Overview

⇒ Chapter 2 Section 2: Analyzing Algorithms
• Chapter 3: Growth of Functions
• Chapter 12: Binary Search Trees
How fast is an algorithm?

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

• How do we time how long an algorithm takes?
  - Want to do this abstractly, so don’t worry about underlying architecture
  - e.g., analysis of a sort algorithm should be predictive of its performance
    on a cell phone, mac, or pc
Random-Access Machine (RAM)

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

• What is the instruction set?
  - Typical of what are found in real computers
    + Adding, multiplying, storing, loading values
    + Conditionals, subroutine calls and returns
    + Actions that can be done in a constant amount of time
  - Don’t include:
    - sort: Not typically found in instructions, does not take constant time
    - Dictionary lookups (associate arrays), not typically found in instructions
  - Exponentiation?
  - Looping constructs?
Data in RAM model

- Integers and floats, but of a fixed sized
- Data should be of fixed size as well
- Don’t model memory hierarchy: caches, virtual memory, paging
Running Time

• Different instructions take different lengths
  - This difference will be drowned out when there are loops, recursion
  - There is a maximum amount of time, regardless of what the data is
    + Even in an if statement, with multiple conditions, there is a maximum time to execute it
    + If there is a subroutine call in the expression, that must be accounted for separately
  - Just assume its time is ‘1’
Size of Input

- Many algorithms work on input data, which can vary in size
  - Sorting a list
  - Parsing a sentence
  - Training a machine learning algorithm on data

- 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:
  - For sorting an array, size of array
  - For multiplying two numbers, number of bits
  - For a graph, number of nodes and edges
Running Time can Depend on Data

**Insertion-Sort**(A)

1. for \( j = 2 \) to \( A.length \)
2. \( key = A[j] \)
3. // Insert \( A[j] \) into the sorted sequence \( A[1 \ldots j - 1] \).
4. \( i = j - 1 \)
5. while \( i > 0 \) and \( A[i] > key \)
7. \( i = i - 1 \)
8. \( A[i + 1] = key \)

- \( t_j \): number of times while loop test in line 5 is executed for value of \( j \)
  - If input is sorted, \( t_j \) is 1. If input is in reverse order, \( t_j = j \)
  - On average, will need to go halfway back in the list \( t_j = j/2 \)
- Why is the while and for statements given a time one greater?

\[ \begin{array}{ll}
\text{cost} & \text{times} \\
\hline
\text{1.} & c_1 \ n \\
\text{2.} & c_2 \ n - 1 \\
\text{3.} & 0 \ n - 1 \\
\text{4.} & c_4 \ n - 1 \\
\text{5.} & c_5 \ \sum_{j=2}^{n} t_j \\
\text{6.} & c_6 \ \sum_{j=2}^{n} (t_j - 1) \\
\text{7.} & c_7 \ \sum_{j=2}^{n} (t_j - 1) \\
\text{8.} & c_8 \ n - 1 \\
\end{array} \]
Worst-case and Average-case Analysis

• Can look at average case or worst-case performance

• Textbook emphasizes worse case running time:
  - Gives an upper bound for any input
  - Worst case might occur fairly often
    + Searching a database and data is not present
  - Average case is often roughly as bad as the worse case
Order of Growth

• Quantify the running time as the input size grows
  - Say worst case running time is $an^2 + bn + c$; where $a$, $b$, $c$ are constants
  - Interested in what happens as $n$ increases
    + First term dominates!
    + Other two terms become noise
    + Can even ignore constant $a$
      + Since not effecting the rate of growth

• Worst case running time for insertion sort: $\Theta(n^2)$
Overview

• Chapter 2 Section 2: Analyzing Algorithms
⇒ Chapter 3: Growth of Functions
• Chapter 12: Binary Search Trees
• 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: \( an^2 + bn + c \)
Theta Notation

- For $f(n)$
  - Is there a function $g(n)$
  - Constants $c_1$, $c_2$, $n_0$
  - $c_1 g(n) \leq f(n) \leq c_2 g(n)$
    + for $n \geq n_0$
  - Then $f(n) = \Theta(g(n))$

\[ f(n) = \Theta(g(n)) \]
More formally

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

- \( \Theta(g(n)) \) is a set of functions that \( g(n) \) can characterize
  - Should write \( f(n) \in \Theta(g(n)) \)
- \( g(n) \) characterizes them for any \( n \) greater than some \( n_0 \)
  - Not interested in small values of \( n \)
- \( g(n) \) characterizes them within constant bounds
- \( c_1, c_2, n_0 \) can depend on the \( f \)
- We say \( g(n) \) is an asymptotically **tight** bound for \( f(n) \)
Example

• Show $\frac{1}{2}n^2 - 3n = \Theta(n^2)$

• Determine $c_1, c_2, n_o > 0$ s.t. that $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2$ for $n \geq n_o$
  - Dividing by $n^2$ yields: $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2$
  - LH inequality: to make $c_1 > 0$, set $n \geq 7$ and so set $c_1 = \frac{1}{14}$
  - RH inequality: holds for any $n \geq 1$ with $c_2 = \frac{1}{2}$

• We can prove it is $\Theta(3n^2)$ or $\Theta(n^2 + 2n)$
  - Want the simplest form for $\Theta(g(n))$

• Constant time algorithms: $\Theta(n^0)$, which can be written as $\Theta(1)$
Big O

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

- Just **upper** bound, not as strong as \( \Theta \), which is a tight bound:
  - In fact, \( \Theta(g(n)) \subseteq O(g(n)) \)
  - \( 2n^2 = O(n^2) \), but also \( 2n^2 = O(n^3) \)
  - Can easily assess \( O \) by looking at nesting of loops

© P. Heeman, 2017 (a) 15 of 23 (b) CS550 Class 03: Chapter 3: Growth of Functions (c)
More on Big O

• For $\Theta$, needed to be clear that it was worst case time (or average time, or best time), since might have different bounds

• 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
  - A bit of an abuse of terminology: each different data of input size $n$ might have a different function for its running time
  - But all of the functions can be bounded above by $O(g(n))$
  - Can say running time (no modifier) of algorithm is $O(g(n))$
Omega

Ω(g(n)) = \{ f(n) : \text{there exists positive constants } c \text{ and } 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 \( f(n) = \Theta(g(n)) \) \iff \( f(n) = O(g(n)) \) and \( f(n) = \Omega(g(n)) \)

• Same comments about \( O \) apply:
  - Lower bound can be specified regardless of data
  - Running time is \( O(n^2) \) and \( \Omega(n) \)
  - Worst case running time is \( \Theta(n^2) \), best case \( \Theta(n) \)
Overview

• Chapter 2 Section 2: Analyzing Algorithms
• Chapter 3: Growth of Functions
⇒ Chapter 12: Binary Search Trees
def InOrderWalk(self):
    if self.left is not None:
        self.left.InOrderWalk()
    print self.key
    if self.right is not None:
        self.right.InOrderWalk()

Theorem 12.1
If $x$ is the root of a tree with $n$ nodes, then $\text{InorderTreeWalk}(x)$
takes $\Theta(n)$ time.

• Let $T(n)$ denote time taken by $\text{InorderTreeWalk}$ when called on
tree with $n$ nodes

• Lower bound:
  - Since it must visit all nodes of the tree, $T(n) = \Omega(n)$
Upper Bound

• Prove by induction that $T(n) = O(n)$
  (textbook refers to this as substitution method).
  - Need more exact formula of its time than just $O(n)$. Let’s guess its time
• 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
  - including time spent on initiating recursive call
  - excluding time spent in the recursive call
  - Will be a constant amount of time, say $d$ and $d \geq c$
When called on a tree with $n$ nodes
- It will split the tree into two parts:
  + right tree $k$ nodes, $0 \leq k \leq n - 1$ (might be an empty subtree)
  + left tree $n - k - 1$ nodes
- $T(n) \leq T(k) + T(n - k - 1) + d$

Assume $T(n) \leq dn$
- Holds for $T(1)$
- Assume true for $1 \leq j < n$, prove true for $n$

\[
T(n) \leq T(k) + T(n - k - 1) + d \leq dk + d(n - k - 1) + d \leq dn
\]
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

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

- What is the lower bound?
  - $\Omega(1)$
  - So it does not have a $\Theta$
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 $O(h)$