CSE1030 – Introduction to Computer Science II

Lecture #20

Recursion II

CSE1030 – Lecture #20

- Review: Recursion
- Iteration versus Recursion
- Examples: Linked-List Functions
- Example: Fractals
- Example: AI Robot Path Planning
- We're Done!

Recursion Review

```
static int factorial(int x)
{
   if(x == 0)
      return 1;
   else
      return x * factorial(x-1);
}


static public void main(String[] args)
{
   int fact = factorial(10);
   System.out.println("fact = " + fact);
}
```

The "Termination Condition" or "Base Case"

Formulation of the "big" problem in terms of a "smaller" version, the "Recursive Case"

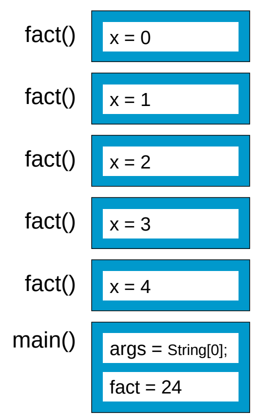
Theory: Definition of Recursion

- A function is **Recursive** if it calls itself (directly or indirectly) from within its own body
- Two components of a Recursive Solution:
  1. A solution to the problem that involves a simpler instance of the problem (called the "**Recursive Case**")
  2. A **Direct Solution** to a simple version of the problem (called the "**Termination Case**", or "**Base Case**")
- Any algorithm can be implemented with either a recursive or iterative algorithm, although some problems are easier to solve one way or the other

Practical: Coding Recursion

- A function is **Recursive** if it calls itself (directly or indirectly) from within its own body
- Recursive Functions always have:
  1. An "if" statement
     - The "if" tests whether the function input is a "**Base Case**"
     - If the input is a Base Case, then a value is returned directly (without calling the function again)
  2. Otherwise, the input requires the "**Recursive Case**"
     - The function calls itself with an argument that is closer to the Base Case than the original argument

How? Recursive Execution Stack



- Every time we recurse, Java creates a new stack frame, within which the variables exist.
- This is how recursion works.

```
int fact(int x)
{
   if(x == 0)
      return 1;
   else
      return x * fact(x-1);
}

public void main(String[] args)
{
   int f = fact(4);
   System.out.println("fact = " + f);
}
```

CSE1030 – Lecture #20

- Review: Recursion
- Iteration versus Recursion
- Examples: Linked-List Functions
- Example: Fractals
- Example: AI Robot Path Planning
- We're Done!

Iterative versus Recursive Solutions

Recursive Solution

```
int factorial(int x)
{
   if(x == 0)
      return 1;
   else
      return x * factorial(x-1);
}
```

Iterative Solution

```
int factorial(int x)
{
   int answer = 1;

   for(int i = 1; i <= x; i++)
      answer *= i;

   return answer;
}
```

Which implementation is faster?

- In class demonstration of:

`benchmark.java`

Comments about Speed and Memory Usage

- Sometimes Speed is very important (real-time applications, games, etc.)
- Sometimes Efficient Memory Usage is very important (embedded programming)
- Most of the time, though, there is lots of time and memory, and so the algorithm can be written either with recursion or with iteration, whichever is easier
- Some people don't like recursive code because of the possibility of stack overflows
  - But running out of memory is running out of memory, regardless of whether the algorithm is recursive or iterative
  - A well-written implementation should be relatively reliable

CSE1030 – Lecture #20

- Review: Recursion
- Iteration versus Recursion
- Examples: Linked-List Functions
- Example: Fractals
- Example: AI Robot Path Planning
- We're Done!

Recursion and Linked-Lists

- The best way to learn recursion is to study lots of examples, and to code some up yourself!
- Linked-Lists provide a great opportunity to use recursion – we will look at several examples…
- In class demonstration of:

`recursiveLinkedLists.java`

- Code Samples follow…

Find the length of a linked-list

```
int length(Node p)
{
   if(p == null)
      return 0;
   else
      return 1 + length(p.next);
}
```

Recursive Solution

```
int length(Node p)
{
   int i = 0;
   while(p != null)
   {
      p = p.next;
      i++;
   }
   return i;
}
```

Iterative Solution

Print a linked-list

```
void printList(Node p)
{
   if (p != null)
   {
      System.out.println(p.data);
      printList(p.next);
   }
}
```

Recursive Solution

```
void printList(Node p)
{
   while(p != null)
   {
      System.out.println(p.data);
      p = p.next;
   }
}
```

Iterative Solution

Printing Forward or Backward?

```
void printList(Node p)
{
   if (p != null)
   {
      System.out.println(p.data);
      printList(p.next);
   }
}
```

Forward

The order that we print and recurse matters!

```
void printReverseList(Node p)
{
   if (p != null)
   {
      printReverseList(p.next);
      System.out.println(p.data);
   }
}
```

Backward

Copying a linked-list

- Copying a linked-list is much easier using recursion…

Recursive Solution:

```
Node copy(Node p)
{
   if(p == null)
      return null;
   else
      return new Node(p.data, copy(p.next));
}
```

Copying a linked-list

Iterative Solution (Part 1)

```
static Node copy(Node pointer)
  {
     Node head = null;
     Node tail = null;

     while(pointer != null)
     {
        Node newNode = new Node(pointer.data, null);

        // Special Case: if the list is empty
        if(head == null)
        {
           head = tail = newNode;
        }
```

Copying a linked-list

Iterative Solution (Part 2)

```
        else
        {
           tail.next = newNode;
           tail = newNode;

        }

        pointer = pointer.next;
     }

     return head;
  }
```

Reversing a linked-list

Recursive Solution:

```
Node reverse(Node p)
{
   return reverse(p, null);
}

Node reverse(Node p, Node ancestor)
{
   if(p == null)                 // empty?
      return ancestor;

   Node theNextNode = p.next;    // remember who's next

   p.next = ancestor;            // point this node backwards

   return reverse(theNextNode, p);   // recurse to next node
}
```

Reversing a linked-list

Iterative Solution:

```
Node invertLinkedList(Node headFrom)
{
   Node headTo = null;

   while(headFrom != null)
   {
      // remove the head of the "From" list
      Node theNodeThatWeAreMoving = headFrom;
      headFrom = headFrom.next;

      // add the node to the "To" list
      theNodeThatWeAreMoving.next = headTo;
      headTo = theNodeThatWeAreMoving;
   }

   return headTo;
}
```

Inserting into an Ordered linked-list

Recursive Solution:

```
Node insertInOrder(String key, Node p)
{
   if(p == null || p.data.compareTo(key) >= 0)
      return new Node(key, p);

   else
   {
      p.next = insertInOrder(key, p.next);
      return p;
   }
}
```

Usage:

```
head = insertInOrder("newdata", head);
```

Remember Inserting into an Ordered a linked-list? (Part 1)

```
void insertInOrder(String data)
{
   // special case for inserting into
   // an empty list
   if(head == null)
   {
      head = new Node(data, null);
      return;
   }
```

Handle insert into empty List

```
   // do we come before the first element?
   // then we have to update the head
   // pointer
   if(head.data.compareTo(data) > 0)
   {
      head = new Node(data, head);
      return;
   }
```

Handle insert at the Head of the List

Remember Inserting into an Ordered a linked-list? (Part 2)

```
   // find the correct spot in the list
   Node pointer = head;
   while(pointer.next != null
      && pointer.next.data.compareTo(data) < 0)
      pointer = pointer.next;
```

Find the node above the correct spot

```
   // create the new node, the 'next' pointer should
   // point to the next node in the list
   Node newNode = new Node(data, pointer.next);
```

New node

```
   // update the 'next' pointer
   // of the previous node
   pointer.next = newNode;
}
```

Update **'next' pointer** of the node **above** the new node

Deleting from an Ordered linked-list

Recursive Solution:

```
Node deleteInOrder(String key, Node p)
{
   if(p == null)
      return p;

   else if (p.data.equals(key))
      return p.next;

   else
   {
      p.next = deleteInOrder(key, p.next);
      return p;
   }
}
```

Usage:

```
head = deleteInOrder("deldata", head);
```

Remember Deleting from an Ordered a linked-list? (Part 1)

```
static Node deleteByValue(String data)
{
   // empty list?
   if(head == null)
      return null;
```

Handle Empty List

```
   // location is the top of the list
   if(head.data.equals(data))
   {
      Node oldHead = head;
      head = head.next;
      return oldHead;
   }
```

Check whether we're deleting from the head of the list, and handle it

Remember Deleting from an Ordered a linked-list? (Part 2)

```
// otherwise, we're looking for the
// node above the one we want to delete
Node pointer = head;
while(pointer.next != null
   && !pointer.next.data.equals(data))
{
   pointer = pointer.next;
}
```

Find the node above the one we want to delete

```
// not found?
if(pointer.next == null)
   return null;
```

The data to be deleted is not found

Remember Deleting from an Ordered a linked-list? (Part 3)

```
   // remember the deleted node, so
   // we can return it
   Node deletedNode = pointer.next;
```

Grab a reference to the node we're deleting, to return

```
   // now, update the 'next' pointer
   pointer.next = pointer.next.next;
```

Here's where we delete the node

```
   return deletedNode;
}
```

Return a reference to the deleted node

Deleting the last Node of a linked-list

Recursive Solution:

```
Node deleteLast(Node p)
{
   if(p == null || p.next == null)
      return null;

   else
   {
      p.next = deleteLast(p.next);
      return p;
   }
}
```

Usage:

```
head = deleteLast(head);
```

Remember Deleting the End of a linked-list? (Part 1)

```
Node deleteFromEnd()
{
   // if the list is empty, then there's
   // nothing to do
   if(head == null)
      return null;


   // if the list only has 1 node
   // delete from head of list
   if(head.next == null)
   {
      Node oldHead = head;
      head = null;
      return oldHead;
   }
```

Handle Empty List

If there is only 1 node, then this is the same as "delete from head of the list"

Deleting the End? (Part 2)

```
   // otherwise, we're looking for the
   // second-last node of the list, which
   // is the node whose '.next.next'
   // pointer is null
   Node pointer = head;
   while(pointer.next.next != null)
      pointer = pointer.next;

   // remember the deleted node, so
   // we can return it (just in case
   // the user wants it)
   Node deletedNode = pointer.next;

   // now, update the 'next' pointer
   pointer.next = null;

   return deletedNode;
}
```

Find the second from last node

Grab a reference to the node we're deleting, to return

Here's where we delete the node

Return a reference to the deleted node

"Two-List" Operations

- All of the Linked-List operations we have seen so far have used only 1 linked-list
- Next, let's look at three operations that combine two linked-lists into one list:
  - Append
  - Shuffle
  - Merge
- For these examples will use the following data:

p =	apple → banana → cherries → fig → grapes → null
q =	aardvark → bat → cat → dragon → elephant → null

"Two-List" Examples

- In class demonstration of:

  recursiveLinkedLists2.java

- Code Samples follow…

Append

Recursive Solution:

```
Node append(Node p, Node q)
{
   if(p == null)
      return q;

   else
   {
      p.next = append(p.next, q);
      return p;
   }
}
```

Result:

```
apple
banana
cherries
fig
grapes
aardvark
bat
cat
dragon
elephant
```

Shuffle

Recursive Solution:

```
Node shuffle(Node p, Node q)
{
   if(p == null)
      return q;

   else if(q == null)
      return p;

   else
   {
      // Note we exchange p and q here
      p.next = shuffle(q, p.next);
      return p;
   }
}
```

Result:

```
apple
aardvark
banana
bat
cherries
cat
fig
dragon
grapes
elephant
```

(Alphabetical) Merge

Recursive Solution:

```
Node merge(Node p, Node q)
{
   if(p == null)
      return q;

   else if(q == null)
      return p;

   else if(p.data.compareTo(q.data) < 0)
   {
      p.next = merge(p.next, q);
      return p;
   }
   else
   {
      q.next = merge(p, q.next);
      return q;
   }
}
```

Result:
(alphabetical)

```
aardvark
apple
banana
bat
cat
cherries
dragon
elephant
fig
grapes
```

CSE1030 – Lecture #20

- Review: Recursion
- Iteration versus Recursion
- Examples: Linked-List Functions
- Example: Fractals
- Example: AI Robot Path Planning
- We're Done!

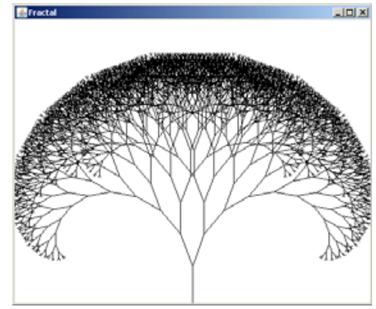
Recursion and Fractals

- **Self Similar** problems are very well suited to Recursion, because they naturally look like a smaller version of themselves as you "zoom in" to them
- Fractals are defined as structures that are self similar
- This means that recursion is very useful for generating fractals…

Fractal Tree Example

- In class demonstration of:

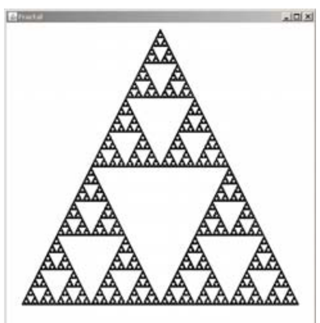
`FractalTree.java`



Sierpinski Triangle Example

- In class demonstration of:

`FractalTriangle.java`



CSE1030 – Lecture #20

- Review: Recursion
- Iteration versus Recursion
- Examples: Linked-List Functions
- Example: Fractals
- Example: AI Robot Path Planning
- We're Done!

Remember last lecture when we said…

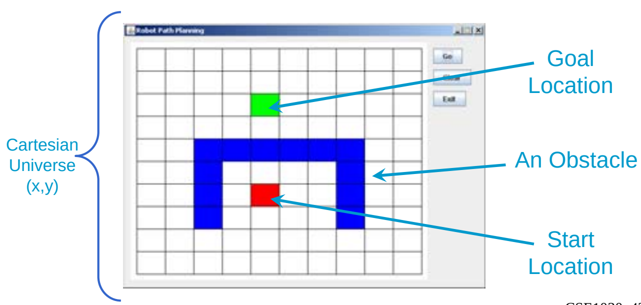
- We don't have to decompose a "big" problem down only into little problems that we can solve
- Some problems can be decomposed into a smaller version of the same problem
- In this case, we don't have to solve the "big" problem or even the "smaller" problem, instead we can get away with solving a very very small version of the problem…

Recursion and Artificial Intelligence

- Because Recursion does not require an explicit solution of a problem, we can use recursion to solve problems for which it is difficult to think of a solution…
- For this reason there is a correlation between recursion and Artificial Intelligence
  - Many of the AI programming languages are strongly recursive (e.g., Lisp, Prolog)

Can you think of an Algorithm that can solve this problem?

- Find a path from the **Start** to the **Goal**
- Robot can only move **up**, **down**, **left**, or **right**



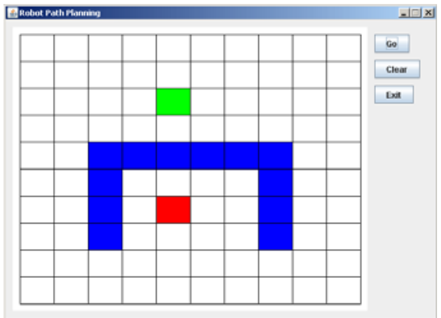
Can You Think of a Solution to This Problem?

- The **Movement Planning Problem** is a difficult AI problem that involves trying to figure-out how to move from a **Start** location to a **Goal** location
- The idea is not to solve the specific problem posed on the previous slide, but to write an algorithm that can solve this problem regardless of the positions of the Start, Goal, and Obstacles
- Through recursion, we don't really have to solve this problem, we just have to know how to get closer to the solution, and how to solve a very easy "Base Case" (like recognising when we have arrived at the Goal)

RobotPlanning Example

- In class demonstration of:

`RobotPlanning.java`



CSE1030 – Lecture #20

- Review: Recursion
- Iteration versus Recursion
- Examples: Linked-List Functions
- Example: Fractals
- Example: AI Robot Path Planning
- We're Done!

Next topic…

Recursion I