# Lecture #0: Introduction

## COSC 3020: Algorithms and Data Structures

Lars Kotthoff[1]
larsko@uwyo.edu

[1] with material from various sources

# Course Information

## Instructor

Lars Kotthoff, `larsko@uwyo.edu`

## Course website

WyoCourses

## Office hours

EERB 422b or on Zoom, by appointment (send me an email)

## TA

Zac Harris, `zharris1@uwyo.edu`

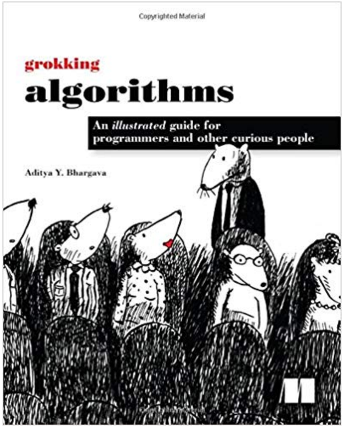
# My Research

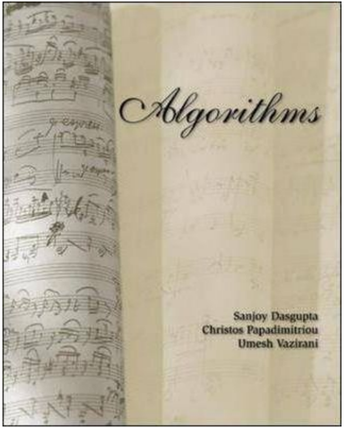
- empirical complexity of algorithms – how do they actually behave in practice
- using machine learning to model the behavior of algorithms
- in particular algorithms to solve challenging AI problems
- also: applying machine learning to advanced materials
- undergraduate internship projects available,
  `https://www.mallet.ai/projects.html`

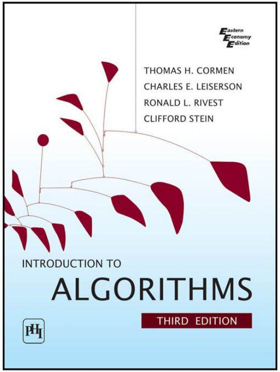
# Course Outline

- Review – this is what you should know
- Sorting
- Graph Algorithms and NP-completeness (hard problems)
- Dynamic Programming
- Asynchronicity and Parallelism

# Optional Textbooks




...and there's many more.

# Course Policies

No late work; may be flexible with advance notice.

| | |
|---|---|
| 5% | attending office hours[2] (except last week of teaching and Final's Week) |
| 15% | labs |
| 20% | assignments |
| 20% | midterm exam |
| 40% | final exam |

Partial points for partially correct answers will be given at the discretion of the grader; if you do not answer the question that was asked you will not get full points.

[2] Or regularly asking/answering questions on WyoCourses.

# JavaScript

- you are expected to know JavaScript or learn on your own
- only basic JavaScript in this course
- you can check out a laptop for the semester from IT
- not a programming class![3]
- can use your browser's JavaScript console, but I recommend node.js (`https://nodejs.org/`)
- some exercises will use server-side JS (=node.js)

[3] exception I'm willing to make: recursion

# Collaboration

You may work in groups of two people on:

- labs
- assignments

Both partners will get the same grade; in particular I will not entertain any accusations that your partner didn't do the work etc.

Always acknowledge any help you had – your lab/homework partner, TA, instructor, website...

# Plagiarism – Don't do it!

Copying without acknowledgment is plagiarism. Minimum penalty 0 points for assignment.

Examples (not exhaustive):

- copying code from a website for part of an assignment, including small modifications to it (e.g. converting to JavaScript from another language)
- copying an answer from an example solution you found online
- working with a partner without acknowledgment

Penalties (not exhaustive):

- no points for an entire assignment (not just the part that was plagiarized)
- F in the course
- suspension

Zero tolerance towards plagiarism. All cases will be referred to the Associate Dean of Undergraduate Education and a note will be made on your student record. If unsure, ask!

# Course Mechanics

- WyoCourses for labs, assignments, slides, discussion `wyocourses.uwyo.edu`
- Labs start next week, (roughly) every week
- feedback – please use "Instructor Feedback" on WyoCourses

# Help

Ask!

- other students
- TAs, instructors
- Tutoring
  - https://www.uwyo.edu/step/
  - https://www.uwyo.edu/ceas/resources/current-students/tutoring.html
- the interwebs (e.g. Stackoverflow for programming questions, see https://stackoverflow.com/help/how-to-ask)
- https://github.com/trekhleb/javascript-algorithms

Your degree is your responsibility.

# Algorithms and Data Structures

- What is an algorithm?

# Algorithms and Data Structures

- What is an algorithm? High-level, language-independent description of step-by-step process for solving a problem.

# Algorithms and Data Structures

- What is an algorithm? High-level, language-independent description of step-by-step process for solving a problem.
- What is a data structure?

# Algorithms and Data Structures

- What is an algorithm? High-level, language-independent description of step-by-step process for solving a problem.
- What is a data structure? Specialized format for organizing and storing data efficiently.

# Algorithms and Data Structures

- What is an algorithm? High-level, language-independent description of step-by-step process for solving a problem.
- What is a data structure? Specialized format for organizing and storing data efficiently.

Particular algorithms may work (better) with particular data structures.

# Observations

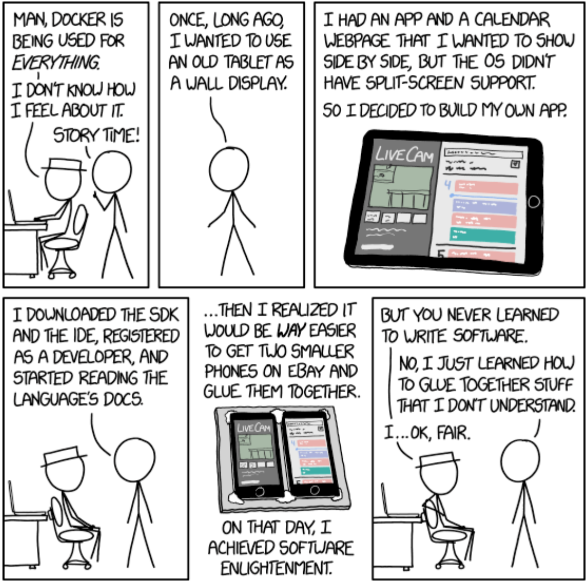
- programs manipulate data
  - programs process, store, display, gather data
  - data can be text, numbers, images, sound
- programs must decide how to store and manipulate data
- choice affects behavior of the program
  - execution speed
  - memory requirements
  - maintenance (debugging, extending, etc.)

Being able to analyze this behavior is what separates good programmers from bad programmers.

# Goals of the Course

- become familiar with some of the fundamental data structures and algorithms in computer science and learn when to use them
- improve your ability to solve problems abstractly with algorithms and data structures as the building blocks
- improve your ability to analyze algorithms (prove correctness; gauge, compare, and improve time and space complexity)
- become modestly skilled with JavaScript (but this is largely on your own)

# Motivation


MAN, DOCKER IS BEING USED FOR EVERYTHING.
I DON'T KNOW HOW I FEEL ABOUT IT.
STORY TIME!
ONCE, LONG AGO, I WANTED TO USE AN OLD TABLET AS A WALL DISPLAY.
I HAD AN APP AND A CALENDAR WEBPAGE THAT I WANTED TO SHOW SIDE BY SIDE, BUT THE OS DIDN'T HAVE SPLIT-SCREEN SUPPORT.
SO I DECIDED TO BUILD MY OWN APP.
LIVE CAM
I DOWNLOADED THE SDK AND THE IDE, REGISTERED AS A DEVELOPER, AND STARTED READING THE LANGUAGE'S DOCS.
...THEN I REALIZED IT WOULD BE WAY EASIER TO GET TWO SMALLER PHONES ON EBAY AND GLUE THEM TOGETHER.
LIVE CAM
ON THAT DAY, I ACHIEVED SOFTWARE ENLIGHTENMENT.
BUT YOU NEVER LEARNED TO WRITE SOFTWARE.
NO, I JUST LEARNED HOW TO GLUE TOGETHER STUFF THAT I DON'T UNDERSTAND.
I...OK, FAIR.

# Motivation – Array Construction

```
var length = 100000;

console.time("array1");
var arr = new Array();
for(let i = 0; i < length; i++) {
  arr.push(1);
}
console.timeEnd("array1");

console.time("array2");
var arr = new Array();
for(let i = 0; i < length; i++) {
  arr.unshift(1);
}
console.timeEnd("array2");
```

# Motivation – Array Destruction

```
console.time("array3");
var arr = new Array(length).fill(1);
for(let i = 0; i < length; i++) {
  arr.pop();
}
console.timeEnd("array3");

console.time("array4");
var arr = new Array(length).fill(1);
for(let i = 0; i < length; i++) {
  arr.shift();
}
console.timeEnd("array4");

console.time("array5");
var arr = new Array(length).fill(1);
arr.reverse();
for(let i = 0; i < length; i++) {
  arr.pop();
}
console.timeEnd("array5");
```

# Motivation – Sorting

```
var arr = new Array();
for(let i = 0; i < length; i++) {
  arr.push(Math.random() * length);
}
var arr1 = JSON.parse(JSON.stringify(arr));

console.time("array6");
arr.sort(function(a, b) { return a - b; });
console.timeEnd("array6");

console.time("array7");
while(true) {
    let swaps = 0;
    for(let i = 1; i < length; i++) {
      if(arr1[i-1] > arr1[i]) {
          let tmp = arr1[i-1];
          arr1[i-1] = arr1[i];
          arr1[i] = tmp;
          swaps++;
      }
    }
    if(swaps == 0) break;
}
console.timeEnd("array7");
```

# Motivation – Sorting

```
var arr = new Array();
for(let i = 0; i < length; i++) {
  arr.push(i);
}
var arr1 = JSON.parse(JSON.stringify(arr));

console.time("array8");
arr.sort(function(a, b) { return a - b; });
console.timeEnd("array8");

console.time("array9");
while(true) {
    let swaps = 0;
    for(let i = 1; i < length; i++) {
      if(arr1[i-1] > arr1[i]) {
          let tmp = arr1[i-1];
          arr1[i-1] = arr1[i];
          arr1[i] = tmp;
          swaps++;
      }
    }
    if(swaps == 0) break;
}
console.timeEnd("array9");
```

# Review – what you should know

# Analyzing Algorithms

- analysis of an algorithm gives insight into
  - how long the program runs (time complexity or runtime) and
  - how much memory it uses (space complexity)
- analysis can provide insight into alternative algorithms
- input size is indicated by a non-negative integer $n$ (sometimes there are multiple measures of an input's size)
- running time is a real-valued function of $n$ such as:
  - $T(n) = 4n + 5$
  - $T(n) = 0.5n \log n - 2n + 7$
  - $T(n) = 2^n + n^3 + 3n$

# Analyzing Code

- single operations: constant time
- consecutive operations: sum operation times
- conditionals: condition time plus max of branch times
- loops: sum of loop-body times
- function call: time for function

Above all, use common sense!

# Rates of Growth

Suppose a computer executes 1 operation per picosecond (trillionth) (for comparison: this laptop does ≈4 500 000 000 operations per second):

| $n =$ | 10 |
|---|---|
| $\log_{10} n$ | 1ps |
| $n$ | 10ps |
| $n \log_{10} n$ | 10ps |
| $n^2$ | 100ps |
| $2^n$ | 1ns |

# Rates of Growth

Suppose a computer executes 1 operation per picosecond (trillionth) (for comparison: this laptop does ≈4 500 000 000 operations per second):

| $n =$ | 10 | 100 |
|---|---|---|
| $\log_{10} n$ | 1ps | 2ps |
| $n$ | 10ps | 100ps |
| $n \log_{10} n$ | 10ps | 200ps |
| $n^2$ | 100ps | 10ns |
| $2^n$ | 1ns | 1Es |

Exasecond(Es) = 32 billion years (age of the universe ≈13.8 billion years)

# Rates of Growth

Suppose a computer executes 1 operation per picosecond (trillionth) (for comparison: this laptop does ≈4 500 000 000 operations per second):

| $n =$ | 10 | 100 | 1,000 |
|---|---|---|---|
| $\log_{10} n$ | 1ps | 2ps | 3ps |
| $n$ | 10ps | 100ps | 1ns |
| $n \log_{10} n$ | 10ps | 200ps | 3ns |
| $n^2$ | 100ps | 10ns | 1$\mu$s |
| $2^n$ | 1ns | 1Es | $10^{289}$s |

Exasecond(Es) $=$ 32 billion years (age of the universe ≈13.8 billion years)

# Rates of Growth

Suppose a computer executes 1 operation per picosecond (trillionth) (for comparison: this laptop does ≈4 500 000 000 operations per second):

| $n =$ | 10 | 100 | 1,000 | 10,000 |
|---|---|---|---|---|
| $\log_{10} n$ | 1ps | 2ps | 3ps | 4ps |
| $n$ | 10ps | 100ps | 1ns | 10ns |
| $n \log_{10} n$ | 10ps | 200ps | 3ns | 40ns |
| $n^2$ | 100ps | 10ns | 1$\mu$s | 100$\mu$s |
| $2^n$ | 1ns | 1Es | $10^{289}$s | |

Exasecond(Es) $=$ 32 billion years (age of the universe ≈13.8 billion years)

# Rates of Growth

Suppose a computer executes 1 operation per picosecond (trillionth) (for comparison: this laptop does ≈4 500 000 000 operations per second):

| $n =$ | 10 | 100 | 1,000 | 10,000 | $10^5$ |
|---|---|---|---|---|---|
| $\log_{10} n$ | 1ps | 2ps | 3ps | 4ps | 5ps |
| $n$ | 10ps | 100ps | 1ns | 10ns | 100ns |
| $n \log_{10} n$ | 10ps | 200ps | 3ns | 40ns | 500ns |
| $n^2$ | 100ps | 10ns | 1$\mu$s | 100$\mu$s | 10ms |
| $2^n$ | 1ns | 1Es | $10^{289}$s | | |

Exasecond(Es) $=$ 32 billion years (age of the universe ≈13.8 billion years)

# Rates of Growth

Suppose a computer executes 1 operation per picosecond (trillionth) (for comparison: this laptop does ≈4 500 000 000 operations per second):

| $n =$ | 10 | 100 | 1,000 | 10,000 | $10^5$ | $10^6$ |
|---|---|---|---|---|---|---|
| $\log_{10} n$ | 1ps | 2ps | 3ps | 4ps | 5ps | 6ps |
| $n$ | 10ps | 100ps | 1ns | 10ns | 100ns | 1$\mu$s |
| $n \log_{10} n$ | 10ps | 200ps | 3ns | 40ns | 500ns | 6$\mu$s |
| $n^2$ | 100ps | 10ns | 1$\mu$s | 100$\mu$s | 10ms | 1s |
| $2^n$ | 1ns | 1Es | $10^{289}$s | | | |

Exasecond(Es) $=$ 32 billion years (age of the universe ≈13.8 billion years)

# Rates of Growth

Suppose a computer executes 1 operation per picosecond (trillionth) (for comparison: this laptop does ≈4 500 000 000 operations per second):

| $n =$ | 10 | 100 | 1,000 | 10,000 | $10^5$ | $10^6$ | $10^9$ |
|---|---|---|---|---|---|---|---|
| $\log_{10} n$ | 1ps | 2ps | 3ps | 4ps | 5ps | 6ps | 9ps |
| $n$ | 10ps | 100ps | 1ns | 10ns | 100ns | 1$\mu$s | 1ms |
| $n \log_{10} n$ | 10ps | 200ps | 3ns | 40ns | 500ns | 6$\mu$s | 9ms |
| $n^2$ | 100ps | 10ns | 1$\mu$s | 100$\mu$s | 10ms | 1s | 11.6 days |
| $2^n$ | 1ns | 1Es | $10^{289}$s | | | | |

Exasecond(Es) $=$ 32 billion years (age of the universe ≈13.8 billion years)

# Rates of Growth

Suppose a computer executes 1 operation per picosecond (trillionth) (for comparison: this laptop does ≈4 500 000 000 operations per second):

| $n =$ | 10 | 100 | 1,000 | 10,000 | $10^5$ | $10^6$ | $10^9$ | $10^{16}$ |
|---|---|---|---|---|---|---|---|---|
| $\log_{10} n$ | 1ps | 2ps | 3ps | 4ps | 5ps | 6ps | 9ps | 16ps |
| $n$ | 10ps | 100ps | 1ns | 10ns | 100ns | 1$\mu$s | 1ms | 2.7h |
| $n \log_{10} n$ | 10ps | 200ps | 3ns | 40ns | 500ns | 6$\mu$s | 9ms | 44.4h |
| $n^2$ | 100ps | 10ns | 1$\mu$s | 100$\mu$s | 10ms | 1s | 11.6 days | $3 \cdot 10^{12}$ years |
| $2^n$ | 1ns | 1Es | $10^{289}$s | | | | | |

Exasecond(Es) $=$ 32 billion years (age of the universe ≈13.8 billion years)

# Analyzing Code

```
// Linear search
function find(key, array) {
    for(var i = 0; i < array.length; i++) {
        if(array[i] == key) return i;
    }
    return -1;
}
```

- ▷ What's the input size, $n$, and how does it affect runtime?

# Analyzing Code

```
// Linear search
function find(key, array) {
    for(var i = 0; i < array.length; i++) {
        if(array[i] == key) return i;
    }
    return -1;
}
```

- What's the input size, $n$, and how does it affect runtime?
- Should we assume a worst-case, best-case, or average-case for input of size $n$?

# Example

```
if(Math.random() == 0) {
  return;
}
if(Math.random() == 0.5) {
  for(var i = 1; i <= n; i++) {
    for(var j = 1; j <= n; j++) {
      k = 1;
    }
  }
  return;
} else {
  for(var i = 1; i <= n; i++) {
    k = 1;
  }
}
return;
```

# Example

```
if(Math.random() == 0) {
  return;
}
if(Math.random() == 0.5) {
  for(var i = 1; i <= n; i++) {
    for(var j = 1; j <= n; j++) {
      k = 1;
    }
  }
  return;
} else {
  for(var i = 1; i <= n; i++) {
    k = 1;
  }
}
return;
```

Best case: $T(n) = 1$

# Example

```
if(Math.random() == 0) {
  return;
}
if(Math.random() == 0.5) {
  for(var i = 1; i <= n; i++) {
    for(var j = 1; j <= n; j++) {
      k = 1;
    }
  }
  return;
} else {
  for(var i = 1; i <= n; i++) {
    k = 1;
  }
}
return;
```

Best case: $T(n) = 1$

Worst case: $T(n) = n^2$

# Example

```
if(Math.random() == 0) {
  return;
}
if(Math.random() == 0.5) {
  for(var i = 1; i <= n; i++) {
    for(var j = 1; j <= n; j++) {
      k = 1;
    }
  }
  return;
} else {
  for(var i = 1; i <= n; i++) {
    k = 1;
  }
}
return;
```

Best case: $T(n) = 1$

Worst case: $T(n) = n^2$

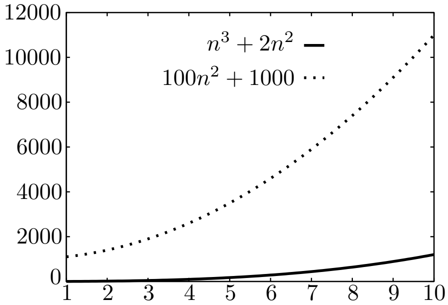
Average case: $T(n) = n$

# Asymptotic Behavior

- we measure runtime as a function of the input size $n$
- we don't care about constants and constant factors
- we are most interested what happens when $n$ gets big
- ...and in particular what happens when the input size changes

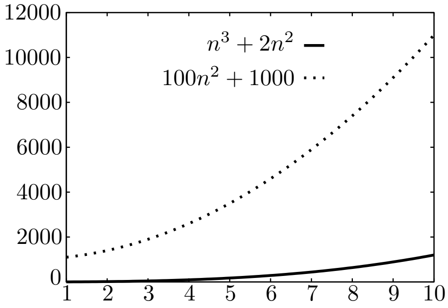
# Example 1

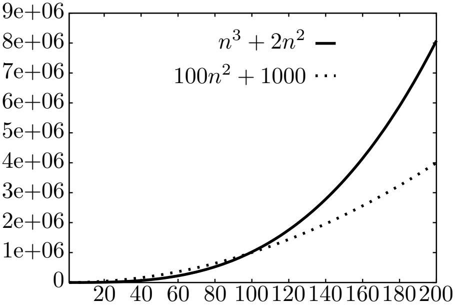
$$n^3 + 2n^2 \qquad \text{versus} \qquad 100n^2 + 1000$$



# Example 1

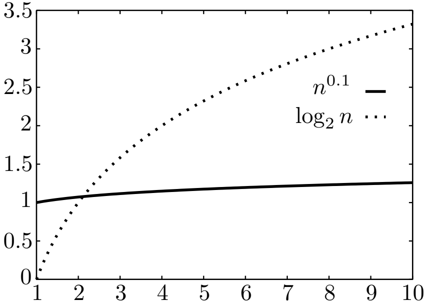
$$n^3 + 2n^2 \qquad \text{versus} \qquad 100n^2 + 1000$$



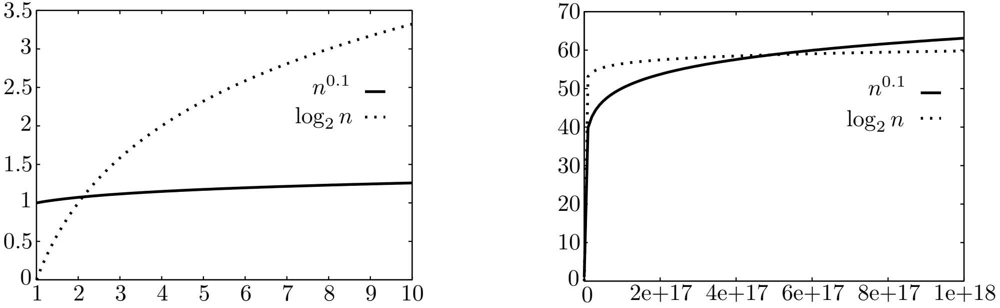
# Example 2

$$n^{0.1} \qquad \text{versus} \qquad \log_2 n$$



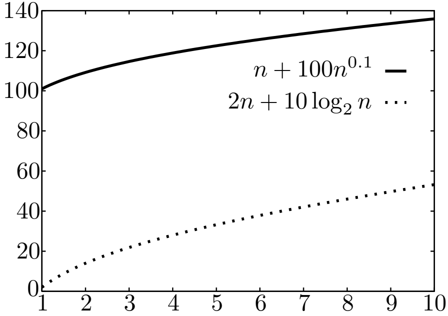
# Example 2

$$n^{0.1} \qquad \text{versus} \qquad \log_2 n$$



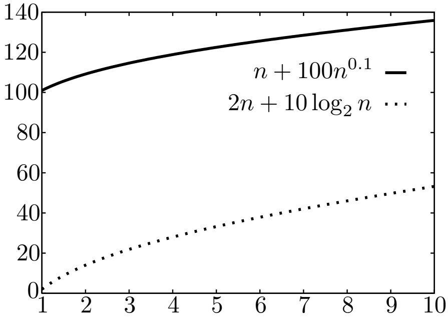
# Example 3

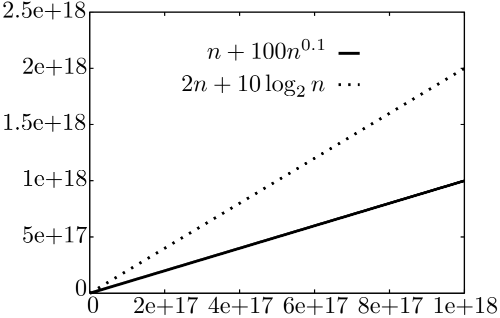
$$n + 100n^{0.1} \qquad \text{versus} \qquad 2n + 10\log_2 n$$



# Example 3

$$n + 100n^{0.1} \qquad \text{versus} \qquad 2n + 10\log_2 n$$



# Asymptotic Notation

- $T(n) \in O(f(n))$ if there are positive[4] constants $c$ and $n_0$ such that $T(n) \leq cf(n)$ for all $n \geq n_0$
- $T(n) \in \Omega(f(n))$ if there are positive constants $c$ and $n_0$ such that $T(n) \geq cf(n)$ for all $n \geq n_0$
- $T(n) \in \Theta(f(n))$ if $T(n) \in O(f(n))$ and $T(n) \in \Omega(f(n))$
- $T(n) \in o(f(n))$ if for any positive constant $c$, there exists $n_0$ such that $T(n) < cf(n)$ for all $n \geq n_0$
- $T(n) \in \omega(f(n))$ if for any positive constant $c$, there exists $n_0$ such that $T(n) > cf(n)$ for all $n \geq n_0$

[4] Remember that 0 is *not* positive.

# Examples

$$10,000n^2 + 25n \in \Theta(n^2)$$

$$10^{-10}n^2 \in \Theta(n^2)$$

$$n \log n \in O(n^2)$$

$$n \log n \in \Omega(n)$$

$$n^3 + 4 \in o(n^4)$$

$$n^3 + 4 \in \omega(n^2)$$

# “Tight” bounds

- informally: want to have “good” bounds
- no better reasonable bound which is asymptotically different
- rigid definition: $\Theta$

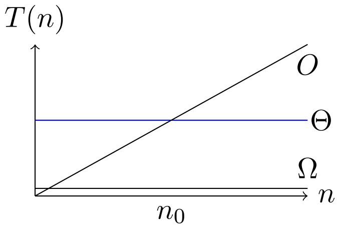
# Example

Best case: $T(n) = 1$

Worst case: $T(n) = n^2$

Average case: $T(n) = n$

# Example




Best case: $T(n) = 1$

Worst case: $T(n) = n^2$

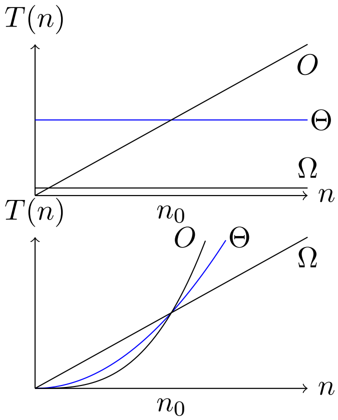
Average case: $T(n) = n$

# Example

Best case: $T(n) = 1$

Worst case: $T(n) = n^2$

Average case: $T(n) = n$

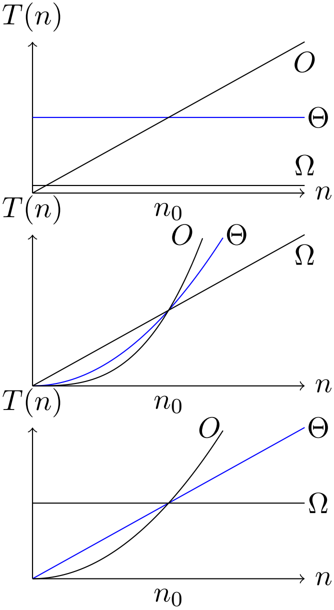


# Example

Best case: $T(n) = 1$

Worst case: $T(n) = n^2$

Average case: $T(n) = n$



# Analysis Cases vs. Asymptotics

- best case $\neq \Omega$, worst case $\neq O$, average case $\neq \Theta$
- execution case provides expression that characterizes execution behavior, which is then categorized according to its asymptotic complexity
- cannot make inferences between different cases in general, i.e. knowing $\Theta$ for the best case does not tell us anything about the average case

# Abstract Data Type

**Formally** mathematical description of an object and the set of operations on the object

**In Practice** interface of a data structure without implementation (think a header file)

# Data Structures as Algorithms

### Algorithm

a high level, language-independent description of a step-by-step process for solving a problem

### Data Structure

a way of storing and organizing data so that it can be manipulated as described by an ADT

A data structure is defined by the algorithms that implement the ADT operations.

# Code Implementation

## Theory

- abstract base class (interface) describes ADT
- concrete classes implement data structures for the ADT
- data structures can change without affecting client code

## Practice

- different implementations sometimes suggest different interfaces (generality vs. simplicity)
- performance of a data structure may influence the form of the client code (time vs. space, one operation vs. another)

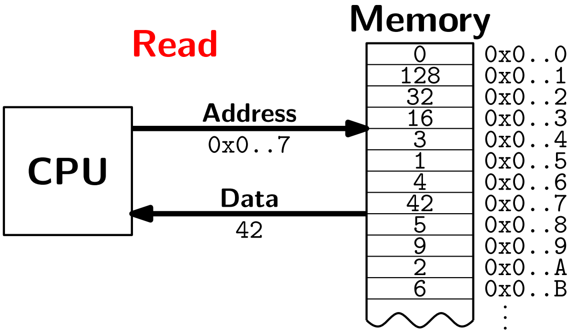
# Some ADTs and Implementations

# Array/List ADT

- store things like integers, strings, etc.
- operations:
  - initialize an empty array
    `var a = [];`
  - access (read or write) the $i$th thing in the array $(0 \leq i \leq n-1)$
    `thing1 = a[i];` Read
    `a[i] = thing2;` Write

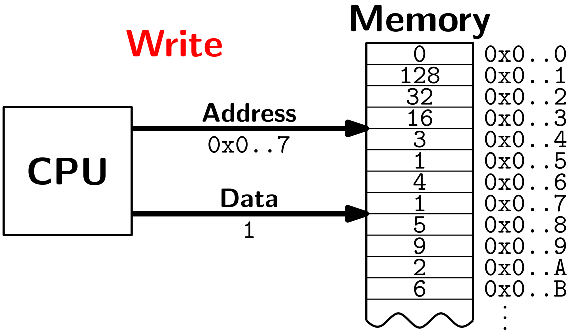
# Why Arrays?

- computer memory is an array
- read: CPU provides address $i$, memory unit returns the data stored at $i$



# Why Arrays?

- computer memory is an array
- write: CPU provides address $i$ and data $d$, memory unit stores data $d$ at $i$



# Why Arrays?

- computer memory is an array
- simple and fast
- used in almost every program
- used to implement other data structures

# Arrays in JavaScript vs. Other Languages

- JavaScript allows you to access undefined elements and use strings as indices[5]

## Other languages (C, C++, Java...)

- need to know size when array is created

[5] Technically you're setting/accessing properties of the array object rather than the array contents this way.

# Arrays in JavaScript vs. Other Languages

- JavaScript allows you to access undefined elements and use strings as indices[5]

## Other languages (C, C++, Java...)

- need to know size when array is created

Fix: Resizeable arrays – if the array fills up, allocate a new, bigger array and copy the old contents to the new array.

[5] Technically you're setting/accessing properties of the array object rather than the array contents this way.

# Arrays in JavaScript vs. Other Languages

- JavaScript allows you to access undefined elements and use strings as indices[5]

## Other languages (C, C++, Java...)

- need to know size when array is created

Fix: Resizeable arrays – if the array fills up, allocate a new, bigger array and copy the old contents to the new array.

- Indices are integers 0,1,2,...

[5] Technically you're setting/accessing properties of the array object rather than the array contents this way.

# Arrays in JavaScript vs. Other Languages

- ▷ JavaScript allows you to access undefined elements and use strings as indices[5]

## Other languages (C, C++, Java...)

- ▷ need to know size when array is created

Fix: Resizeable arrays – if the array fills up, allocate a new, bigger array and copy the old contents to the new array.

- ▷ Indices are integers 0,1,2,...

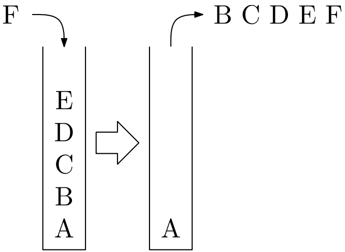
Fix: Hashing.

[5] Technically you're setting/accessing properties of the array object rather than the array contents this way.

# Stack ADT

## Stack operations

- create
- destroy
- push
- pop
- top
- is_empty




## Stack property

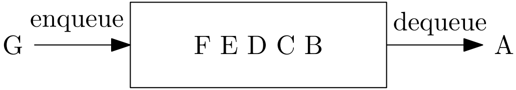
If $x$ is pushed before $y$ is pushed, then $x$ will be popped after $y$ is popped.

LIFO: Last In First Out

# Queue ADT

## Queue operations

- create
- destroy
- enqueue
- dequeue
- is_empty




## Queue property

If $x$ is enqueued before $y$ is enqueued, then $x$ will be dequeued before $y$ is dequeued.

FIFO: First In First Out

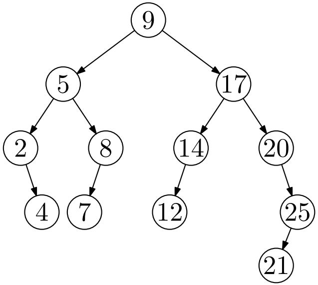
# Binary Search Trees

## Binary tree property

- each node has $\leq 2$ children

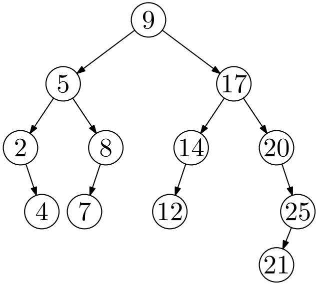
## Search tree property

- all keys in left subtree smaller than node's key
- all keys in right subtree larger than node's key



# Aside: Tree Terminology

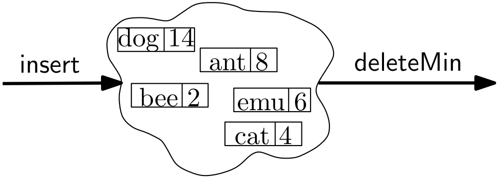
root, leaf, parent, child, sibling, ancestor, descendent, subtree, depth, height, degree, branching factor, complete



# Priority Queue ADT

## Priority Queue operations

- create
- destroy
- insert
- deleteMin
- is_empty




## Priority Queue property

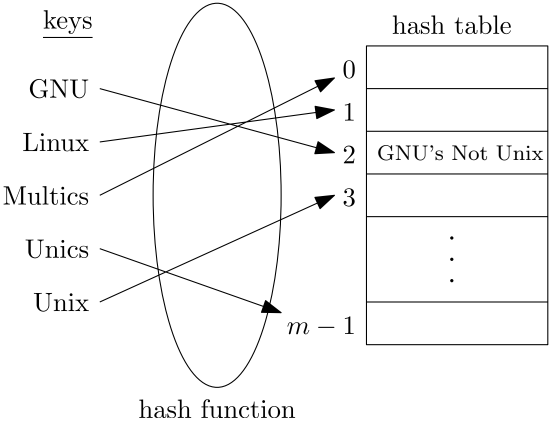
For two elements in the queue, $x$ and $y$, if $x$ has a lower priority value than $y$, $x$ will be deleted before $y$.

# Dictionary ADT

- stores pairs of strings: (word, definition)
- operations:
  - insert(word,definition)
  - delete(word)
  - find(word) $\rightarrow$ definition

# Hash Tables

Use a hash function to map keys to indices.




hash("GNU") $= 2$

# Hash Collisions

## Pigeonhole principle

If more than $m$ pigeons fly into $m$ pigeonholes then some pigeonhole contains at least two pigeons.

Unless we know all the keys in advance and design a perfect hash function, we must handle collisions.

What do we do when two keys hash to the same entry?

- separate chaining: store multiple items in each entry
- open addressing: pick a next entry to try

# Up Next: Sorting