Recursion
Recursion

• Recursion is a **fundamental concept** of Computer Science.
• It usually help us to write simple and elegant solutions to programming problems.
• You will learn to program recursively by working with many examples to develop your skills.
Recursion

• But I know this, from high school math! They just call it "induction" (μαθηματική επαγωγή)

• Goal: show $P(n)$ for all $n \geq n_0$

• Method
  – base case: show $P(n_0)$
  – induction hypothesis: assume $P(k-1)$
    (or even better: assume $P(n)$ for any $n < k$)
  – induction step: show $P(k)$

eg. show $n! > 2^n$ for all $n \geq 4$
Why induction works?

• Me:
  – I don't believe P(3) is true, I want a full (non-inductive) proof

• You:
  – ok, let me prove P(2) first
  – for P(2), I now need P(1)
  – P(1) this is the base case, I have proven it!
  – from P(1) I now construct a proof of P(2)
  – and from P(2) a proof of P(3)
Recursive Programs

• Same idea, but for computing!
• I want a function `DoSomething(int n)`
  (that works for all non-negative `n`)
• Base case:
  – if `n==0` compute it directly
• Inductive case:
  – assume `DoSomething(m)` works for `m < n`
  – so we can call `DoSomething(m)` to solve
    a smaller sub-problem (or many sub-problems!)
  – use its result to compute the return value
Example: Computing the Factorial

• Let us consider a simple program to compute the factorial $n!$ of $n$.

• An **iterative function** to do this is the following:

```c
int Factorial(int n)
{
    int i, f;

    f=1;
    for (i=2; i<=n; ++i) f*=i;
    return f;
}
```
Recursive Factorial

int Factorial(int n)
{
    if (n==1) {
        return 1;
    } else {
        return n*Factorial(n-1);
    }
}
Recursive Factorial

```
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

6 = 3!

```
n = 3
else
    return n * factorial(n - 1);
```

```
recursive call
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

```
n = 2
else
    return n * factorial(n - 1);
```

```
recursive call
int factorial(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * factorial(n - 1);
}
```

```
n = 1
else
    return 1;
```

```
return n * factorial(n - 1);
```

Data Structures and Programming Techniques
Another Example

• Let us consider a simple program to add up all the squares of integers from \( m \) to \( n \).

• An **iterative function** to do this is the following:

```c
int SumSquares(int m, int n)
{
    int i, sum;

    sum=0;
    for (i=m; i<=n; ++i) sum +=i*i;
    return sum;
}
```
Recursive Sum of Squares

```
int SumSquares(int m, int n)
{
    if (m<n) {
        return m*m + SumSquares(m+1, n);
    } else {
        return m*m;
    }
}
```
Comments

• In the case that the range \( m:n \) contains more than one number, the solution to the problem can be found by adding (a) the solution to the smaller subproblem of summing the squares in the range \( m+1:n \) and (b) the solution to the subproblem of finding the square of \( m \). (a) is then solved in the same way (recursion).

• We stop when we reach the base case that occurs when the range \( m:n \) contains just one number, in which case \( m==n \).

• This recursive solution can be called “going-up” recursion since the successive ranges are \( m+1:n \), \( m+2:n \) etc.
Going-Down Recursion

```c
int SumSquares(int m, int n)
{
    if (m<n) {
        return SumSquares(m, n-1) + n*n;
    } else {
        return n*n;
    }
}
```

Recursive call

Base case
Recursion Combining Two Half-Solutions

int SumSquares(int m, int n)
{
    int middle;

    if (m==n) {
        return m*m;
    } else {
        middle=(m+n)/2;
        return
            SumSquares(m,middle)+SumSquares(middle+1,n);
    }
}
Comments

• The recursion here says that the sum of the squares of the integers in the range \( m:n \) can be obtained by adding the sum of the squares of the left half range, \( m:\text{middle}, \) to the sum of the squares of the right half range, \( \text{middle}+1:n. \)

• We stop when we reach the base case that occurs when the range contains just one number, in which case \( m==n. \)

• The middle is computed by using integer division (operator \( / \)) which keeps the quotient and throws away the remainder.
Call Trees and Traces

• We can depict graphically the behaviour of recursive programs by drawing call trees or traces.
Annotated Call Trees

SumSquares(5,10)

SumSquares(5,7)

SumSquares(5,6)

SumSquares(5,5)

SumSquares(6,6)

SumSquares(8,7)

SumSquares(8,9)

SumSquares(8,8)

SumSquares(9,9)

SumSquares(10,10)
Traces

\[ \text{SumSquares}(5,10) = \text{SumSquares}(5,7) + \text{SumSquares}(8,10) = \]
\[ = \text{SumSquares}(5,6) + \text{SumSquares}(7,7) + \text{SumSquares}(8,9) + \text{SumSquares}(10,10) \]
\[ = \text{SumSquares}(5,5) + \text{SumSquares}(6,6) + \text{SumSquares}(7,7) + \text{SumSquares}(8,8) + \text{SumSquares}(9,9) + \text{SumSquares}(10,10) \]
\[ = ((25+36)+49) + ((64+81)+100) \]
\[ = (61+49) + (145+100) \]
\[ = (110+245) \]
\[ = 355 \]
Computing the Factorial (cont’d)

• This is a “going-down” recursion.
• Can you do it “going-up” for factorial?
• Can you combine two half-solutions?
• The above tasks do not appear to be easy.

```c
int Factorial(int n)
{
    if (n==1) {
        return 1;
    } else {
        return n*Factorial(n-1);
    }
}
```
Computing the Factorial (cont’d)

• It is easier to first write a function \texttt{Product(m,n)} which \texttt{multiplies} together the numbers in the range \texttt{m:n}.

• Then \texttt{Factorial(n)=Product(1,n)}.
Multiplying \( m : n \) Together Using Half-Ranges

int Product(int m, int n)
{
    int middle;

    if (m==n) {
        return m;
    } else {
        middle=(m+n)/2;
        return Product(m,middle)*Product(middle+1,n);
    }
}
Reversing Linked Lists

• Let us now consider the problem of reversing a linked list \( L \).
• The type \( \text{NodeType} \) has been defined in the previous lecture as follows:

```c
typedef char AirportCode[4];
typedef struct NodeTag {
    AirportCode Airport;
    struct NodeTag *Link;
} NodeType;
```
Reversing a List Iteratively

- An **iterative function for reversing a list** is the following:

```c
void Reverse(NodeType **L)
{
    NodeType *Prev, *Cur, *TempCur;

    Cur = *L;
    Prev = NULL;
    while (Cur != NULL) {
        TempCur = Cur;
        Cur = Cur->Link;
        TempCur->Link = Prev;
        Prev = TempCur;
    }
    *L = Prev;
}
```
Question

• If in our main program we have a list with a pointer $A$ to its first node, how do we call the previous function?
Answer

• We should make the following call:
  \texttt{Reverse(&A)}
Reversing Linked Lists (cont’d)

• Goal: recursive solution for the problem of reversing a list $L$

• A little trick:
  – partition the list in 2 lists: its $\text{head Head}(L)$ and $\text{tail Tail}(L)$
  – reverse the $\text{Tail}(L)$
  – concatenate $\text{Head}(L)$ in the result
Head and Tail of a List

- Let $L$ be a list. $\text{Head}(L)$ is a list containing the first node of $L$. $\text{Tail}(L)$ is a list consisting of $L$’s second and succeeding nodes.
- If $L == \text{NULL}$ then $\text{Head}(L)$ and $\text{Tail}(L)$ are not defined.
- If $L$ consists of a single node then $\text{Head}(L)$ is the list that contains that node and $\text{Tail}(L)$ is $\text{NULL}$.
Example

• Let \( L = (SAN, ORD, BRU, DUS) \). Then
  Head\((L)\) = (SAN) and
  Tail\((L)\) = (ORD, BRU, DUS).
Reversing Linked Lists (cont’d)

NodeType *Reverse(NodeType *L)
{
    NodeType *Head, *Tail;

    if (L == NULL) {
        return NULL;
    } else {
        Partition(L, &Head, &Tail);
        return Concat(Reverse(Tail), Head);
    }
}
void Partition(NodeType *L, NodeType **Head, NodeType **Tail)
{
    if (L != NULL) {
        *Tail=L->Link;
        *Head=L;
        (*Head)->Link=NULL;
    }
}

Reversing Linked Lists (cont’d)

NodeType *Concat(NodeType *L1, NodeType *L2)
{
    NodeType *N;

    if (L1 == NULL) {
        return L2;
    } else {
        N=L1;
        while (N->Link != NULL) N=N->Link;
        N->Link=L2;
        return L1;
    }
}

Infinite Regress

• Let us consider again the recursive factorial function:
  int Factorial(int n);
  {
      if (n==1) {
          return 1;
      } else {
          return n*Factorial(n-1);
      }
  }

• What happens if we call Factorial(0)?
Infinite Regress (cont’d)

Factorial(0) = 0 \times \text{Factorial}(-1) \\
= 0 \times (-1) \times \text{Factorial}(-2) \\
= 0 \times (-1) \times \text{Factorial}(-3) \\

and so on, in an infinite regress.

When we execute this function call, we get “Segmentation fault (core dumped)”.
The Towers of Hanoi
The Towers of Hanoi (cont’d)

- To Move 4 disks from Peg 1 to Peg 3:
  - Move 3 disks from Peg 1 to Peg 2
  - Move 1 disk from Peg 1 to Peg 3
  - Move 3 disks from Peg 2 to Peg 3
Move 3 Disks from Peg 1 to Peg 2
Move 1 Disk from Peg 1 to Peg 3
Move 3 Disks from Peg 2 to Peg 3
Done!
A Recursive Solution

```c
void MoveTowers(int n, int start, int finish, int spare)
{
    if (n==1){
        printf("Move a disk from peg %ld to peg %ld\n", start, finish);
    } else {
        MoveTowers(n-1, start, spare, finish);
        printf("Move a disk from peg %ld to peg %ld\n", start, finish);
        MoveTowers(n-1, spare, finish, start);
    }
}
```
Analysis

• Let us now compute the number of moves $L(n)$ that we need as a function of the number of disks $n$:

\[ L(1) = 1 \]
\[ L(n) = L(n-1) + 1 + L(n-1) = 2L(n-1) + 1, \quad n > 1 \]

The above are called recurrence relations. They can be solved to give:

\[ L(n) = 2^n - 1 \]
Analysis (cont’d)

• Techniques for solving recurrence relations are taught in the Algorithms and Complexity course.

• The running time of algorithm MoveTowers is exponential in the size of the input.
Readings

• T. A. Standish. *Data structures, algorithms and software principles in C.* Chapter 3.

• (προαιρετικά) R. Sedgewick. *Αλγόριθμοι σε C.* Κεφ. 5.1 και 5.2.