How fast is an algorithm?

• Important part of designing and analyzing an algorithm - efficiency: how long does it take?

Want to do this abstractly, so don’t worry about underlying architecture

How do we time how long an algorithm takes?

- e.g., analysis of a sort algorithm should be predictive of its performance

Overview

⇒ Chapter 2: Analyzing Algorithms
• Chapter 3: Growth of Functions
⇒ Chapter 2: Analyzing Algorithms
Don't model memory hierarchy: caches, virtual memory, paging
Data should be of fixed size as well
Integers and floats, but of a fixed size

Data in RAM model

Random-Access Machine (RAM)

- Assumes a random-access machine (RAM)
- Any memory location can be accessed in the same amount of time
- No concurrent operations (one instruction executed after another)

What is the instruction set?
- Don't include:
  - Actions that can be done in a constant amount of time
  - Dictionary lookups (associate arrays), not typically found in instructions, does not take constant time
  - Don't include:

What is a Random-Access Machine (RAM)?

Instructions

Exponential instructions

- Addition, multiplication, squaring, taking square roots
- Conditional, subroutine calls and returns
- Dictionary lookups (associate arrays), not typically found in instructions
- Sort: Not typically found in instructions, does not take constant time
Many algorithms work on input data, which can vary in size. For many algorithms, effect of input size can be huge. Size of input usually determines size of loops, or depth of recursion. So determine running time with respect to size of input, \( n \).

Different ways of measuring input size:

- Size of input
- Number of elements in an array
- Number of bits
- Number of nodes and edges
- Number of loops or recursion

For sorting an array, size of array
For multiplying two numbers, number of bits
For a graph, number of nodes and edges

Running Time

Different instructions take different lengths. Even in an if statement, with multiple conditions, there is a maximum amount of time. If there is a subroutine call in the expression, it must be accounted for. Even if there is a maximum amount of time, regardless of what the data is, this difference will be drowned out when there are loops, recursion and different instruction lengths.
Worst-case and Average-case Analysis

- Average case is often roughly as bad as the worst case
- Searching a database and data is not present
- Worst case might occur fairly often
- Gives an upper bound for any input
- Textbook emphasizes worst case running time:
  - Can look at average case or worst-case performance

Running Time can Depend on Data

```plaintext
\text{INSERTION-SORT}(A)
```

\( n \): number of times while loop test in line 5 is executed for value of \( j \)
Overview

Chapter 12

• Chapter 3: Growth of Functions
⇒ Chapter 2: Analyzing Algorithms

Order of Growth

- Quantify the running time as the input size grows

- Worst case running time for insertion sort: \( \Theta(n^2) \)

- Since not affecting the rate of growth, can even ignore constants
- Other terms become noise, first term dominates
- Interested in what happens as \( n \) increases

- Say worst case running time is \( an + b \) where \( a, b \) are constants

- First term dominates

⇒ Chapter 2: Analyzing Algorithms
\[(u) \Theta = (u)f\]

\[\text{For } n \geq n_{0}, \quad c_1 g(n) \leq f(n) \leq c_2 g(n)\]

Then \(f(n) = \Theta(g(n))\)

Example: \(an^2 + bn + c\)

Asymptotic Notation

Running time versus size of data using asymptotic analysis

Focus on what happens to a function as \(n\) gets bigger and bigger

Function can represent anything: worst case running time of algorithm, or how much space it needs

Example: \(an^2 + bn + c\)
Example

\[(u) \Theta = (u) f \quad (u) \Theta = (u) g \quad (u) \Theta = (u) \Theta \]

\[
\begin{align*}
\frac{u}{f} & = \frac{\Theta}{f} = \frac{\theta}{f} \\
\frac{u}{f} & = \frac{\Theta}{f} = \frac{\theta}{f}
\end{align*}
\]

We can prove it is \( \Theta (3n^2) \) or \( \Theta (n^2 + 2n) \).

More formally

\[
\forall u \geq u \text{ for all } (u) f \geq (u) g \geq (u) \Theta \geq 0, \text{ and } \exists \text{ such that } (u) f \leq (u) g.
\]

More formally
More on Big O

- Can say running time (no matter) of algorithm is \( ((u)b)O \)

- But all of the functions can be bounded above by

  - A bit of an abuse of terminology: each different data of input size \( n \)
  - A bit of an abuse of terminology: each different data of input size \( n \)
  - A bit of an abuse of terminology: each different data of input size \( n \)
  - A bit of an abuse of terminology: each different data of input size \( n \)

  - Since Big O is just an upper bound, when we use it to upper bound worst-case, it is upper bounding algorithm for any data

  - Since Big O is just an upper bound, when we use it to upper bound worst-case, it is upper bounding algorithm for any data

- Just needed to be clear that it was worst case time (or average)

\[
O(g(n)) = \{ f(n) : \text{there exists positive constants } c_1, n_0 \text{ s.t. } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}
\]

- Just upper bound, not as strong as \( \Theta \), which is a tight bound:

\[
\{(u)O = \Theta(u)O \}
\]

\[
\{(u)bO \subseteq (u)bO \}
\]

- Can easily assess \( O \) by looking at nesting of loops

\[
\{ f(n) \leq g(n) \}
\]

\[
\{(u)f \leq (u)g \}
\]
Overview

Chapter 2: Analyzing Algorithms

Chapter 3: Growth of Functions

Chapter 12

\[ \Omega(g(n)) = \{ f(n) : \text{there exists positive constants } c, n_0 \text{ s.t. } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \} \]

Theorem: 3.1

For any two functions \( f(n) \) and \( g(n) \), we have

\[ f(n) = \Theta(g(n)) \text{ iff } f(n) = O(g(n)) \text{ and } f(n) = \Omega(g(n)) \]

Same comments about \( O \) apply:

- Worst case running time is \( \Theta \), best case \( \Theta \)
- Running time is \( O \) and \( \Omega \) regardless of data
- Lower bound can be specified regardless of data

Theorem: 3.1

\[(u)\Theta = (u)f \text{ and } ((u)\Theta)O = (u)f \text{ iff } \]

\[ ((u)\Theta)\Theta = (u)f \text{ and } ((u)\Theta)\Theta = (u)f \]

For any two functions \( f(n) \) and \( g(n) \), we have

\[ \{u \geq u \text{ for all } (u)f \geq (u)\Theta \geq 0 \} \]

There exists positive constants \( c \) and \( n_0 \).

Omega
Upper Bound

\[ T(n) = O(n) \]  

(proof by induction)

- When called on a leaf, takes constant time \( T(1) \) = \( c \) for some constant \( c > 0 \)
- How much time will it take when it is not a leaf
- \( T(n) \) = \( \Omega(n) \) for \( n \geq p \) and any \( p \)
- Excluding time spent in the recursive call
- Including time spent on initiating recursive call

Lower Bound

- Since it must visit all nodes of the tree, \( \Omega(n) \)
- Let \( T(u) \) denote time taken by InorderTreeWalk when called on node \( u \)
- If \( u \) is the root of a tree with \( n \) nodes, then InorderTreeWalk(\( x \))

**Theorem 12.1**

- \( (u)\Omega = (u)\Theta = (u)O \)
Theorem 12.2: 
Search runs in \( O(h) \) time on a binary tree of height \( h \)

```python
def Search(self, k):
    if k == self.key:
        return self
    if k < self.key and self.left is not None:
        return self.left.Search(k)
    if k > self.key and self.right is not None:
        return self.right.Search(k)
    return None
```

Continued

Theorem 12.2: 
Search runs in \( O(h) \) time on a binary tree of height \( h \)

Continued
Complexity of Insert

```python
def Insert(self, z):
    y = None
    x = self.root
    while x is not None:
        y = x
        if z.key < x.key:
            x = x.left
        else:
            x = x.right
    z.p = y
    if y is None:
        self.root = z
    elif z.key < y.key:
        y.left = z
    else:
        y.right = z
```

Time $O(h)$