Recursion

CSE 2320 – Algorithms and Data Structures
University of Texas at Arlington

adapted from slides provided by Vassilis Athitsos
Recursion

• Recursion is a fundamental concept in computer science.
  – In all recursive concepts, there are one or more **base cases**.

• **Recursive [math] functions**: functions that call themselves.
  – Example:
    • Base case:

• **Recursive data types**: data types that are defined using references to themselves.
  – Example:
    • Base case:

• **Recursive algorithms**: algorithms that solve a problem by solving one or more smaller instances of the same problem.
  – Example:
    • Base case:
Recursion

- Recursion is a fundamental concept in computer science.
  - In all recursive concepts, there are one or more **base cases**.

- **Recursive [math] functions**: functions that call themselves.
  - Example: $N! = N \times (N-1)!$
    - Base case: $N = 0$
    - Example: $A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1,A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0 \end{cases}$
  - Used as benchmark for compiler optimization for recursion.

- **Recursive data types**: data types that are defined using references to themselves.
  - Example: Nodes in the implementation of linked lists.
    - Base case: NULL

- **Recursive algorithms**: algorithms that solve a problem by solving one or more smaller instances of the same problem.
  - Example: binary search, mergesort
    - Base case: one or no element in collection.
Recursive Algorithms

- **Recursive algorithms**: solve a problem by solving one or more smaller instances of the same problem
  - It can easily be implemented using recursive functions, and
  - There is also a corresponding algorithm that solves it without recursion.

- Example of a **recursive function**: the factorial.
  - Recursive definition?
  - Non-recursive definition?
  - Try it yourself! Start with the recursive one if you can.
Factorial

**Recursive Definition:**

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

**Iterative Definition:**

```c
int factorial_iter(int N) {
    int result = 1;
    int i;
    for (i = 2; i <= N; i++)
        result *= i;
    return result;
}
```

My terminology:
- When talking about the algorithm or paper definition: base case, recursive case
- When talking about the implementation: base step, recursive step
- I will probably end up mixing these terms.

- The **recursive call** is the actual (self) function call. E.g. `factorial(N-1)` above.
Factorial

Recursive Definition:

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

How is factorial(3) evaluated?
Analyzing a Recursive Program

- Two main questions:
  - Does it always **terminate**?
  - Does it always **compute the right result**?
- Both questions answered by induction.
- Example: does the factorial function on the right always compute the right result?
- Proof: by induction.

Recursive Definition:

```c
int factorial(int N) {
    if (N == 0) return 1;
    return N*factorial(N-1);
}
```
Analyzing a Recursive Program –

Factorial computes correct result

• Proof: by induction.

• Step 1: (the base case)
  – For N = 0, factorial(0) returns 1, which is correct.

• Step 2: (using the inductive hypothesis)
  – Suppose that factorial(N) returns the right result for N = K, where K is an integer >= 0.
    ( factorial(K) = K! )
  – Then, for N = K+1, factorial(N) returns:
    N * factorial(N-1) = (K+1) * factorial(K) = (K+1) * K! = (K+1)! = N!
  – Thus, for N = K+1, factorial(N) also returns the correct result.

• Thus, by induction, factorial(N) computes the correct result for all N.

Recursive Definition:

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

Where precisely was the inductive hypothesis used?
Analyzing a Recursive Program

factorial computes correct result

• Proof: by induction.
• Step 1: (the base case)
  – For \( N = 0 \), factorial(0) returns 1, which is correct.
• Step 2: (using the inductive hypothesis)
  – Suppose that factorial(N) returns the right result for \( N = K \), where \( K \) is an integer \( \geq 0 \).
    \[ \text{factorial}(K) = K! \]
  – Then, for \( N = K+1 \), factorial(N) returns:
    \[ N \times \text{factorial}(N-1) = (K+1) \times \text{factorial}(K) = (K+1) \times K! = (K+1)! = N! \]
  – Thus, for \( N = K+1 \), factorial(N) also returns the correct result.
• Thus, by induction, factorial(N) computes the correct result for all \( N \).

Recursive Definition:

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

Where precisely was the inductive hypothesis used?

In substituting \( K! \) for factorial(K).
Guidelines for Designing Recursive Functions

• They must explicitly solve one or more base cases.
• Each recursive call must involve smaller values of the arguments, or smaller sizes of the problem.
• Some local work must be done in the recursive step. You cannot simply make the recursive call. (e.g.: return factorial(N-1);
• (Strictly speaking, the only way to verify the correctness is a mathematical proof.)
• (You can see the correspondence between recursion and loops.)
• Return computed value vs update an argument variable
Example Violation of the Guidelines

```c
int puzzle(int N)
{
    if (N == 1) return 1;
    if (N % 2 == 0)
        return puzzle(N/2);
    else return puzzle(3*N+1);
}
```

• The function does NOT always call itself with smaller values.
• Consequence: it is hard to prove if this function always terminates.
  – No one has actually been able to prove or disprove that!!!

How is puzzle(3) evaluated?
Euclid's Algorithm

int gcd(int m, int n)
{
    if (n == 0) return m;
    return gcd(n, m % n);
}

• Recursive algorithm
• One of the most ancient algorithms.
• Computes the greatest common divisor of two numbers.
• It is based on the property that if T divides X and Y, then T also divides X mod Y.
• How is gcd(96, 36) evaluated?
Euclid's Algorithm

int gcd(int m, int n)
{
    if (n == 0) return m;
    return gcd(n, m % n);
}

• How is gcd(96, 36) evaluated?
• gcd(96, 36) = gcd(36, 24) = gcd(24, 12) = gcd(12, 0) = 12.
Recursive Vs. Non-Recursive Implementations

• In some cases, recursive functions are much easier to read.
  – They make crystal clear the mathematical structure of the algorithm.
  – To process recursive data types, such as nodes, oftentimes it is easy to write recursive functions.

• Example: \textbf{int count(link x)}
  – count how many links there are between \(x\) and the end of the list (\(x\) should be included in the count).
  – Recursive solution?
  – Base case?
  – Recursive function?
Recursive Vs. Non-Recursive Implementations

• In some cases, recursive functions are much easier to read.
  – They make crystal clear the mathematical structure of the algorithm.
  – To process recursive data types, such as nodes, oftentimes it is easy to write recursive functions.

• Example: \textbf{int count(link x)}
  – count how many links there are between \textit{x} and the end of the list (\textit{x} should be included in the count).
  – Recursive solution? \( \text{count}(x) = 1 + \text{count}(x\rightarrow \text{next}) \)
  – Base case: \( x = \text{NULL}, \quad \text{count}(\text{NULL}) = 0. \)
  – Recursive function:
    \begin{verbatim}
    int count(link x)
    {
      if (x == NULL) return 0;
      return 1 + count(x\rightarrow \text{next});
    }
    \end{verbatim}
Recursive Vs. Non-Recursive Implementations

• In some cases, recursive functions are much easier to read.
  – They are simpler (less code, fewer loops, “smaller problem”).
  – To process recursive data types, such as nodes, oftentimes it is easy to write recursive functions.

• Any recursive function can also be written in a non-recursive way.

• Oftentimes recursive functions run slower. Why?
Recursive Vs. Non-Recursive Implementations

- Oftentimes recursive functions run slower. Why?
  - Recursive functions generate many function calls.
  - The CPU has to pay a price (perform a certain number of operations) for each function call.
  - Possible solution: use tail-recursion when possible (some compilers have optimizations for it)

- Non-recursive implementations can be uglier (and more buggy, harder to debug) but more efficient.
  - Compromise: make first version recursive, second non-recursive.
    (You can use the recursive one to test the correctness of the non-recursive one.)
Addressing the inefficiency of recursive functions

Tail-recursion
- There is only one recursive call.
- The recursive call is returned directly, not used in a computation.
- E.g. tail recursion: `return factorial(…);`
- E.g. not tail recursion: `return N*factorial(…);`
- Where/how will the work be done?
More on Functions

• The **data computed** by a recursive function can be ‘passed back up’ to the caller function in 2 ways:
  
  – Actually returned (left example)
  – By modifying a pointer or a reference variable (right example)

  • This is needed for tail-recursion

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

int main() {
    int N = 3;
    int res = factorial(N);
    printf("N = %d , res = %d ", N, res);
}
```

```c
void fact_tail_helper(int N, int* res) {
    if (N <= 0) return;
    (*res) = (*res) * N;
    fact_tail_helper(N-1, res);
}

// Wrapper function to set-up parameters.
int fact_tail(int N) {
    int res = 1;
    fact_tail_helper(N, &res);
    // note diff between N and res when fct finishes
    return res;
}
```
TRAPS: Pointers to Local Variables in C

• Pointers to local variables.
  – OK to pass to the fct being called (e.g. fact_tail_helper) a reference/pointer to a local variable (e.g. &res).
  – BAD to return a pointer to a local variable.

```c
void fact_tail_helper(int N, int* res) {
    if (N == 0) return;
    (*res) = (*res) * N;
    fact_tail_helper (N-1, res);
}

int main() {
    int N = 3;
    int res = 1;
    fact_tail_helper (N, &res); // Ok. out->in
    printf("N = %d , res = %d ", N, &res);
}
```
Passing Pointers to Functions in C

- Managing the memory yourself may be safer.

```c
void fact_tail_helper(int N, int* res) {
    if (N == 0) return;
    (*res) = (*res) * N;
    fact_tail_helper (N-1, res);
}
```

// Referencing a local variable.
// OK if done in the correct ‘direction’.
// Easier.
int fact_tail(int N) {
    int res = 1;
    fact_tail_helper (N, &res);
    return res;
}

// Managing the memory yourself.
// Make sure you do it right.
int fact_tail_2(int N) {
    int * res = (int*)malloc(sizeof(int));
    (*res) = 1;
    fact_tail_helper (N, res);
    int temp = (*res);
    free res;
    return temp;
}
Variables: Local vs Static Local

- What will $\text{fact}_v^3(3)$ evaluate to?
- What will $\text{fact}_v^4(3)$ evaluate to?
- Is test_4 ok? (Yes, because res is static.)

```c
int fact_v3(int N) {
    int res = 1;
    if (N == 0) return res;
    res = res * N;
    fact_v3(N-1);
    return res;
}

int fact_v4(int N) {
    static int res = 1;
    if (N == 0) return res;
    res = res * N;
    fact_v4(N-1);
    return res;
}

int* test_4(int N) {
    static int res = 1;
    return &res;
}
```

// static variables have issues as well:
int N = 5;
printf("fact_v4(%d) = %d\n", N, fact_v4(N));
N = 3;
printf("fact_v4(%d) = %d\n", N, fact_v4(N));
Practice Recursive Implementation:

– Binary search
– Sum of elements in an array
– Selection sort (or Insertion sort)
  • Remember the processing that happens?
  • What would make a smaller problem?
– Place N queens on an NxN checkerboard.
– Generate permutations with repetitions
  • Lock combinations: 3 spots, each spot can have anyone of the 0-9 digits
  • Give a recursive function that will print all the possible permutations. The recursive function will populate and update the array with the permutation
    ```
    void perm(int* perm_arr, int spots,...){
        // if basecase: print perm_arr
    }
    ```
Binary Search - Recursive

/* Adapted from Sedgewick */

int search(int A[], int left, int right, int v)
{
    int m = (left+right)/2;
    if (left > right) return -1;
    if (v == A[m]) return m;
    if (left == right) return -1;
    if (v < A[m])
        return search(A, left, m-1, v);  // recursive call
    else
        return search(A, m+1, right, v);  // recursive call
}

- How many recursive calls?
- Any correspondence between the recursive and non-recursive implementations?