Asymptotic Analysis

Comparing algorithms
- How do we compare different algorithms that do the same thing?
- What are important properties of algorithms that are independent of language and speed of machines?

Best case, worst case, average case
- Some algorithms take a varying amount of time depending on the pattern of the data or "luck"
- Some algorithms are generally very fast but might, in exceptional circumstances, take a very long time.
- Some algorithms have guaranteed average time behavior but might take a long time some of the time.

Asymptotic Analysis
- How does an algorithm perform when the problem is very large?
- What is the increase in running time when the problem size increases?
- How do we make this precise?

Characterization of the problem
- To analyze an algorithm we need to define a number that characterizes the size of the problem
- We need a concept of an "elementary operation" whose count is a meaningful measure of the running time of the program
- Sometimes it is important how much storage space a program needs

Definitions

Big-O Notation
- A function \( g(n) \) is said to be of order \( f(n) \) if for large values of \( n \) the ratio \( \frac{g(n)}{f(n)} \) is less than some constant \( c \), or more precisely
- \( g(n) = O(f(n)) \) if and only if there exist constants \( c \) and \( n_0 \) such that \( g(n) < c f(n) \) for all \( n > n_0 \)

Theta Notation
- \( g(n) = \Theta(f(n)) \) if and only if there exist constants \( c_1, c_2, n_0 \) such that \( c_1 f(n) \leq g(n) \leq c_2 f(n) \) for all \( n \geq n_0 \)

Multiplicative constants
- Different on different computers
- Rate-of-growth more important than constant
- Simple instructions execute in constant time
- Vector inserts are not constant time

Examples
- \( 100n = \Theta(n) \)
- \( 100n = O(n) \)
- \( n \log n = O(n^2) \)
- \( n \log n = \Theta(n^2) \)
- \( .000001 n \log n = O(n) \)

Here are some commonly occurring functions
- Indexing an array
- Any primitive operation
- \( \log n \)
- Binary search
- Length of a decimal/binary number \( n \)
- Time to examine \( n \) objects
- Time to find the maximum of \( n \) numbers
- \( n \log n \)
- Time to efficiently sort \( n \) numbers
- \( n^2 \)
- Time for insertion sort or bubble sort

2\(^n\)
- Number of possible binary numbers of length \( n \)

We can ignore lower additive terms
- \( n^2 + n = O(n) \)

Counting steps in the Eiffel Tower
(see Main pp18-27)
- Technique 1
  - Walk down and keep a tally
  - \( O(n) \)
- Technique 2
  - Walk down but let your friend at the top keep a tally
  - \( O(n^2 + 2n) = O(n^2) \)
- Technique 3
  - Have a friend tell you the answer
  - \( O(\log n) \)

Best case, Worst case, Average case
- We can use our big-O or theta notation to describe the best case, worst case, average case

Examples of big-O notation
- \( n = O(n) \)
- \( 0.0001n = O(n) \)
- \( 100n = O(n) \)
- \( 100 \log n = O(n) \)
- \( 100 \log n + 1000 \log n = O(n) \)
- \( n = O(n^2) \)
- \( 10n^2 + 2367n + 12 = O(n^2) \)
- \( n \log n = O(n^2) \)
- \( n^2 + 0.000000001n^3 \neq O(n^2) \)
- \( n^2 + 0.000000001n^2 = O(n^2) \)
- \(.000001n \log n \neq O(n) \)

Which of the above are still true if we replace \( O \) with \( \Theta \)?
- \( n = \Theta(n) \)
- \( 0.0001n = \Theta(n) \)
- \( 100n = \Theta(n) \)
- \( 100 \log n \neq \Theta(n) \)
- \( 100 \log n + 1000 \log n \neq \Theta(n) \)
- \( n \neq \Theta(n^2) \)
- \( 10n^2 + 2367n + 12 = \Theta(n^2) \)
- \( n \log n \neq \Theta(n^2) \)
- \( n^2 + 0.000000001n^3 \neq \Theta(n^2) \)
- \( n^2 + 0.000000001n^2 = \Theta(n^2) \)
- \(.000001n \log n \neq \Theta(n) \)
Here are some commonly occurring big-O functions

\( O(1) \)
Indexing an array
Any primitive operation
\( O(\log n) \)
Binary search
Length of a decimal/binary number \( n \)
\( O(n) \)
Time to examine \( n \) objects
Time to find the maximum of \( n \) numbers
\( O(n \log n) \)
Time to efficiently sort \( n \) numbers
\( O(n^2) \)
Time for insertion sort or bubble sort
\( O(2^n) \)
Number of possible binary numbers of length \( n \)

We can ignore lower additive terms
\( n^2 + n = O(n) \)

Uses of Big-O and Theta notation
We can use our big-O or theta notation to describe
the best case, worst case, average case
We can use our big-O or theta notation to describe
the memory required
We can use our big-O or theta notation to describe
any other resource that a program uses

Searching
Linear Search
Binary Search
Hashing

Searching - Linear Search
public class LinearSearch {
    public static void main(String args[]) {
        int[] a = new int[13];
        for (int i = 0; i < a.length; i++) {
            a[i] = 2 * i + 1;
        }
        System.out.println("Value "+i+", is at index "+ find(a, i));
    }
    public static int find(int[] a, int search) {
        for (int i = 0; i < a.length; i++) {
            if (a[i] == search) {
                return i;
            }
        }
        return -1;
    }
} // LinearSearch

Output of program
Value 0 is at index -1
Value 1 is at index 0
Value 2 is at index -1
Value 3 is at index 1
Value 4 is at index -1
Value 5 is at index 2
Value 6 is at index -1
Value 7 is at index 3
Value 8 is at index -1
Value 9 is at index 4
Value 10 is at index -1
Value 11 is at index 5
Value 12 is at index -1
Value 13 is at index 6
Value 14 is at index -1
Value 15 is at index 7
Value 16 is at index -1
Value 17 is at index 8
Value 18 is at index -1
Value 19 is at index 9
Value 20 is at index -1
Value 21 is at index 10
Value 22 is at index -1
Value 23 is at index 11
Value 24 is at index -1
Value 25 is at index 12
Value 26 is at index -1

Recursive Binary Search
public class RBinarySearch {
    public static void main(String args[]) {
        int[] a = new int[13];
        for (int i = 0; i < a.length; i++) {
            a[i] = 2 * i + 1;
        }
        for (int i = 0; i <= 2 * a.length; i++) {
            System.out.println("Value "+i+", is at index "+
                + find(a, i, 0, a.length - 1));
        }
    }
    public static int find(int[] a, int search, int low, int high) {
        int middle = (low + high) / 2;
        if (a[middle] == search) {
            return middle;
        }
        if (a[middle] < search) {
            return find(a, search, middle + 1, high);
        } else {
            return find(a, search, low, middle - 1);
        }
    }
} // RBinarySearch

Non-Recursive Binary Search
public class BinarySearch {
    public static void main(String args[]) {
        int[] a = new int[13];
        for (int i = 0; i < a.length; i++) {
            a[i] = 2 * i + 1;
        }
        for (int i = 0; i <= 2 * a.length; i++) {
            System.out.println("Value "+i+", is at index "+
                + find(a, i));
        }
    }
    public static int find(int[] a, int search) {
        int low = 0;
        int high = a.length - 1;
        while (low <= high) {
            int middle = (low + high) / 2;
            if (a[middle] == search) {
                return middle;
            }
            if (a[middle] < search) {
                low = middle + 1;
            } else {
                high = middle - 1;
            }
        }
        return -1;
    }
} // BinarySearch
Hashing
Suppose you need to construct a system to quickly check if a credit card number is in a list of 50,000 stolen cards

Idea - Make a hash table of the cards
  Make an array of 100,000 entries and use the last 5 digits of the card to index this array
  If the entry is empty then put the card number there
  Otherwise, look for the next free location in the array and put the card number there

Lookup process
  Look in the array using the last 5 digits of the card to be checked
  If the card is there then return success
  Otherwise, keep looking until we find the card or we come to an empty location
  If we find the card then return success
  If we find a free location then return failure

Sorting
Selection Sort
Insertion Sort
Merge Sort
Quick Sort
Heap Sort

Selection Sort
Problem - Sort an array of items
Idea - build the correct sorted array one element at a time
Precondition (and postcondition): All of the items whose index is above i are in the correct place.
Algorithm step - find the largest item below i and move it to index i and decrement i
If we start with no items in place (i = length - 1) and keep executing the algorithm step
  The precondition will be satisfied each time
  The postcondition will be satisfied after each step
When i reaches zero the post condition states that the array is sorted

Selection Sort Code
public class SelectionSort {
    public static void selectionSort(int[] data) {
        for (int i = data.length - 1; i > 0; i--) {
            int big = 0;
            for (int j = 1; j <= i; j++) {
                if (data[big] < data[j]) {
                    big = j;
                }
            }
            int temp = data[i];
            data[i] = data[big];
            data[big] = temp;
        }
    }
} // SelectionSort

Insertion Sort
Problem - Sort an array of items
Idea - make a small list that is sorted and keep adding items to it in the proper order
Precondition (and postcondition): all items that have index below i are in order
Algorithm step: Take the next item and find out where it belongs in the sorted part of the array and put it there, moving all items following it up one location
If we start with no elements sorted (i = 0) and keep executing the algorithm step
  The precondition will be satisfied each time
  The postcondition will be satisfied after each step
When i reaches the length of the array the post condition states that the array is sorted

Insertion Sort Code
public class InsertionSort {
    public static InsertionSort(int[] data) {
        for (int i = 1; i < data.length; i++) {
            int temp = data[i];
            int j = i - 1;
            while (j >= 0 && data[j] > temp) {
                data[j + 1] = data[j];
                j--;
            }
            data[j + 1] = temp;
        }
    }
} // InsertionSort

Merge Sort
Problem - Sort an array of items
Idea - if we had an algorithm that sorted an array we could divide our array into two pieces and sort each piece using this algorithm; then we could merge these two pieces together to solve our problem
Precondition: two sorted arrays
Postcondition: one sorted array containing the elements from the two sorted arrays
Recursive algorithm:
  If the array contains 1 or fewer elements return success
  Split the array into two pieces and recursively sort each piece
  Merge these two pieces together
How do we merge?
  Keep taking the smallest remaining item from the two pieces and storing it in a temporary array
  Note that the smallest must be the first remaining item in one of the pieces
  Copy the temporary array back to where it belongs
Merge Sort Code

```java
public class MergeSort {
    public static void mergeSort(int[] data) {
        mergeSortR(data, 0, data.length);
    }
    public static void mergeSortR(int[] data, int first, int n) {
        if (n > 1) {
            int n1 = n / 2;
            int n2 = n - n1;
            mergeSortR(data, first, n1);
            mergeSortR(data, first + n1, n2);
            merge(data, first, n1, n2);
        }
    }
    public static void merge(int[] data, int first, int n1, int n2) {
        int[] temp = new int[n1 + n2];
        int copied0 = 0, copied1 = 0, int copied2 = 0;
        while (copied1 < n1 && copied2 < n2) {
            if (data[first + copied1] < data[first + n1 + copied2]) {
                temp[copied2++] = data[first + copied1++];
            } else {
                temp[copied1++] = data[first + n1 + copied2++];
            }
        }
        while (copied1 < n1) {
            temp[copied2++] = data[first + copied1++];
        }
        while (copied2 < n2) {
            temp[copied1++] = data[first + n1 + copied2++];
        }
        for (int i = 0; i < n1 + n2; i++) {
            data[first + i] = temp[i];
        }
    }
}
// MergeSort
```

Merge Sort Running Time

merges take time of $O(n)$
copies take time $O(n)$
Each level of the algorithm merges $2^k$ pieces of length $1/2^k$ for time $O(n)$
There are $\log n$ levels
Total time is $O(n \log n)$

Comparison table

<table>
<thead>
<tr>
<th>n</th>
<th>n \log n</th>
<th>n^2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>8</td>
<td>24</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>160</td>
<td>1024</td>
</tr>
<tr>
<td>128</td>
<td>896</td>
<td>16,384</td>
</tr>
<tr>
<td>512</td>
<td>4,608</td>
<td>262,144</td>
</tr>
<tr>
<td>2048</td>
<td>22,528</td>
<td>4,194,304</td>
</tr>
</tbody>
</table>

Quick Sort

Problem - Sort an array of items

Idea - guess at a "middle" or "pivot" value and
partition the items so that all items less than this
value are before all items greater than this item then
recurse on each half

How do we partition the data?

Precondition: a piece of the array with at least 2 items in
it
Postcondition: all items less than some pivot value are
before all items greater than this pivot value

Partition algorithm

Search from the beginning for some item larger than the pivot

Quick Sort Code

```java
public class QuickSort {
    public static void sort(int[] a) {
        qsort(a, 0, a.length - 1);
    }
    private static void qsort(int[] a, int lo, int hi) {
        int i, j, guess_median, tmp;
        if (lo < hi) {
            i = lo;
            j = hi;
            guess_median = a[(lo + hi) / 2];
            while (i < j) {
                if (a[i] < guess_median) {
                    i++;
                } while (a[j] > guess_median) {
                    j--;
                } if (i < j) { // swap
                    tmp = a[i]; a[i] = a[j]; a[j] = tmp;
                    i++;
                    j--;
                } else if (i == j) { // middle item
                    if (a[i] < guess_median) {
                        j--;
                    } else {
                        i++;
                    }
                }
                qsort(a, lo, j);
                qsort(a, i, hi);
            }
        }
    }
    // QuickSort
```
Trees

Main: Chapter 9
A tree consists of a finite number of nodes
If the tree is not empty then there is one special node called the “root”
Each node except for the root has exactly one parent
The root does not have a parent
The root is the ancestor of all other nodes

Some Tree Definitions
The root node of a tree is the unique node in the tree without a parent
A child of a given node is a node that has the given node as a parent
A grandchild of a node is a child of a child of the node
A descendant of a node is a node that is a child of the node or the child of a descendant of the node
A sibling node is a node with the same parent
A grandparent of a node is the parent of the parent of the node
An ancestor of a node is the parent of the node or the parent of an ancestor of the node
A leaf node or exterior node is a node without children
An interior node is a node with children
A branch is the path from a node to one of its children
The level or depth of a node is how many nodes there are to the root node (the root is at level 0 or depth 0)
The depth of a tree is the maximum depth of a node in the tree
A subtree is a tree consisting of a node and all of its descendants

Binary Trees
A binary tree is a tree where every node has at most two children, a left child and a right child
The subtree with the left child as the root is called the left subtree and the subtree with the right child as the root is called the right subtree
A full binary tree is a binary tree where every leaf node is at the same depth and every interior node has two children
How many nodes does a full binary tree of depth $n$ have?
A complete binary tree is a full binary tree where some of the rightmost leaf nodes may be missing

Sometimes we will look at other kinds of trees that have other kinds of restrictions such as ternary trees with no more than 3 children per node or $k$-ary trees, balanced trees etc.

Representing binary trees by arrays
Complete binary trees can be represented by arrays as follows:
Put the root at element 0
Put nodes at level 1, in order, next
Put nodes at level 2, in order, next etc.
If a node is at index $i$
What is the index of its parent?
What is the index of its left child (if it exists)?
What is the index of its right child (if it exists)?
The following tree would be represented by the array
$[1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 ]$

This representation is not suitable for all binary trees
Why not?

Representing binary tree nodes with Objects
We can use the definition of a binary tree directly to implement a binary tree node that contains integer data:

Public class IntBinaryTree {
    private IntBinaryTree left; // left tree
    private IntBinaryTree right; // right tree
    private int data; // int node data
    ...
} // IntBinaryTree

To make a tree that can associate arbitrary data with a node:

Public class BinaryTree {
    private BinaryTree left; // left subtree
    private BinaryTree right; // right subtree
    private Object data; // associated node data
    ...
} // BinaryTree
Building Trees

We need to add some methods to our Tree class so we can make trees and examine them:

```java
public class IntBT {
    private IntBT left; // the left subtree
    private IntBT right; // the right subtree
    private int data; // associated int node data
    public IntBT (IntBT left, IntBT right, int data) {
        this.left = left; this.right = right;
        this.data = data;
    }
    public IntBT getLeft() {
        return left;
    }
    public IntBT getRight() {
        return right;
    }
    public int getData() {
        return data;
    }
    public void setLeft(IntBT node) {
        left = node;
    }
    public void setRight(IntBT node) {
        right = node;
    }
    public void setData(int newData) {
        data = newData;
    }
} // IntBinaryTree
```

Building a small tree

From the bottom up

```java
IntBT left = new IntBT(null, null, 1);
IntBT right = new IntBT(null, null, 2);
IntBT root = new IntBT(left, right, 3);
```

From the top down

```java
IntBT root = new IntBT(null, null, 3);
root.setLeft( new IntBT(null, null, 1) );
root.setRight( new IntBT(null, null, 2) );
```

Additional Tree Methods

How would we add a method isLeaf() that returns true if and only if the node is a leaf node?

How would we add a method size() that returns the number of nodes of the subtree rooted at this node?

How would we add a method depth() that returns the integer depth of the subtree rooted at this node?

How would we add a method copy() that returns a copy of the subtree rooted at this node?

Also look at Main, 9.3

The Animal Guessing Game

Main, 9.3, pp 435-445

The computer asks a player a series of yes/no questions

The computer then guesses the answer

If the computer is wrong, the player must give the correct answer and type in a yes/no question that would distinguish between the computer’s answer and the right answer

Using Binary Trees To Play The Animal Game

Let all interior nodes of a binary tree contain a yes/no question

Let the left subtree contain all answers consistent with a yes answer

Let the right subtree contain all answers consistent with a no answer

Let all leaf nodes contain an answer that is consistent with the questions on the path to the root

Animal Game Program Execution

Set the current node to the root

While the current node is not a leaf node

Ask the question contained in the current node

If the answer is yes, make the current node refer to the left subtree

If the answer is no, make the current node refer to the right subtree

We are at a leaf node - give the answer contained in the current node

If the answer is wrong

Get a question and the correct answer

Make the left and right references of the current node refer to two new nodes

Put the correct answer in the correct new node

Copy the wrong answer from the current node to the other new node

Put the new question in the current node

Tree Traversals

How can we visit the nodes of the following tree?

```
1
/ \
2   3
/   / \
4   5   6
    /   /
     7
```

Inorder Traversal

Recursively visit the left subtree (if any)

Visit the current node

Recursively visit the right subtree (if any)

4 2 5 1 6 3 7

Preorder Traversal

Visit the current node

Recursively visit the left subtree (if any)

Recursively visit the right subtree (if any)

1 2 4 5 3 6 7

Postorder Traversal

Recursively visit the left subtree (if any)

Recursively visit the right subtree (if any)

Visit the current node

4 5 2 6 7 3 1
Processing data in Binary Trees
How would we add up all of the node data values in a tree?
Would it matter which traversal we used?
How would we find the maximum of all the node data values in a tree?
Would it matter which traversal we used?

Adding up all nodes

```java
public static int sumNodes( IntBT tree ) {
    if ( tree == null ) {
        return 0;
    } else {
        return tree.getData()
            + sumNodes( tree.getLeft() )
            + sumNodes( tree.getRight() );
    }
}
```

Finding the maximum

```java
public static int getMax( IntBT tree, int currentMax ) {
    if ( tree == null ) {
        return currentMax;
    } else {
        int data = tree.getData();
        int max = ( data > currentMax
                ? data : currentMax );
        max = getMax( tree.getLeft(), max );
        max = getMax( tree.getRight(), max );
        return max;
    }
}
```

How would we separate the traversal part of the algorithm from the data processing part of the algorithm?

Define an interface that declares a process method

```java
public interface IntProcessor {
    void process( int data, int depth );
} // IntProcessor
```

Define a class that implements this interface

```java
public class SumAndPrint implements IntProcessor {
    private int sum;
    public SumAndPrint() {
        sum = 0;
    }

    public void process( int data, int depth ) {
        for( int i = 0; i < depth; i++ ) {
            System.out.println(" ");
        } System.out.println( data );
        sum = sum + data;
    }

    public int getSum() {
        return sum;
    }
} // SumAndPrint
```

Using the data processor to sum nodes

Write a tree method that traverses the tree and calls the process method appropriately

```java
public class IntBT {
    private IntBT left, right;
    private int data; // associated int node data
    public IntBT( IntBT left, IntBT right, int data ) {
        this.left = left;
        this.right = right;
        this.data = data;
    }
    // accessor
    // setters
    public void inorder( IntProcessor p, int depth ) {
        if ( left != null )
            left.inorder(p, depth+1);
        p.process(data, depth);
        if ( right != null )
            right.inorder(p, depth+1);
    }
    public void preorder( IntProcessor p, int depth ) {
        p.process(data, depth);
        if ( left != null )
            left.preorder(p, depth+1);
        if ( right != null )
            right.preorder(p, depth+1);
    }
    public void postorder(IntProcessor p, int depth) {
        if ( left != null )
            left.postorder(p, depth+1);
        if ( right != null )
            right.postorder(p, depth+1);
        p.process(data, depth);
    }
} // IntBinaryTree
```

Testing the data processor

```java
public class TestIntBT {
    public static void main( String args[] ) {
        IntBT root = new IntBT{
            new IntBT( null, null, 4 ),
            new IntBT( null, null, 5 ),3),
            new IntBT( null, null, 6 ),
            new IntBT( null, null, 7 ),2),1);
        SumAndPrint sap;
        sap = new SumAndPrint();
        System.out.println( "Preorder" );
        root.preorder( sap, 0 );
        System.out.println();
        System.out.println( sap.getSum() );
        System.out.println();
        sap = new SumAndPrint();
        System.out.println( "Postorder" );
        root.postorder( sap, 0 );
        System.out.println();
        System.out.println( sap.getSum() );
        System.out.println();
        sap = new SumAndPrint();
        System.out.println( "Inorder" );
        root.inorder( sap, 0 );
        System.out.println();
        System.out.println( sap.getSum() );
        System.out.println();
        sap = new SumAndPrint();
        System.out.println( "Preorder" );
        root.preorder( sap, 0 );
        System.out.println();
        System.out.println( sap.getSum() );
        System.out.println();
        sap = new SumAndPrint();
        System.out.println( "Postorder" );
        root.postorder( sap, 0 );
        System.out.println();
        System.out.println( sap.getSum() );
        System.out.println();
    }
} // TestIntBT
Using the data processor to find the maximum

Define a class that implements the process interface

```java
public class FindMax implements IntProcessor {
    private int max = 0;
    private boolean flag = false;

    public FindMax () {
    }

    public void process( int data, int depth ) {
        if ( flag ) {
            if ( data > max ) {
                max = data;
            }
        } else {
            max = data;
            flag = true;
        }
    }

    public int getMax() {
        if ( flag ) {
            return max;
        } else {
            return 0; // ?? no data
        }
    }
}
```

Testing the maximum finder

```java
public class TestMax {
    public static void main ( String args[] ) {
        IntBT root
            = new IntBT(
                    new IntBT( null, null, 4),
                    new IntBT( null, null, 5),3),
                    new IntBT(
                            new IntBT( null, null, 6),
                            new IntBT( null, null, 7),2),1);
        FindMax fm;

        fm = new FindMax();
        root.preorder( fm, 0 );
        System.out.println( fm.getMax() );

        fm = new FindMax();
        root.postorder( fm, 0 );
        System.out.println( fm.getMax() );

        fm = new FindMax();
        root.inorder( fm, 0 );
        System.out.println( fm.getMax() );
    }
} // TestMax
```