Recursion and Trees

Recursive Code – A program that calls itself and stops when a *termination condition* is achieved.

Recursive algorithms

- Solves a problem by solving one or more of smaller instances of the same problem
- Recursive functions in programming languages, like C, C++, or Pascal, correspond to recursive definitions of mathematical functions
  - Languages such as Fortran and Cobol do not support recursion while Lisp and Prolog do not know any other control structure except recursion
- Recursive definition of factorial function
  - In mathematics, $n! = n \times (n-1)!$
  - In C or C++

```
int factorial ( const int n )
{
    return ( n ? n * factorial ( n - 1 ) : 1 );
}
```

  - The same computation could be performed by the following iterative code

```
for ( t = 1, i = 1; i <= n; t *= i++ );
```

- Recursion allows us to express complex algorithms in compact form
  - We do not sacrifice efficiency in terms of writing the code but the machine has to do a lot of work behind our back that may affect performance
  - The extra work involves maintaining the stack, and creating new local variables (look at the code above for factorial)
- Use mathematical induction to show that the recursive version of `factorial( n )` terminates

  Base case. If $n = 0$, the program terminates as a given

  Inductive hypothesis. Assume that the program terminates for all values of k such that $1 < n \leq k$

  Induction step. `factorial ( k + 1 )` is computed by the expression `( k + 1 ) * factorial ( k )`
  Since we know from induction hypothesis that `factorial ( k )` terminates, hence the proof □
- From mathematical induction, we can see that our recursive code must have two basic properties:
  1. It must explicitly solve a basic case
  2. Each recursive call must involve smaller values of the argument
     - Consider what happens when you try to use the recursive and iterative codes to compute $(-5)!$
       * Not a problem in terms of math as factorial of negative numbers is undefined
       * The recursive code gets into infinite recursion
       * So, we should have a statement such as

```
if ( n < 0 ) throw ( "Cannot take factorial of negative numbers" );
```

         as the first statement in the code
- Let us look at another recursive code

```
int puzzle ( const int n )
{
    if ( n == 1 ) return 1;
    return ( ( n % 2 ) ? puzzle ( 3 * n + 1 ) : puzzle ( n / 2 ) );
}
```

  - Cannot use mathematical induction to prove that the code will terminate as the recursive call may operate on a larger value than the one it started with
  - Try the code with `puzzle ( 3 )`

- Euclid's algorithm to find the greatest common divisor of two integers

```
int gcd ( const int m, const int n )
{
    return ( n ? gcd ( n, m % n ) : m );
}
```

- Evaluating prefix expressions

```
char *a;                  // Global character array
int i ( 0 );              // Global counter to point to an element in array
int eval()
{
    int x ( 0 );          // Local variable; unique for every instance
    for ( ; a[i] == ' '; i++ );        // Skip blanks
    if ( a[i] == '+' )                 // Add
    {
        i++;
        return ( eval() + eval() );
    }
    if ( a[i] == '*' )                 // Multiply
    {
        i++;
        return ( eval() * eval() );
    }
    while ( isdigit ( a[i] ) )         // Return the number by itself
        x = 10 * x + ( a[i++] - '0' );
    return ( x );
}
```

- Depth of recursion
  - Maximum degree of nesting of function calls over the course of computation
- Recursive functions for linked lists
  - Counting the number of elements in a list

    ```
    int count ( const link x )
    {
        return ( x ? 1 + count ( x -> next ) : 0 );
    }
    ```

  - Traversing a list

```
void traverse ( link h, void visit ( link ) )
{
    if ( h )
    {
        visit ( h );
        traverse ( h -> next, visit );
    }
}
```

– Traversing a list in reverse

```
void traverse_reverse ( link h, void visit ( link ) )
{
    if ( h )
    {
        traverse_reverse ( h -> next, visit );
        visit ( h );
    }
}
```

- Tail recursion

Divide and conquer

- Finding the maximum element in an array

```
item max ( const item * a, const int l, const int r )
{
    if ( l == r )
        return ( a[l] );
    int m = ( l + r ) / 2;
    item u = max ( a, l, m );
    item v = max ( a, m + 1, r );
    return ( u > v ? u : v );
}
```

Property 5 *A recursive function that divides a problem of size N into two independent (nonempty) parts that it solves recursively calls itself less than N times.*

If the parts are one of size k and one of size $N-k$, then the total number of recursive function calls that we use is

$$T_N = T_k + T_{N-k} + 1, \qquad \text{for } N \geq 1 \text{ with } T_1 = 0$$

The solution is: $T_N = N - 1$
Proof by induction:

Base case. $N = 1$ implies $T_1 = 0$ which is given.

Inductive hypothesis. Assume true for all values of i such that $1 \leq N \leq i$

Induction step. Prove for T_{i+1}, using the fact that $k \geq 1$

$$\begin{aligned} T_{i+1} &= T_k + T_{i+1-k} + 1 \\ &= (k-1) + (i+1-k-1) + 1 \\ &= i \end{aligned}$$

- Since the program `max()` does constant amount of work on each function call, its total running time is linear (recall the recurrence $T_n = 2T_{n/2}$); however some divide and conquer algorithms may require more work on each function call
- Other divide and conquer algorithms may require less work because of division (case in point: binary search), or may even require more work possibly to assemble the results (mergesort)
- Towers of Hanoi
  - Three pegs and n disks that fit onto the pegs, with all disk being of different size
  - Initially the disks are on one peg with the largest at the bottom and arranged by size so that the smallest is at the top
  - Since you do not have a life, you can move the disks to a different peg while retaining the order, with the following conditions
    1. Only one disk may be shifted at a time
    2. No disk may be placed on top of a smaller one at any time
  - To move N disks
    * Move top $N-1$ disks to the peg on left
    * Move the last disk to the peg on right
    * Move the $N-1$ disks one more peg to the left
  - Recursive code to solve the problem

```
// Positive peg means move to right of current peg,
// negative peg means move to left of current peg

void hanoi ( const int disk, const int peg )
{
    if ( disk )
    {
        hanoi ( disk - 1, -peg );
        shift ( disk, peg );
        hanoi ( disk - 1, -peg );
    }
}
```

  - For five disks (numbered 0 to 4) and three pegs, the code is called by

```
hanoi ( 4, +1 )
```

Property 6 *The recursive divide-and-conquer algorithm for the towers of Hanoi problem produces a solution that has $2^n - 1$ moves, for n disks*

From the code, the recurrence is given by:

$$T_n = 2T_{n-1} + 1, \qquad \text{for } N \geq 2 \text{ with } T_1 = 1$$

Proof by induction:

Base case. $T_1 = 2^1 - 1 = 1$

Induction hypothesis . Assume that the expression is satisfied for all values of n such that $1 \leq n \leq k$ for some constant k

Induction step. Test for $k+1$

$$\begin{aligned} T_{k+1} &= 2T_k + 1 \\ &= 2(2^k - 1) + 1 \\ &= 2^{k+1} - 2 + 1 \\ &= 2^{k+1} - 1 \end{aligned}$$

□

- Drawing a ruler
  - Each inch on a ruler is divided into two 1/2 inch parts, separated by a bar whose size is proportional to the resolution indicated by it; the mark for 1/2 inch is twice as high as the mark for 1/4 inch
  - The code is given as:

```
void rule ( const int l, const int r, const int ht )
{
    if ( ht > 0 )                       // Height of mark is positive
    {
        int m ( ( l + r ) / 2 );
        rule ( l, m, ht - 1 );          // Mark in left half
        mark ( m, ht );
        rule ( m, r, ht - 1 );          // Mark in right half
    }
}
```

  - The code seems to be like in-order traversal of binary tree
  - Equivalent iterative code for the same

```
void rule ( const int l, const int r, const int ht )
{
    for ( int t = 1, j = 1; t <=h; j *= 2, t++ )
        for ( int i ( 0 ); l + j + i <= r; i += 2*j )
            mark ( l + j + i, t );
}
```

Dynamic programming

- Divide-and-conquer partitions a problem into independent subproblems
- Dynamic programming is required to take into account the fact that the problems may not be partitioned into *independent* subproblems
  - Direct recursive implementation can require too much time
- Computing the nth Fibonacci number

```
int fib ( const int n )
{
    if ( n <= 0 ) return ( 0 );
    if ( n == 1 ) return ( 1 );
    return ( fib ( n - 1 ) + fib ( n - 2 ) );
}
```

  - This code is extremely inefficient; why?
  - How does this function differ from the `eval()` function?
- An efficient code using an array of size n

```
int fib ( const int n )
{
    if ( n <= 0 ) return ( 0 );
    if ( n == 1 ) return ( 1 );
    int * a = new int[n+1];
    a[0] = 0;
```

```
    a[1] = 1;
    for ( int i ( 2 ); i <= n; i++ )
        a[i] = a[i-1] + a[i-2];
    int tmp ( a[n] );
    delete[] a;
    return tmp;
}
```

- If we want to save space as well, the following code is more appropriate

```
int fib ( const int n )
{
    if ( n <= 0 ) return ( 0 );
    if ( n == 1 ) return ( 1 );
    int f0 ( 0 ), f1 ( 1 ), f;
    for ( int i ( 2 ); i <= n; i++ )
    {
        f = f1 + f0;
        f0 = f1;
        f1 = f;
    }
    return f;
}
```

  The above solution is known as *bottom-up dynamic programming* – we compute the smallest values first and build the solution using the solution to smaller problems; most of the real dynamic programming situations refer to *top-down dynamic programming* (also known as *memoization*) as you will see next in knapsack problem and in your homework assignment on traveling salesman problem

- Knapsack problem
  - Given n objects with the weight of ith object as w_i and a knapsack with a capacity m
  - If a fraction x_i, $0 \leq x_i \leq 1$, of object i is placed into the knapsack, then a profit of $p_i x_i$ is earned
  - Objective is to obtain a filling of the knapsack that maximizes the total profit earned, under the constraint of the capacity of the knapsack
  - Formal definition of the problem:

    Definition 1 *Maximize* $\sum_{1 \leq i \leq n} p_i x_i$ *such that* $\sum_{1 \leq i \leq n} w_i x_i \leq m$, *and* $\forall i, p_i > 0$ *and* $w_i > 0$
  - A feasible solution is any set $(x_1, \ldots, x_n)$ satisfying the above constraints; an optimal solution is one for which the expression $\sum_{1 \leq i \leq n} p_i x_i$ is maximized
  - Example of the problem: $n = 3$, $m = 20$, $p = (25, 24, 15)$, and $w = (18, 15, 10)$; four feasible solutions are given by:

(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum p_i x_i$
(1/2, 1/3, 1/4)	16.5	24.25
(1, 2/15, 0)	20.0	28.20
(0, 2/3, 1)	20.0	31.00
(0, 1, 1/2)	20.0	31.50

  - Greedy solution
  - Recursive solution
    - Each time you choose an item, you assume that you can optimally find a solution to pack the rest of the knapsack

```
struct item
{
    int   size;
    int   val;
};

int knapsack ( const int capacity )
{
    int t;
    // N is the number of item types
    for ( int i ( 0 ), int max ( 0 ); i < N; i++ )
        if ( ( int space = capacity - items[i].size ) >= 0 )
            if ( ( t = knapsack ( space ) + items[i].val ) > max )
                max = t;
    return ( max );
}
```

– Dynamic programming solution

```
int knapsack ( const int capacity )
{
    int maxi, t;
    if ( maxknown[capacity] )
        return ( maxknown[capacity] );
    for ( int i ( 0 ), int max ( 0 ); i < N; i++ )
        if ( ( int space = capacity - items[i].size ) >= 0 )
            if ( ( t = knapsack ( space ) + items[i].val ) > max )
            {
                max = t;
                maxi = i;
            }
    maxknown[capacity] = max;
    itemknown[capacity] = items[maxi];
    return ( max );
}
```

- Dynamic programming eliminates all recomputation in any recursive program, by saving intermediate values in variables whose scope is designed to allow them to be visible in more than one local context

  Property 7 *Dynamic programming reduced the running time of a recursive function to be at most the time required to evaluate the function for all arguments less than or equal to the given argument, treating the cost of a recursive call as constant.*

- Property 7 implies that the running time for the knapsack problem is $O(NM)$
- Dynamic programming becomes ineffective when the number of possible function values that may be needed is so high that we cannot afford to save or precompute all of them

Graphs

Directed graph or **Digraph** $G = (V, E)$
V – Finite set of vertices – **vertex set**
E – Binary relation on V – **edge set**

Vertex or **Node** is a simple object that can have a name and can carry other associated information

Self-loop is an edge from a vertex to itself

Undirected graph $G = (V, E)$
Edge set E consists of *unordered* pairs of vertices
Edge is a set $\{u, v\}$ where $u, v \in E$ and $u \neq v$
No self-loops allowed in undirected graphs

Edge (u, v) is a connection between two vertices u and v

- incident from (leaves) vertex u
- incident to (enters) vertex v
- vertex v is *adjacent to* vertex u
- vertex u is *adjacent from* vertex v

Degree of vertex v

- number of edges incident on the vertex
- *Out-degree* – number of edges leaving the vertex
- *In-degree* – number of edges entering the vertex

Path of *length* k from vertex u to vertex u' is sequence

$$\langle u_0, u_1, u_2, \ldots, u_k \rangle$$

where $u = u_0$ and $u' = u_k$ and $(u_{i-1}, u_i) \in E$ for $i = 1, 2, \ldots, k$
Length of the path = number of edges in the path
Path *contains*

- vertices $u_0, u_1, u_2, \ldots, u_k$
- edges $(u_0, u_1), (u_1, u_2), \ldots, (u_{k-1}, u_k)$

v is *reachable* from u if there is a path from u to v
Simple path – all the vertices are distinct
subpath – contiguous subsequence of vertices in the path
Cycle – Path $u_0, u_1, u_2, \ldots, u_k$ in a directed graph such that $u_0 = u_k$ and there is at least one edge in the path
Simple cycle – Cycle with only two common vertices, other vertices are distinct
Acyclic graph – has no cycle

Connectivity in graphs

Connected graph Every pair of vertices is connected by an edge

Connected components Equivalence class of vertices under the "is reachable from" relation

Strongly connected directed graph – Every two vertices are reachable from each other

Strongly connected components – equivalence classes of vertices under the "mutually reachable from" relation

Isomorphic graph Two graphs $G = (V, E)$ and $G' = (V', E')$ are isomorphic if there exists a bijection $f : V \to V'$ such that $(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$.

Subgraph A graph $G' = (V', E')$ is a subgraph of $(G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$

Induced subgraph Given a set $V' \subseteq V$, the subgraph of G induced by V' is the graph $G' = (V', E')$, where

$$E' = \{(u, v) \in E \ : \ u, v \in V'$$

Directed version of an undirected graph Each undirected edge (u, v) is replaced by two directed edges (u, v) and (v, u).

Neighbor of a vertex

Complete graph Undirected graph in which every pair of vertices is adjacent

Bipartite graph An undirected graph $G = (V, E)$ in which V can be partitioned into two sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$, or $u \in V_2$ and $v \in V_1$.

Multigraph An undirected graph that can have both multiple edges between vertices and self-loops.

Trees

- A mathematical abstraction that play a central role in the design and analysis of algorithms
  - Used to describe dynamic properties of algorithms
  - Explicit data structures that are concrete realization of trees
- Free tree, or simply tree
  - Nonempty collection of vertices and edges
  - Connected, acyclic, undirected graph
  - "Free" often omitted
  - Disconnected, acyclic, undirected graph will be called a **forest**
- Let $G = (V, E)$ be an undirected graph. The following statements are equivalent
  - G is a free tree
  - Any two vertices in G are connected by a unique simple path
  - G is connected, but if any edge is removed from E, the resulting graph is disconnected
  - G is connected, and $|E| = |V| - 1$
  - G is acyclic, and $|E| = |V| - 1$
  - G is acyclic, but if any edge is added to E, the resulting graph contains a cycle
- Rooted and ordered trees
  - An ordered tree in which one of the vertices is distinguished from others, and is called the $root$ r
  - Node of the tree
  - Ancestor of x – Any node y on the unique path from r to x
  - Descendant of y – Any node x such that y is ancestor of x
  - Proper ancestor
  - Proper descendant
  - Subtree rooted at x is the tree induced by descendants of x
  - Parent
  - Child
  - Sibling
  - External node or Leaf
  - Internal node
  - Degree of a node x in a rooted tree – number of children of x
  - Depth of a node x – Length of the path from root to node x
  - Height of the tree – Largest depth of any node

  - Ordered tree – Children of each node are ordered

- Binary and positional trees
  - Structure defined on a finite set of nodes that either
    - contains no nodes
    - is comprised of three disjoint sets of nodes: a *root* node, a binary tree called its *left subtree*, and a binary tree called its *right subtree*
  - Left child
  - Right child
  - Full binary tree – Each node is either a leaf or has exactly two children
  - Complete tree

Mathematical properties of binary trees

Property 8 *A binary tree with N internal nodes has $N + 1$ external nodes.*

Proof by induction:

Base case. Let there be no internal node, or $N = 0$. Such a tree has one external node, hence the proof for base case.

Inductive hypothesis. Assume that the property holds for all values of N such that $0 \leq N \leq k$.

Induction step. Consider a tree with $k+1$ internal nodes. Such a tree has L internal nodes in the left subtree and $k-L$ internal nodes in the right subtree, with the root providing for the other internal node. By inductive hypothesis, the left subtree will have $L + 1$ external nodes and the right subtree will have $k - L + 1$ external nodes. The total number of external nodes is: $(L + 1) + (k - L + 1)$ which reduces to $k + 2$. Hence the proof.

Property 9 *A binary tree with N internal nodes has $2N$ links: $N - 1$ links to the internal nodes and $N + 1$ links to the external nodes.*

Every internal node except the root has a unique parent, hence the $N - 1$ links to internal nodes; every external node has one link to its unique parent, hence $N + 1$ links to external nodes.

Level of a node in a tree is one higher than the level of its parent, with the root at level 0

Height of a tree is the maximum of the levels of the tree's nodes

Path length of a tree is the sum of the levels of all the tree's nodes

Internal path length of a binary tree is the sum of the levels of all the tree's internal nodes

External path length of a binary tree is the sum of the levels of all the tree's external nodes

Property 10 *The external path length of any binary tree with N internal nodes is $2N$ greater than the internal path length.*

Property 11 *The height of a binary tree with N internal nodes is at least $\lg N$ and at most $N - 1$.*

Property 12 *The internal path length of a binary tree with N internal nodes is at least $N \lg(N/4)$ and at most $\frac{N(N-1)}{2}$.*

Tree traversal

- Preorder traversal

```
void preorder ( node * r )
{
    if ( ! r ) return;
    visit ( r );
    preorder ( r->left );
    preorder ( r->right );
}
```

- Postorder traversal

```
void postorder ( node * r )
{
    if ( ! r ) return;
    postorder ( r->left );
    postorder ( r->right );
    visit ( r );
}
```

- Inorder traversal

```
void inorder ( node * r )
{
    if ( ! r ) return;
    inorder ( r->left );
    visit ( r );
    inorder ( r->right );
}
```

- Level order traversal

Recursive binary-tree algorithms

- Computing number of nodes in a tree

```
int count ( const tree root ) const
{
    return ( root ? count ( root->left ) + count ( root->right ) + 1 : 0 );
}
```

- Computing the height of a binary tree

```
int height ( const tree root ) const
{
    return ( root ? 1 + max( height(root->left), height(root->right) ) : 0 );
}
```