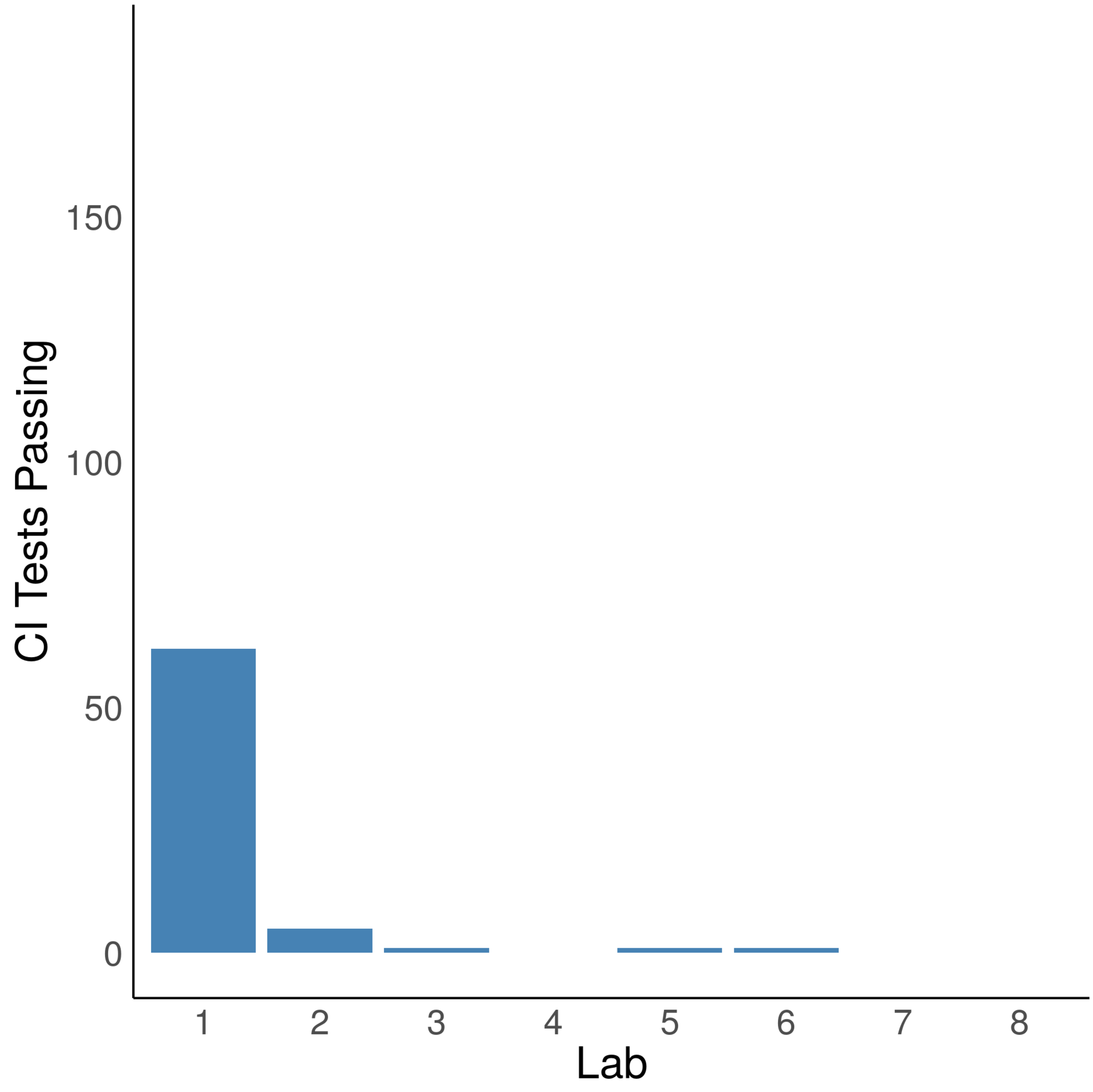
Seminar today…


CI Tests Passing
150
100
50
0
1
2
3
4
5
6
7
8
Lab

Adding Javadoc using Eclipse:

1. Click your method's name
2. Linux and Windows: press keyboard keys: Alt + Shift + j
3. Mac OSX: ⌘ + Alt + J

- *Credit: William Thorenfeldt*

"How to Write Doc Comments for the Javadoc Tool", Oracle. https://www.oracle.com/technetwork/java/javase/tech/index-137868.html

Reminder about GitLab membership:

- Project *membership* for peer feedback from week 3
- Always 1 week after lab deadline
- **Don't add anyone as project member before then**
- GitLab server set up to monitor memberships
- University disciplinary process for code plagiarism

Software Development 3 (F27SG)

Lecture 3

Introduction to Complexity

Rob Stewart

Outline

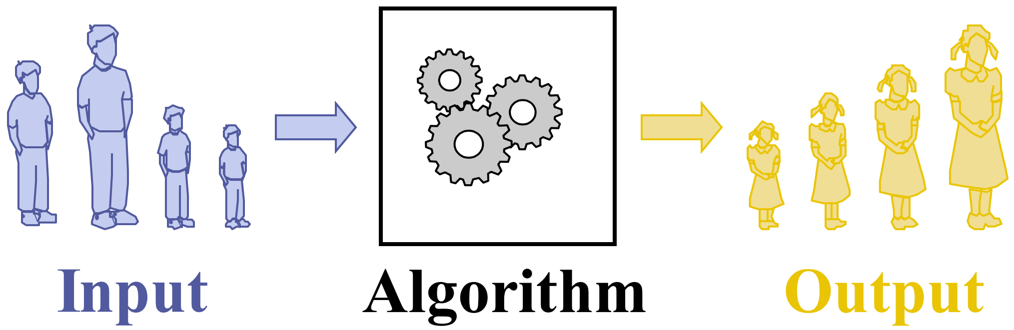
- By the end of this lecture you should
  - understand the importance of analysing algorithms
  - Be able to analyse algorithms using the big-Oh notation for time complexity
  - Abstract Java code to Big-Oh notation

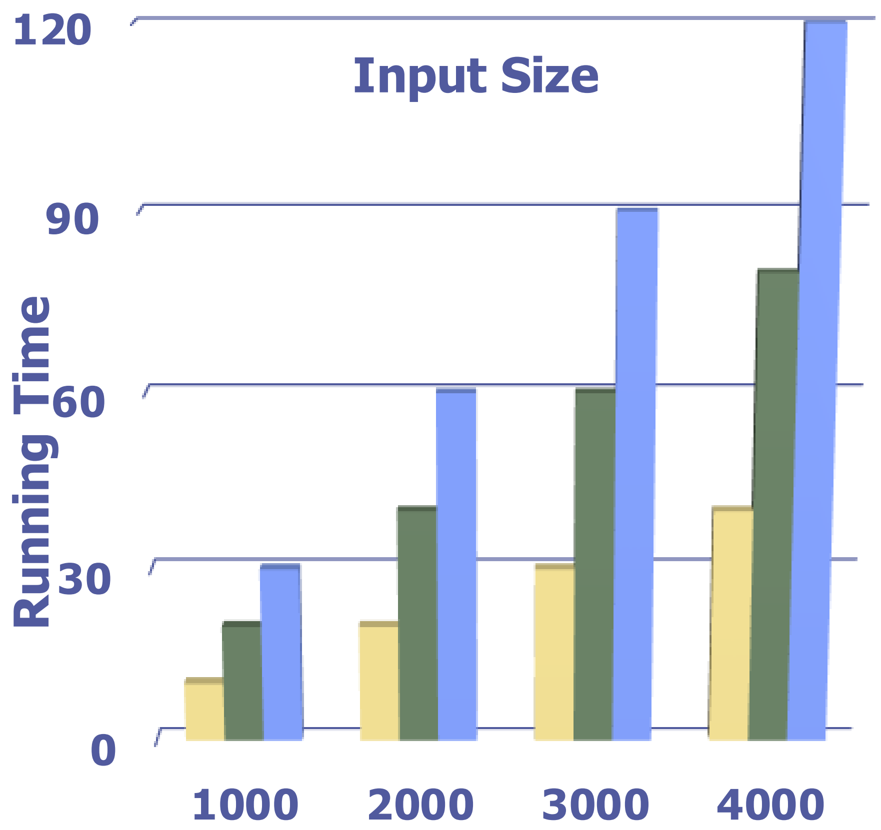
Introduction to Complexity Analysis

- Choice of algorithm & data structure is important
  - in particular when working with large amounts of data
- We need to be able to
  - analyse algorithms & data structures
  - based on this compare them for the problem in hand
- Typically this is with respect to
  - **time** usage
  - **space** usage
  - our focus will be on **time** usage

Running Time

- Most algorithms transforms input objects to output objects
  - running time grows with input
- We can use
  - best case
  - average case
  - worst case
- We focus on worst case
  - easier to analyse
  - most important in many applications
    - banks, car control systems, real time systems





Comparing Algorithms

- Consider the following (correct) ways of
  - multiplying two positive numbers

A
```
int posmult1(int m,int n){
   return m*n;
}
```

B
```
int posmult2(int m,int n){
  int result = 0;
  for(int i = 0; i < m;i++){
    result += n;
   }
   return result;
}
```

Which one do you think is best?

Give an argument why you think one is better

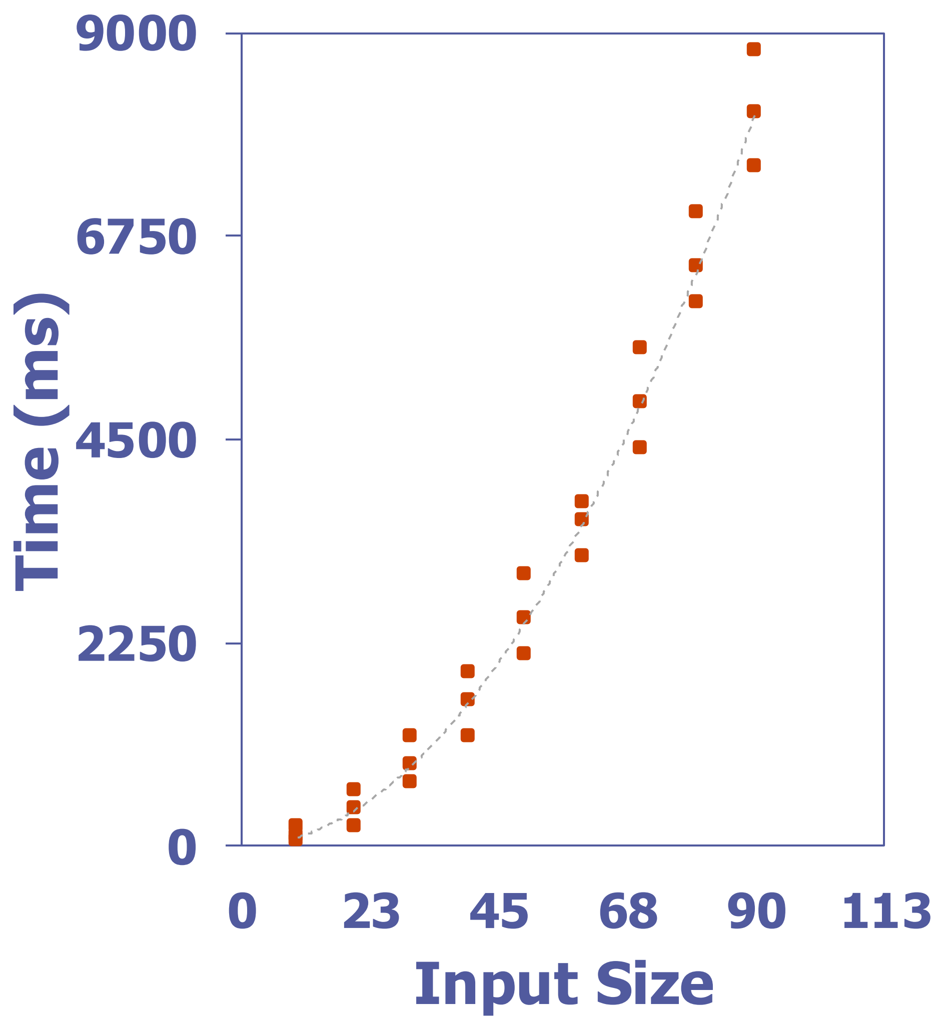
www.socrative.com / `student login' / room: SD32019

Measuring Time

- We can measure an algorithm by implementing it and experimentally
  - run it for different input sizes
  - measure time usage, e.g by

```
long before = System.currentTimeMillis();
program(n);
long after = System.currentTimeMillis();
long time = after - before;
```

  - plot the result



- Problems
  - other inputs may behave differently
  - depends on underlying architecture
  - external events may have impact
    - e.g. other processes using CPU, garbage collection, ...

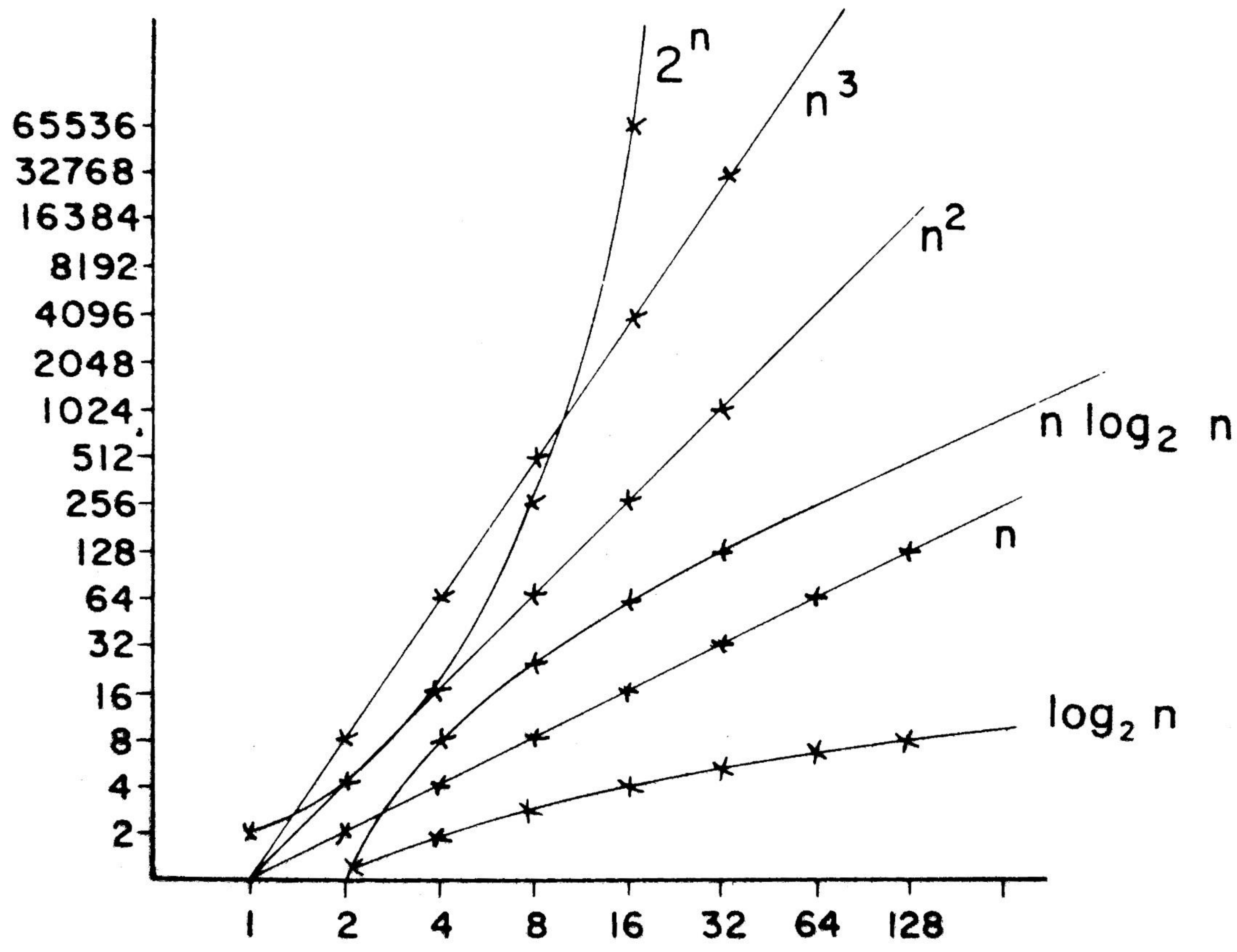
Big-O Notation

- **Big-O** follows a more theoretical approach
- Characterises the running time as a function on the input size **n**
  - don’t take input here too literally,
  - it can e.g. be an array stored as a field in the object

Big-O Notation

- It describes the **growth rate** with respect to the input size
  - *All possible inputs* are taken into account
  - Evaluation is independent of the underlying software and hardware environment
- We take a “big picture” approach where we try to be simple
  - ... but also as close to the actual value as we can
- This is computed by a process called asymptotic analysis

Illustration of Growth Rates



Fundamentals of Data Structures, E Horowitz & S Sahni, 1998.

Complexity in Words

- ***O(1) - "****constant time"*
  - *the time taken doesn't change regardless of input size*
- **O(log n) - "***logarithmic time"*
  - *any algorithm which cuts the problem in half each time.*
  - Operation will take longer as the input size increases, but once the input gets fairly large it won't change enough to worry about.
- ***O(n) -*** *"linear time"*
  - *for every element in 1..n, you are doing a constant number of operations, such as comparing each element to a known value.*
  - Every time you double *n*, the operation will take twice as long.
- ***O(n log n) -*** *"log linear time"*
  - *perform an* ***O(log n)*** *operation for each item in your input.*
  - Every time you double *n*, you spend twice as much time plus a little more.
- ***O(n^2) -*** *"quadratic time"*
  - *for every element in 1..n, you do something with every other element 1..n, such as comparing them.*
  - Every time *n* doubles, the operation takes four times as long. Only practical up to a certain input size.

CompSci 101 - Big-O Notation, December 2010, Dave Perrett.
https://www.daveperrett.com/articles/2010/12/07/comp-sci-101-big-o-notation

Shopping Basket

Shopping list

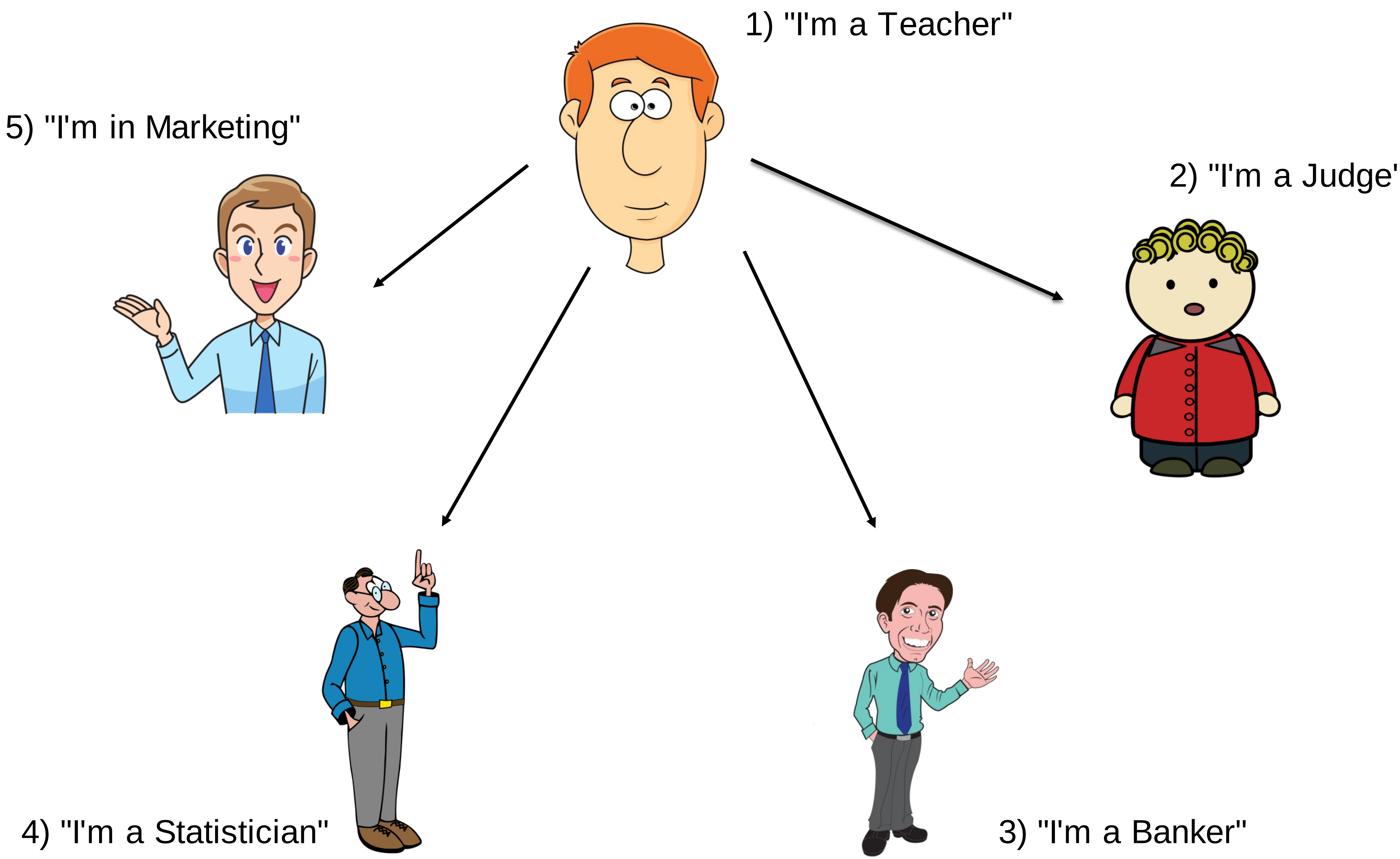
Shopping basket

- Apple
- Banana
- Shampoo
- Bread
- Orange juice

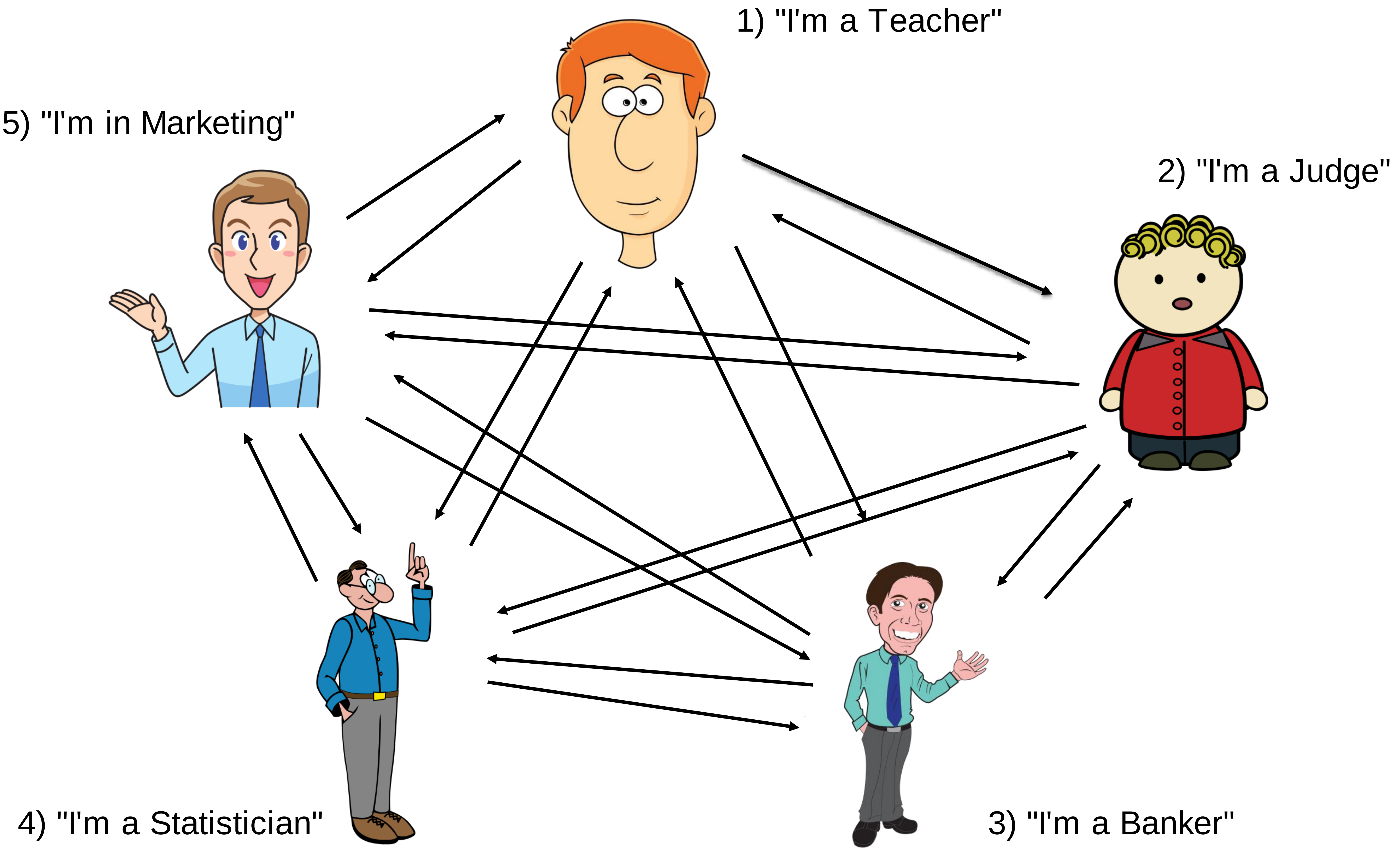
Step 1) check next list item

Step 2) check if it’s in the basket

Job announcements



Job announcements



Address book

How could I find someone in an address book?

Address book

- Finding someone in address book
- 1) start at A, scan to Z (linear)
- 2) start in the middle, letter M
  Either: go to A-L, or N-Z (recursive)

O(N) versus O(N^2)

- Linear **O(N)** complexity of **reading a book**

```
for (int page = 0; page  < N; page++) {
    readPage(page);
}
```

- Quadratic **O(N^2)** complexity of **person introductions:**

```
for (int personA = 0; personA < N; personA++) {
    for (int personB = 0; personB < N; personB++) {
        sayHello(personA , personB);
    }
}
```

- Quadratic **O(N^2)** complexity of **checking your shopping basket**:

```
for (int listItem = 0; listItem < N; listItem ++) {
    for (int basketItem = 0; basketItem < M; basketItem++) {
        if (shoppingList[listItem]  == basket[basketItem])
           break;
    }
}
```

The Big-O Approach

A set of **primitive operations** are defined

- each assumed to have the same running time

1. We find (count) the **worst-case** number of primitive operations
   - expressed as a function on the input size
2. We then **simplify** this function to Big-O notation

Abstracting Code to Big-O formula

- The following constitutes as a primitive
  - assignment
  - calling a method
  - arithmetic operation
  - comparison (e.g. of two numbers)
  - index into an array
  - following an object reference
  - returning from a method

Counting Primitives Example

```
public int count(int [] arr){
  int MAX = arr.length;
  int total = 0;
  int i = 0;
  while (i < MAX){
      total = total + arr[i];
      i++;
  }
return total
}
```

- **Primitives**
  - assignment
  - calling a method
  - arithmetic operation
  - comparison
  - index into an array
  - following an object reference
  - returning from a method

Counting Primitives Example

```
public int count(int [] arr){
  int MAX = arr.length; // assignment + follow ref = 2
  int total = 0;  // assignment = 1
  int i = 0; // assignment = 1
  while (i < MAX){ I/ compare = 1
      total = total + arr[i]; // arith + array lookup + assign = 3
      i++ // (i = i+1) : assign + arith = 2
  }
return total // return = 1
}
```

- Before loop:4
- Loop: 6 per iteration
- After loop: 1
- Total: 6N + 5 (where N is size of array)

Exercises

- **Primitives**
  - assignment
  - calling a method
  - arithmetic operation
  - comparison
  - index into an array
  - follow object reference
  - returning from a method

Q1: How many primitive operations does the following code have (worse case):

```
public boolean isEmpty(){
  return curr == 0; }
```

Q2: How many primitive operations does the following code have (worst case):

```
if (i < j){
  int tmp = i;
  i = j;
  j = tmp;
}else{
  i = j;    }
```

www.socrative.com / `student login' / room: SD32019

Exercises

- **Primitives**
  - assignment
  - calling a method
  - arithmetic operation
  - comparison
  - index into an array
  - follow object reference
  - returning from a method

Q1: How many primitive operations does the following code have (worse case):

```
public boolean isEmpty(){
  return curr == 0; }
```

2

Q2: How many primitive operations does the following code have (worst case):

```
if (i < j){
  int tmp = i;
  i = j;
  j = tmp;
}else{
  i = j;    }
```

4

Common Functions

- We then need to simplify the function
- The following functions are very commonly used:
  - **$O(1)$ - the constant function**
  - **$O(\log n)$ - the logarithmic function**
  - **$O(n)$ - the linear function**
  - **$O(n \log n)$ - the n-log-n function**
  - **$O(n^2)$ - the quadratic function**
  - $O(n^3)$ - the cubic function
  - $O(2^n)$ - the exponential function
- This are listed in *order of complexity*
  - *$O(1)$* is the simplest, while *$O(2^n)$* is the most complex
- *We will come across and introduce the **bold** functions during this course*

Simplifying Functions

- In Big-O we try to write the functions in the **simplest terms**
- The following rules are used to simplify terms
  - **drop lower-order terms**
  - **drop constant factors**
- For our example: *6N+5*
  - we drop the lower-order term *5* (left with 6N)
  - we drop the constant *6* (left with N)
  - meaning it is expressed by the linear function O(N)
- Another example: $5n^3 + 2n^2 + 3n - 4$
  - drop the lower terms: $2n^2$, $3n$ and 4
  - drop the constant 5
  - meaning it is the cubic function $O(n^3)$

Exercise [revisited]

- Compute big-O for each program and figure out which is better:

```
int posmult1(int m,int n){
  return m*n;
}
```

```
int posmult2(int m,int n){
  int result = 0;
  for(int i = 0; i < m;i++){
    result += n;
  }
  return result;
}
```

- **Primitives**
  - assignment
  - calling a method
  - arithmetic operation
  - comparison
  - index into an array
  - follow object reference
  - returning from a method

- **Simplify terms**
  - drop lower-order terms
  - drop constant factors

www.socrative.com / `student login' / room: SD32019

Solution

```
int posmult1(int m,int n){
   return m*n;
}
```

```
int posmult2(int m,int n){
  int result = 0;
  for(int i = 0; i < m;i++){
     result += n;
   }
   return result;
}
```

- posmult1: 2 operation = O(1)
- posmult2:
  - 2 before loop
  - 5 in loop
  - 1 after loop
  - = 5N+3 which is simplified to **O(N)**
- **Conclusion: posmult1 is preferable**

Complexity of Data Structures (weeks 2-7)

DATA STRUCTURE OPERATIONS

Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Stack	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Singly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Doubly-Linked List	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Skip List	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)
Hash Table	–	O(1)	O(1)	O(1)	–	O(n)	O(n)	O(n)
Binary Search Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	–	O(log(n))	O(log(n))	O(log(n))	–	O(n)	O(n)	O(n)
B-Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Red-Black Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))
Splay Tree	–	O(log(n))	O(log(n))	O(log(n))	–	O(log(n))	O(log(n))	O(log(n))
AVL Tree	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))

https://github.com/ro31337/bigoposter

Complexity of Sorting (weeks 9-10)

ARRAY SORTING ALGORITHMS

Algorithm	Time Complexity		
	Best	Average	Worst
Quicksort	O(n log(n))	O(n log(n))	O(n^2)
Mergesort	O(n log(n))	O(n log(n))	O(n log(n))
Timsort	O(n)	O(n log(n))	O(n log(n))
Heapsort	O(n log(n))	O(n log(n))	O(n log(n))
Bubble Sort	O(n)	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)
Shell Sort	O(n)	O((nlog(n))^2)	O((nlog(n))^2)
Bucket Sort	O(n+k)	O(n+k)	O(n^2)
Radix Sort	O(nk)	O(nk)	O(nk)

https://github.com/ro31337/bigoposter

Complexity Space Race

Exercise

Simplify the following function using Big-Oh notation and order them according to growth rate (smallest first):

1. $n \log n + 5 \log n + 3n + 5$
2. $n^2 + n \log n - 2n$
3. 34
4. $\log n + 5$
5. $n^3 + 2^n + 100$

- **Simplify terms**
  - drop lower-order terms
  - drop constant factors

Solution

- **Primitives**
  - assignment
  - calling a method
  - arithmetic operation
  - comparison
  - index into an array
  - follow object reference
  - returning from a method

Simplify the following function using Big-Oh notation and order them according to growth rate (smallest first):

- 34 **-> $O(1)$**
- $\log n + 5$ **-> $O(\log n)$**
- $n \log n + 5 \log n + 3n + 5$ **-> $O(n \log n)$**
- $n^2 + n \log n - 2n$ **-> $O(n^2)$**
- $n^3 + 2^n + 100$ **-> $O(2^n)$**

Exercises

- **Primitives**
  - assignment
  - calling a method
  - arithmetic operation
  - comparison
  - index into an array
  - follow object reference
  - returning from a method

Compute big-O for sumBetween and find_max by

- **first finding primitive operations**
- **then simplify terms**

```
int sumBetween(int[] arr, int min,int max){
  int tmp;
  int result = 0;
  for(int i = 0; i < arr.length;i++){
    tmp = arr[i];
    if (tmp <= max && tmp >= max)
        result += tmp;
  }
  return result;
}
```

```
public int find_max(int x,int y,int z){
    int res = x;
    if (y > res) res = y;
    if (z > res) res = z;
    return res;
}
```

- **Simplify terms**
  - drop lower-order terms
  - drop constant factors

More Examples

- Determine if a number is odd or even
  - isOdd(x) = (mod (x , 2) == 1)

- Reading a book

- How to sort a deck of playing cards?

Real World Examples

- Determine if a number is odd or even
  - **O(1):** isOdd(x) = mod (x , 2) == 1
  - Doesn't matter how big the number *x* is
- Finding a word in the dictionary
  - **O(log N)**: use binary search
  - Size of dictionary has impact on runtime
- Reading a book
  - **O(N)**: read from start to end
- Checking if you have everything on your shopping list in trolley
  - **O(N^2)**: for each shopping list item, check every time in the trolley
- Person introductions
  - **O(N)**: one to all
  - **O(N^2)**: all to all

Summary

- In this lecture we have
  - motivated the importance of analysing algorithms
  - introduced big-O notation
- Attendance sheet
- Next lecture: **stacks**
- **Lab 2:** stacks. Try to complete this week.
- **Lab 1 deadline**
  - Group 1: Friday 18th January
  - Group 2: Monday 21st January