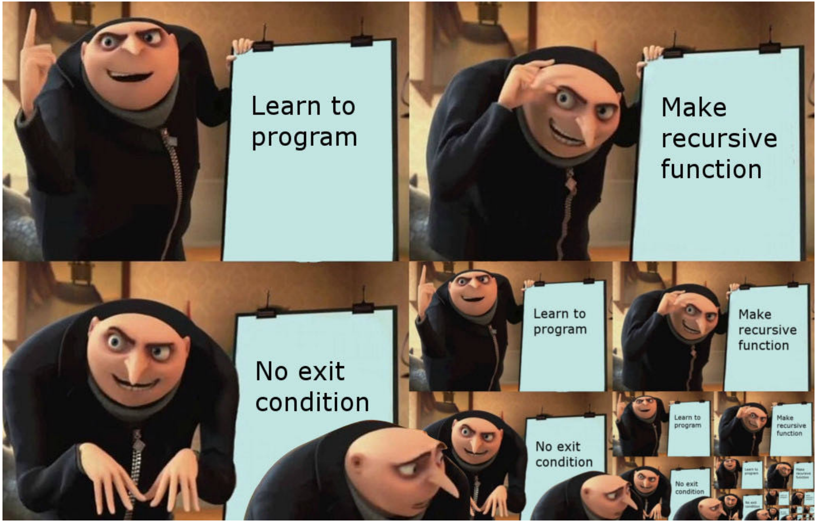
COMP2521 23T3
Recursion

Kevin Luxa

cs2521@cse.unsw.edu.au


Learn to program
Make recursive function
No exit condition
Learn to program
Make recursive function
No exit condition
Learn to program
Make recursive function
No exit condition

Motivation

- 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

Definition

- 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

Example - Factorial

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 - Factorial

Example:

```
factorial(3) = 3 * factorial(2)
             = 3 * (2 * factorial(1))
             = 3 * (2 * (1 * factorial(0)))
             = 3 * (2 * (1 * 1))
             = 3 * (2 * 1)
             = 3 * 2
             = 6
```

Example - Factorial

Recursive factorial:

```
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

How Recursion Works

- 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

How Recursion Works

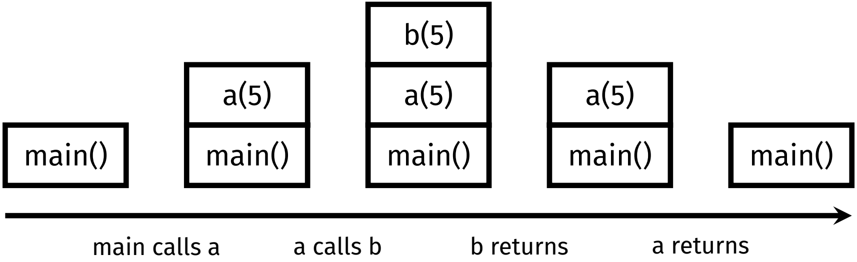
Consider the following program:

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

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

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

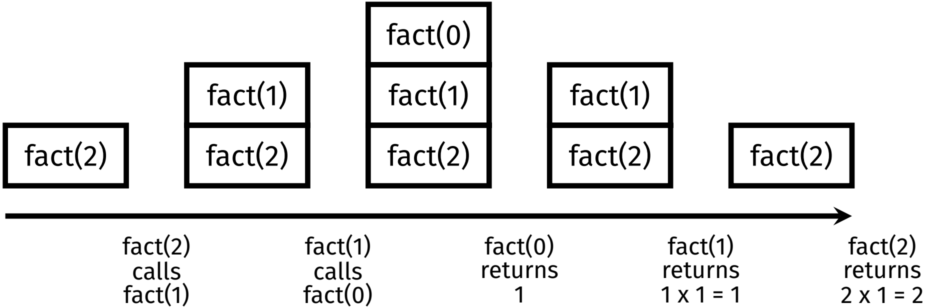
This is how the state of the stack changes:



How Recursion Works

Let's consider `factorial(2)`:

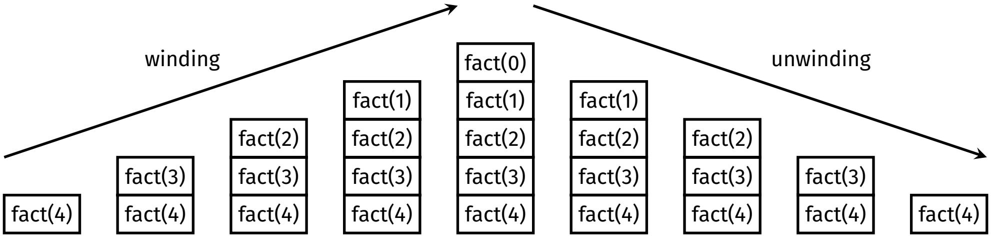
```c
int factorial(int n) {
    if (n == 0) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```



How Recursion Works

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

When the stack is shrinking, that is called ”unwinding”



Example - Summing a List

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 - Summing a List

Example:

```
listSum([3, 1, 4]) = 3 + listSum([1, 4])
                   = 3 + (1 + listSum[4])
                   = 3 + (1 + (4 + 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

Example - List Append

Let's implement this function:

```
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.

Example - List Append

What's wrong with this?

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

Example - List Append

What's wrong with this?

```
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`.

Example - List Append

A more elegant solution:

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

Recursive Helper Functions

Sometimes, recursive solutions require recursive helper functions

- Data structure uses a ”wrapper” struct
- Recursive function needs to take in extra information (e.g., state)

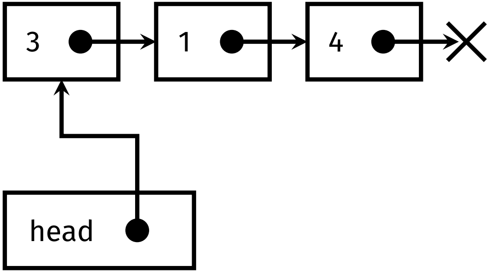
Recursive Helper Functions

Wrapper structs

Consider the following list representation:

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

struct list {
    struct node *head;
};
```



Let's implement this function:

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

Recursive Helper Functions

Wrapper structs


```c
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


```
void listAppend(struct list *list, int value) {
    list->head = doListAppend(list->head, value);
}

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

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

Recursive Helper Functions

Passing extra information

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
```

Recursive Helper Functions

Passing extra information

We need to keep track of the current number.

Solution:

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

```
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);
}
```

Example - Fibonacci

Although recursive solutions are often simple and elegant, they can be horribly inefficient!

Example: Computing Fibonacci numbers

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

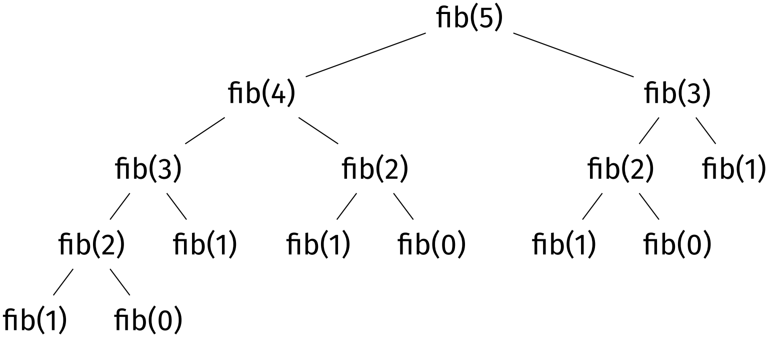
Example - Fibonacci

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);
    }
}
```

Example - Fibonacci

Computation of `fib(5)`:



The number of recursive calls, and hence the time taken by the function, grows exponentially as `n` increases.

Example - Fibonacci

Much more efficient iterative implementation:

```
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

Feedback

`https://forms.office.com/r/aPF09YHZ3X`