CSCI 241

Lecture 6

Radix Sort

A bit of runtime, if time allows

Announcements

- Quiz 1 graded and available on Gradescope
- A1 is due Monday night
- If gradle test hangs, you probably have an infinite loop.
- You can run a single test with e.g.:
  - `gradle test --tests SortsTest.test00Insertion`

Goals:

- Be prepared to implement radix sort.
- Understand how to go from operation count to a Big-O runtime class.
- Be able to count primitive operations in the non-recursive sorting algorithms we’ve covered so far.

Comparison sorts operate by comparing pairs of elements.

How do you sort without comparing elements?

Suppose I gave you 10 sticky notes with the digits 0 through 9.

What algorithm would you use to sort them?

How many times did you need to look at each sticky note?

What if there are duplicates?

Refresher:
Stacks and Queues

(LIFO) (FIFO)

```
Stack s;
Queue q;

for i in 1..5:
  s.add(i) // push
  q.add(i) // enqueue
for i in 1..5:
  print s.remove() // pop
  print q.remove() // dequeue
```

Q1: **What is printed?**

Stability

Objects can be sorted on **keys** - **different** objects may have the same value.

- e.g., sorting on 10's place only

A **stable** sort maintains the order of distinct elements with the same key.

Exercise - Q2:

- Sort the following array **stably** on the 1’s digit:

[7, 19, 61, 11, 14, 54, 1, 8]

Exercise - Q2:

- Sort the following array **stably** on the 1's digit:

[**7**, 1**9**, 6**1**, 1**1**, 1**4**, 5**4**, **1**, **8**]

LSD Radix Sort

```
/** least significant digit radix sort A */
LSDRadixSort(A):
max_digits = max # digits in any element of A
for d in 0..max_digits:
  do a stable sort of A on the dth least
  significant digit

// A is now sorted(!)
```

LSD Radix Sort

```
/** least significant digit radix sort A */
LSDRadixSort(A):
max_digits = max # digits in any element of A
for d in 0..max_digits:
  do a stable sort of A on the dth least
  significant digit

// A is now sorted(!)
```

Don’t believe me? https://visualgo.net/en/sorting

LSD Radix Sort using queue buckets

```
Pseudocode from visualgo.net:

LSDRadixSort(A):
 create 10 buckets (queues) for each digit (0 to 9)
 for each digit (least- to most-significant):
   for each element in A:
     move element into its bucket based on digit
   for each bucket, starting from smallest digit
     while bucket is non-empty
       restore element to list
```

LSD Intuition: sort on most-significant digit **last**; if tied, yield to the next most significant digit, and so on.
Only works because **stability** preserves orderings from less significant (previously sorted) digits.

Exercise: Radix sort this

[7, 19, 21, 11, 14, 54, 1, 8]

Hint: [07, 19, 21, 11, 14, 54, 01, 08]

```
LSDRadixSort(A):
 create 10 buckets (queues) for each digit (0 to 9)
 for each digit (least- to most-significant):
   for each element in A:
     move element into its bucket based on digit
   for each bucket, starting from smallest digit
     while bucket is non-empty
       restore element to list
```

Exercise - Q3: Radix sort this

[07, 19, 61, 11, 14, 54, 01, 08]

Buckets on 1's place:

0	1	2	3	4	5	6	7	8	9

Sorted on 1's place:

Buckets on 10's place:

0	1	2	3	4	5	6	7	8	9

Sorted on 10's place:

Exercise: Radix sort this

[07, 19, 61, 11, 14, 54, 01, 08]

Buckets on 1's place:

0	1	2	3	4	5	6	7	8	9
	01								
	11			54					
	61			14			07	08	19

Sorted on 1's place: 61 11 01 14 54 07 08 19

Buckets on 10's place:

0	1	2	3	4	5	6	7	8	9
08	19								
07	14								
01	11				54	61			

Sorted on 10's place: 01 07 08 11 14 19 54 61

LSD Radix Sort using counting sort

```
/** least significant digit radix sort A */
LSDRadixSort(A):
max_digits = max # digits in any element of A
for d in 0..max_digits:
  counting sort A on the dth least
  significant digit

// A is now sorted(!)
```

Counting Sort

Formalizes what you did with the 1-9 sticky notes:

- Handles duplicates
- Stable sort

Intuition:

http://www.cs.miami.edu/home/burt/learning/Csc517.091/workbook/countingsort.html

Pseudocode in CLRS (and reproduced on the next slide).

Counting Sort - from CLRS

Notes:

- k is the base or radix (10 in our examples)
- B is filled with the sorted values from A.
- C maintains counts for each bucket.
- The final loop **must** go back-to-front to guarantee stability.

```
COUNTING-SORT(A, B, k)
let C[0..k] be a new array
for i = 0 to k
    C[i] = 0
for j = 1 to A.length
    C[A[j]] = C[A[j]] + 1
// C[i] now contains the number of elements equal to i.
for i = 1 to k
    C[i] = C[i] + C[i - 1]
// C[i] now contains the number of elements less than or equal to i.
for j = A.length downto 1
    B[C[A[j]]] = A[j]
    C[A[j]] = C[A[j]] - 1
```

Runtime Analysis: Overview

- Why? We want a measure of performance that is

  - **Independent** of what computer we run it on.

    Solution: count **operations** instead of clock time.

  - Dependence on **problem size** is made explicit.

    Solution: express runtime as a function of **n**
    (or whatever variables define problem size)

  - **Simpler** than a raw count of operations and focuses on performance on **large problem sizes**.

    Solution: ignore constants, analyze **asymptotic** runtime.

Runtime Analysis: Overview

- How?

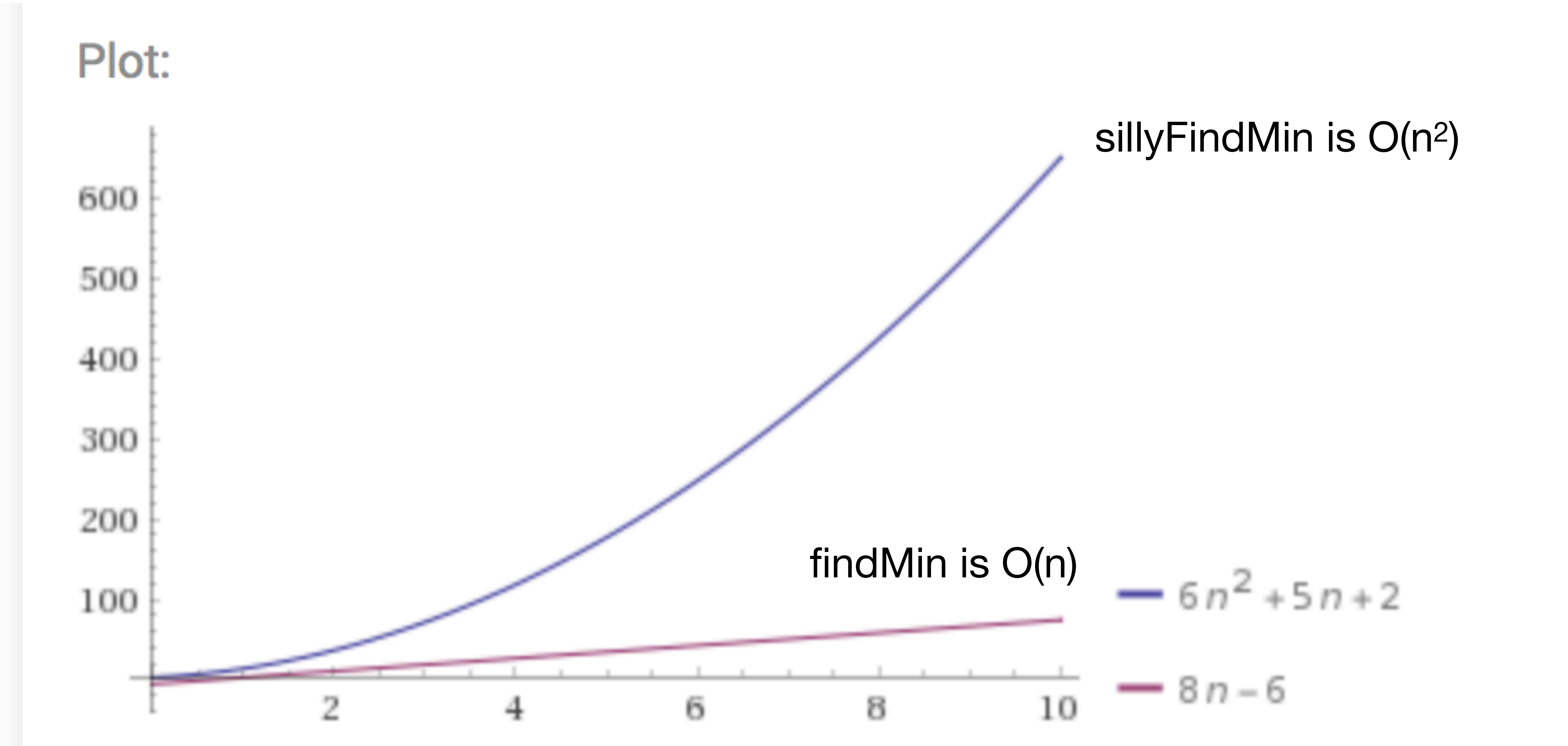
1. Count the number of primitive (constant-time) operations that occur over the entire execution of the algorithm.

2. Drop constants and lower-order terms to find the **asymptotic runtime class**.

Comparing findMaxes

- findMax: 8N - 5
- sillyFindMax: 2 + 5N + 6N²

Drop constants and lower-order terms to find the **asymptotic runtime class**.



From operation count to Big-O runtime class

- findMax: $8N - 5$ findMax is $O(n)$

- sillyFindMax: $2 + 5N + 6N^2$ sillyFindMax is $O(n^2)$

How do we get from 8N-5 to O(N)?

1. Drop all constant factors.

   $8N - 5$ becomes $N - 1$

From operation count to Big-O runtime class

- findMax: 8N - 5 — findMax is O(n)
- sillyFindMax: $2 + 5N + 6N^2$ — sillyFindMax is $O(n^2)$

How do we get from $2 + 5N + 6N^2$ to $O(N^2)$?

1. Drop all constant factors.

   8N - 5 becomes N - 1

2. Drop all but the most-significant term

   N - 1 becomes O(N)

From operation count to Big-O runtime class

- findMax: $8N - 5$

findMax is $O(n)$

- sillyFindMax: $2 + 5N + 6N^2$

sillyFindMax is $O(n^2)$

How do we get from $2 + 5N + 6N^2$ to $O(N^2)$?

1. Drop all constant factors.

$2 + 5N + 6N^2$ becomes $N + N^2$

2. Drop all but the most-significant term

$N + N^2$ becomes $O(N^2)$

From operation count to Big-O runtime class

- findMax: 8N - 5

findMax is O(n)

- sillyFindMax: 2 + 5N + 6N²

sillyFindMax is O(n²)

What if the count is 8?

1. Drop all constant factor factors.

   8 becomes 1

2. Drop all but the most-significant term

   1 becomes O(1) = “constant time”

From operation count to Big-O runtime class

- findMax: 8N - 5

findMax is O(n)

- sillyFindMax: 2 + 5N + $6N^2$

sillyFindMax is $O(n^2)$

What if the count is 800?

1. Drop all constant factor factors.

   800 becomes 1

2. Drop all but the most-significant term

   1 becomes O(1) = “constant time”

Counting Operations

What's a constant-time operation?

- Anything that doesn't depend on the input size:
  - Reading/writing from/to a variable or array location.
  - Evaluating an arithmetic or boolean expression.
  - Returning from a method.

Counting Operations

What's a constant-time operation?

- Anything that doesn't depend on the input size:
    - Reading/writing from/to a variable or array location.
      ```
      int i = 2; int b = 4; a[i] = b;
      ```
    - Evaluating an arithmetic or boolean expression.
      ```
      int i = 0; int j = i+4; int k = i*j;
      ```
    - Returning from a method. `return k;`

Key intuition:

- These don't take identical amounts of time, but the times are within a **constant factor** of each other.
- Same for running the **same** operation on a **different** computer.