Recursion
What is recursion

Recursion is a process for solving problems by subdividing a larger problem into smaller cases of the problem itself and then solving the smaller, more trivial parts.

• A mathematical example: the definition of Fibonacci sequence
  \[ F(n) = \begin{cases} 
  1, & n = 1, 2 \\
  F(n - 1) + F(n - 2), & \text{others} 
\end{cases} \]

• A computer science example: recursive function
  • Can be used to solve a wide range of problems.
  • Not always the most efficient.
Recursive Function

• Recall the preorder traversal on a binary tree.

```python
def preorderTrav( subtree ):
    if subtree is not None:
        print( subtree.data )
        preorderTrav( subtree.left )
        preorderTrav( subtree.right )
```

• Other examples

```python
def printRev( n ):
    if n > 0:
        print( n )
        printRev( n-1 )
```

```python
def printInc( n ):
    if n > 0:
        printInc( n-1 )
        print( n )
```
Recursive Flow

What is the output of each function?
Recursive Function

Function “a” is recursive function

Is function “a” recursive function

Is function “b” recursive function
Recursive Function

<table>
<thead>
<tr>
<th>Direct Recursive Function</th>
<th>Indirect Recursive Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>A function is called by itself directly.</td>
<td>A function is called not by itself but by another function that it called (either directly or indirectly).</td>
</tr>
</tbody>
</table>

**Remark:**
Indirect recursive function is also called mutual recursive function because it is symmetric (recall the function a and b in last page)
Three Rules of Recursion

1. A recursive solution must contain a base case.
2. A recursive solution must contain a recursive case.
3. A recursive solution must make progress toward the base case.

Remark:
• 1 & 3 guarantee the recursion can stop.
• 2 divides the problem into smaller problems (divide and conquer).
Recursive Solutions

• A recursive solution subdivides a problem into smaller versions of itself.
  • Must consist of a data set or a term that can be divided into smaller sets or a smaller term.
  • The base case is the terminating case and represents the smallest subdivision.
  • If a solution does not make progress toward the base case, the recursion will never end.
How to Use Recursion Solve Problem

• Whether this problem is solvable by recursion?
  Can this problem be divided into smaller but the same problem?

• Steps to solve this problem.
  1. Find based case
  2. Find recursion case
Example: Use Recursion to Solve Factorials

• Factorial of $n$ is the product of $n$ and factorial of $n-1$. Therefore, recursion can solve this problem.

• Based base $n = 0$, $n! = 1$.

• Recursion case $n! = n(n-1)!$.

```python
1 # Compute n!
2 def fact( n ):
3     assert n >= 0, "Factorial not defined for negative values."
4     if n < 2 :
5         return 1
6     else :
7         return n * fact(n - 1)
```
Recursive Call Tree of Factorial: fact(5)

Note:
Function Factorial call itself only once. This is called single recursion.
Recursive Call Trees

- Each box represents a function call.
  - Function name.
  - Actual arguments used to invoke it.

- The directed edges indicate the flow of execution.
  - Solid edges indicate function calls.
  - Dashed edges indicate function returns.
  - Return values of a function shown on dashed lines.
Call Trees: Multiple Calls

• If a function makes multiple function calls, each call is indicated in the tree.

```python
main():
  y <-- foo(3)
  bar(2)

main()

foo(x):
  if x % 2 != 0:
    return 0
  else:
    return x + foo(x-1)

bar(n):
  if n > 0:
    print(n)
    bar(n-1)
```
Call Trees: Multiple Calls

- The edges are listed left to right in the order the calls are made.
Multiple Recursion

- Factorial is a single recursion.
- The Fibonacci sequence

\[ \text{fib}(n) = \begin{cases} 
  1, & n = 1, 2 \\
  \text{fib}(n - 1) + \text{fib}(n - 2), & \text{others}
\end{cases} \]

\[ 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \ldots \]

can be solved by multiple recursion.

```python
fib(n): //assuming n >= 1
    if n=1 or n=2:
        return 1
    else:
        return fib(n - 1)+fib(n-2)
```
Call Tree of Fibonacci Sequence: \( \text{fib}(6) \)
Recurrence Equation

• We can describe the running time of a recursive algorithm as a recurrence equation or recurrence.
  • describes the overall running time of a problem of size $n$ in terms of that of smaller inputs.
  • use mathematical tools to solve the recurrence and bound the running time.
Recurrence Equation

- Let $T(n)$ be the running time of a problem of size $n$.
- We divide the problem into $k$ subproblems, each of which has size $n/b$.
- Then,

$$ T(n) = \begin{cases} 
O(1) & \text{if } n < c \\
 kT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{otherwise}
\end{cases} $$

- if $n < c$ for some small enough constant $c$, then the problem is trivial and can be solved easily.
- $D(n)$: time spent to divide the problem into subproblems
- $C(n)$: time spent to combine the solutions to the subproblems into that of the original problem.
The Runtime Stack

```python
def main():
    y = fact(2)
main()
```

![Diagram showing the runtime stack](image)
Relationship between Recursion and Loop

• All recursion can be solved by stack + loop.
• Single recursion can be solved by loop without stack.
• Multiple recursion (cannot be reduced to a single recursion) can only be solved by stack.

A problem can be solved by stack. Can it be solved by recursion?

Hint:
Think about the limitation of the run-time stack.
Solve Fibonacci without Recursion

Fib is a multiple recursion. Why can it be solved by a loop without stack?
Applications

• Many applications can be solved using recursion.
  • Binary Search
  • Towers of Hanoi
  • Tic-Tac-Toe
Recursive Binary Search

• The binary search algorithm can be implemented recursively.
• The problem is expressed in smaller versions of itself.
  • At each split, the list is cut in half.
  • Continues until the target is located or the sub list contains no items.
Recursive Binary Search

```python
recBinarySearch(target, A, first, last):

    //if the sequence cannot be subdivided further, we are done.
    if first > last:
        return False

else:
    mid = (last + first) // 2
    if A[mid] == target:
        return True

    else if target < A[mid]:
        return recBinarySearch(target, A, first, mid-1)

    else:
        return recBinarySearch(target, A, mid+1, last)
```
Efficiency: Call Tree

• We can use a call tree.