Reduction & Recursion
Overview

Reduction definition
Reduction techniques
Recursion definition
Recursive thinking
(Many) recursion examples
Indirect recursion
Runtime stack

23 December 2007 © Ariel Shamir

Reduction

Reduction is a technique for solving problems by rephrasing the problem in terms of another problem whose solution is known.

We say that we reduce the problem into another known problem.

Examples for reduction in mathematics.

Reduction as a programming technique.

23 December 2007 © Ariel Shamir

Reduction Examples

\[ P(x) \rightarrow Q_1(x), Q_2(x), \ldots Q_k(x) \]

Where \( Q_i(x) \) is simpler than \( P(x) \)

- Sorting \( \rightarrow \) Finding minimum (selection sort)
- Polynomial multiplication \( \rightarrow \) Integer multiplication and addition
- More…

23 December 2007 © Ariel Shamir
Recursion As Reduction

\[ P(x) \Rightarrow P(x_1), P(x_2), \ldots P(x_k) \]

When \( x_i \ll x \) (\( x_i \) is smaller than \( x \))

Recursion is a fundamental programming technique that can provide an elegant solution to certain kinds of problems. In recursion we solve a problem by reducing it to a similar, smaller problem whose input might be different.

Recursion In Programs

```java
public static void f() {
    f();
}

public static void f(int n) {
    f(n-1); // reduction!
}

public static void f(int n) {
    if (n == 0) // base case!
        return;
    f(n-1); // reduction!
}
```

Effective Recursion Rules

1. **Self call**
   
   \[ P \Rightarrow P \]

2. **Reduction of some sort and merging**
   
   \[ P(n) \Rightarrow P(x_1) \cup P(x_2) \ldots \cup P(x_k) \]

3. **Base case (recursion halt)**
   
   \[ P(0) \text{ or } P(1) \text{ etc.} \]
Recursive Methods

A method in Java that invokes itself is called a recursive method.

An effective recursive method has a base case which solve the task without a recursive call and the recursive part which solve the task by reducing it and using a recursive call.

If there is no base case and sometimes if there is no reduction, the sequence will not come to an end. We call this case an infinite recursion.

Factorial Example

If n is a positive integer, we define n! - the factorial of n - as follows:

\[ n! = 1 \times 2 \times 3 \times \ldots \times n \]

A simple implementation for a method that computes the factorial of a number would be with a loop that goes over the expression and keep track of the partial multiplication (next slide)

Factorial Code (Loop)

```java
/**
 * Computes the factorial of a number.
 * @param n The given number.
 * @return n! - The factorial of n.
 */
public static long factorial(long n) {
    long fact = 1;
    for (int i=1; i<=n; i++) {
        fact *= i;
    }
    return fact;
}
```
Factorial Reduction

The previous solution is simple, but we want to find a recursive solution:

For every integer n, such that n>1:

\[ n! = 1 \times 2 \times ... \times n = (1 \times 2 \times ... \times (n-1)) \times n = (n-1)! \times n \]

So we have:

\[ 1! = 1 \]

\[ n! = (n-1)! \times n, \text{ for } n>1 \]

Factorial Recursive Solution

Recursion: \( n! \rightarrow (?)! \)

Reduction: \( n \rightarrow n-1 \)

Merge: use \( \times \) operator

Base: if \( n == 1 \) then \( 1! = 1 \)

From (n-1)! to n!

We can use these observations to derive a new implementation for computing \( n! \).

If \( n \) is 1, there is no problem, we know that \( 1! = 1 \).

If \( n > 1 \) we can't say immediately the result, but if we only knew how to compute \( (n-1)! \) we could compute \( n! \) using the fact that \( n! = (n-1)! \times n \).
How to find \((n-1)!\) ?

Note, that we can make the reduction for any arbitrary \(n\), such that \(n>1\). So we can reduce the problem from \(n!\) to \((n-1)!\) to \((n-2)!\) etc. until we will face the problem of computing \(1!\) which is known.

Factorial Recursive Computation

\[

t_5 \rightarrow 120 \\
t_4 \rightarrow 24 \\
t_3 \rightarrow 6 \\
t_2 \rightarrow 2 \\
t_1 \rightarrow 1
\]

Factorial Recursive Code

```java
/**
 * Computes the factorial of a number.
 * @param n A positive integer.
 * @return n! - The factorial of n.
 */
public static long factorial(long n) {
    if (n==1) {
        return 1;
    }
    return n*factorial(n-1);
}
```
Recursive Factorial Code (Version 2)

```java
/**
 * Computes the factorial of a number.
 * @param n A positive integer.
 * @return n! - The factorial of n.
 */
public static long factorial(long n) {
    return ((n==1) ? 1 : n*factorial(n-1));
}
```

Factorial Code Proof

Usually recursive algorithms are proved using mathematical induction.
In our implementation of ‘factorial’ we have to show the following:

- Factorial give the correct result for n=1.
- If ‘factorial’ returns the correct answer for an arbitrary number n such that n>1, then ‘factorial’ returns the correct answer for n+1.

Recursive Complexity?

Factorial Recursion:

\[
T(N) = T(N-1) + C_0 \\
T(N-1) = T(N-2) + C_0 \\
\vdots \\
T(1) = C_1
\]

\[
T(N) = (N-1)C_0 + C_1 = O(N)
\]

measuring complexity:

- Non-recursive programs \(\rightarrow\) counting loops
- Recursive programs \(\rightarrow\) solving recursion!
isNumeric() Example

Consider the task of checking if a given string consist only of digits. We want to write a method `isNumeric()` that for a given string would return true if and only if all the characters in the string are from '0', '1', ..., '9'.
A simple implementation would loop on the characters of the string and return false if one found that is not a digit (next slide).

isNumeric() Code

```java
/** *
 * Checks if a given string consists only *
 * from digit symbols. *
 */
public static boolean isNumeric(String s) {
    for (int i=0; i<s.length(); i++) {
        char c = s.charAt(i);
        if (c<'0' || c>'9') {
            return false;
        }
    }
    return true;
}
```

IsNumeric Recursive Solution

Recursion: `isNum(S) \rightarrow isNum(???)`
Reduction: `S \rightarrow S.substring(1)`
Merge: use `&&` operator
Base: if `S` is empty then it is numeric (because of the `&&` operator)
Recursive isNumeric()

Observe the following facts:

- An empty string doesn’t contain non-digit characters, therefore the method should return true for the empty string.
- If the first character of s is non-digit then the method should return false.
- If the first character of s is a digit symbol, then the string s is numeric if and only if the rest of s (without the first character) is numeric.

```java
/**
 * Checks if a given string consists only from digit symbols.
 */
public static boolean isNumeric(String s) {
    if (s.length()==0) {
        return true;
    }
    char c = s.charAt(0);
    return (c>='0' && c<='9' &&
            isNumeric(s.substring(1)));
}
```

Recursive Programming

Note that just because we can use recursion to solve a problem, it doesn't mean we should.

For instance, we usually would not use recursion to solve the previous problem. However, for some problems, recursion provides an elegant solution, often cleaner than an iterative version.
Fibonacci Example

There are many cases where a recursive solution is very appropriate. Fibonacci series is a series of integers \( f_0, f_1, f_2, \ldots \) that is defined as follows:

\[
\begin{align*}
f_0 &= 1 \\
f_1 &= 1 \\
f_n &= f_{n-1} + f_{n-2} \text{ for } n > 1
\end{align*}
\]

A recursive solution seems very suitable.

---

Fibonacci Recursive Solution

Recursion: \( \text{Fib}(n) \rightarrow \text{Fib}(???) \)

Reduction: \( n \rightarrow n-1, n-2 \)

Merge: use + operator

Base: if \( n == 0 \) or \( n == 1 \) then we have 1

---

Fibonacci Code

```java
/**
 * Returns the n number in fibonacci series
 */
public static int fibonacci(int n) {
    if (n<=1) {
        return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
```

1 1 2 3 5 8 13 21 34 55 89 144...
Fibonacci Computation

\[ F(4) \]
\[ F(3) \]
\[ F(2) \]
\[ F(1) \]
\[ F(0) \]

Fibonacci Complexity

\[ f_n = f_{n-1} + f_{n-2} \]
\[ T(n) = T(n-1) + T(n-2) + C_0 \]
\[ T(n) = f_n + (f_{n-1} - 1) C_0 \] (Induction)
\[ T(n) = (C_0 + 1) f_n - C_0 \approx O(f_n) \]
\[ f_n \approx (0.5(1+\sqrt{5}))^n = O(\phi^n) \]
Exponential!

Can We Do Better?

**Trivially:** just sum up from \( f_0 \) and \( f_1 \)

**Where is the price of recursion?**

**Doing redundant work:**
- To compute \( f_n \), we need \( f_{n-1} \) and \( f_{n-2} \) but when computing \( f_{n-1} \), we didn’t use \( f_{n-2} \) but computed it again “from scratch”!
Binary Enumeration

Print all possible combinations of 0 and 1 in a given range:

000
001
010
011
100
101
110
111

Non-recursive

int[] a = new int[n];
for (a[0]= 0 ; a[0] < 2 ; a[0]++){
    for (a[1]= 0 ; a[1] < 2 ; a[1]++){
        ...// ← n-2 loops come here
        printArray(a);
    }
}

Enumeration Recursive Solution

Recursion: enum(n) → enum(???)
Reduction: n → n-1,n-1
Merge: use concatenation
Base: if n == 0 then we print
New: we send something (the array) through the recursion
Recursive

```java
void printCombRec(int[] a, int i) {
    if (i == a.length) {
        printArray(a);
        return;
    }
    a[i] = 0;
    printCombRec(a, i+1);
    a[i] = 1;
    printCombRec(a, i+1);
}
```

GCD Example

The Greatest Common Divisor of two integers \(a, b\) is defined as the largest integer that divides both \(a\) and \(b\). We denote the greatest common divisor of \(a\) and \(b\) by \(\text{gcd}(a, b)\).

Finding the GCD

Suppose \(a > b\).
If \(b\) divides \(a\), then \(\text{gcd}(a, b) = b\) (because \(b\) also divides itself and there is no greater integer that divides \(b\)). – This will be the base case.
If \(b\) doesn’t divide \(a\), let \(m\) be the reminder of the division of \(a\) by \(b\) (\(m = a \mod b\)).
We can write: \(a = k \cdot b + m, \ m < b\)
Finding the GCD (Cont.)

Denote \( x = \gcd(a,b) \). \( x \) must also divide \( m \) (since \( a = kb + m \)). Then \( x \) divides \( a,b,m \!\). 

Denote \( y = \gcd(b,m) \). \( y \) must also divide \( a \) (since \( m = a - kb \)). Then \( y \) divides \( a,b,m \!\). 

Since \( y \) divides \( a \) and \( b \), \( y \leq \gcd(a,b) = x \) 

Since \( x \) divides \( m \) and \( b \), \( x \leq \gcd(b,m) = y \) 

\( y \leq x \leq y \rightarrow \gcd(a,b) = x = y = \gcd(b,m) \!\) 

Therefore, \( \gcd(a,b) = b \) if \( b \) divides \( a \) or else \( \gcd(a,b) = \gcd(b,a \% b) \!\). 

Recursive \( \text{gcd()} \) Code

```java
/**
 * Finds the greatest common divisor of
 * two input numbers.
 */
public static long gcd(long a,long b) {
    if (b < a) {
        long m = a%b;
        return (m == 0 ? b : gcd(b,m));
    } else {
        long m = b%a;
        return (m == 0 ? a : gcd(a,m));
    }
}
```

GCD Complexity

Assume we have \( a > b \). 
If \( b < a/2 \) then \( a\%b < b < a/2 \) 
If \( b > a/2 \) then \( a\%b = a-b < a/2 \) 

So after 2 GCD steps we replace \( a \) with a number smaller than \( a/2 \). 
At most after \( 2 \log_2 a \) we get a number smaller than 2. 

So GCD is better than \( O(\log a) \).
GCD Worst Case Complexity

Fibonacci consecutive pair:
\[ f_n / f_{n-1} = (f_{n-1} + f_{n-2}) / f_{n-1} = 1 + f_{n-2} / f_{n-1} \]
i.e. \[ f_n \% f_{n-1} = f_{n-2} \]
Therefore we move from \((f_n, f_{n-1})\) to \((f_{n-1}, f_{n-2})\).

Partial Sum Example

Problem: given a series of integers \(a_1, a_2, ..., a_n\) and a number \(s\), we want to determine weather there is a partial series whose elements sum up to \(s\).

13, 5, 10, 3, 12, 7 Sum = 21?
13 + 5 + 3 = 21!

Non Recursive Solution

Sort?
Start with minimum?
Start with maximum?
Loop?
Search for all possible sub-sums!
How many sub-sums? \(2^N\) Why?
\[ 2^2 = 4, 2^3 = 8, ... \]
\[ 2^{10} = 1024, 2^{20} > 1000000 \]
Partial Sum Solution

If the series is empty the answer is true iff s is zero.
Otherwise, we should check the following two options:

- The last element is not included in the partial series - in this case we reduce the problem to finding a partial sum s in the rest of the series.
- The last element is included - we reduce the problem to finding a partial sum \( s - a_n \) in the rest.

hasPartialSum()

Example

```java
/**
 * Checks if a given series of integers has a partial series whose sum of elements is equal to a given integer.
 * @param elements The series of integers.
 * @param sum The requested sum
 */
public static boolean hasPartialSum(int[] series, int sum) {
    return hasPartialSum(series, sum, series.length);
}
```

Partial Sum Code

```java
// Checks for a partial sum in a given series
// length indicates the length of the series
private static boolean hasPartialSum(
    int[] series, int sum, int length) {
    if (length==0) {
        return (sum ==0); 
        return hasPartialSum(series, sum, length-1) ||
            hasPartialSum(series, sum-series[length-1], length-1);
    }
```
Evaluation Order

// Checks for a partial sum in a given series
// length indicates the length of the series
private static boolean hasPartialSum(
    int[] series, int sum, int length) {
    if (length==0)
        return (sum ==0);
    return
        hasPartialSum(series, sum, length-1) ||
        hasPartialSum(series, sum-series[length-1],length-1);
}

Depth First Evaluation

- hasPartialSum(series, sum, length-1)
- hasPartialSum(series, sum, length-2)
- hasPartialSum(series, sum, length-3)
  ...
- hasPartialSum(series, sum, 1)
- hasPartialSum(series, sum, 0)
Following the Path

\[ F(S, x_1 \ldots x_n) \]
\[ F(S, x_1 \ldots x_{n-1}) \]
\[ F(S-x_n, x_1 \ldots x_{n-1}) \]
\[ F(S-x_{n-1}, x_1 \ldots x_{n-2}) \]
\[ F(S-x_n-x_{n-1}, x_1 \ldots x_{n-2}) \]
\[ F(S, x_1 \ldots x_{n-2}) \]

Stopping the Path

// Checks for a partial sum in a given series
// length indicates the length of the series
private static boolean hasPartialSum(
    int[] series, int sum, int length) {
    if (sum == 0)
        return true;
    if (length == 0 || sum < 0)
        return false;
    return
        hasPartialSum(series, sum, length-1) ||
        hasPartialSum(series, sum-series[length-1], length-1);
}

Catalan Numbers

Example

The \(n\)th Catalan number \(- C_n \) - Comes up in many problems.

The number of valid strings of length 2n consisting of \(n\) ‘(’ symbols and \(n\) ’)’ symbols, which satisfy that in every prefix of the string the number of ‘(’ symbols is greater than or equal to the number of ’)’ symbols. For example "(()())" is a valid string, while "()(()" is not.
**Enumeration of Valid Strings**

1: ()
2: () () ()
3: ((())) () () () () () ()
4: () ((())) () (()) () () () () ()

---

**Number of (N+2) Polygon Triangulations**

---

**Number of Groupings for Multiplication**

1: (a b)
2: ((a b c) (a b c))
3: (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d)
4: (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d) (a (b c) d)
Catalan Numbers
Recursion

The base case is \( C_0 = 1 \)

For \( n > 0 \) we have

\[
C_n = \sum_{k=1}^{n} (C_{k-1} \times C_{n-k})
\]

i.e. loop and sum previous Catalan numbers.

Catalan Example

```java
/**
 * Returns the n’th Catalan number.
 */
public static int catalan(int n) {
    if (n==0) {
        return 1;
    }
    int sum = 0;
    for (int k=1; k<=n; k++) {
        sum += catalan(k-1)*catalan(n-k);
    }
    return sum;
}
```

Towers of Hanoi Example

```
```
Solution for 3 Rings

Recursive Solution

Towers of Hanoi Program

// Simulates the Towers of Hanoi problem for a
// given number of disks.
private static void solve(
    Pile from, Pile to, Pile using, int n) {
    if (n==1) {
        to.addDisk(from.removeDisk());
        return;
    }
    solve(from, using, to, n-1);
    solve(from, to, using, 1);
    solve(using, to, from, n-1);
}
Listing Files Example

Suppose we want to list all the files under a given folder. We can do it as follows:

List folders under folder $f$:

Let $L$ be the list of all files and folders directly under $f$.

For each element $f_i$ in $L$:

- If $f_i$ is a folder: list folders under $f_i$.
- Otherwise: print the name of $f_i$.

Listing File Code

```java
import java.io.File;
// List all the files under a given folder
// Usage: java ListFiles <folder-name>
class ListFiles {
    /**
     * List of the files under a given folder
     * given as an argument.
     */
    public static void main(String[] args) {
        listFiles(new File(args[0]));
    }
}
```

ListFiles Recursion Code

```java
// List all files under a given folder
private static void listFiles(File folder) {
    String[] files = folder.list();
    for (int i=0; i<files.length; i++) {
        File file = new File(folder, files[i]);
        if (file.isDirectory()) {
            listFiles(file);
        } else {
            System.out.println(file);
        }
    }
}
```
Depth First Search

Listing files is just an example of a well known algorithm for search on a graph, which is called depth first search. DFS first follows path up to their end (deep search) and then continues to other paths (breadth search).

DFS Example

Other Search Possibility?

BFS: Breadth First Search
Indirect Recursion

A method invoking itself is considered to be direct recursion.

- A method m1 could invoke another method m2, which invokes another m3, etc., until eventually the original method m1 is invoked again.

This is called indirect recursion, and requires all the same care as direct recursion. It is often more difficult to trace and debug.

Fractals

Hilbert Curve: Space Filling Curves

X \rightarrow (\text{count} > 0 ? \text{YFXFXFY : return )}
Y \rightarrow (\text{count} > 0 ? \text{XFYFYFX : return )}
F \rightarrow (\text{count} > 0 ? \text{F : draw)
How Does Recursion Work?

If we examine again the implementation of factorial, the non-recursive solution describes the whole process of the computation. Instead of using variables inside the method, the return values are accumulating the product.

How Is 5! Computed?

We call factorial(4) to compute 4! when the computation is finished the result (return value of factorial(4)) is multiplied by 5 and is returned. How do we compute factorial(4)... recursively you know the answer! ☺

Recursive Solutions

When trying to understand a recursive solution, or when seeking a recursive solution to a problem try to avoid thinking of the problem in this way. Assume that the method works for smaller inputs and try to understand how it uses this assumption to solve the problem with the current input.
Recursion Execution

However, we do explain the process by which the computer executes a recursive method. This will reveal us more about the process of executing a program in general.

Executing a Program

When we execute a Java program, the virtual machine creates a thread that processes the program code. The thread begins in the first statement of the main() method of our program, and advances according to the flow of the program until it reaches the end of main().
The Runtime Stack

The thread uses a section of memory that is called its Runtime Stack. At the beginning the Runtime stack is empty.

Method ‘Frames’

As the thread enters the main() method, it allocates a ‘frame’ on the stack for the execution of main. The frame stores all the local variables defined in main etc.
A Method Call

If the thread encounters a method call, say to \texttt{foo()}, it pushes a new frame on the stack for the variables of \texttt{foo()}.

A Method Return

When the execution of the method \texttt{foo()} is finished (the method returns) the frame that was created for it is removed from the stack.

Many Method Calls

If main calls \texttt{m1} which calls \texttt{m2} which calls \texttt{m3}, all three frames will be accumulated one on top of the other in the stack.
Other Method Calls

m3 returns and m4 is called from m2

Main()
m1()
m2()
m3()
m4()

The Runtime Stack in Recursion

public static long factorial(int n) {
    //...
    return factorial(n-1)*n;
}

The Runtime stack is filled by the frames of the same function called with different parameters

Main()
factorial(n)
factorial(n-1)
factorial(n-2)
...

Folding Back the Stack

When the recursion reaches the base state (here n==1), The stack begins to unfold back by popping the frames out.

Main()
factorial(n)
factorial(n-1)
factorial(n-2)

23 December 2007 © Ariel Shamir
End of Recursion

When it reaches the first call, all the calculations have been accumulated and the last call returns the result.

factorial(n-1)
factorial(n)
Main()