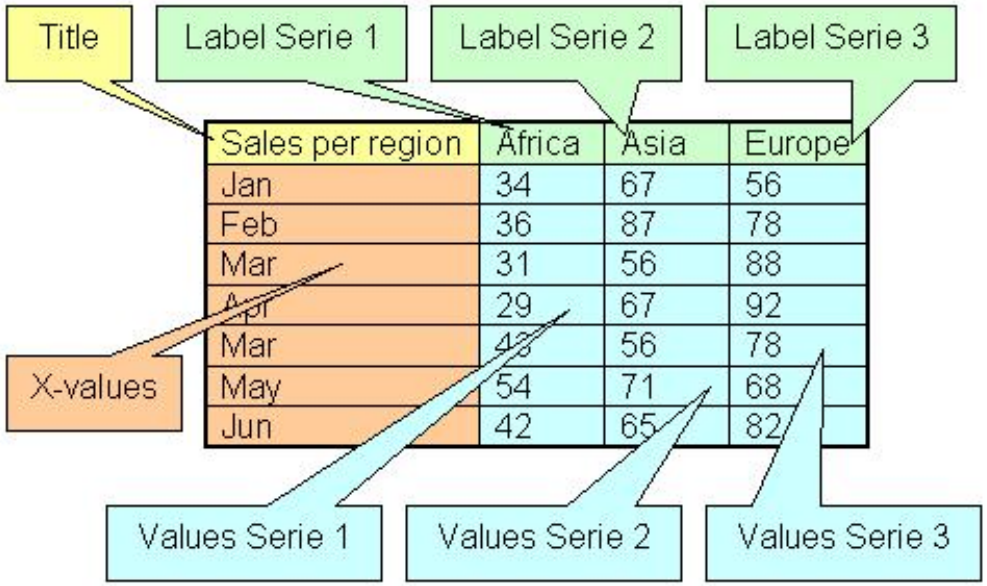
- Java and Python provide the split(String regex) method for simple tokenizing of a String
  - Can use regular expressions for more complex splits
  - Use packages/libraries if you need more functionality
    - e.g. nltk in python – Natural Language Toolkit

Comma Separated Values

- We'll take a little detour and introduce comma separated values
  - We'll use this as an example later on
- We often need to store data to a file
  - The question is, in what form should we store it?
- If the data is in table or matrix form, one can write it out as a set of rows and columns in a certain format:
  - One row is written per line
  - Each row contains multiple fields, one per column
  - Each field's value is separated by a comma ','

CSV Example

Content of File




```
Sales per region,Africa,Asia,Europe
Jan,34,67,56
Feb,36,87,78
Mar,31,56,88
Apr,29,67,92
Mar,43,56,78
May,54,71,68
Jun,42,65,82
```

The CSV File

CSV Notes

- The delimiting commas have no trailing space
  - It might look better to have a space after the comma, but it just makes it harder to parse when reading the file later!
- Column sizes don’t have to be consistent across rows
  - *e.g.,* the first row (headings) has much longer fields than the same columns in subsequent rows
    - It would be a waste of space to pad out the fields
    - Parsing/tokenizing must handle these variable-length fields
- Numeric data is converted into its textual equivalent
  - If we saved integers directly, we might get things that look like text ‘\n’ (ASCII 13) or ',' (ASCII 44) but are merely part of the data

Text vs Binary

- You don't *have* to save table data in CSV format
  - In fact, dumping raw binary data is often more efficient, and you know how large each field is (*e.g.,* ints = 4 bytes)
- However, CSV text data has some advantages:
  - Easy for humans to read and edit
  - Highly portable to different platforms
    - big endian vs little endian
  - Fields are explicitly separated with commas
  - CSV is a widely-known format
    - XML is now beginning to supplant CSV as the standard format for data interchange between companies, but CSV is still used

Parsing a Single CSV Row (Java)

```
private void processLine(String csvRow) {
   String thisToken = null;

   String tokens = csvRow.split(",");          ← Split string on comma to parse the csvRow

   int tokenCount = tokens.length();           ← get number of tokens

   for (int j = 0; j < tokenCount; j++) {      ← Iterate over tokens
      thisToken = tokens[j];
      System.out.print(thisToken + " ");       ← Print out each column for this row
   }
   System.out.println("");
}
```

Parsing a Single CSV Row (Python)

```
def processLine(csvRow):

    tokens = csvRow.split(",")

    for thisToken in tokens:

        print(thisToken, end=" ")

    print()
```

← Split string on comma to parse the csvRow
← returns a list, but that's ok in this case

← Iterate over tokens

← Print out each column for this row, add space at end

- In your own coding, you should use csv_reader and other methods to make coding easier.
- In this unit, we expect you to parse code manually to learn the techniques
- You shouldn't be using lists in DSA, but file I/O will be an exception ;-

Output

- Given the following CSV data file:

```
97452,James,88,96,82,86
99576,Alan,6,46,34,38
99888,Geoff,100,68,72,75
```

- The following would be the output of having readFileExample() call the shown processLine():

```
97452 James 88 96 82 86
99576 Alan 6 46 34 38
99888 Geoff 100 68 72 75
```

Parsing CSV Files with Known Format

- Combine the previous method with the line-reading algorithm for a *general-purpose* CSV parser
- But what if you have very specific data?
  - Consider the following CSV file contents:

```
97452,James,88,96,82,86
99576,Alan,6,46,34,38
99888,Geoff,100,68,72,75
```

  - Note that the *meaning* of each column is not in the CSV
    - We must get this information from somewhere else and hardcode it into the tokenizing/parsing application (clumsy!)
    - Format for the above data happens to be:

```
<StdntID>,<Name>,<Assign%>,<Test%>,<Exam%>,<Overall%>
```

Parsing CSV Files with Known Format

- In this case, we know exactly what each column should contain
- Thus we may want to handle each column specially
  - The next slide considers an application that prints the correct label in front of each field

Parsing a Known-Format CSV Row (Java)

```
private void processLine(String csvRow) throws IllegalStateException
{
   int id;
   String name;
   double assign, test, exam, overall;

   String tokens = csvRow.split(","); ← Split string on comma to parse the csvRow
   try {
      id = Integer.parseInt(tokens[0]);          ← DON'T loop: just assume the format is
      name = tokens[1];                             exactly what we expect it to be, catching
      assign = Double.valueOf(tokens[2]);           exceptions if it turns out to be a bad format
      test = Double.valueOf(tokens[3]);
      exam = Double.valueOf(tokens[4]);
      overall = Double.valueOf(tokens[5]);

      System.out.println(" ID:" + id + "Name:" + name + "Assign:" + assign +
                         "Test:" + test + "Exam:" + exam + "Overall:" + overall);
   }
   catch (Exception e) {
      throw new IllegalStateException("CSV row had invalid format");
   }
}
```

Parsing a Known-Format CSV Row (Python)

```
def processLine(csvRow):

    tokens = csvRow.split(",")

    try:
        id = int(tokens[0])
        name = tokens[1]
        assign = float(tokens[2])
        test = float(tokens[3])
        exam = float(tokens[4])
        overall = float(tokens[5])

        print(" ID:" + id + " Name:" + name + " Assign:" + str(assign) +
          " Test:" + str(test) + " Exam:" + str(exam) + " Overall:" + str(overall))
    except TypeError:
         raise TypeError("CSV row had invalid format")
```

← Split string on comma to parse the csvRow

← DON'T loop: just assume the format is exactly what we expect it to be, catching exceptions if it turns out to be a bad format

Output

- Given the CSV data from a previous slide:

```
97452,James,88,96,82,86
99576,Alan,6,46,34,38
99888,Geoff,100,68,72,75
```

- The following would be the output of having readFileExample() call the last processLine():

```
ID:97452 Name:James Assign:88 Test:96 Exam:82 Overall:86
ID:99576 Name:Alan Assign:6 Test:46 Exam:34 Overall:38
ID:99888 Name:Geoff Assign:100 Test:68 Exam:72 Overall:75
```

Writing Text Files

- Writing files is actually simpler than reading them
  - Since you don’t have to worry about parsing!
- The overall approach is the same:
  - Open a File (Java: FileOutputStream, then create a Writer)
  - Output data to the file
- One thing to be careful with is that you must ensure that newlines and commas are put in the right place
  - Assuming you are outputting CSV format, of course!

Writing a CSV Row (Java)

```
private void writeOneRow(String filename, int ID, String name, double assign, double test,
                         double exam, double overall) {
   FileOutputStream fileStrm = null;
   PrintWriter pw;

   try {
      fileStrm = new FileOutputStream(filename);          ← Open the file for writing
      pw = new PrintWriter(fileStrm);                     ← Initialise writer

      pw.println(id + "," + name + "," + assign + "," + test + "," + exam + "," + overall);
      pw.close();                                         ← Clean up the stream
   }
   catch (IOException e) {                                ← MUST catch IOExceptions
      if (fileStrm != null) {                             ← Clean up the stream if it was opened
         try { fileStrm.close(); } catch (IOException ex2) { } // We can't do anything more!
      }
      System.out.println("Error in writing to file: " + e.getMessage());     ← Or do a throw
   }
}
```

Introduction to Time Complexity Analysis

- In computers, time in seconds is not a useful measure of an algorithm since faster hardware can reduce the time
- Instead, we need to talk about how many steps are needed
  - Which is independent of hardware speed, so is a better ‘absolute’ measure of speed
  - And where ‘steps’ really is CPU instructions
- Unfortunately, we can never know *exactly* how many CPU instructions something takes
  - Different CPUs have different instruction sets and are faster/slower with different instructions vs other CPUs

Big-O Notation

- Instead we give an estimate of the number of steps, focusing on how the algorithm scales with more data
  - We want to know whether the algorithm will handle lots of data well or if it will become quickly unusable
- Big-O notation was developed for this purpose
  - Indicates the 'order-of' the algorithm (ie: what ballpark it is in)
  - Notation: O(<numSteps>), eg: O(N), O(N log N), $O(N^2)$
  - Ignores multiplying constants
    - Only concerned with scalability as the amount of data increases
    - *e.g.,* O(N) means "double the data N, and you double the time"
    - *e.g.,* $O(N^2)$ means "triple the data N, and you 9x the time"
  - We will use this to compare algorithms

Sample Values

- O(1)
  - Number of steps is constant, no matter N
- O(log N)
  - N=100 steps=7, N=1000 steps=10, N=10^6 steps=20
- O(N)
  - N=100 steps=100, N=1000 steps=1000, N = 10^6 steps = 10^6
- O(N log N)
  - N=100 steps=700, N=1000 steps=10000, N=10^6 steps = 2 *10^7
- O(N^2)
  - N=10 steps=100, N=1000 steps=1000000, N=10^6 steps = 10^{12}

Searching through a list of values

- To see how Big-O works, lets analyse find() on an unsorted array
  - Best case: myArray[0] is element to find
    - This is one step, so O(1)
  - Worst case: myArray[n-1] is the match, which is n steps
    - O(N) in Big-O notation
    - Each step involves multiple CPU instructions, but we aren't concerned with these details, so we don't talk about O(5N)
  - Average case: On average, we must go halfway: N/2 steps
    - O(N) – again, the constant multiplying factor of ½ is irrelevant
  - We are mostly interested in the average and worst cases

Searching a sorted list

- Searching a list for an item (or items) is common
  - Naïve search: go sequentially through the list → O(N)
    - Also called a 'linear search'
    - As simple as it gets, but still pretty effective

```
METHOD LinearSearch IMPORT array, searchTgt EXPORT matchIdx

  ii ← 0
  matchIdx ← -1                                    ← Assume we won't find the target
  WHILE (ii < array.length) AND (matchIdx == -1) DO   ← Find the first match
     IF (array[ii] == searchTgt) THEN
        matchIdx  ← ii
     ENDIF
  ENDWHILE
```

Binary Search

- Linear search is OK, but nothing to brag about
- A faster alternative: Binary search
  - Takes advantage of (and needs) sorted data to ‘jump around’

    Step 1. Set an upper and lower bound on the search location
    - The target is known to exist within these bounds
    - Start off with the full range and refine the search from there

    Step 2. Check the value halfway between these bounds and use this to update the bounds
    - If the halfway value still too low, it becomes the new lower bound
    - If it’s too high, it becomes the new upper bound

    Repeat until you hit the target value being searched for

Binary Search – Strategy

- Analogy: Guessing a number between 0..100
  - Example target: 85; Initial bounds: 0 .. 100

• Initial guess: (0+100)/2 = 50	Response: "No; higher"
• New bound: 50 .. 100	
• Guess: (50+100)/2 = 75	Response: "No, higher"
• New bound: 75 .. 100	
• Guess: (75+100)/2 = 88	Response: "No, lower"
• New bound: 75 .. 88	
• Guess: (75+88)/2 = 82	Response: "No, higher"
• New bound: 82 . .88	
• Guess: (82+88)/2 = 85	Response: "Got it!"

Binary Search – Discussion

- Search time complexity:
  - Halving bounds on each iteration, means at most $\log_2 N$ iterations
    - Since after $\log_2 N$, the bounds will shrink to one element in size
  - Average/Worst Cases: O(log N)
    - Best case: O(1), but that only happens due to luck!
- Lends itself to recursive implementation
  - Change the bounds and repeat on the simpler sub-problem
  - Base cases: 1) target found OR 2) bounds are equal (*i.e.,* not found)
- An iterative solution isn't more difficult
  - You don't need to remember past upper/lower bounds
    - Unlike MergeSort or QuickSort, which has to keep track of *all* merge indexes

Binary Search – Algorithm

- Replace ‘guessing the number’ with ‘guessing the index in a sorted list’ and you have binary search

```
METHOD BinarySearch IMPORT sortedArr, searchTgt
                    EXPORT matchIdx, which will be -1 if not found
matchIdx ← -1                                    ← Assume failure to find the target
lowerBd ← 0                                        could throw exception
upperBd ← sortedArr.length

WHILE (NOT found) AND (lowerBd <= upperBd) DO
   chkIdx ← (lowerBd + upperBd) / 2
   IF (sortedArr[chkIdx] < searchTgt) THEN
      lowerBd ← chkIdx+1                         ← target must be in the upper half
   ELSEIF (sortedArr[chkIdx] > searchTgt) THEN
      upperBd ← chkIdx-1                         ← target must be in the lower half
   ELSE
      matchIdx ← chkIdx                          ← found our target
      found ← TRUE
   ENDIF
ENDWHILE
```

Binary Search – Limitations

- Requires data to be in sorted order
  - Needs sorting so that upper and lower bound indexes are guaranteed to contain the target
  - Thus although binary searches are a very fast O(log N), it requires an O(N log N) pre-step to sort the data first
    - Hence one-off searches are better done with O(N) linear search
    - Only for repeated searching is it worth the sorting pre-step

Sorting: Why Sort?

- For clear presentation of data:
  - We often need to present data in an organised manner to a person so that they can make sense of it
  - Sorting is usually a good way to organise data
    - Just imagine using a randomly-ordered phone book!
- To facilitate efficient processing:
  - Selecting a range is simple if the data is in a sorted list
    - *e.g.,* in a supermarket database, finding all transactions between 01/01/2010 and 31/01/2010 is easy if the data is sorted by date
  - Sorting also allows us to search for an item quickly
    - Analogy: finding a name in a phone book
      - you go directly to first letter, then second, etc

Sorting: Time Complexity

- Sorting is concerned with manipulating *all* N items
  - Specifically: to put them in sorted order
- So O(1) for sorting is impossible
  - *At minimum* we must check all N items to see what kind of order they are in – so O(N) is an absolute lower limit
    - *e.g.,* if the elements are already in sorted order
    - … which almost never happens!
  - In the average case, we can expect worse than O(N)
    - Not a surprise – sorting is pretty involved

Sorting: Time Complexity

- So what average-case sorting time can we hope for?
  - Naïve approaches to sorting quickly becomes $O(N^2)$
  - More sophisticated sorts are O(N log N)
    - In fact, it has been mathematically proven that no general sorting algorithm can be faster than O(N log N) in the average case
      - Some sorts are faster only by exploiting characteristics of the data
      - *e.g.,* radix sort is O(kN): needs integer data with known min & max
- Remember: the focus is on average and worst cases
  - Why put much stock in a best case when most of the time it will be the average case that is happening?
  - Worst case also important: shows how bad things might get!

Scaling with N

- The following table illustrates how different complexities scale

N	$N \log_2 N$	N^2
4	8	16
8	24	64
16	64	256
32	160	1,024
64	384	4,096
128	896	16,384
256	2,048	65,536
512	4,608	262,144
1,024	10,240	1,048,576
2,048	22,528	4,194,304
4,096	49,152	16,777,216
8,192	106,496	67,108,865
16,384	229,376	268,435,456
32,768	491,520	1,073,741,824
65,536	1,048,576	4,294,967,296

Sorting Visualisations

- Each semester we find new sorting visualisation videos
- This is an old one, but a good one:
  - https://www.youtube.com/watch?v=BeoCbJPuvSE
- We'll also use a simulation to hep understand the sorts:
  - https://visualgo.net/en/sorting

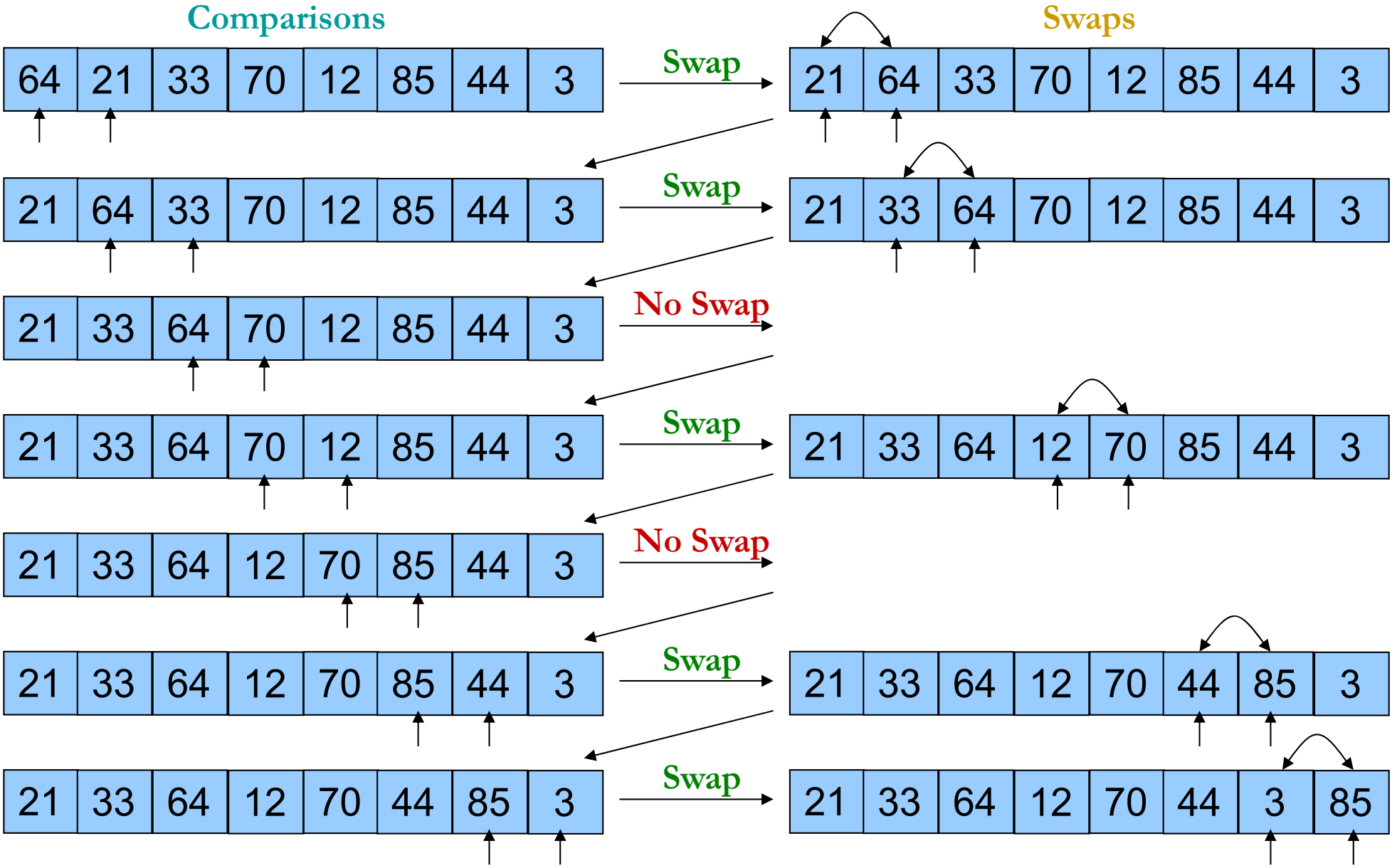
$O(N^2)$ Sorting Algorithms

- We'll start by examining the $O(N^2)$ sort algorithms
  - These all have two loops (nested), each iterating ~N times
    - ~N iterations of length ~N = ~N*~N = ~N^2 = $O(N^2)$
  - The inner loop defines a single pass through the data, performing checks and swaps to improve sorted order
  - The outer loop forces multiple inner-loop passes until the array is completely sorted
    - Exactly how many passes depends on the algorithm and on the initial state of the array (*i.e.,* how unsorted it is)

Bubble Sort – Strategy

- Incrementally sorts the array by comparing adjacent pairs and swapping them if they are not in order
  - One pass through the array will only *improve* ordering
  - Need multiple passes P (up to N) to fully sort the array
  - Each pass will ‘bubble up’ the largest value to the end
- Some ‘optimisations’ can be done:
  - Can stop sorting when no swaps are needed in a pass
    - Detects sorted: if all adjacent pairs are in order, array is sorted
  - Don’t need to check the last P values in the array
    - The previous P passes have put the largest P values at the end

Bubble Sort – Example (one pass)



Examples

- A helpful online resource for visualising sorting algorithms (as well as other algorithms we'll cover)
  - http://www.cs.usfca.edu/~galles/visualization/Algorithms.html
  - This plays through animated examples of many algorithms.
- We will look at http://visualgo.net/.
  - The examples are somewhat smaller, making the graphics larger.
  - This is better for lectures.
  - It shows the pseudo-code as it steps through.

Bubble Sort – Time Complexity

- Each pass P does (N – P) comparisons
  - N-1 on first pass, N-2 on second pass, etc
  - Best case: Elements are already in sorted order
    - *i.e.,* Only one pass needed, which finds no swaps
    - Complexity: O(N) : N-1 compares + 0 swaps
  - Worst case: All N elements are in *reverse-sorted order*
    - *i.e.,* need P = N passes, with compares/swaps N-1 + N-2 + … + 1
    - = N(N+1)/2 compares + N(N+1)/2 swaps ≈ N^2
    - Complexity: $O(N^2)$
  - Average case: Surprisingly, ~N passes are typically needed
    - Still N(N+1)/2 compares ≈ N^2 (usually fewer swaps though)
    - Complexity: $O(N^2)$

Proof: 1+2+3+…+N = N(N+1)/2

$$S_N = 1+2+3+\ldots+(N-1)+N$$

$$2S_N = \left[1+2+\ldots+(N-1)+N\right]+\left[1+2+\ldots+(N-1)+N\right]$$

$$= \left[1+2+\ldots+(N-1)+N\right]+\left[N+(N-1)+\ldots+2+1\right]$$

$$= (1+N)+\left(2+(N-1)\right)+\ldots+\left((N-1)+2\right)+(N+1)$$

$$= (N+1)+(N+1)+\ldots+(N+1)+(N+1)$$

$$2S_N = N(N+1)$$

$$S_N = \frac{N(N+1)}{2}$$

1+N = N+1
2+N-1 = N+1
3+N-2 = N+1
…
N-1+2 = N+1
N+1 = N+1
(N total)

Bubble Sort Algorithm – Basic Version

```
METHOD BubbleSort IMPORT array EXPORT array

FOR pass ← 0 TO (array.length-1)-1 DO                ← Need N-1 passes to guarantee
  sorted
    FOR ii ← 0 TO (array.length-1 - pass)-1 DO       ← NOTE: 0-based array
  indexing
        IF (array[ii] > array[ii+1]) THEN            ← Avoid >= to keep the sort
  stable
            temp ← array[ii]                  ← Swap out-of-order elements ii and ii+1
            array[ii] ← array[ii+1]
            array[ii+1] ← temp
        ENDIF
    ENDFOR
ENDWHILE
```

Bubble Sort Algorithm – 'Optimised'

```
METHOD BubbleSort IMPORT array EXPORT array

pass ← 0
DO
   sorted ← TRUE                              ← Assume sorted – we'll find out if it's not
   FOR ii ← 0 TO (array.length-1 - pass)-1 DO   ← NOTE: 0-based array indexing
      IF (array[ii] > array[ii+1]) THEN
         temp ← array[ii]                     ← Swap out-of-order elements ii and ii+1
         array[ii] ← array[ii+1]
         array[ii+1] ← temp
         sorted ← FALSE                       ← Still need to continue sorting
      ENDIF
   ENDFOR
   pass ← pass + 1                            ← Next pass
WHILE (NOT sorted)                            ← Stop sorting passes when the array is sorted
```

Bubble Sort – Discussion

- Problem: Lots of swaps as well as comparisons
- Good example of why we *don't* focus on best case
  - Best case looks great: O(N) if sorted or almost-sorted
    - BUT: 'almost-sorted' has a *very* specific meaning here!
    - In particular, it requires that small elements start *not very **far** from their final sorted position.* Unlikely except if already-sorted
      - *e.g.,* An array with its smallest value at the end *but is otherwise perfectly sorted* will still take $O(N^2)$ for bubble sort!!!
      - … since the smallest value needs to 'bubble down' to the front
- Average/worst cases of $O(N^2)$ are even worse than they look due to the sheer amount of swaps involved
  - Other $O(N^2)$ algorithms manage with far fewer swaps

Insertion Sort – Strategy

- Inspired from the idea of adding items to an array in sorted order
  - Every time a new item is added, insert it in sorted position
- Can also be applied to sorting an existing array
  - Maintain a marker (index) and insertion-sort the element at the marker into the items to the left of the marker
    - *i.e.,* take the next element and insert it in sorted order into the sub-array that precedes the element
    - Start the marker at index 1 and move it up by one after each inserted element. Then elements before marker will be sorted
    - Searches for the insert position *backwards* so that we can take advantage of semi-sorted arrays

Insertion Sort – Time Complexity

- For each element, we must find the place to insert at
  - Best case: each element already is at insertion point
    - *i.e.,* the N elements are being added in *sorted order*
    - Complexity: O(N) : O(1) compares, done N times
  - Worst case: must go through all elements in each pass
    - *i.e.,* the N elements are being added in *reverse sorted order*
    - 1 + 2 + … + N compares + 1 + 2 + … + N swaps
    - = N(N+1)/2 compares + N(N+1)/2 swaps $\approx N^2$
    - Complexity: $O(N^2)$
  - Average case: go through half the elements in each pass
    - 1/2 + 2/2 + 3/2 + 4/2 + … + N/2 steps = N(N+1)/4 steps $\approx N^2$
    - Complexity: $O(N^2)$

Insertion Sort Algorithm

```
METHOD InsertionSort IMPORT array EXPORT array

FOR nn ← 1 TO array.length-1 DO                          ← Start inserting at element 1 (0 is pointless)
  ii ← nn                                                ← Start from the last item and go backwards
                                                           NOTE: 0-based array indexing
   WHILE (ii > 0) AND (array[ii-1] > array[ii]) DO       ← Insert into sub-array to left of nn
                                                           Use > to keep the sort stable
      temp ← array[ii]                                   ← Move insert val (at ii) up by one via
      array[ii] ← array[ii-1]                              swapping [ii-1] with [ii]
      array[ii-1] ← temp

      ii ← ii - 1
   ENDWHILE

ENDFOR
```

Insertion Sort Alternative

```
METHOD InsertionSort IMPORT array EXPORT array

FOR nn ← 1 TO array.length-1 DO
   ii ← nn

   temp ← array[ii]
   WHILE (ii > 0) AND (array[ii-1] > temp) DO

       array[ii] ← array[ii-1]
       ii ← ii - 1
   ENDWHILE
   array[ii] ← temp
ENDFOR
```

← Start inserting at element 1 (0 is pointless)
← Start from the last item and go backwards
NOTE: 0-based array indexing

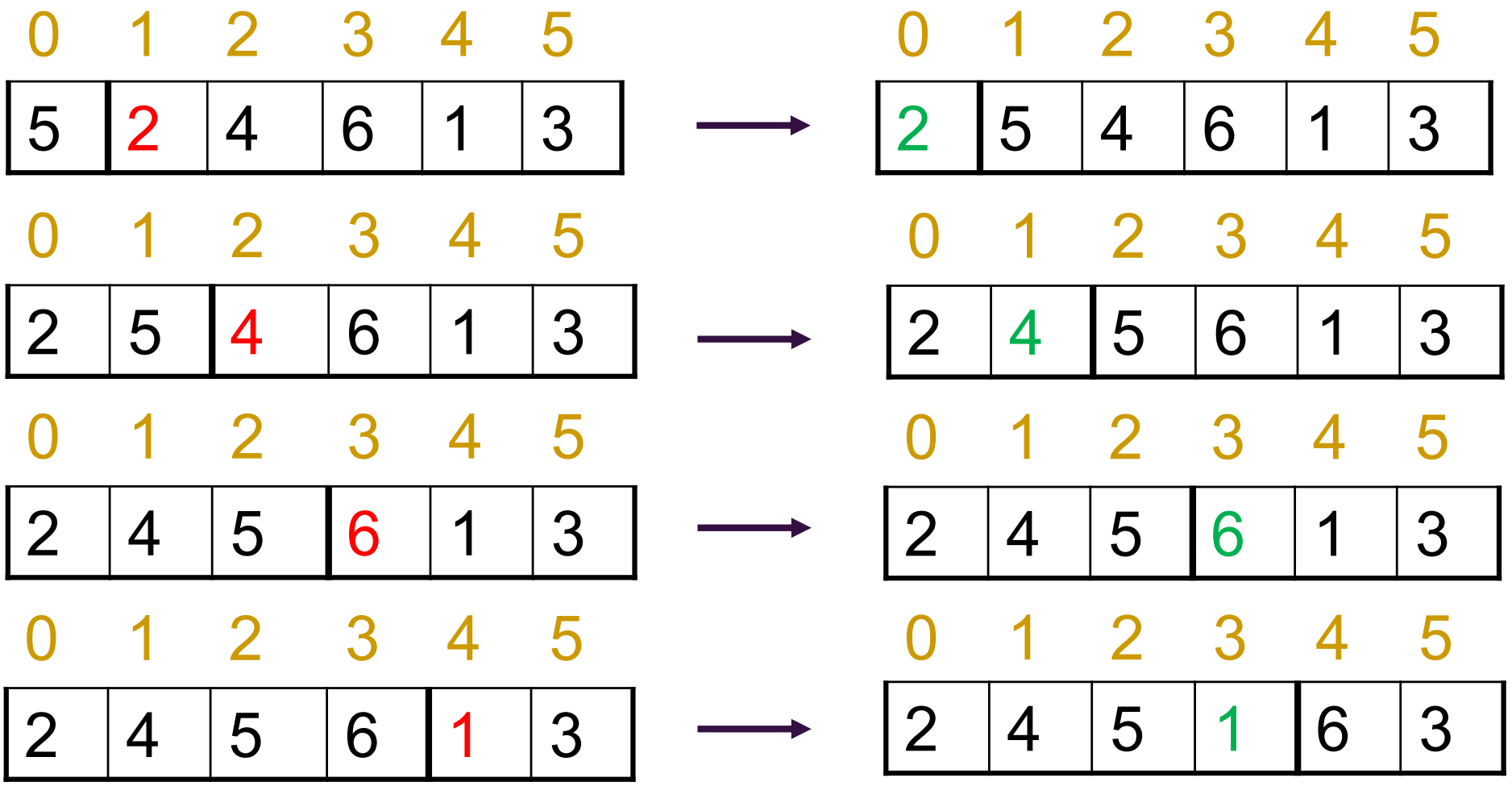
← Insert into sub-array to left of nn
Use > to keep the sort stable
← Shuffle until correct location

Only requires half the work inside the loop!

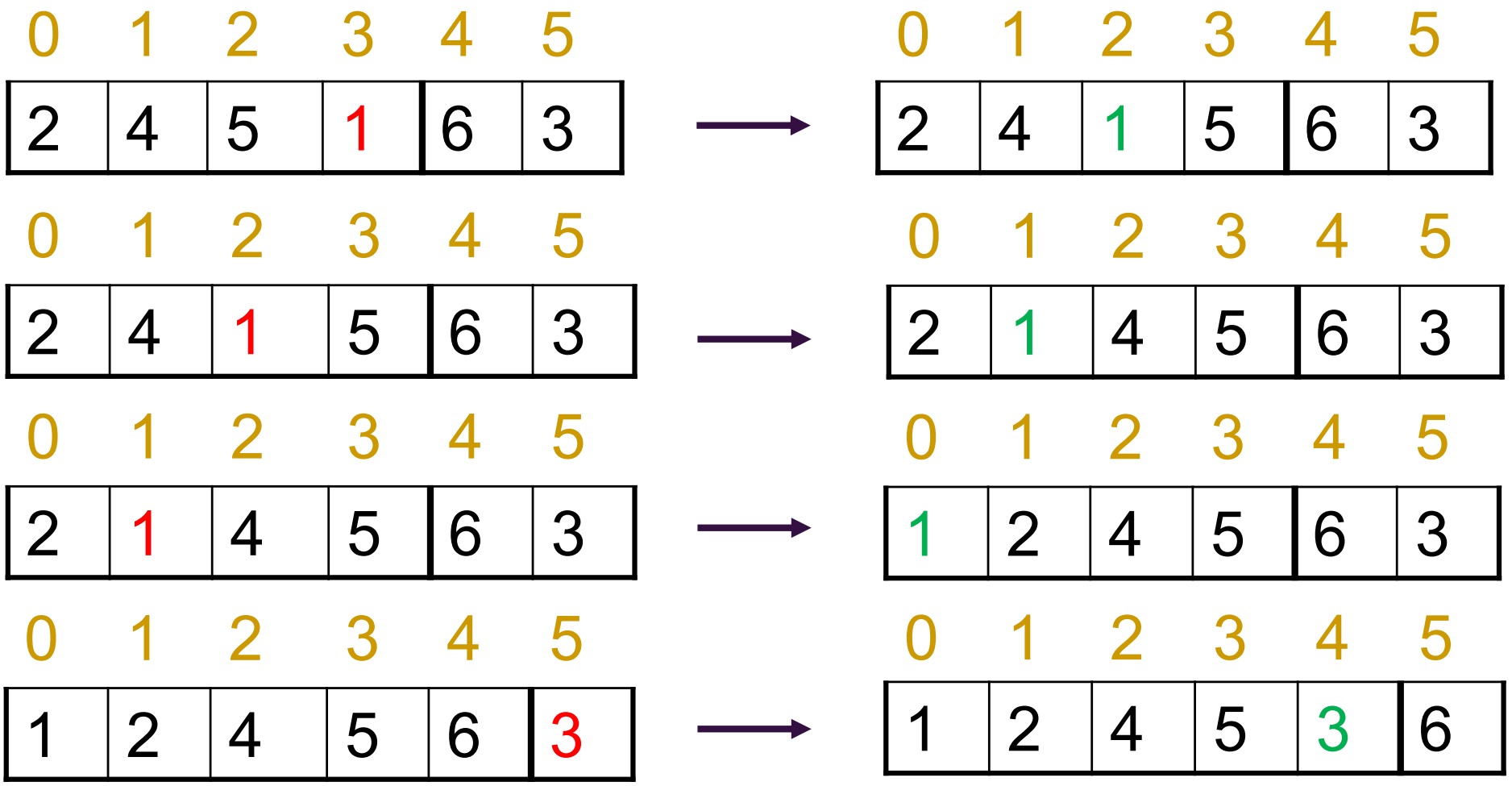
Insertion Sort – Discussion

- Although it looks superficially like Bubble Sort in its time complexity, insertion sort is generally superior
  - Better at being efficient with semi-sorted data
    - Elements are placed in their sorted position *directly*
      - … whereas Bubble Sort is swapping things around all the time
  - Shell Sort is a faster $O(N^{3/2})$ variant that exploits semi-sorted data even better than Insertion Sort
- But Insertion Sort still does a lot of swaps per pass
  - In fact, the same number of swaps as compares
    - …because once the insertion point is found, the pass is complete
  - On average, P/2 compares + P/2 swaps (P = pass number)

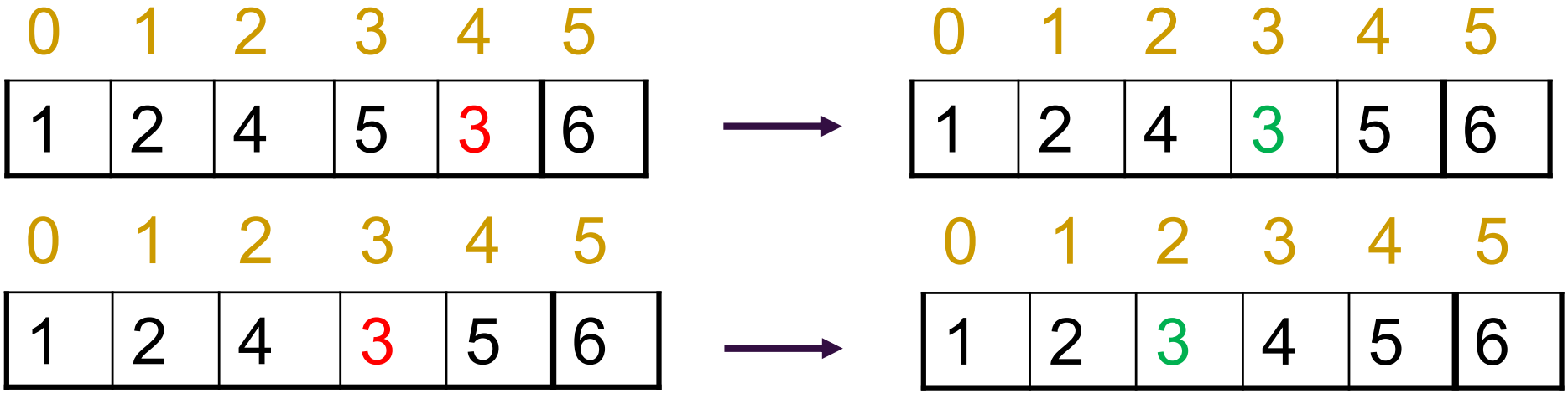
Example of Insertion Sort



Example of Insertion Sort



Example of Insertion Sort



Selection Sort – Strategy

- Select smallest item and swap it with the first item
  - Requires one full pass through the array to find smallest
- Repeat with the second-smallest item, swapping it with the second item
- Repeat until all items have been selected and placed in their sorted position
  - *i.e.,* needs N-1 passes (N^{th} pass isn't needed since its job will be done automatically by the previous passes)

Selection Sort – Time Complexity

- N-1 passes in total
- The P^{th} pass does (N – P) comparisons for selection
  - Don't need to check first P – they're already sorted
- Only one swap per pass
  - The pass only *identifies* the smallest element – it is swapped after the pass finishes
- Thus Best = Average = Worst case
  - There are *always* N-1 passes of: N-P compares + 1 swap
    - N-1+1 + N-2+1 + N-3+1 + … + 1+1
    - = N + N-1 + N-2 + … + 1 = N(N+1)/2 steps ≈ N^2
  - Complexity: $O(N^2)$

Selection Sort Algorithm

```
METHOD SelectionSort IMPORT array EXPORT array

FOR nn ← 0 TO array.length-1 DO                    ← NOTE: 0-based indexing
   minIdx ← nn
   FOR jj ← nn+1 TO array.length-1 DO              ← Ignore nn – that's already our initial minIdx
      IF (array[jj] < array[minIdx]) THEN
         minIdx ← jj                               ← Update newly-found minimum val position
      ENDIF
   ENDFOR

   temp ← array[minIdx]                            ← Now do the swap
   array[minIdx] ← array[nn]
   array[nn] ← temp

ENDFOR
```

Selection Sort – Discussion

- Only one swap per pass
  - Each selected item is placed directly in the position it will ultimately occupy and never swapped again
- But always do N-P comparisons per pass P
  - With N passes, selection sort is thus $O(N^2)$ in all cases
- Summary: many passes with minimal work per pass
  - Consistent speed: bit faster on avg than other $O(N^2)$ sorts
  - But does not take advantage of semi-sorted data like Insertion Sort or (to a lesser extent) Bubble Sort

$O(N^2)$ Sorts – Comparison

- That's all the $O(N^2)$-class sorting algorithms that we will be looking at
  - So how do they compare to one another?

Bubble Sort

- ☑ Simple to implement
- ☑ Can finish early (*i.e.,* faster) if data is almost-sorted
  - But ‘almost-sorted’ has a *very* specific meaning here!
  - In particular, it requires that smaller elements start *not very **far** from their final sorted position*
    - This rarely happens, hence the O(N) best case degrades *very quickly* to $O(N^2)$ on pretty much all but already-sorted data
- ☒ Lots of work per pass – constantly swapping
- ☒ Very slow on reverse-ordered data
  - Ends up swapping *every* element on *each* pass

Insertion sort

- ☑ Very fast with almost-sorted data (minimal swaps/compares)
  - Plus, performance degrades 'gently' from best case
    - *e.g.,* single out-of-place elements don't destroy efficiency
  - Hence effective with most arrays that are partially ordered
- ☑ Stable sort (discussed in later slides)
- ☒ Conceptually trickier than other $O(N^2)$ sorts
- ☒ Lots of swaps – need to shuffle larger elements up
- ☒ Very slow on reverse-ordered data

Selection Sort

- ☑ Simple to implement
- ☑ Minimal work per pass – only one swap
  - Thus generally faster in the average case than the others
- ☑ Best case = average case = worst case
  - *i.e.,* always has to perform the full N-1 passes, and each pass always involves the same amount of work no matter what the unsorted array looks like
    - P checks + 1 swap
  - But this consistency could be considered a positive too, depending on the situation
- ☑ In-place (discussed next)
- ☒ Unstable sort (we'll discuss what this means next)

Other Factors in Sorting

- Speed is not the only consideration in algorithms
- Extra memory use (memory overhead) is another
  - In sorting, extra memory is often needed for temporary storage to help organise the data
    - In-place vs not in-place sorting
- And different problem domains have their own particular issues
  - For sorting, one of them is whether two identical values will stay in the same relative order after sorting
    - Stable vs unstable sorting

In-Place Sorting

- All the $O(N^2)$ sorting algorithms we've looked at didn't need much temporary storage
  - Only enough to handle the swap, *i.e.,* one temp element
- This makes them 'in-place' sorting algorithms
  - *i.e.,* they sort the array in the same place as the array, without needing to copy chunks to another temp array
- Some sorting algorithms are not in-place
  - For very large data sets that fill up almost all RAM, the extra space of non-in-place sorting can be a problem!

Stable vs Unstable Sorting

- If you look closely at the $O(N^2)$ sorting algorithms, you'll see that the compares were done carefully
  - *i.e.,* we chose *very deliberately* whether to do > or >=
  - Goal: choose the compare that will ensure that duplicate values will remain in the same order w.r.t. each other
- Why bother? So what if identical items get swapped?
  - Ah, but we rarely sort just a list of *single values*
  - Instead, we usually sort *rows* of data, by choosing one of the *columns* to use as the sorting key (*i.e.,* the comparer)
    - The other columns get 'dragged along' with the sort column
    - These other columns *won't* be identical, hence we'd like to preserve the relative order of rows with identical sort keys

Stable vs Unstable Sorting

- Thus a 'stable sort' is one which *guarantees* that identically-valued sort keys will keep their ordering
  - And conversely, an unstable sort is one that does not
- Bubble sort and Insertion sort are both stable
  - But only because we carefully selected > or >=
    - (they all ended up as >, but that's not a blanket rule!)
- Selection sort is unstable
  - Consider selection sort on the list of values 8, 8, 3
  - The first 8 will swap with the 3, then no more swaps
    - This puts the first 8 after the second 8 – unstable
  - No matter how you code it, Selection Sort cannot guarantee stability for all possible arrays

Summary of Sorting Algorithms

	Pros	Cons
Bubble Sort	Simple to implement Fast if already sorted In-place, stable sort	Generally poor speed – too many swaps In practice, only ever comes close to best case performance if data is *already* sorted
Insertion Sort	Works relatively fast (vs other $O(N^2)$) with a variety of semi-sorted data In-place, stable sort	Not particularly simple to implement Slow on reversed / near-reversed data
Selection Sort	Simple to implement Minimal work per pass (only one swap so worst case no worse than average case) In-place	Best, avg and worst cases are all identical -ie: takes no advantage of semi-sorted data Unstable sort

Summary of Sorting: Big-O

	Best Case	Average Case	Worst Case
Bubble Sort	$O(N)^{\dagger}$	$O(N^2)$	$O(N^2)$
Insertion Sort	$O(N)$	$O(N^2)$	$O(N^2)$
Selection Sort	$O(N^2)$	$O(N^2)$	$O(N^2)$

† In practice this case really only occurs if the data is already sorted

Sorting Visualisations Revisited

- Each semester we find new sorting visualisation videos
- Here's some more colourful ones:
  - https://www.youtube.com/watch?v=IjIViETya5k
  - https://www.youtube.com/watch?v=bcwwM6EoveA
- And my new favourite:
  - https://www.youtube.com/watch?v=QOYcpGnHH0g
- And if you've got lots of time…
  - https://www.youtube.com/watch?v=YyerMJlmtts
  - https://www.youtube.com/watch?v=lyZQPjUT5B4

Practicals and Next Week

- Practicals:
  - File I/O
  - Sorting algorithms

- Next Week:
  - Revising Objects
  - Stacks and Queues