Big O

David Kauchak
cs140
Spring 2024

1

Administrative

Assignment 0 out and due on Sunday

Mentor hours up soon!

Slack channel

2

Proofs

What is a proof?
- A deductive argument showing a statement is true based on previous knowledge (axioms)

Why are they important/useful?
- Allows us to be sure that something is true
- In algs: allow us to prove properties of algorithms

3

An example

Prove the sum of two odd integers is even

4

An example

Prove the sum of two odd integers is even

Odd number: n = 2k + 1 for some integer k

Even number: n = 2k for some integer k

5

An example

Prove the sum of two odd integers is even

Odd number: n = 2k + 1 for some integer k

Even number: n = 2k for some integer k

Let a and b be odd numbers

By definition: $a = 2i + 1$ and $b = 2j + 1$ where i and j are integers

$$
\begin{aligned}
a + b &= 2i + 1 + 2j + 1 \\
&= 2i + 2j + 2 \\
&= 2(i + j + 1)
\end{aligned}
$$

since i and j are integers then $i + j + 1$ is an integer, so the number is even

6

Proof techniques?

example/counterexample

enumeration

by cases

by inference (aka direct proof)

trivially

contrapositive

contradiction

induction (strong and weak)

7

Proof by induction (weak)

Proving something about a sequence of events by:

1. first: proving that some starting case is true and
2. then: proving that if a given event in the sequence were true then the next event would be true

8

Proof by induction (weak)

1. **Base case:** prove some starting case is true
2. **Inductive case:** Assume some event is true and prove the next event is true
   a. **Inductive hypothesis:** Assume the event is true (usually k or k-1)
   b. **Inductive step to prove:** What you're trying to prove *assuming* the inductive hypothesis is true
   c. **Proof of inductive step**

9

Proof by induction example

Prove: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

1. **Base case:** prove some starting case is true
2. **Inductive case:** Assume some event is true and prove the next event is true
   a. **Inductive hypothesis:** Assume the event is true (usually k or k-1)
   b. **Inductive step to prove:** What you're trying to prove *assuming* the inductive hypothesis is true
   c. **Proof of inductive step**

10

Base case

Prove: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Show it is true for n = 1

$$\sum_{i=1}^{n} i = 1 = \frac{1 * 2}{2}$$

11

Inductive case

Prove: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Inductive hypothesis: assume n = k – 1 is true

$$\sum_{i=1}^{k-1} i = \frac{(k-1) * k}{2}$$

12

Inductive case

Prove: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$

Inductive hypothesis: assume n = k – 1 is true

$$\sum_{i=1}^{k-1} i = \frac{(k-1)k}{2}$$

Prove:

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$$

13

Inductive case: proof

Prove: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ IH: $\sum_{i=1}^{k-1} i = \frac{(k-1)k}{2}$

$$\sum_{i=1}^{k} i =$$

$$= \frac{k(k+1)}{2}$$

14

Inductive case: proof

Prove: $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ IH: $\sum_{i=1}^{k-1} i = \frac{(k-1)k}{2}$

$$\sum_{i=1}^{k} i = k + \sum_{i=1}^{k-1} i$$ by definition of sum

$$= k + \frac{(k-1)*k}{2}$$ by IH

$$= \frac{2k}{2} + \frac{(k-1)*k}{2}$$

$$= \frac{2k + (k-1)*k}{2}$$

$$= \frac{k^2+k}{2}$$

$$= \frac{k(k+1)}{2}$$

Why does induction work as a proof?

15

Layout of a proof by induction

1. State what you're trying to prove
   We show that XXX using proof by induction
2. Prove base case
3. State the inductive hypothesis
4. Inductive proof
   a. State what you want to show (may include a variable change, e.g., k in instead of n)
   b. Show a step-by-step derivation from the left-hand side resulting in the right-hand side. Give justifications for steps that are non-trivial

16

1. We show that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ using proof by induction
2. Base case: n = 1 $\sum_{i=1}^{n} i = 1 = \frac{1*2}{2}$
3. IH, Assume it holds for k-1: $\sum_{i=1}^{k-1} i = \frac{(k-1)k}{2}$
4. Inductive step: want to show $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$

$$\sum_{i=1}^{k} i =$$

$$\ldots$$

$$= \frac{k(k+1)}{2}$$

17

Inductive proofs

Weak vs. strong?

18

Inductive proofs

Weak: inductive hypothesis only assumes it holds for some step (e.g., *k*th step)

Strong: inductive hypothesis assumes it holds for all steps from the base case up to *k*

19

Sorting

Input: An array of numbers A

Output: The number in sorted order, i.e.,

$$A[i] \leq A[j]\ \forall i < j$$

20

Sorting

What sorting algorithm?

```
for j ← 2 to length[A]
    current ← A[j]
    i ← j − 1
    while i > 0 and A[i] > current
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← current
```

21

Sorting

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    current ← A[j]
    i ← j − 1
    while i > 0 and A[i] > current
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← current
```

22

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    current ← A[j]
    i ← j − 1
    while i > 0 and A[i] > current
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← current
```

Does it terminate?

Is it correct?

How long does it take to run?

Memory usage?

23

Insertion-sort

```
INSERTION-SORT(A)
for j ← 2 to length[A]
    current ← A[j]
    i ← j − 1
    while i > 0 and A[i] > current
        A[i + 1] ← A[i]
        i ← i − 1
    A[i + 1] ← current
```

Does it terminate?

24

Insertion-sort

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2        current ← A[j]
3        i ← j − 1
4        while i > 0 and A[i] > current
5              A[i + 1] ← A[i]
6              i ← i − 1
7        A[i + 1] ← current
```

Is it correct? Can you prove it?

25

Loop invariant

Loop invariant: A statement about a loop that is true *before* the loop begins and *after each iteration* of the loop.

Upon termination of the loop, the invariant should help you show something useful about the algorithm.

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2        current ← A[j]
3        i ← j − 1
4        while i > 0 and A[i] > current
5              A[i + 1] ← A[i]
6              i ← i − 1
7        A[i + 1] ← current
```

Loop invariant?

26

Loop invariant

Loop invariant: A statement about a loop that is true *before* the loop begins and *after each iteration* of the loop.

At the start of each iteration of the for loop of lines 1-7 the subarray *A[1..j − 1]* is the sorted version of the original elements of *A[1..j − 1]*

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2        current ← A[j]
3        i ← j − 1
4        while i > 0 and A[i] > current
5              A[i + 1] ← A[i]
6              i ← i − 1
7        A[i + 1] ← current
```

Proof?

27

Loop invariant

At the start of each iteration of the for loop of lines 1-7 the subarray *A[1..j − 1]* is the sorted version of the original elements of *A[1..j − 1]*

Proof by induction
- Base case: invariant is true before loop
- Inductive case: it is true after each iteration

```
INSERTION-SORT(A)
1  for j ← 2 to length[A]
2        current ← A[j]
3        i ← j − 1
4        while i > 0 and A[i] > current
5              A[i + 1] ← A[i]
6              i ← i − 1
7        A[i + 1] ← current
```

28

Insertion-sort

```
INSERTION-SORT(A)
for j ← 2 to length[A]
        current ← A[j]
        i ← j − 1
        while i > 0 and A[i] > current
                A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← current
```

How long will it take to run?

29

Asymptotic notation

How do you answer the question: "what is the running time of algorithm *x*?"

We want to talk about the computational cost of an algorithm that focuses on the essential parts and ignores irrelevant details

You've seen some of this already:

- linear
- *n* log *n*
- n^2

30

Asymptotic notation

Precisely calculating the actual steps is tedious and not generally useful

Different operations take different amounts of time. Even from run to run, things such as caching, etc. cause variations

We want to identify **categories** of algorithmic runtimes

31

For example…

$f_1(n)$ takes n^2 steps

$f_2(n)$ takes $2n + 100$ steps

$f_3(n)$ takes $3n+1$ steps

Which algorithm is better?

Is the difference between f_2 and f_3 important/significant?

32

Runtime examples

	n	$n \log n$	n^2	n^3	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 18 min	10^{25} years
$n = 100$	< 1 sec	< 1 sec	1 sec	1s	10^{17} years	very long
$n = 1000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long
$n = 10,000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long
$n = 100,000$	< 1 sec	2 sec	3 hours	32 years	very long	very long
$n = 1,000,000$	1 sec	20 sec	12 days	31,710 years	very long	very long

(adapted from [2], Table 2.1, pg. 34)

33

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

34

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* above by some constant factor of *g(n)*

35

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* above by some constant multiplied by *g(n)*

For some increasing range

36

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

$$O(n^2) = \begin{array}{lcl} f_1(x) & = & 3n^2 \\ f_2(x) & = & 1/2n^2 + 100 \\ f_3(x) & = & n^2 + 5n + 40 \\ f_4(x) & = & 6n \end{array}$$

37

Big O: Upper bound

O(g(n)) is the set of functions:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Generally, we're most interested in big O notation since it is an upper bound on the running time

38

Omega: Lower bound

Ω(g(n)) is the set of functions:

$$\Omega(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

39

Omega: Lower bound

Ω(g(n)) is the set of functions:

$$\Omega(g(n)) = \left\{ f(n) : \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* below by some constant factor of *g(n)*

40

Omega: Lower bound

Ω(g(n)) is the set of functions:

$$\Omega(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\ 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

$$\Omega(n^2) = \begin{array}{lcl} f_1(x) & = & 3n^2 \\ f_2(x) & = & 1/2n^2 + 100 \\ f_3(x) & = & n^2 + 5n + 40 \\ f_4(x) & = & 6n^3 \end{array}$$

41

Theta: Upper and lower bound

Θ(g(n)) is the set of functions:

$$\Theta(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

42

Theta: Upper and lower bound

Θ(g(n)) is the set of functions:

$$\Theta(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

We can bound the function *f(n)* above **and** below by some constant factor of *g(n)* (though different constants)

43

Theta: Upper and lower bound

Θ(g(n)) is the set of functions:

$$\Theta(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

Note: A function is theta bounded **iff** it is big O bounded and Omega bounded

44

Theta: Upper and lower bound

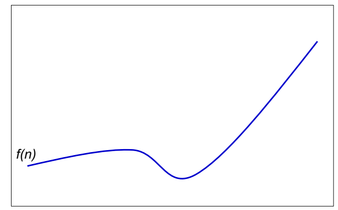
$\Theta(g(n))$ is the set of functions:

$$\Theta(g(n)) = \left\{ f(n): \begin{array}{l} \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that} \\ 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0 \end{array} \right\}$$

$$\Theta(n^2) = \begin{array}{lcl} f_1(x) & = & 3n^2 \\ f_2(x) & = & 1/2n^2 + 100 \\ f_3(x) & = & n^2 + 5n + 40 \\ f_4(x) & = & 3n^2 + n \log n \end{array}$$

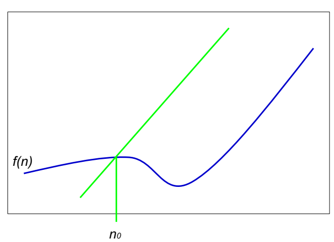
45

Visually



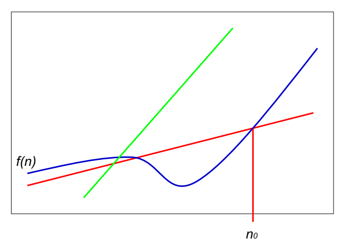
46

Visually: upper bound



47

Visually: lower bound



48

worst-case vs. best-case vs. overall

worst-case: what is the worst the running time of the algorithm can be?

best-case: what is the best the running time of the algorithm can be?

overall: given some data, what is the running time of the algorithm? (Sometimes can think about this as any data or random data)

Don't confuse this with O, Ω and Θ. The cases above are *situations*, asymptotic notation is about bounding particular situations

49

Some rules of thumb

Multiplicative constants can be omitted

- $14n^2$ becomes n^2
- 7 log n become log n

Lower order functions can be omitted

- $n + 5$ becomes n
- $n^2 + n$ becomes n^2

n^a dominates n^b if $a > b$

- n^2 dominates n, so n^2+n becomes n^2
- $n^{1.5}$ dominates $n^{1.4}$

50

Some rules of thumb

a^n dominates b^n if $a > b$

- 3^n dominates 2^n

Any exponential dominates any polynomial

- 3^n dominates n^5
- 2^n dominates n^c

Any polynomial dominates any logorithm

- n dominates log n or log log n
- n^2 dominates n log n
- $n^{1/2}$ dominates *log n*

Do **not** omit lower order terms of different variables ($n^2 + m$) does not become n^2

51

Big O

$n^2 + n \log n + 50$

$2^n - 15n^2 + n^3 \log n$

$n^{\log n} + n^2 + 15n^3$

$n^5 + n! + n^n$

52

Insertion-sort

```
INSERTION-SORT(A)
for j ← 2 to length[A]
        current ← A[j]
        i ← j − 1
        while i > 0 and A[i] > current
                A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← current
```

How long will it take to run?

53

Insertion-sort

```
INSERTION-SORT(A)
for j ← 2 to length[A]
        current ← A[j]
        i ← j − 1
        while i > 0 and A[i] > current
                A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← current
```


How long will it take to run?
Best case? Worst case? Overall?
Use theta when you can, O otherwise.

54

Insertion-sort

```
INSERTION-SORT(A)
for j ← 2 to length[A]
        current ← A[j]
        i ← j − 1
        while i > 0 and A[i] > current
                A[i + 1] ← A[i]
                i ← i − 1
        A[i + 1] ← current
```


Best case (sorted): $\Theta(n)$

Worst case (reverse sorted): $\Theta(n^2)$

Overall: $O(n^2)$

55

Some examples

- O(1) – constant. Fixed amount of work, regardless of the input size
  - add two 32 bit numbers
  - determine if a number is even or odd
  - sum the first 20 elements of an array
  - delete an element from a doubly linked list
- O(log *n*) – logarithmic. At each iteration, discards some portion of the input (i.e. half)
  - binary search

56

Some examples

- O(n) – linear. Do a constant amount of work on each element of the input
  - find an item in a linked list
  - determine the largest element in an array
- O($n \log n$) log-linear. Divide and conquer algorithms with a linear amount of work to recombine
  - Sort a list of number with MergeSort
  - FFT

57

Some examples

- O(n^2) – quadratic. Double nested loops that iterate over the data
  - Insertion sort
- O(2^n) – exponential
  - Enumerate all possible subsets
  - Traveling salesman using dynamic programming
- O(n!)
  - Enumerate all permutations
  - determinant of a matrix with expansion by minors

58