C I T

5 9 4 0

ALGORITHM ANALYSIS

Definitions

Problem: a task to be performed

Algorithm: a method or a process followed to solve a problem

Program: an instance, or concrete representation, of an algorithm in some programming language

Check-in: Come up with an **algorithm** for the **problem** of finding the biggest element in the list.

Different Algorithms, Same Problem

The same problem can be solved with multiple different algorithms.

We decide which algorithm to use based on its **complexity**, or the amount of resources that it requires in order to execute. These resources can be:

- *time* (how many CPU cycles required)
- *space* (the number & size of records that need to be saved in program memory)

Motivating Example: Linear Search

If `inputs` has a length of N,

- What situation leads to the smallest possible number of iterations before the function returns? (What is that number?)
- What situation leads to the largest possible number of iterations before the function returns? (What is that number?)

```java
public static boolean contains(int[] inputs, int target) {
  for (int i = 0; i < inputs.length; i++) {
    if (inputs[i] == target) {
      return true;
    }
  }
  return false;
}
```

Motivating Example: Linear Search

If inputs has a length of N,

- The `target` might be the first element of `inputs`, meaning that we stop when $i = 0$.
- The `target` might not be in `inputs` at all, meaning that we stop when $i = N$.

```
public static boolean contains(int[] inputs, int target) {
  for (int i = 0; i < inputs.length; i++) {
    if (inputs[i] == target) {
      return true;
    }
  }
  return false;
}
```

Sunny Days and Doomsdays

For an algorithm, *on a given size of input* (e.g. for an array of a given length), we can define its:

- **Best case** as the scenario where the algorithm does the minimum possible number of operations
- **Worst case** as the scenario where the algorithm does the maximum possible number of operations.

What were the best and worst cases for Linear Search?

Motivating Example: Binary Search

```java
public static int binarySearch(String[] inputs, String target) {
        int left = 0;
        int right = inputs.length - 1;
        while (right >= left) {
                int middle = (left + right) / 2;
                String middleElem = inputs[middle];
                if (middleElem.compareTo(target) > 0) {
                        left = middle + 1;
                } else if (middleElem.compareTo(target) < 0) {
                        right = middle - 1;
                } else {
                        return middle;
                }
        }
        return -1;
}
```

Motivating Example: Binary Search

If `inputs` is sorted and has a length of N,

- What situation leads to the smallest possible number of iterations before the function returns?
- What situation leads to the largest?

Motivating Example: Binary Search

If `inputs` is sorted and has a length of *N*,

- Middle element might be the target, so 1 iteration is the best case.
- Element might not be present at all, causing us to throw out half of the elements each time until none remain.
    - If we start with 8 elements, we would throw out 4, then 2, then 1, then 1 again, for a total of 4 iterations.

Which is "faster?"

How many iterations will it take to determine that the target is not in the array?

Length of the array	Linear Search	Binary Search
2	2	2
4	4	3
8	8	4
16	16	5
100	100	7

As the size of the array grows, the number of iterations required grows at different rates for the two algorithms.

Growth: Run-Time Complexity

Important: the implementation of an algorithm or data structure can require a computer to spend more or fewer CPU cycles (and therefore more *time* and *energy*) in order to solve a problem.

We want to write programs to be as fast as possible—🕐 is 💰

We'll need a way of analyzing and categorizing the run-time complexity of different algorithms in order for us to understand how efficient we're being.

Definitions: Growth Rate & Upper/Lower Bounds

Growth rate of an algorithm is a function, $T(N)$, that represents the number of constant time operations performed by the algorithm on an input of size N.

An algorithm with runtime complexity $T(N)$ has a lower bound and an upper bound.

- **Lower bound**: A function $f(N) \leq T(N)$ for all positive values of N past a certain point.*
- **Upper bound**: A function $f(N) \geq T(N)$ for all positive values of N past a certain point.*

**more formal specification coming soon*

Size of the input

If we say that the N in $T(N)$ corresponds to the size of the input, what does that mean?

- The size of the input can be quantified with one (or a few) numbers.
  - Algorithm for parsing Strings $\rightarrow$ input size is the # of chars in the String
  - Sorting or searching Lists $\rightarrow$ # of elements in the List
  - Binary exponentiation $\rightarrow$ length of the integer in bits

Constant Time Operations

Basic operations (variable assignment, arithmetic, conditional checking) each take a small, constant amount of time.

- You can assume that all basic operations are equally as fast. **BUT,** they might need to be done many times!

Anything that is not a basic operation incurs a cost that is proportional in some way to the size of the input that it's being executed on.

Definitions: Constant Time Operations

Operation	Example
Addition, subtraction, multiplication, and division of fixed size values.	`w = 10.4`, `x = 3.4`, `y = 2.0`, `z = (w - x) / y`
Assignment of a reference, pointer, or other fixed size data value.	`x = 1000`, `y = x`, `a = true`, `b = a`
Comparison of two fixed size data values.	`a = 100`, `b = 200`, `if (b > a) {...}`
Read or write an array element at a particular index.	`x = `, `arr[index]`, `arr[index + 1] = x + 1`

A ***constant time operation*** is an operation that, for a given processor, always operates in the same amount of time, regardless of input values.

Constant Time or Not?

```java
int[] a = {3, 4, 5, 6, 7, 8};
a[a.length - 1] = a[0] + a[a.length - 2];
```

Constant Time or Not?

```
int[] a = {3, 4, 5, 6, 7, 8};
a[a.length - 1] = a[0] + a[a.length - 2];
```

Yes, array getting/setting and addition are all constant time.

Constant Time or Not?

```java
List<String> listOne = ...;
List<String> listTwo = ...;
setUpLists(listOne, listTwo);
if (listOne.equals(listTwo)) { // is this line "constant time"?
        doSomeStuff();
}
```

Constant Time or Not?

```
List<String> listOne = ...;
List<String> listTwo = ...;
setUpLists(listOne, listTwo);
if (listOne.equals(listTwo)) { // is this line "constant time"?
        doSomeStuff();
}
```

No! List equality requires us to compare all elements, of which there are possibly very many.

Bounds vs. Cases

"Best Case" and "Worst Case" refer to variations of an algorithm’s performance based on specific input classes to the problem that the algorithm is designed to solve.

- “fix the algorithm & input length, find the inputs that will make it run the fastest/slowest”

"Upper Bound" and "Lower Bound" refer to measures of an algorithm’s performance as we vary the size of the input.

- “for a fixed algorithm, as the inputs grow, how does the cost of the algorithm grow?”

Growth Rates Examples

What will be the growth rate of the number of constant-time operations performed in the best case? Worst case?

```java
public static int linearSearch(int[] x, int target) {
  for(int i=0; i < x.length; i++) {
    if (x[i] == target)
      return i;
  }
  return -1; // target not found
}
```

Growth Rates Examples

```
public static int linearSearch(int[] x, int target) {
  for(int i=0; i < x.length; i++) {
    if (x[i] == target)
      return i;
  }
  return -1; // target not found
}
```

Linear growth rate in the worst case, $T(n) = c_1 n + c_0$, constant growth rate in the best case, $T(n) = c$

Growth Rates Examples

```java
public static boolean checkDuplicates(int[] arr) {
  for (int i = 0; i < arr.length; i++) {
    for (int j = i + 1; j < arr.length; j++) {
      if (arr[i] == arr[j]) {
        return true;
      }
    }
  }
  return false;
}
```

Growth Rates Examples

```
public static boolean checkDuplicates(int[] arr) {
  for (int i = 0; i < arr.length; i++) {
    for (int j = i + 1; j < arr.length; j++) {
      if (arr[i] == arr[j]) {
        return true;
      }
    }
  }
  return false;
}
```

$T(n) = c(n-1) + c(n-2) + \ldots c \times 1 = c \times \frac{n^2-n}{2}$, quadratic growth rate in the worst case.

Growth Rates Examples

```java
public static <T extends Comparable<T>> int binarySearch(List<T> inputs, T target) {
        int left = 0;
        int right = inputs.size() - 1;
        while (right >= left) {
                int middle = (left + right) / 2;
                T middleElem = inputs.get(middle);
                if (middleElem.compareTo(target) > 0) {
                        left = middle + 1;
                } else if (middleElem.compareTo(target) < 0) {
                        right = middle - 1;
                } else {
                        return middle;
                }
        }
        return -1;
}
```

Growth Rates Examples

In binary search, we throw away half of the remaining inputs with each iteration of the while loop. We are guaranteed to terminate by the time we have thrown out all of the elements.

$$T(N) = c + T(N/2)$$
$$T(N) = c + (c + T(N/4))$$
$$T(N) = c + (c + (c + T(N/8)))$$

...

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

How many times are we going to spend c? $log_2(N)$ times. So: $T(N) = c \times log_2(N)$.

Upper bound: Big-Oh

For $T(n)$, a non-negatively valued function, $T(n) \in O(f(n))$ if there exist two positive constants c and n_0 such that $T(n) \leq c \times f(n)$ for all $n > n_0$.

"Past a certain point, the runtime of the algorithm will always be less than a certain factor of another function."

Big-Oh Exercises

Show that if $T(n) = 3n^2, T(n) \in O(n^2)$.

Big-Oh Exercises

Show that if $T(n) = 3n^2, T(n) \in O(n^2)$.

Need to choose c, n_0 such that $3n2 \leq cn^2 \forall n > n_0$.

Big-Oh Exercises

Show that if $T(n) = 3n^2, T(n) \in O(n^2)$.

Need to choose c, n_0 such that $3n2 \leq cn^2 \forall n > n_0$.

- Try... $c = 3, n_0 = 1$
- Is $3n^2 \leq 3n^2$ for all $n > 1$? Yes!

$\Rightarrow T(n) \in O(n^2)$

Big-Oh Exercises

Show that if $T(n) = 3n^2 + 4n, T(n) \in O(n^2)$.

Need to choose c, n_0 such that $3n2 \leq cn^2 \; \forall \; n > n_0$.

Big-Oh Exercises

Show that if $T(n) = 3n^2 + 4n, T(n) \in O(n^2)$.

Need to choose c, n_0 such that $3n2 \leq cn^2 \; \forall \; n > n_0$.

Try... $c = 8, n_0 = 1$. Is $3n^2 + 4n \leq 8n^2 \; \forall \; n > 1$?

$$\begin{aligned} 8n^2 &= 4n^2 + 4n^2 \\ &\geq 3n^2 + 4n^2 \\ &\geq 3n^2 + 4n \; \square \end{aligned}$$

Big-Oh: Simplifying Rules

If $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$ then $f(n)$ is in $O(h(n))$

```
public void method1(int n){
    int i=0;
    while (i < n){
        //do something
        i = i + 1;
    }
}
```

`method1` is in $O(n)$ and $O(n^2)$ since $O(n) \in O(n^2)$.

Big-Oh: Simplifying Rules

If $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$

```
public void method1(int n){
    int i=0;
    while (i < n){
        //do something
        i = i + 1;
    }
}
```

`method1` is in $O(n)$ and, say, $O(\frac{1}{2}n)$, but we'll **always** drop the constant.

Big-Oh: Simplifying Rules

If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then
$f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$

```
public void method2(int n) {
  for (int i = 0; i < n; i++) {
    doSomethingConstantTime();
  }
  for (int i = 0; i < n * n; i++) {
    doSomethingConstantTime();
  }
}
```

`method2` is in $O(n^2)$ since $O(n) + O(n^2) \in O(n^2)$

Big-Oh: Simplifying Rules

If $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$, then $f_1(n)f_2(n)$ is in $O(g_1(n)g_2(n))$

```
public void method3(int n) {
  for (int i = 0; i < n; i++) {
    method2(n);
  }
}
```

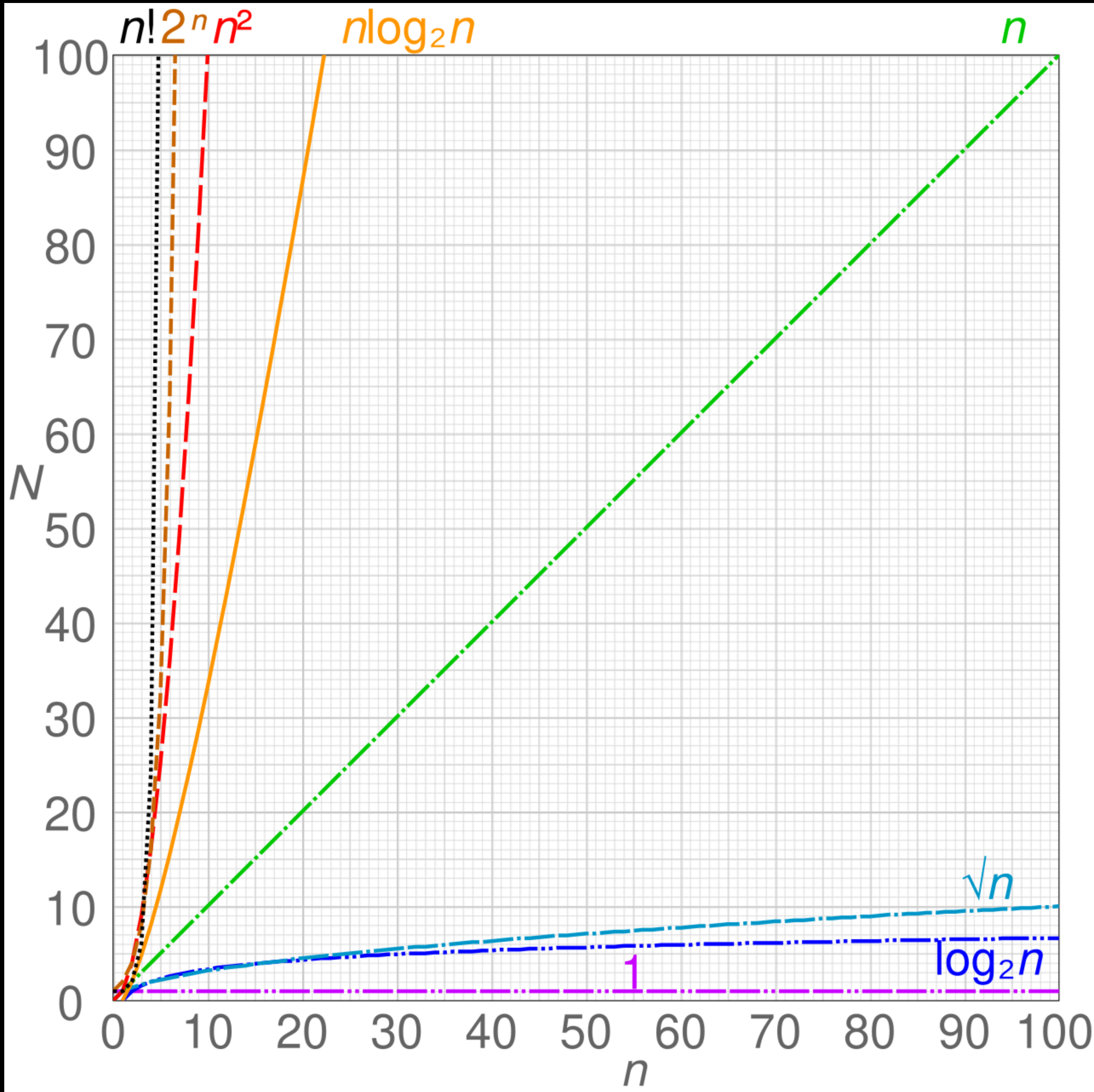
`method3` is in $O(n^3)$ since $O(n) \times O(n^2) \in O(n^3)$

Big-Oh Table

Families above are contained in the families below (e.g. if you show that $f(n) \in O(n^3)$, it follows that $f(n) \in O(2^n)$ but $f(n) \notin O(n^2)$.

Expression	Name
$O(1)$	constant
$O(log(n))$	logarithmic
$O(n)$	linear
$O(nlog(n))$	linearithmic
$O(n^2)$	quadratic
$O(n^3)$	cubic
$O(2^n)$	exponential

Growth, Visualized



Class Activity

Expression	Dominant term(s)	Big-Oh
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n\log_{10}(n)$	$100n^{1.5}$	$O(n^{1.5})$ or $O(n^2)$
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$	$2.5 \cdot n^{1.75}$	$O(n^{1.75})$ or $O(n^2)$
$n^2 \log_2 n + n(\log_2 n)^2$		
$n \log_3 n + n \log_2 n$		
$100n + 0.01n^2$		
$0.01n + 100n^2$		
$0.01n \log_2 n + n(\log_2 n)^2$		

Class Activity

Expression	Dominant term(s)	Big-Oh
$5 + 0.001n^3 + 0.025n$	$0.001n^3$	$O(n^3)$
$500n + 100n^{1.5} + 50n\log_{10}(n)$	$100n^{1.5}$	$O(n^{1.5})$ or $O(n^2)$
$0.3n + 5n^{1.5} + 2.5 \cdot n^{1.75}$	$2.5 \cdot n^{1.75}$	$O(n^{1.75})$ or $O(n^2)$
$n^2 \log_2 n + n(\log_2 n)^2$	$n^2 \log_2 n$	$O(n^2 \log_2 n)$
$n \log_3 n + n \log_2 n$	either	$O(n \log_2 n)$
$100n + 0.01n^2$	$0.01n^2$	$O(n^2)$
$0.01n + 100n^2$	$100n^2$	$O(n^2)$
$0.01n \log_2 n + n(\log_2 n)^2$	$n(\log_2 n)^2$	$O(n(\log_2 n)^2)$

Growth: Space

Time isn't the only resource! Space counts, too.

Recall that data structures store **records,** which are the individual units of information.

- The size of a record changes based on an implementation: a single `int` is 32 bits, but a 2D coordinate would require 64 bits.
- The number of records stored depends on the context of the problem, too.

Example: Who's the Tallest Person to Pass By?

Imagine you have a camera pointed at a street and you want to know the height of the tallest person who passes the camera.

Example Solution #1

- Initialize a list, empty to start.
- For each person that passes by, append their height to the list.
- At the end of the day, sort the list.
- Return the height at the end of the list.

Example Solution #1

- Initialize a list, empty to start.
- For each person that passes by, append their height to the list.
- At the end of the day, sort the list.
- Return the height at the end of the list.

If N people pass by, you have to write down N numbers.

Example Solution #2

- Initialize a variable, `max`, initialized to `-1`.
- For each person that passes by, compare that person's height to `max`.
    - If that person's height is larger than `max`, update `max`
    - Else, do nothing.
- At the end of the day, return the value of `max`

Example Solution #1

- Initialize a variable, `max`, initialized to `-1`.
- For each person that passes by, compare that person's height to `max`.
    - If that person's height is larger than `max`, update `max`
    - Else, do nothing.
- At the end of the day, return the value of `max`

Even if N people pass by, you only have to write down 1 number.

Collections Runtime Cheat Sheet

	LinkedList	ArrayList	TreeSet/Map	HashSet/Map
add	**O(1)** to the head/tail, **O(n)** to the middle	**O(1)** to the end, **O(n)** elsewhere	**O(log n)**	**O(1)** 😉
get(int i)	**O(i)**	**O(1)**	**n/a**	**n/a**
remove	**O(1)** from the head/tail, **O(n)** from the middle	**O(1)** to the end, **O(n)** elsewhere	**O(log n)**	**O(1)** 😉
contains	**O(n)**	**O(n)**	**O(log n)**	**O(1)** 😉
size/clear	**O(1)**	**O(1)**	**O(1)**	**O(1)**