Big O
David Kauchak
cs140
Spring 2024

Administrative
Assignment 0 out and due on Sunday
Mentor hours up soon!
Slack channel

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 alg's: allow us to prove properties of algorithms

An example
Prove the sum of two odd integers is even
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: 
\[
\begin{align*}
a &= 2i + 1 \\
b &= 2j + 1
\end{align*}
\]
where \( i \) and \( j \) are integers
\[
\begin{align*}
a + b &= (2i + 1) + (2j + 1) \\
&= 2i + 2j + 2 \\
&= 2(i + j + 1)
\end{align*}
\]
since \( i \) and \( j \) are integers then \( i + j + 1 \) is an integer, so the number is even

Proof techniques?
example/counterexample
enumeration
by cases
by inference (aka direct proof)
trivially
contrapositive
contradiction
induction (strong and weak)

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
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**

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**

**Base case**

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

Show it is true for \( n = 1 \)

\[ \sum_{i=1}^{1} i = 1 = \frac{1 + 2}{2} \]

**Inductive case**

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

Inductive hypothesis: assume \( n = k - 1 \) is true

\[ \sum_{i=1}^{k-1} i = \frac{(k-1) \times k}{2} \]
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:

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

Inductive case: proof

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

IH: \[ \sum_{i=1}^{k-1} i = \frac{(k-1)k}{2} \]

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

Why does induction work as a proof?

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
   a. Inductive proof
      i. State what you want to show (may include a variable change, e.g., \( k \) in instead of \( n \))
      ii. 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
4. Inductive proof

Why does induction work as a proof?
1. We show that $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ using proof by induction.

2. Base case: $n = 1$ $\sum_{i=1}^{1} i = 1 = \frac{1 \times 2}{2}$

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

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

18

Inductive proofs

Weak vs. strong?

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.,

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

What sorting algorithm?

```
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
```

Does it terminate? Is it correct? How long does it take to run? Memory usage?

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
```

Does it terminate?
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?

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

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
How long will it take to run?

**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
```

**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
- \( n \log n \)
- \( n^2 \)

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.

**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?
Runtime examples

<table>
<thead>
<tr>
<th>n</th>
<th>$n^2$</th>
<th>$2^n$</th>
<th>O($n^2$)</th>
<th>O($2^n$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>100</td>
<td>1024</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>20</td>
<td>400</td>
<td>1,048</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>100</td>
<td>10,000</td>
<td>1,024</td>
<td>&gt; 1 min, very long</td>
<td>&gt; 1 min, very long</td>
</tr>
<tr>
<td>1000</td>
<td>1,000,000</td>
<td>1,024</td>
<td>&gt; 1 min, very long</td>
<td>&gt; 1 min, very long</td>
</tr>
</tbody>
</table>

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

Big O: Upper bound

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

$$O(g(n)) = \{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$$

We can bound the function $f(n)$ above by some constant factor of $g(n)$ for some increasing range

Big O: Upper bound

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

$$O(g(n)) = \{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$$

We can bound the function $f(n)$ above by some constant multiplied by $g(n)$.
**Big O: Upper bound**

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

\[
O(g(n)) = \left\{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 < f(n) \leq cg(n) \text{ for all } n > n_0 \right\}
\]

- \( f_1(x) = 3n^3 \)
- \( f_2(x) = \frac{1}{2}n^2 + 100 \)
- \( f_3(x) = n^2 + 5n + 40 \)
- \( f_4(x) = 6n \)

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

---

**Omega: Lower bound**

\( \Omega(g(n)) \) is the set of functions:

\[
\Omega(g(n)) = \left\{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 < cg(n) \leq f(n) \text{ for all } n > n_0 \right\}
\]

We can bound the function \( f(n) \) below by some constant factor of \( g(n) \).
Omega: Lower bound

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

$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c, n_0 \text{ such that } 0 \leq c \cdot g(n) \leq f(n) \text{ for all } n \geq n_0 \}$

$f_1(x) = 3n^2$
$f_2(x) = 1/2n^2 + 100$
$\Omega(n^2)$
$f_3(x) = n^2 + 5n + 40$
$f_4(x) = 6n^2$

Theta: Upper and lower bound

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

$\Theta(g(n)) = \{ f(n) : \text{there exist positive constants } c_1, c_2, n_0 \text{ such that } 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \text{ for all } n \geq n_0 \}$

Note: A function is theta bounded if it is big O bounded and Omega bounded
Theta: Upper and lower bound

θ(𝑔(𝑛)) is the set of functions:

\[ θ(𝑔(𝑛)) = \{ f(𝑛) \mid \text{there exists positive constants } c, c_1, \text{ and } c_2 \text{ such that } \]

\[ 0 < c_1 \cdot g(n) = f(n) = c_2 \cdot g(n) \text{ for all } n = n_0. \]

\[ f_1(n) = 3n^2 \]
\[ Θ(n^2) = f_2(n) = 1/2n^2 + 100 \]
\[ f_3(n) = n^2 + 5n + 40 \]
\[ f_4(n) = 3n^2 + n \log n \]

Visually: upper bound

Visually: lower bound
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.

Some rules of thumb

**Multiplicative constants can be omitted**
- $54n^2$ becomes $n^2$
- $7 \log n$ becomes $\log n$

**Lower order functions can be omitted**
- $n^3 = 5n^2$ becomes $n^2$
- $n^3 + n^2$ becomes $n^3$

$n^a$ dominates $n^b$ if $a > b$
- $n^a$ dominates $n^b$, so $n^a n^b$ becomes $n^a$
- $n^a$ dominates $n^b$

Any exponential dominates any polynomial
- $2^n$ dominates $n$
- $2^n$ dominates $n^2$

Any polynomial dominates any logarithm
- $n$ dominates $\log n$ or $\log \log n$
- $n^2$ dominates $\log n$
- $n^m$ dominates $\log n$

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

Big O

- $n^2 + n \log n + 50$
- $2^n - 15n^2 + n^3 \log n$
- $n^{\log n} + n^2 + 15n^3$
- $n^5 + n! + n^n$
How long will it take to run?

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

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
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

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