Big O

David Kauchak

cs140

Fall 2022
Administrative

Assignment 1

Peer learning groups

Mentor hours

Slack channel
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 $k$
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
```
Sorting

**insertion-sort**($A$)

1. for $j \leftarrow 2$ to $\text{length}[A]$
2. \hspace{1em} $\text{current} \leftarrow A[j]$
3. \hspace{1em} $i \leftarrow j - 1$
4. \hspace{1em} while $i > 0$ and $A[i] > \text{current}$
5. \hspace{2em} $A[i + 1] \leftarrow A[i]$
6. \hspace{2em} $i \leftarrow i - 1$
7. \hspace{1em} $A[i + 1] \leftarrow \text{current}$
Does it terminate?

Is it correct?

How long does it take to run?

Memory usage?

**INSERTION-SORT(A)**

1. \( \text{for } j \leftarrow 2 \text{ to } \text{length}[A] \)
2. \( \quad \text{current } \leftarrow A[j] \)
3. \( \quad i \leftarrow j - 1 \)
4. \( \quad \text{while } i > 0 \text{ and } A[i] > \text{current} \)
5. \( \quad \quad A[i + 1] \leftarrow A[i] \)
6. \( \quad \quad i \leftarrow i - 1 \)
7. \( \quad A[i + 1] \leftarrow \text{current} \)
Insertion-sort

\textbf{Insertion-Sort}(A)

1 \hspace{1em} \textbf{for} \ j \leftarrow 2 \ \textbf{to} \ \text{length}[A] \\
2 \hspace{1em} \hspace{1em} \textit{current} \leftarrow A[j] \\
3 \hspace{1em} i \leftarrow j - 1 \\
4 \hspace{1em} \hspace{1em} \textbf{while} \ i > 0 \ \text{and} \ A[i] > \textit{current} \\
5 \hspace{1em} \hspace{1em} \hspace{1em} A[i + 1] \leftarrow A[i] \\
6 \hspace{1em} \hspace{1em} \hspace{1em} i \leftarrow i - 1 \\
7 \hspace{1em} \hspace{1em} \hspace{1em} A[i + 1] \leftarrow \textit{current}

Does it terminate?
Insertion-sort

**Insertion-Sort(A)**

1. for $j \leftarrow 2$ to \text{length}[A]
2. \quad \text{current} \leftarrow A[j]
3. \quad i \leftarrow j - 1
4. \quad \text{while } i > 0 \text{ and } A[i] > \text{current}
5. \quad \quad A[i + 1] \leftarrow A[i]
6. \quad \quad i \leftarrow i - 1
7. \quad A[i + 1] \leftarrow \text{current}

Is it correct? Can you prove it?
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.

```plaintext
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: 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
```
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           A[i + 1] ← current
7   i ← i - 1
```
Insertion-sort

**FUNCTION**

\[
\text{INSERTION-SORT}(A)
\]

1. \( \text{for } j \leftarrow 2 \text{ to } \text{length}[A] \)
2. \( \quad \text{current} \leftarrow A[j] \)
3. \( \quad i \leftarrow j - 1 \)
4. \( \quad \text{while } i > 0 \text{ and } A[i] > \text{current} \)
5. \( \quad \quad A[i + 1] \leftarrow A[i] \)
6. \( \quad \quad i \leftarrow i - 1 \)
7. \( \quad A[i + 1] \leftarrow \text{current} \)

**Query**

How long will it take to run?
Asymptotic notation

How do you answer the question: “what is the running time of algorithm \(x\)?”

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
- \(n \log n\)
- \(n^2\)
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
For example...

\[ f_1(n) \text{ takes } n^2 \text{ steps} \]
\[ f_2(n) \text{ takes } 2n + 100 \text{ steps} \]
\[ f_3(n) \text{ takes } 3n+1 \text{ steps} \]

Which algorithm is better?

Is the difference between \( f_2 \) and \( f_3 \) important/significant?
## Runtime examples

<table>
<thead>
<tr>
<th>n</th>
<th>$n \log n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$2^n$</th>
<th>n!</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 18 min</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>1s</td>
<td>$10^{17}$ years</td>
</tr>
<tr>
<td>$n = 1000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12 days</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>32 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>31,710 years</td>
<td>very long</td>
</tr>
</tbody>
</table>

(Adapted from [2], Table 2.1, pg. 34)
Big O: Upper bound

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

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

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

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

We can bound the function $f(n)$ above by some constant factor of $g(n)$
Big O: Upper bound

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

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

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

For some increasing range
Big O: Upper bound

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

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

\[
O(n^2) = f_1(x) = 3n^2
\]
\[
f_2(x) = 1/2n^2 + 100
\]
\[
f_3(x) = n^2 + 5n + 40
\]
\[
f_4(x) = 6n
\]
Big O: Upper bound

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

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

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:

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

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

$$\Omega(g(n)) = \left\{ f(n) : \begin{align*}
\text{there exists positive constants } c \text{ and } n_0 \text{ such that} \\
0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0
\end{align*} \right\}$$

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:

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

$$
\begin{align*}
\Omega(n^2) &= f_2(x) = \frac{1}{2}n^2 + 100 \\
f_1(x) &= 3n^2 \\
f_3(x) &= n^2 + 5n + 40 \\
f_4(x) &= 6n^3
\end{align*}
$$
Theta: Upper and lower bound

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

$$\Theta(g(n)) = \left\{ f(n) : \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \right\}$$
\( \Theta(g(n)) \) is the set of functions:

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

We can bound the function \( f(n) \) above and below by some constant factor of \( g(n) \) (though different constants)
Theta: Upper and lower bound

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

\[
\Theta(g(n)) = \left\{ f(n) : \text{there exists positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \right\}
\]

Note: A function is theta bounded \textbf{iff} it is big O bounded and Omega bounded
Theta: Upper and lower bound

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

\[
\Theta(g(n)) = \left\{ f(n) : \exists c_1, c_2, n_0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0 \right\}
\]

\[
\Theta(n^2) = \begin{cases} 
 f_1(x) & = 3n^2 \\
 f_2(x) & = 1/2n^2 + 100 \\
 f_3(x) & = n^2 + 5n + 40 \\
 f_4(x) & = 3n^2 + n \log n 
\end{cases}
\]
Visually

\[ f(n) \]
Visually: upper bound
Visually: lower bound
worst-case vs. best-case vs. average-case

**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?

**average-case**: given random data, what is the running time of the algorithm?

**Don’t** confuse this with $O$, $\Omega$, and $\Theta$. The cases above are *situations*, asymptotic notation is about bounding particular situations.
Proving bounds: find constants that satisfy inequalities

Show that $5n^2 - 15n + 100$ is $\Theta(n^2)$

Step 1: Prove $O(n^2)$ – Find constants $c$ and $n_0$ such that $5n^2 - 15n + 100 \leq cn^2$ for all $n > n_0$

$$cn^2 \geq 5n^2 - 15n + 100$$

$$c \geq 5 \frac{-15}{n} + \frac{100}{n^2}$$

Let $n_0 = 1$ and $c = 5 + 100 = 105$.
100/n^2 only get smaller as $n$ increases and we ignore $-15/n$ since it only varies between -15 and 0
Proving bounds

Step 2: Prove $\Omega(n^2)$ – Find constants $c$ and $n_0$ such that $5n^2 - 15n + 100 \geq cn^2$ for all $n > n_0$

\[
\begin{align*}
cn^2 & \leq 5n^2 - 15n + 100 \\
c & \leq 5 - \frac{15}{n} + \frac{100}{n^2}
\end{align*}
\]

Let $n_0 = 4$ and $c = 5 - 15/4 = 1.25$ (or anything less than 1.25). $15/n$ is always decreasing and we ignore $100/n^2$ since it is always between 0 and 100.
Bounds

Is $5n^2 \in O(n)$? No

How would we prove it?

$O(g(n)) = \left\{ f(n) : \text{there exists positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \right\}$
Is $5n^2$ $\mathcal{O}(n)$?

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

Assume it’s true.

That means there exists some $c$ and $n_0$ such that

$5n^2 \leq cn$ for $n > n_0$

$5n \leq c$ contradiction!
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 $n^{1.4}$
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 logarithm
  - \( n \) 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 \( n^2 \)
Big O

\( n^2 + n \log n + 50 \)

\( 2^n - 15n^2 + n^3 \log n \)

\( n^{\log n} + n^2 + 15n^3 \)

\( n^5 + n! + n^n \)
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