Motivation
Definition
Example - Factorial
How
Recursion
Works
Example - List
Sum
How to Use
Recursion
Example - List
Append
Recursive
Helper
Functions
Example - Fibonacci
Recursion vs.
Iteration

COMP2521 23T3
Recursion

Kevin Luxa
cs2521@cse.unsw.edu.au
Recursion

Motivation

Definition

Example - Factorial

How Recursion Works

Example - List Sum

How to Use Recursion

Example - List Append

Recursive Helper Functions

Example - Fibonacci

Recursion vs. Iteration

Learn to program

Make recursive function

No exit condition
• Recursion is a fundamental idea in computer science
• Recursion is a technique that can be used to produce logically simple and elegant solutions
• Problems on some data structures are easily solved with recursion but are more difficult to solve with iteration
• Learning and practicing recursion will improve your logical thinking skills
• Recursion is a powerful problem solving strategy where problems are solved by solving smaller instances of the same problem

• Solving a problem recursively in a program involves writing functions that call themselves from within their own code
Simple example: computing factorial ($n!$)

- $0! = 1$
- $1! = 1$
- $2! = 2 \times 1$
- $3! = 3 \times 2 \times 1$

\[ \vdots \]

- $(n-1)! = (n-1) \times \cdots \times 3 \times 2 \times 1$
- $n! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1$
Simple example: computing factorial \((n!)\)

- \(0! = 1\)
- \(1! = 1\)
- \(2! = 2 \times 1\)
- \(3! = 3 \times 2 \times 1\)
- \[\vdots\]
- \((n-1)! = (n-1) \times \cdots \times 3 \times 2 \times 1\)
- \((n)! = n \times (n-1) \times \cdots \times 3 \times 2 \times 1\)

\[n! = n \times (n-1)!\]
Example: 

\[
\text{factorial}(3) = 3 \times \text{factorial}(2) \\
= 3 \times (2 \times \text{factorial}(1)) \\
= 3 \times (2 \times (1 \times \text{factorial}(0))) \\
= 3 \times (2 \times (1 \times 1)) \\
= 3 \times (2 \times 1) \\
= 3 \times 2 \\
= 6
\]
Recursive factorial:

```c
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```
Some Terminology

**base case** (or *stopping case*)
no recursive call is needed

**recursive case**
calls the function on a smaller version of the problem
• Recursion involves a function calling itself
• Won’t the program get confused?
• No, because each call to the function is a separate instance
  • Each function call creates a new mini-environment
  • This holds all of the data needed by the function
    • Local variables
• The mini-environments are called stack frames
  • They are created as part of the function call
  • They are removed when the function returns
Consider the following program:

```c
int main(void) {
    a(5);
}

void a(int val) {
    b(val);
}

void b(int val) {
    printf("%d\n", val);
}
```

This is how the state of the stack changes:
Let's consider factorial(2):

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

```
fact(2) calls fact(1)
fact(1) calls fact(0)
fact(0) returns 1
fact(1) returns 1 x 1 = 1
fact(2) returns 2 x 1 = 2
```
How Recursion Works

When the stack is growing, that is called "winding"

When the stack is shrinking, that is called "unwinding"
Another simple example: summing integer values in a list

- **Base case: empty list**
  - Sum of an empty list is zero

- **Non-empty lists**
  - I can’t solve the whole problem directly
  - But I do know the first value in the list
  - And if I can sum the rest of the list (smaller than whole list)
  - Then I can add the first value to the sum of the rest of the list, giving the sum of the whole list
Example: 

\[
\text{listSum([3, 1, 4])} = 3 + \text{listSum([1, 4])} \\
= 3 + (1 + \text{listSum([4])}) \\
= 3 + (1 + (4 + \text{listSum([[]]}))) \\
= 3 + (1 + (4 + 0)) \\
= 3 + (1 + 4) \\
= 3 + 5 \\
= 8
\]
Example - Summing a List

```c
struct node {
    int value;
    struct node *next;
};

int listSum(struct node *list) {
    if (list == NULL) {
        return 0;
    } else {
        return list->value + listSum(list->next);
    }
}
```
How to Write a Recursive Function

- Consider whether using recursion is appropriate
  - Can the solution be expressed in terms of a smaller instance of the same problem?
- Identify the base case
- Think about how the problem can be reduced
- Think about how results can be built from the base + recursive cases
Let’s implement this function:

```c
struct node *listAppend(struct node *list, int value) {
    ...
}
```

`listAppend` should insert the given value at the end of the given list and return a pointer to the start of the updated list.
What's wrong with this?

```c
struct node *listAppend(struct node *list, int value) {
    if (list == NULL) {
        return newNode(value);
    } else {
        listAppend(list->next, value);
        return list;
    }
}
```

If `list->next` is NULL, the new node does not get attached to the list.
What's wrong with this?

```c
struct node *listAppend(struct node *list, int value) {
    if (list == NULL) {
        return newNode(value);
    } else {
        listAppend(list->next, value);
        return list;
    }
}
```

If `list->next` is NULL, the new node does not get attached to the list.
Example - List Append

```
struct node *listAppend(struct node *list, int value) {
    if (list == NULL) {
        return newNode(value);
    } else if (list->next == NULL) {
        list->next = newNode(value);
        return list;
    } else {
        listAppend(list->next, value);
        return list;
    }
}
```

This works, but is not very elegant, as it repeats the call to newNode and repeats return list.
A more elegant solution:

```c
struct node *listAppend(struct node *list, int value) {
    if (list == NULL) {
        return newNode(value);
    } else {
        list->next = listAppend(list->next, value);
        return list;
    }
}
```
Sometimes, recursive solutions require recursive helper functions

- Data structure uses a "wrapper" struct
- Recursive function needs to take in extra information (e.g., state)
Consider the following list representation:

```c
struct node {
    int value;
    struct node *next;
};

struct list {
    struct node *head;
};

Let's implement this function:

```c
void listAppend(struct list *list, int value);
```
void listAppend(struct list *list, int value);

We can’t recurse with this function because our recursive function needs to take in a struct node pointer.

Solution: Use a recursive helper function!
Recursive Helper Functions

Wrapper structs

Our convention for naming recursive helper functions is to prepend "do" to the name of the original function.
Problem:

- Print a linked list in a numbered list format, starting from 1.

```
void printNumberedList(struct node *list);
```

Example:

- Suppose the input list contains the following elements: [11, 9, 2023]
- We expect the following output:

  1. 11
  2. 9
  3. 2023
We need to keep track of the current number.

Solution:

- Use a recursive helper function that takes in an extra integer

```c
void printNumberedList(struct node *list) {
    doPrintNumberedList(list, 1);
}

void doPrintNumberedList(struct node *list, int num) {
    if (list == NULL) return;
    print("%d. %d\n", num, list->value);
    doPrintNumberedList(list->next, num + 1);
}
```
Although recursive solutions are often simple and elegant, they can be horribly inefficient!

Example: Computing Fibonacci numbers

\[
F(0) = 0 \\
F(1) = 1 \\
F(n) = F(n - 1) + F(n - 2)
\]
Recursive Fibonacci:

```c
int fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```
Computation of \( \text{fib}(5) \):

```
\text{fib}(5) \\
\downarrow \\
\text{fib}(4) \\
\downarrow \\
\text{fib}(3) \\
\downarrow \\
\text{fib}(2) \\
\downarrow \\
\text{fib}(1) \\
\downarrow \\
\text{fib}(0)
```

The number of recursive calls, and hence the time taken by the function, grows exponentially as \( n \) increases.
Much more efficient iterative implementation:

```c
int fib(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    int prevPrev = 0;
    int prev = 1;
    int curr = 1;
    for (int i = 2; i <= n; i++) {
        curr = prev + prevPrev;
        prevPrev = prev;
        prev = curr;
    }

    return curr;
}
```
Recursion vs. Iteration

• If there is a simple iterative solution, a recursive solution will generally be slower
  • Due to a stack frame needing to be created for each function call
• A recursive solution will generally use more memory than an iterative solution
https://forms.office.com/r/aPF09YHZ3X