Recursion vs Induction

22c:31
Chapter 1
Hantao Zhang
Recursion

- Recursion means defining something, such as a function, in terms of itself
  - For example, let $f(x) = x!$
  - We can define $f(x)$ as
    
    $f(x) = \text{if } x < 2 \text{ then } 1 \text{ else } x \ast f(x-1)$
Recursion example

• Sequences are functions from natural numbers to reals:
  \[ f(i) = a_i \]
  \[ a_0, a_1, a_2, a_3, \ldots, a_n. \]

Find \( f(1), f(2), f(3), \) and \( f(4), \) where \( f(0) = 1 \)

Let \( f(n+1) = f(n)^2 + f(n) + 1 \)
\[ f(1) = f(0)^2 + f(0) + 1 = 1^2 + 1 + 1 = 3 \]
\[ f(2) = f(1)^2 + f(0) + 1 = 3^2 + 3 + 1 = 13 \]
\[ f(3) = f(2)^2 + f(0) + 1 = 13^2 + 13 + 1 = 183 \]
\[ f(4) = f(3)^2 + f(0) + 1 = 183^2 + 183 + 1 = 33673 \]
A tree is a set of nodes (vertices) and edges (links, lines) connecting the nodes such that there are no cycles and that every two nodes are connected by at least one path (a sequence of adjacent edges).

In many Computer Science applications, one particular node of the tree is designated as the root, other nodes are arranged by levels below the root level.
A Binary Tree: A rooted tree such that each node has at most two nodes connected to it at the next level, distinguished as the left child node and the right child node (if present).

A recursive definition for binary trees:
(Basis) An empty tree is a binary tree;
(Recursive step) A root whose two child nodes are binary trees;
(Closure) Every binary tree must be constructed by repeated use of the recursive steps.
A BinaryNode class in Java:

```java
public class BinaryNode
{
  public:

    // Data in each node; accessible by other package routines
    Comparable element;    // The data in the node
    BinaryNode left;       // Left child
    BinaryNode right;      // Right child

    // Two Constructors
    BinaryNode(Comparable theElement )
    {
        this( theElement, null, null );
    }
    BinaryNode(Comparable theElement, BinaryNode lt, BinaryNode rt)
    {
        element = theElement;
        left    = lt;
        right   = rt;
    }

    ... ...
}
```
Computing the height of a binary tree

public int height () {  
    // Pre: assuming the current node is not null.  
    // Post: return the height of the binary tree  
    // rooted by the current node.  
    int h1 = 1, h2 = 1;  
    if (left != null) h1 = left.height() + 1;  
    if (right != null) h2 = right.height() + 1;  
    return (h1 < h2)? h2 : h1;  
} // height
public void traverse () {
    // preorder action, e.g., display(element);
    BinaryNode child = left;
    if (child != null) child.traverse();
    // inorder action, e.g., display(element);
    child = right;
    if (child != null) child.traverse();
    // postorder action, e.g., display(element);
} // traverse
Preorder Binary Tree Traversal (Print)

```java
public void preorder () {
    System.out.print(data + " ");
    if (left!=null) left.preorder();
    if (right!=null) right.preorder();
}
```

Assume that we use this on an expression tree with the labels being operators or simple one-letter variable names. The expression \((\sim A - B) * (C / (D + E))\) is represented as

```
*  
/  /
/~ ~\  
  \\
\  
\  \  
\  \  
\  \  
B C +
```

The preorder would print \(* - \sim A B / C + D E\)
The expression \((\sim A - B) \ast (C / (D + E))\)

has postorder \(A \sim B - C D E + / \ast\)

Each of preorder and postorder provides an unambiguous way to denote expressions without parentheses. This is more compact than infix notation and can lead to very efficient expression evaluations.
Inorder Traversal of a Binary Expression Tree

public void inorder () {
    System.out.print("(");
    if (left!=null) left.inorder();
    System.out.print(data);
    if (right!=null) right.inorder();
    System.out.print(")");
}

The expression  (~A - B) * ( C / (D + E) )
is printed as

((~(A)) -(B)) * (((C) /(D+E)))

The correct fully parenthesized version.
Evaluation of a binary tree:

```java
public int eval () {
    switch(element) {
        case '~' : return -right.eval(); break;
        case '+' : return left.eval() + right.eval(); break;
        case '-' : return left.eval() - right.eval(); break;
        case '*' : return left.eval() * right.eval(); break;
        case '/' : return left.eval() / right.eval(); break;
        default: return toInt(element);
    }
}
```

This expression evaluates as –28.
A binary tree representation of general rooted trees:

Each node has two link fields, in which the left link points to the leftmost child node while the right link points to the first sibling node on the right side.

A general rooted tree

A binary tree implementation:
- **Left link**
- **Right link**
public int height () {
    // **Pre:** assuming the current node is not null.
    // **Post:** return the max height of the nonbinary trees
    //            rooted by the current node or its siblings.
    int h1 = 1, h2 = 1;
    if (left != null) h1 = left.height() + 1;
    if (right != null) h2 = right.height();
    return (h1 < h2)? h2 : h1;
} // height
Common Tree Traversal Template for Non-binary Tree

```java
public void traverse () {
    // preorder action, e.g., display(element);
    BinaryNode child = left;
    if (child != null) child.traverse();
    child = right;
    if (child != null) child.traverse();
    // postorder action, e.g., display(element);
} // traverse
```
Hanoi Tower - *Instructions*

1. Transfer all the disks from pole A to pole B.

2. You may move only ONE disk at a time.

3. A large disk may not rest on top of a smaller one at anytime.
Try this one!

Shortest number of moves??
And this one

Shortest number of moves??
Now try this one!

Shortest number of moves??
How to solve Tower of Hanoi of n disks?

- If \( n = 1 \), “move disk 1 from A to B”, done.
- If \( n > 1 \),
  1. Solve the Tower of Hanoi of \( n-1 \) disks, from A to C;
  2. “move disk \( n \) from A to B”
  3. Solve the Tower of Hanoi of \( n-1 \) disks, from C to B.

```cpp
Hanoi ( int n, char A, char B, char C ) {
    if (n==1)  cout << "move disk 1 from " << A << " to " << B << endl;
    else {
        Hanoi(n-1, A, C, B);
        cout << "move disk " << n << " from " << A << " to " << B << endl;
        Hanoi(n-1, C, B, A);
    }
}
```

Counting the moves:
Let \( f(n) \) be the number of moves for \( n \) disks.

\[
f(1) = 1; \\
f(n) = 2f(n-1) + 1.
\]
Let $f(n)$ be the number of moves for $n$ disks.

\[ f(1) = 1; \]
\[ f(n) = 2f(n-1) + 1. \]

<table>
<thead>
<tr>
<th>Number of Disks</th>
<th>Number of Moves</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$f(1) = 1$</td>
</tr>
<tr>
<td>2</td>
<td>$f(2) = 2*1 + 1 = 3$</td>
</tr>
<tr>
<td>3</td>
<td>$f(3) = 2*3 + 1 = 7$</td>
</tr>
<tr>
<td>4</td>
<td>$f(4) = 2*7 + 1 = 15$</td>
</tr>
<tr>
<td>5</td>
<td>$f(5) = 2*15 + 1 = 31$</td>
</tr>
<tr>
<td>6</td>
<td>$f(6) = 2*31 + 1 = 63$</td>
</tr>
</tbody>
</table>
Let $f(n)$ be the number of moves for $n$ disks.

$f(1) = 1$;

$f(n) = 2f(n-1) + 1$.

Prove: $f(n) = 2^n - 1$.

By induction.

- Base step: $n = 1$.
  - Left = $f(1) = 1$;
  - Right = $2^1 - 1 = 1$

- Induction hypothesis: $f(n-1) = 2^{n-1} - 1$.

- Inductive step:
  - Left = $f(n) = 2f(n-1) + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$.
  - Right = $2^n - 1$. 
Fascinating fact

So the formula for finding the number of steps it takes to transfer $n$ disks from post A to post C is:

$$2^n - 1$$

- If $n = 64$, the number of moves of single disks is $2$ to the $64$th minus $1$, or $18,446,744,073,709,551,615$ moves! If one worked day and night, making one move every second it would take slightly more than $580$ billion years to accomplish the job! - far, far longer than some scientists estimate the solar system will last.
Fibonacci sequence

• Definition of the Fibonacci sequence

  – Non-recursive:
    \[ F(n) = \frac{(1 + \sqrt{5})^n - (1 - \sqrt{5})^n}{\sqrt{5} \cdot 2^n} \]

  – Recursive:
    \[ F(n) = F(n-1) + F(n-2) \]
    or:
    \[ F(n+1) = F(n) + F(n-1) \]

• Must always specify base case(s)!
  – \( F(1) = 1, F(2) = 1 \)
  – Note that some will use \( F(0) = 1, F(1) = 1 \)
Fibonacci sequence in Java

```java
long Fibonacci (int n) {
    if ( (n == 1) || (n == 2) )
        return 1;
    else
        return Fibonacci (n-1) + Fibonacci (n-2);
}

long Fibonacci2 (int n) {
    return (long) ((Math.pow((1.0+Math.sqrt(5.0)),n)-
        Math.pow((1.0-Math.sqrt(5.0)),n)) /
        (Math.sqrt(5) * Math.pow(2,n)));
}
```
Recursion pros

- Easy to program
- Easy to understand
Recursion cons

- Consider the recursive Fibonacci generator
- How many recursive calls does it make?
  - $F(1): 1$
  - $F(2): 1$
  - $F(3): 3$
  - $F(4): 5$
  - $F(5): 9$
  - $F(10): 109$
  - $F(20): 13,529$
  - $F(30): 1,664,079$
  - $F(40): 204,668,309$
  - $F(50): 25,172,538,049$
  - $F(100): 708,449,696,358,523,830,149 \approx 7 \times 10^{20}$
    - At 1 billion recursive calls per second (generous), this would take over 22,000 years
    - But that would also take well over $10^{12}$ Gb of memory!
Defining sets via recursion

• Same as mathematical induction:
  – Base case (or basis step)
  – Recursive step

• Example: the set of positive integers
  – Basis step: $1 \in S$
  – Recursive step: if $x \in S$, then $x+1 \in S$
Defining sets via recursion

• Give recursive definitions for:
  
a) The set of odd positive integers
    ■ 1 ∈ S
    ■ If x ∈ S, then x+2 ∈ S
  
b) The set of positive integer powers of 3
    ■ 3 ∈ S
    ■ If x ∈ S, then 3*x ∈ S
  
c) The set of polynomials with integer coefficients
    ■ 0 ∈ S
    ■ If p(x) ∈ S, then p(x) + cx^n ∈ S
      - c ∈ Z, n ∈ Z and n ≥ 0
Defining strings via recursion

• Terminology
  – $\lambda$ is the empty string: “”
  – $\Sigma$ is the set of all letters: \{ a, b, c, …, z \}
    • The set of letters can change depending on the problem
• We can define a set of strings $\Sigma^*$ as follows
  – Base step: $\lambda \in \Sigma^*$
  – If $w \in \Sigma^*$ and $x \in \Sigma$, then $wx \in \Sigma^*$
  – Thus, $\Sigma^*$ s the set of all the possible strings that can be generated with the alphabet
Defining strings via recursion

• Let $\Sigma = \{ 0, 1 \}$

• Thus, $\Sigma^*$ is the set of all binary all binary strings
  – Or all possible computer files
String length via recursion

• How to define string length recursively?
  – Basis step: \( \text{len}(\lambda) = 0 \)
  – Recursive step: \( \text{len}(wx) = \text{len}(w) + 1 \) if \( w \in \Sigma^* \) and \( x \in \Sigma \)

• Example: \( \text{len}(\text{“aaa”}) \)
  – \( \text{len}(\text{“aaa”}) = \text{len}(\text{“aa”}) + 1 \)
  – \( \text{len}(\text{“aa”}) = \text{len}(\text{“a”}) + 1 \)
  – \( \text{len}(\text{“a”}) = \text{len}(\text{“”}) + 1 \)
  – \( \text{len}(\text{“”}) = 0 \)
  – Result: 3
Strings via recursion example

• Given a string \( x = a_1a_2\ldots a_n \) \( x^R \) stands for its reversal: \( x^R = a_na_{n-1}\ldots a_1 \). Eg. \( x = abc, \ x^R = cba. \)

• A string \( x \) is a palindrome if \( x = x^R \). Eg. \( x = aba. \)

• Give a recursive definition for the set of string that are palindromes
  – We will define set \( P \), which is the set of all palindromes

• **Basis step:** \( \lambda \in P \)

• **Second basis step:** \( x \in P \) when \( x \in \Sigma \)

• **Recursive step:** \( xpx \in P \) if \( x \in \Sigma \) and \( p \in P \)
How to enumerate all subsets?

• A set of n elements has $2^n$ subsets

• An n-bit integer has $2^n$ values, each value corresponds to one subset.

• Can we use the values of an n-bit integer for printing out each subset?
How to enumerate all subsets?

• A set of n elements has $2^n$ subsets
• A n-bit integer has $2^n$ values, each value corresponds to one subset.

```java
class EnumerateDemo {
    public static void main(String[] args) {
        enumerateSubsets(10);
    }

    public static void enumerateSubsets (int n) {
        // Pre: n < 32
        for (int x = 0; x < (1 << n); x++) {
            System.out.print("{");
            for (int j = 1; j <= n; j++) if (x & (1 << (j-1)) != 0)
                System.out.print(j + ", ");
            System.out.print("}\n");
        }
    }
}
```
How to enumerate k-combinations of n numbers?

• Let $N$ be the set \{ 1, 2, 3, ..., $n$ \} and $0 < k < n$.
• Any subset $X$ of $N$, $|X| = k$, is a $k$-combination of $N$.
• There are $C(k, n) = n!/(k!(n – k)!)$ such combinations.
• How to enumerate all of them? E.g., if $n = 9$, $k = 5$, how can we enumerate them from \{ 1, 2, 3, 4, 5 \} to \{ 5, 6, 7, 8, 9 \}?
How to enumerate $k$-combinations of $n$ numbers?

- Let $N$ be the set $\{1, 2, 3, \ldots, n\}$ and $0 < k < n$.
- Any subset $X$ of $N$, $|X| = k$, is a $k$-combination of $N$.
- There are $C(k, n) = \frac{n!}{(k!(n - k)!)}$ such combinations.
- How to enumerate all of them? E.g., if $n = 9$, $k = 5$, how can we enumerate them from $\{1, 2, 3, 4, 5\}$ to $\{5, 6, 7, 8, 9\}$?
- If $k$ is fixed, we can use $k$-nested loops:

```java
public static void enumerate5Combinations (int n) {
    // Pre: k = 5
    for (int x1 = 1; x1 <= (n - 4); x1++)
        for (int x2 = x1+1; x2 <= (n - 3); x2++)
            for (int x3 = x2+1; x3 <= (n - 2); x3++)
                for (int x4 = x3+1; x4 <= (n - 1); x4++)
                    for (int x5 = x4+1; x5 <= n; x5++) {
                        System.out.print("{ ");
                        System.out.print(x1+", "+x2+, "+x3+, "+x4+, "+x5); \\
                        System.out.print(" }\n");
                    }
}
```
How to enumerate \( k \)-combinations of \( n \) numbers?

- Let \( N \) be the set \( \{ 1, 2, 3, \ldots, n \} \) and \( 0 < k < n \).
- Any subset \( X \) of \( N \), \( |X| = k \), is a \( k \)-combination of \( N \).
- There are \( C(k, n) = \frac{n!}{(k!(n-k)!)} \) such combinations.
- We may use the idea in `enumerate5Combinations` for general \( k \).

```java
public static void enumerateCombinations (int k, int n) {
    int x[] = new int[100];    // k <= 100
    for (int j = 0; j < k; j++) x[j] = j+1;
    while (true) {
        printCombination(x, k);
        if (nextCombination(x, k, n) == false) break;
    }
}

public static boolean nextCombination (int x[], int k, int n) {
    for (int j = k-1; j >= 0; j--) if (x[j] < (n - k + j)) {
        x[j]++;
        for (int i = 1; i < k - j;  i++) x[i+j] = x[j]+i;
        return true; }
    return false;
}
```
How to enumerate all permutations of n numbers?

• Let $N$ be the set $\{ 1, 2, 3, \ldots, n \}$.
• There are $n!$ different permutations on $N$.
• How to enumerate all of them? E.g., if $n = 5$, how can we enumerate them from $\{ 1, 2, 3, 4, 5 \}$ to $\{ 5, 4, 3, 2, 1 \}$?
How to enumerate all permutations of n numbers?

• Let \( N \) be the set \{ 1, 2, 3, \ldots, n \}.
• There are \( n! \) different permutations on \( N \).
• How to enumerate all of them? E.g., if \( n = 5 \), how can we enumerate them from \{ 1, 2, 3, 4, 5 \} to \{ 5, 4, 3, 2, 1 \}?
• If \( n \) is fixed, we can use \( n \)-nested loops:

```java
public static void enumerate5Permutations () {
    // Pre: n = 5
    for (int x1 = 1; x1 <= 5; x1++)
        for (int x2 = 1; x2 <= 5; x2++) if (x1 != x2)
            for (int x3 = 1; x3 <= 5; x3++) if (x3 != x1 && x3 != x2)
                for (int x4 = 1; x4 <= 5; x4++) if (x4 != x1 && x4 != x2 && x4 != x3)
                    for (int x5 = 1; x5 <= 5; x5++)
                        if (x5 != x1 && x5 != x2 && x5 != x3 && x5 != x4) {
                            System.out.print("{ ");
                            System.out.print(x1+", "+x2+", "+x3+", "+x4+", "+x5);
                            System.out.println(" }\n");
                        }
}
```
How to enumerate k-combinations of n numbers?

- Let $N$ be the set $\{1, 2, 3, \ldots, n\}$ and $0 < k < n$.
- Any subset $X$ of $N$, $|X| = k$, is a $k$-combination of $N$.
- There are $C(k, n) = \frac{n!}{k!(n – k)!}$ such combinations.
- How to enumerate all of them? E.g., if $n = 9$, $k = 5$, how can we enumerate them from $\{1, 2, 3, 4, 5\}$ to $\{5, 6, 7, 8, 9\}$?
- If $k$ is fixed, we can use $k$-nested loops:

```java
public static void enumerate5Combinations (int n) {
    // Pre: k = 5
    for (int x1 = 1; x1 <= (n - 4); x1++)
        for (int x2 = x1+1; x2 <= (n - 3); x2++)
            for (int x3 = x2+1; x3 <= (n - 2); x3++)
                for (int x4 = x3+1; x4 <= (n - 1); x4++)
                    for (int x5 = x4+1; x5 <= n; x5++) {
                        System.out.print("{ ");
                        System.out.print(x1+", "+x2+", "+x3+", "+x4+", "+x5);
                        System.out.print(" }\n");
                    }
}
```