The Tower of Hanoi

Recursion Solution

Recursive Thinking: ignore everything but the bottom disk.

Recursive Function

Hanoi (n, src, dest, temp):
  If (n > 0) then
    Hanoi (n - 1, src, temp, dest)
    Move disk n from src to dest
    Hanoi (n - 1, temp, dest, src)

Recursive Thinking

Examples: Recursive Definitions of Mathematical Formulas
- Factorial
- Powers
- Greatest common divisor
- Practical example: Directory Search!

Why Recursion?

Time Complexity

T(n) = 2T(n - 1) + 1
= 2^2T(n - 2) + 2 + 1
= ... 
= 2^iT(n - i) + 2^{i-1} + 2^{i-2} + ... + 1
= ... 
= 2^{n-1}T(1) + 2^{n-2} + ... + 1
= 2^{n-1} + 2^{n-2} + ... + 1
= (2^n - 1)/(2 - 1) = 2^n - 1

Recursion splits a problem:
- Into one or more simpler versions of itself
- E.g.
  n! = n * (n-1)!
  x^0 = x * x^{n-1}
Simple Example: Factorial

- Recall from math that factorial is defined as
  
  \[ 0! = 1 \]
  \[ n! = n \times (n-1)! \]

  where \( n \) is not negative.

Implementation

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

A simple definition of recursion

- Recursion simply means a function that calls itself.
- In order to keep the recursion from going on forever, you must make sure you hit a termination condition called the base case.

General Recursive Design Strategy

- Identify the base case(s) (for direct solution).
- Devise a problem splitting strategy.
  - Subproblems must be smaller.
  - Subproblems must work towards a base case.
- Devise a solution combining strategy.

Outline of a Recursive Function

```
if (answer is known)
    provide the answer
else
    make a recursive call to solve a smaller version of the same problem
```

Recursive Definitions of Mathematical Formulas

- Mathematicians often use recursive definitions.
- These lead naturally to recursive algorithms.
- Examples include:
  - Factorial
  - Powers
  - Greatest common divisor
### Example: Factorial

```java
int factorial(int N) {
    if (N == 0)
        return 1;  // Base Case
    return N*factorial(N-1);  // Recursive Case
}
```

### How Recursion Works

- A recursive function call is handled like any other function call.
- Each recursive call has an activation record on the stack.
  - Stores values of parameters and local variables.
- When base case is reached return is made to previous call - the recursion “unwinds”.

### Example: Trace Factorial(3)

1. **n! = n * (n-1)!**
2. **0! = 1**
3. If a recursive function never reaches its base case, a stack overflow error occurs.

### Recursive Definitions: Power

- $x^0 = 1$
- $x^n = x \cdot x^{n-1}$

```java
public static double power(double x, int n) {
    if (n == 0)
        return 1;
    else
        return x * power(x, n-1);
}
```

### Greatest Common Divisor

Definition of \( \text{gcd}(m, n) \), for integers \( m > n > 0 \):
1. \( \text{gcd}(m, n) = n \), if \( n \) divides \( m \) evenly
2. \( \text{gcd}(m, n) = \text{gcd}(n, m \mod n) \), otherwise

```java
public static int gcd (int m, int n) {
    if (m < n)
        return gcd(n, m);
    else if (m % n == 0)
        return n;
    else
        return gcd(n, m % n);
}
```

### Recursion Vs. Iteration

- Recursion and iteration are **similar**.
- **Iteration:**
  - Loop repetition test determines whether to exit.
- **Recursion:**
  - Condition tests for a base case.
  - Can always write iterative solution to a problem solved recursively, but:
    - Recursive code often simpler than iterative.
    - Thus easier to write, read, and debug.
Factorial (n) - iterative

Factorial (n) = n * (n-1) * (n-2) * ... * 1     for  n > 0

Factorial (0) = 1

```c
int IterFactorial (int n)
{
    int fact =1;
    for (int i = 1; i <= n; i++)
        fact = fact * i;
    return fact;
}
```

Efficiency of Recursion

- Recursive method often **slower** than iterative;  
  **why?**
  - Overhead for loop repetition smaller than overhead for call and return.

Recursive Data Structures

- Just as we have recursive **algorithms**, we can have recursive **data structures**.
- Like algorithms, a recursive data structure has:
  - A **base case**, a simple data structure, or null
  - A recursive case: includes a **smaller** instance of the **same data structure**.

Recursive Data Structures

- Computer scientists often **define** data structures recursively.
  - Trees are defined recursively.
  - Linked lists can also be defined recursively.
  - Recursive methods are very natural in processing recursive data structures.

Recursive Definition of Linked List

A **linked list** is either:
- An **empty list** ← the base case, or
- A head node, consisting of:
  - A **data item** and
  - A reference to a **linked list** (rest of list)

Divide-and-Conquer

- The most-well known algorithm design strategy:
  - Divide instance of problem into two or more smaller instances
  - Solve smaller instances **recursively**
  - Obtain solution to original (larger) instance by combining these solutions
Divide-and-Conquer Technique

It generally leads to a recursive algorithm!

Divide-and-Conquer Examples

- Binary search (?)
- Sorting: mergesort and quicksort
- Binary tree traversals

Binary Search

Very efficient algorithm for searching in a sorted array:

$K$ vs $A[0] \ldots A[m] \ldots A[n-1]$  

If $K = A[m]$, stop (successful search); otherwise, continue searching by the same method in $A[0..m-1]$ if $K < A[m]$ and in $A[m+1..n-1]$ if $K > A[m]$

Time efficiency:

- worst-case recurrence: $T(n) = 1 + T(\lceil n/2 \rceil)$, $T(1) = 1$
- solution: $T(n) = \lceil \log_2(n+1) \rceil$

Optimal for searching a sorted array

Limitations: must be a sorted array (not linked list)

Merge-Sort

- Merge-sort on an input sequence $S$ with $n$ elements consists of three steps:
  - Divide: partition $S$ into two sequences $S_1$ and $S_2$ of about $n/2$ elements each
  - Recur: recursively sort $S_1$ and $S_2$
  - Conquer: merge $S_1$ and $S_2$ into a unique sorted sequence

Pseudocode of Mergesort

```
ALGORITHM Mergesort($A[0..n-1]$)
// Sorts array $A[0..n-1]$ by recursive mergesort
// Input: An array $A[0..n-1]$ of orderable elements
// Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lceil n/2 \rceil..n-1]$ to $C[0..\lfloor n/2 \rfloor - 1]$
    Mergesort($B[0..\lfloor n/2 \rfloor - 1]$)
    Mergesort($C[0..\lfloor n/2 \rfloor - 1]$)
    Merge($B$, $C$, A)
```
### Pseudocode of Merge

**Algorithm** Merge(B[0, p - 1], C[0, q - 1], A[p + q - 1])

//Merges two sorted arrays into one sorted array
//Input: Arrays B[0, p - 1] and C[0, q - 1] both sorted
//Output: Sorted array A[p + q - 1] of the elements of B and C

\[
\begin{align*}
& i = 0, \quad j = 0, \quad k = 0 \\
& \text{while } i < p \text{ and } j < q \text{ do} \\
& \quad \text{if } B[i] \leq C[j] \\
& \qquad A[k] \leftarrow B[i], \quad i \leftarrow i + 1 \\
& \quad \text{else} \quad A[k] \leftarrow C[j], \quad j \leftarrow j + 1 \\
& \quad k \leftarrow k + 1 \\
& \quad \text{if } i = p \\
& \qquad \text{copy } C[j, q - 1] \text{ to } A[k, p + q - 1] \\
& \quad \text{else} \quad \text{copy } B[i, p - 1] \text{ to } A[k, p + q - 1]
\end{align*}
\]

**Time complexity:** \(O(p+q) = O(n)\) comparisons

### Analysis of Mergesort

- **Merge sort has an average and worst-case performance:** \(O(n \log n)\)

\[
T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + \Theta(n)
\]

\[
T(1) = 1
\]

- **Space requirement:** \(O(n)\)

### Mergesort Example

```
8  3  2  9  7  1  5  4
8  3  2  9
7  1  5  4
8  3
2  9
7  1
5  4
8
3
2
9
7
1
5
4
3  8
2  9
1  7
4  5
2  3  8  9
1  4  5  7
1  2  3  4  5  7  8  9
```

### Time Complexity

\[
T(n) = 2 \cdot T\left(\frac{n}{2}\right) + cn
\]

\[
= 2 \cdot 2 \cdot T\left(\frac{n}{4}\right) + c \frac{n}{2} + cn
\]

\[
= 4 \cdot T\left(\frac{n}{4}\right) + 2 \cdot cn
\]

\[
= 4 \cdot 2 \cdot T\left(\frac{n}{8}\right) + c \frac{n}{4} + 2 \cdot cn = 8 \cdot T\left(\frac{n}{8}\right) + 3 \cdot cn
\]

\[
\vdots
\]

\[
= 2^k \cdot T\left(\frac{n}{2^k}\right) + k \cdot cn
\]

\[
= 2^k \cdot T\left(\frac{n}{2^k}\right) + kn \cdot cn \quad (2^k = n \Rightarrow k = \log_2 n)
\]

\[
\vdots
\]

\[
= n \cdot T(1) + cn \cdot \Theta(n)
\]

\[
= \Theta(n^2)
\]