Recursive Algorithms

CS 180
Sunil Prabhakar
Department of Computer Science
Purdue University
Recursive Algorithms

- Within a given method, we are allowed to call other accessible methods.
- It is also possible to call the same method from within the method itself.
- This is called a recursive call.
- For certain problems a recursive solution is very natural and simple.
- It is possible to implement a recursive algorithm without using recursion, but the code can be more complex.
Example of Recursion

- The **factorial of N** is the product of the first N positive integers:
  \[ N! = N \times (N - 1) \times (N - 2) \times \cdots \times 2 \times 1 \]

- This is useful for many situations, e.g.
  - there are \( n! \) possible sequences of \( n \) objects
  - there are \( n!(n-k)!/k! \) unique subsets of size \( k \), from a set of size \( n \).

- The factorial of N can be defined *recursively* as

\[
\text{factorial}(n) = \begin{cases} 
  n \times \text{factorial}(n-1) & \text{if } n > 1 \\
  1 & \text{otherwise}
\end{cases}
\]
An **recursive method** is a method that contains a statement (or statements) that makes a call to itself.

Implementing the factorial of N recursively will result in the following method.

```java
public int factorial(int N) {
    if (N == 1) {
        return 1;
    } else {
        return N * factorial(N-1);
    }
}
```

Test to stop or continue.

End case: recursion stops.

Recursive case: recursion continues.
As with any call, a recursive call results in the creation of temporary workspace for the called method and copying of parameters.

Each call to a method results in the creation of a separate workspace with new copies of variables (local and formal parameters).

When a recursive call ends, flow returns to the caller.
public int factorial(int N) {
    if (N == 1) {
        return 1;
    } else {
        return N * factorial(N-1);
    }
}

factorial(3);
Example

```java
public int factorial(int N) {
    if (N == 1) {
        return 1;
    } else {
        return N * factorial(N-1);
    }
}
```

Tuesday, April 10, 2012
Directory Listing

List the names of all files in a given directory and its subdirectories.

```java
public void directoryListing(File dir) {
    // assumption: dir represents a directory
    String[] fileList = dir.list(); // get the contents
    String dirPath = dir.getAbsolutePath();

    for (int i = 0; i < fileList.length; i++) {
        File file = new File(dirPath + "/" + fileList[i]);

        if (file.isFile()) {
            // it's a file
            System.out.println(file.getPath());
        } else {
            // it's a directory
            directoryListing(file); // so make a recursive call
        }
    }
}
```
List all anagrams of a given word.

Word: CAT

Anagrams:
- CTA
- ATC
- ACT
- TCA
- TAC
Anagram Solution

The basic idea is to make recursive calls on a sub-word after every rotation. Here’s how:

- **Rotation Left**
  - Recursion
  - *C A T*
  - Rotate Left
  - *A T C*
  - Rotate Left
  - *T C A*
  - Rotate Left
  - *T A C*

- **Recursion**

---

Tuesday, April 10, 2012
public void anagram(String prefix, String suffix) {
    String newPrefix, newSuffix;
    int numOfChars = suffix.length();
    if (numOfChars == 1) {
        // End case: print out one anagram
        System.out.println(prefix + suffix);
    } else {
        for (int i = 1; i <= numOfChars; i++) {
            newSuffix = suffix.substring(1, numOfChars);
            newPrefix = prefix + suffix.charAt(0);
            anagram(newPrefix, newSuffix);
            // recursive call
            // rotate left to create a rearranged suffix
            suffix = newSuffix + suffix.charAt(0);
        }
    }
}
The goal of the Towers of Hanoi puzzle is to move N disks from Peg 1 to Peg 3:

- Only one disk can be moved at a time.
- A larger disk cannot be placed on top of a smaller disk.
Towers of Hanoi Solution
public void towersOfHanoi(int N, //number of disks
            int from, //origin peg
            int to, //destination peg
            int spare ) { //"middle" peg

if ( N == 1 ) {
    moveOne( from, to );
}
else {
    towersOfHanoi( N-1, from, spare, to );
    moveOne( from, to );
    towersOfHanoi( N-1, spare, to, from );
}
}

private void moveOne( int from, int to ) {
    System.out.println( from + " ---> " + to );
}
Recursion vs Iteration

- Recursion has greater overhead due to method calls, variable setups etc.
- Recursion provides cleaner solutions
- Can be simulated using iteration and a stack-like structure
  - may add too much complexity (consider an iterative solution for Towers of Hanoi)
When Not to Use Recursion

- When recursive algorithms are designed carelessly, it can lead to very inefficient and unacceptable solutions.
- For example, consider the following:

```java
public int fibonacci(int N) {
    if (N == 0 || N == 1) {
        return 1;
    } else {
        return fibonacci(N-1) + fibonacci(N-2);
    }
}
```
Excessive Repetition

Recursive Fibonacci ends up repeating the same computation numerous times.

- fibonacci(5)
  - fibonacci(4)+fibonacci(3)
    - fibonacci(2)+fibonacci(1)
      - fibonacci(1)+fibonacci(0)
    - fibonacci(3)+fibonacci(2)
      - fibonacci(2)+fibonacci(1)
        - fibonacci(1)+fibonacci(0)
      - fibonacci(1)+fibonacci(0)
    - fibonacci(1)+fibonacci(0)
      - fibonacci(1)+fibonacci(0)
public int fibonacci(int N) {
    int fibN, fibN1, fibN2, cnt;
    if (N == 0 || N == 1) {
        return 1;
    } else {
        fibN1 = fibN2 = 1;
        cnt = 2;
        while (cnt <= N) {
            fibN = fibN1 + fibN2; //get the next fib no.
            fibN1 = fibN2;
            fibN2 = fibN;
            cnt++;
        }
        return fibN;
    }
}
What does the following method compute for N>0?

```java
public int recurse( int N ) {
    if (N == 1) {
        return 2;
    } else {
        return 2 * recurse(N-1);
    }
}
```

A. $N^2$
B. $2^N$
C. $N!$
D. $(N-1)!$
Overhead of recursion

Remember that each recursive call is expensive
- create local variables for each call
- copy arguments into local variables
- track the execution point for each call
- returned values are copied back
- local space is reclaimed
When Not to Use Recursion

In general, use recursion if

- A recursive solution is natural and easy to understand.
- A recursive solution does not result in excessive duplicate computation.
- The equivalent iterative solution is too complex.
Trees

- Trees are a very commonly used data structure in Computer Science.
- For example, simple binary trees can be used to maintain a sorted list of strings.
- Suppose we had an unknown number of strings to input, sort, then output.
- We could use linked lists:
  - Have to modify insert to ensure sorted order.
- Or, we can use trees:
  - More efficient.
A Sorted Binary Tree

d, b, a, e, f, c, do, dot

```
d
  b
  a
c
  e
  do
  f
  dot
```
The Sorted Binary Tree Node

class SBTNode {
   private SBTNode left, right;
   private String content;

   public SBTNode (String c) {
      left = right = null;
      content = c;
   }

   public void insert(String c) {
      if(c.compareTo(this.content)<=0) {
         if(left==null)
            left = new SBTNode(c);
         else
            left.insert(c);
      } else {
         if(right==null)
            right = new SBTNode(c);
         else
            right.insert(c);
      }
   }

   ...
public void print()
{
    if(left!=null)
        left.print();
    System.out.println(content);
    if(right!=null)
        right.print();
}

Example Use

```java
public static void main (String[] args) {

    SBTNode root = null;
    String input;

    input = JOptionPane.showInputDialog(null, "Enter String");
    if(input.length()>0) {
        root = new SBTNode(input);
        while(true){
            input= JOptionPane.showInputDialog(null, "Enter String");
            if(input.length()<1)
                break;
            root.insert(input);
        }
    root.print();
}
```
Expression Trees

\[ 5x + 2y \]

\[ 5 \times x \times 2 \times y \]

\[ 5 \times x + 2 \times y + \frac{z}{v} \]

\[ 5 \times x + 2 \times y + z \div v \]
Tree traversal

- Visiting all the nodes of a tree is called a tree traversal
  - e.g., in earlier example, we visited and printed out each node

- Important types of traversals
  - In order: left child, parent, right child
    - e.g., sorted output
  - Post order: left child, right child, parent
    - e.g., to get postfix version of expression
    - useful for expression evaluation using a stack
  - Pre order: parent, left child, right child
    - e.g., to get prefix version of expression
Expression Tree Traversals

In: $5 \times x + 2 \times y$
Post: $5x \times 2y \times +$
Pre: $\times + 5x \times 2y$

In: $5 \times x + 2 \times y + z \div v$
Post: $5x \times 2y \times + z v \div +$
Pre: $\times + 5x \times 2y \div z v$

Tuesday, April 10, 2012
General Trees

- In general, trees may have
  - more than two children
  - no ordering among children

- They don’t have:
  - cycles, multiple parents, single root

- More general structure: graph

- Applications:
  - class hierarchy,
  - index structures (databases)
Tree Traversal

```java
public void inOrderPrint()
{
    if(left!=null)
        left.print();
    System.out.println(content);
    if(right!=null)
        right.print();
}
```

```java
public void preOrderPrint()
{
    System.out.println(content);
    if(left!=null)
        left.print();
    if(right!=null)
        right.print();
}
```
Searching

- When we maintain a collection of data, one of the operations we need is a search routine to locate desired data quickly.
- Array searching
  - Given an array of values and a search key, return
    - the index of the key if found
    - -1 if key is not in the array
- Linear search
- Binary search
  - logarithmic performance, but requires sorted data
Search Result

Unsuccessful Search: \( \text{search}(45) \rightarrow \text{NOT FOUND} \)

Successful Search: \( \text{search}(12) \rightarrow 4 \)

```
number
0  1  2  3  4  5  6  7  8
23 17  5 90 12 44 38 84 77
```
Linear Search

- Search the array from the first to the last position in linear progression.

```java
public int linearSearch (int[] number, int searchValue) {
    int loc = 0;
    while (loc < number.length && number[loc] != searchValue) {
        loc++;
    }
    if (loc == number.length) {
        // Not found
        return NOT_FOUND;
    } else {
        return loc;  // Found, return the position
    }
}
```
public int binarySearch (int[] number, int searchValue) {
    int low   = 0, 
        high = number.length - 1, 
        mid  = (low + high) / 2; 
    while (low <= high && number[mid] != searchValue) {
        if (number[mid] < searchValue) {
            low = mid + 1;
        } else {  //number[mid] > searchValue 
            high = mid - 1;
        } 
        mid = (low + high) / 2;  //integer division will truncate 
    }
    if (low > high) {
        mid = NOT_FOUND;
    }
    return mid;
}
Binary Search Performance

- **Successful Search**
  - Best Case – 1 comparison
  - Worst Case – $\log_2 N$ comparisons

- **Unsuccessful Search**
  - Best Case = Worst Case – $\log_2 N$ comparisons

Since the portion of an array to search is cut into half after every comparison, we compute how many times the array can be divided into halves.

- After $K$ comparisons, there will be $N/2^K$ elements in the list. We solve for $K$ when $N/2^K = 1$, deriving $K = \log_2 N$. 
# Comparing N and $\log_2 N$ Performance

<table>
<thead>
<tr>
<th>Array Size</th>
<th>Linear – N</th>
<th>Binary – $\log_2 N$</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>4</td>
</tr>
<tr>
<td>50</td>
<td>50</td>
<td>6</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>7</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>9</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>2000</td>
<td>2000</td>
<td>11</td>
</tr>
<tr>
<td>3000</td>
<td>3000</td>
<td>12</td>
</tr>
<tr>
<td>4000</td>
<td>4000</td>
<td>12</td>
</tr>
<tr>
<td>5000</td>
<td>5000</td>
<td>13</td>
</tr>
<tr>
<td>6000</td>
<td>6000</td>
<td>13</td>
</tr>
<tr>
<td>7000</td>
<td>7000</td>
<td>13</td>
</tr>
<tr>
<td>8000</td>
<td>8000</td>
<td>13</td>
</tr>
<tr>
<td>9000</td>
<td>9000</td>
<td>14</td>
</tr>
<tr>
<td>10000</td>
<td>10000</td>
<td>14</td>
</tr>
</tbody>
</table>
When we maintain a collection of data, many applications call for rearranging the data in certain order, e.g. arranging Person information in ascending order of age.

Here’s the problem statement:

Given an array of N integer values, arrange the values into ascending order.

We will count the number of comparisons the algorithms make to analyze their performance.

- The ideal sorting algorithm will make the least possible number of comparisons to arrange data in a designated order.

We will compare different sorting algorithms by analyzing their worst-case performance.
Selection Sort

1. Find the smallest element in the list.

2. Exchange the element in the first position and the smallest element. Now the smallest element is in the first position.

3. Repeat Step 1 and 2 with the list having one less element (i.e., the smallest element is discarded from further processing).

This is the result of one pass.
### Selection Sort Passes

<table>
<thead>
<tr>
<th>Pass #</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
<td>17</td>
<td>23</td>
<td>90</td>
<td>12</td>
<td>44</td>
<td>38</td>
<td>84</td>
<td>77</td>
</tr>
<tr>
<td>sorted</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>12</td>
<td>23</td>
<td>90</td>
<td>17</td>
<td>44</td>
<td>38</td>
<td>84</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>5</td>
<td>12</td>
<td>17</td>
<td>90</td>
<td>23</td>
<td>44</td>
<td>38</td>
<td>84</td>
<td>77</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>12</td>
<td>17</td>
<td>23</td>
<td>38</td>
<td>44</td>
<td>77</td>
<td>84</td>
<td>90</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>12</td>
<td>17</td>
<td>23</td>
<td>38</td>
<td>44</td>
<td>77</td>
<td>84</td>
<td>90</td>
</tr>
</tbody>
</table>

Result AFTER one pass is completed.
public void selectionSort(int[] number) {
    int startIndex, minIndex, length, temp;
    length = number.length;

    for (startIndex = 0; startIndex <= length-2; startIndex++) {
        // each iteration of the for loop is one pass
        minIndex = startIndex;

        // find the smallest in this pass at position minIndex
        for (i = startIndex+1; i <= length-1; i++) {
            if (number[i] < number[minIndex]) minIndex = i;
        }

        // exchange number[startIndex] and number[minIndex]
        temp = number[startIndex];
        number[startIndex] = number[minIndex];
        number[minIndex] = temp;
    }
}
Selection Sort Performance

- We derive the total number of comparisons by counting the number of times the inner loop is executed.
- For each execution of the outer loop, the inner loop is executed $\text{length} - \text{start}$ times.
- The variable $\text{length}$ is the size of the array. Replacing $\text{length}$ with $N$, the array size, the sum is derived as...
Bubble Sort

- With selection sort, we make one exchange at the end of one pass.
- Bubble sort improves the performance by making more than one exchange during each pass.
- By making multiple exchanges, we will be able to move more elements toward their correct positions using the same number of comparisons as the selection sort makes.
- The key idea of the bubble sort is to make pairwise comparisons and exchange the positions of the pair if they are out of order.
One Pass of Bubble Sort

The largest value 90 is at the end of the list.
public void bubbleSort(int[] number) {
    int temp, bottom, i;
    boolean exchanged = true;
    bottom = number.length - 2;

    while (exchanged) {
        exchanged = false;
        for (i = 0; i <= bottom; i++) {
            if (number[i] > number[i + 1]) {
                temp = number[i];    // exchange
                number[i] = number[i + 1];
                number[i + 1] = temp;
                exchanged = true;     // exchange is made
            }
        }
        bottom--;}
}
Bubble Sort Performance

- In the worst case, the outer while loop is executed $N-1$ times for carrying out $N-1$ passes.

- For each execution of the outer loop, the inner loop is executed $\text{bottom}+1$ times. The number of comparisons in each successive pass is $N-1$, $N-2$, $\ldots$, $1$. Summing these will result in the total number of comparisons.

- So the performances of the bubble sort and the selection sort are approximately equivalent. However, on the average, the bubble sort performs much better than the selection sort because it can finish the sorting without doing all $N-1$ passes.

$$
(N - 1) + (N - 2) + \cdots + 1 = \sum_{i=1}^{N-1} i = \frac{N(N-1)}{2}
$$

$$
= \frac{N^2 - N}{2} \approx N^2
$$
Quicksort

- To sort an array from index low to high, we first select a pivot element $p$.
  - Any element may be used for the pivot, but for this example we will use `number[low]`.
- Move all elements less than the pivot to the first half of an array and all elements larger than the pivot to the second half. Put the pivot in the middle.
- Recursively apply quicksort on the two halves.
Any element can be used as a pivot. For simplicity, we use number[low] as pivot.
public void quickSort(int[] number, int low, int high) {
    if (low < high) {
        int mid = partition(number, low, high);
        quickSort(number, low, mid - 1);
        quickSort(number, mid + 1, high);
    }
}
Quicksort Performance

- In the worst case, quicksort executes roughly the same number of comparisons as the selection sort and bubble sort.
- On average, we can expect a partition process to split the array into two roughly equal subarrays.