Examples of Iterative and Recursive Algorithms

Fast Exponentiation

Recursive Definition:

$$a^n = \begin{cases} 1, & \text{if } n = 0, \\ (a^{\lfloor n/2 \rfloor})^2 & \text{if } n > 0 \text{ and } n \text{ is even,} \\ (a^{\lfloor n/2 \rfloor})^2 a & \text{if } n \text{ is odd.} \end{cases}$$

Problem: Given integers a, n, and m with $n \geq 0$ and $0 \leq a < m$, compute a^n (mod m).

Input: Integers a, n, and m, with $0 \leq n$ and $0 \leq a < m$.

Output: a^n (mod m)

Algorithm (recursive):

```
Integer fastExp( Integer a, Integer n, Integer m )
    if ( n == 0 )
        return 1;
    if ( n == 1 )
        return a;
    x = fastExp( a, ⌊n/2⌋, m);
    if ( even(n) )
        return x^2 (mod m);
    else
        return x^2 a (mod m);
```

Greatest Common Divisor (Euclid's Algorithm)

Recursive Definition: For $a,b \geq 0$, $gcd(a,b) = \begin{cases} a & \text{if } b = 0, \\ gcd(b,\ a \bmod b) & \text{otherwise.} \end{cases}$

Problem: Given nonnegative integers a and b, not both 0, compute $gcd(a,b)$.

Input: Nonnegative integers a and b, not both zero.

Output: The greatest common divisor of a and b.

Algorithm (recursive)

```
Integer gcd( Integer a, Integer b )
    if ( b == 0 )
        return a;
    else
        return gcd( b, a mod b);
```

Notes:

1) If $b > a$, the first recursive call effectively exchanges a and b.
2) In many applications, we need an extended version of Euclid's algorithm, one that also produces integers u and v such that $ua+vb = gcd(a,b)$. The algorithm below outputs a triple (d,u,v) such that $d = gcd(a,b)$ and $ua+vb = d$

```
TripleOfIntegers ext_gcd( Integer a, Integer b )
    if ( b == 0 )
        return (a, 1, 0);
    else
        (d,u,v) = ext_gcd(b, a mod b);
        return (d, v, u−v⌊a/b⌋);
```

Fibonacci Numbers

Recursive definition: $F_0 = 0,\ F_1 = 1,\ F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$.

Problem: Given a nonnegative integer n, compute F_n.

Input: A nonnegative integer n.

Output: The Fibonacci number F_n.

Algorithm (recursive):

```
Integer fibon( Integer n)
    if ( n ≤ 1 )
        return n;
    else
        return fibon(n–1) + fibon(n–2);
```

Caution: *A C/C++ function or Java method based on this description will be <u>hopelessly inefficient</u>, unless n is very small. If we attempt to compute F_{200} (a 41-digit number) using such a function, the program will not finish in the lifetime of the earth, even with a computer millions of times faster than present ones. By contrast, with the iterative algorithm below, we can compute F_{200} easily in a tiny fraction of a second.*

Algorithm (alternate iterative description)

```
Integer fibon( Integer n)
    if ( n ≤ 1 )
        return n;
    b = 0;
    c = 1;
    for ( i = 2,3,...,n )    // c = F_{i–1}, b = F_{i–2}, a = F_{i–3} (except when i=2).
        a = b;
        b = c;
        c = b + a;           // Now c = F_i, b = F_{i–1}, a = F_{i–2}.
    return c;
```

Rank Search

Problem: Find the k^{th} smallest element of a set S.

Input: A non-empty set S (distinct elements), a total ordering < on S, and an integer k with $1 \leq k \leq |S|$.

Output: The k^{th} smallest element of S. (Numbering starts at 1; $k = 1$ gives smallest.)

Algorithm (recursive)

```
Element rankSearch( Set S, Integer k)
    Choose an element p of S;     // A good strategy: p = random elt of S.
    S1 = ∅; S2 = ∅;
    for ( each element x of S–{p} )
        if ( x < p )
            S1 = S1 ∪ {x};
        else if ( x > p )
            S2 = S2 ∪ {x};
    // Now S = S1 ∪ {p} ∪ S2, each elt of S1 is < p, and each elt of S2 is > p.
    if ( k ≤ |S1| )
        return rankSearch( S1, k);
    else if ( k ≥ |S1|+2 )
        return rankSearch( S2, k–1–|S1|);
    else
        return p;
```

Notes:

1) *This algorithm may be used to find the median of S.*
2) *The for-loop partitions S into S_1, {p}, and S_2. Partitioning takes n–1 comparisons, where n = |S|. If the elements of S are stored in an array of size n, there is a particularly efficient algorithm that performs the partitioning in place. This same partitioning algorithm is used in quicksort.*
3) *This is probably the most efficient algorithm known for finding the k^{th} smallest in the <u>expected case</u>, but it is rather slow in the <u>worst case</u> (to be discussed in class.)*

Height of a Binary Tree

Recursive definition: For a binary tree t,

$$height(t) = \begin{cases} -1 & \text{if } t \text{ is empty,} \\ 1 + \max(\, height(\, leftSubtree(t)\,), \\ \qquad height(\, rightSubtree(t)\,)\,) & \text{otherwise.} \end{cases}$$

Problem: Given a binary tree t, find its height.

Input: A binary tree t.

Output: An integer, the height of t. (The empty tree has height –1; the tree whose left and right subtrees are empty has height 0.)

Algorithm (recursive)

```
Integer height( BinaryTree t)
    if ( empty(t) )
        return –1;
    else
        return 1 + max( height(leftSubtree(t)), height( rightSubtree(t)) );
```