Lesson 46: Iteration Revisited #3 (W13D4)

Balboa High School

Michael Ferraro

November 13, 2015

Do Now

Come up with an algorithm for finding *all* integer factors of a given number. For example, the number 16 has these factors: $\{1, 2, 4, 8, 16\}$.

- **Specific/Concrete Case:** If you are given the number 28, describe a process (set of steps) that you can follow that will guarantee that you find every factor of 28.
- **General/Abstract Case:** Once you figure out the steps that need to be carried out for 28, describe the process for finding all factors of any given integer.

Acceptable ways of showing process: Flowchart or Pseudocode

Aim

Students will develop algorithms for factoring integers and determining whether integers are prime.

Factoring: Your Ideas

Let's discuss your ideas for finding all factors for a given integer, `n`...

Factoring: A Naïve Approach

- In the design of an algorithm, we typically start with a simple — or naïve[1] — approach, one that is simple and we know works, but isn't necessarily efficient.

[1]means there's a lack of sophistication or experience

Factoring: A Naïve Approach

- In the design of an algorithm, we typically start with a simple — or naïve[1] — approach, one that is simple and we know works, but isn't necessarily efficient.
- Once we have something that works, we can *tweak* the algorithm.

[1]means there's a lack of sophistication or experience

Factoring: A Naïve Approach

- In the design of an algorithm, we typically start with a simple — or naïve[1] — approach, one that is simple and we know works, but isn't necessarily efficient.
- Once we have something that works, we can *tweak* the algorithm.
  - make it work using fewer steps (faster)

[1]means there's a lack of sophistication or experience

Factoring: A Naïve Approach

- In the design of an algorithm, we typically start with a simple — or naïve[1] — approach, one that is simple and we know works, but isn't necessarily efficient.
- Once we have something that works, we can *tweak* the algorithm.
  - make it work using fewer steps (faster)
  - try some different method altogether

[1] means there's a lack of sophistication or experience

Factoring: A Naïve Approach

- In the design of an algorithm, we typically start with a simple — or naïve[1] — approach, one that is simple and we know works, but isn't necessarily efficient.
- Once we have something that works, we can *tweak* the algorithm.
    - make it work using fewer steps (faster)
    - try some different method altogether
    - if we fail, can fall back on our naïve solution

[1]means there's a lack of sophistication or experience

Factoring: A Naïve Approach

- In the design of an algorithm, we typically start with a simple — or naïve[1] — approach, one that is simple and we know works, but isn't necessarily efficient.
- Once we have something that works, we can *tweak* the algorithm.
    - make it work using fewer steps (faster)
    - try some different method altogether
    - if we fail, can fall back on our naïve solution
- Common way to write software, especially when we have deadlines.

[1]means there's a lack of sophistication or experience

Factoring: A Naïve Approach

We need to find all integer factors for `n`, an integer…

- Start with 1; is that a factor? Yes, always!

Factoring: A Naïve Approach

We need to find all integer factors for `n`, an integer. . .

- Start with 1; is that a factor? Yes, always!
- Increment to 2; is that a factor? Well, does $n\%2 == 0$?

Factoring: A Naïve Approach

We need to find all integer factors for `n`, an integer...

- Start with 1; is that a factor? Yes, always!
- Increment to 2; is that a factor? Well, does $n\%2 == 0$?
- Increment to 3; is that a factor? Well, does $n\%3 == 0$?

Factoring: A Naïve Approach

We need to find all integer factors for `n`, an integer...

- Start with 1; is that a factor? Yes, always!
- Increment to 2; is that a factor? Well, does $n\%2 == 0$?
- Increment to 3; is that a factor? Well, does $n\%3 == 0$?
- Increment to `x`; is that a factor? Well, does $n\%x == 0$?

Factoring: A Naïve Approach

We need to find all integer factors for `n`, an integer…

- Start with 1; is that a factor? Yes, always!
- Increment to 2; is that a factor? Well, does $n\%2 == 0$?
- Increment to 3; is that a factor? Well, does $n\%3 == 0$?
- Increment to `x`; is that a factor? Well, does $n\%x == 0$?
- Increment to `n`; is that a factor? Yes, always!

Factoring: A Naïve Approach

We need to find all integer factors for `n`, an integer. . .

- We need an `int` variable that goes from `1` $\rightarrow$ `n`.
    - with each iteration, test whether the `int` evenly divides `n`
    - if so, it's a factor — print it.

Factoring: A Naïve Approach

- Write a class called `FactorFinder` in new project `Lesson46`.
- Include a method called `printFactors()` that accepts an `int` argument.
- Use the `main()` method of this class to ask `printFactors()` to print the factors of 96.

Efficiency of Naïve Approach

- How many numbers needed to be tested in order to get all factors of 96?

Efficiency of Naïve Approach

- How many numbers needed to be tested in order to get all factors of 96?
    - Answer: 96

Efficiency of Naïve Approach

- How many numbers needed to be tested in order to get all factors of 96?
  - Answer: 96
- Is this algorithm efficient, running in the least steps to find the integer factors of n?

A Better Approach

- How might we develop a more efficient algorithm, one that takes fewer steps? *Ideas?*

A Better Approach

- How might we develop a more efficient algorithm, one that takes fewer steps? *Ideas?*
- Look at how I've had Algebra I students factor 24:

A Better Approach

- How might we develop a more efficient algorithm, one that takes fewer steps? *Ideas?*
- Look at how I've had Algebra I students factor 24:
    - $1 \times 24 \leftarrow$ when you find one factor, find its partner!

A Better Approach

- How might we develop a more efficient algorithm, one that takes fewer steps? *Ideas?*
- Look at how I've had Algebra I students factor 24:
  - $1 \times 24 \leftarrow$ when you find one factor, find its partner!
  - 2×12

A Better Approach

- How might we develop a more efficient algorithm, one that takes fewer steps? *Ideas?*
- Look at how I've had Algebra I students factor 24:
  - $1 \times 24 \leftarrow$ when you find one factor, find its partner!
  - 2×12
  - 3×8

A Better Approach

- How might we develop a more efficient algorithm, one that takes fewer steps? *Ideas?*
- Look at how I've had Algebra I students factor 24:
  - $1 \times 24 \leftarrow$ when you find one factor, find its partner!
  - 2×12
  - 3×8
  - 4×6

A Better Approach

- How might we develop a more efficient algorithm, one that takes fewer steps? *Ideas?*
- Look at how I've had Algebra I students factor 24:
  - $1 \times 24 \leftarrow$ when you find one factor, find its partner!
  - 2×12
  - 3×8
  - 4×6
  - test 5. . . doesn't work.

A Better Approach

- How might we develop a more efficient algorithm, one that takes fewer steps? *Ideas?*
- Look at how I've had Algebra I students factor 24:
  - $1 \times 24 \leftarrow$ when you find one factor, find its partner!
  - 2×12
  - 3×8
  - 4×6
  - test 5 . . . doesn't work.
  - test $6 \leftarrow$ we already saw 6 . . . stop!

A Better Approach

- So whenever we find a factor, find its “partner” factor (e.g., 2 and partner 12 in the case of 24)

A Better Approach

- So whenever we find a factor, find its “partner” factor (e.g., 2 and partner 12 in the case of 24)
- Keep track of the last partner factor we saw

A Better Approach

- So whenever we find a factor, find its “partner” factor (e.g., 2 and partner 12 in the case of 24)
- Keep track of the last partner factor we saw
- If the lower factor we’re trying out is equal to the last partner factor we saw, stop!

A Better Approach

- So whenever we find a factor, find its “partner” factor (e.g., 2 and partner 12 in the case of 24)
- Keep track of the last partner factor we saw
- If the lower factor we’re trying out is equal to the last partner factor we saw, stop!
- In the same class you already have, write a new method to find factors using this more efficient technique. In case it doesn’t work out, we still have our original method!

Prime Numbers

- *Prime Numbers:* Integers whose only factors are 1 and itself

Prime Numbers

- *Prime Numbers:* Integers whose only factors are 1 and itself
- Examples: $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, \ldots\}$

Prime Numbers

- *Prime Numbers:* Integers whose only factors are 1 and itself
- Examples: $\{2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, \ldots\}$
- *Let's discuss how you might write a program to find prime numbers...*

Prime Numbers

Write `PrimeFinder`:

1. Write a method called `isPrime()` that returns a `boolean`, indicating whether the integer sent to it is prime.
2. Have `main()` see if 327 is prime and print out the result.
3. Have `main()` see if 83 is prime and print out the result.
4. Once you think the program is working correctly, put a `for()` loop in `main()` that sends 2 through 501 to `isPrime()` and print out which ones are prime!

Next. . .

Work on PS #7, §5 — problems posted here.

HW

Finish §5 of PS #7.