Definition
Example - Pyramid
Example - Factorial
Example - Fibonacci

How Recursion Works
Recursion on Linked Lists
Example - List Sum
Example - List Append

How to Use Recursion
Exercises
Recursive Helper Functions
Recursion vs. Iteration

COMP2521 24T1
Recursion

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

Definition

Example - Pyramid

Example - Factorial

Example - Fibonacci

How Recursion Works

Recursion on Linked Lists

Example - List Sum

Example - List Append

How to Use Recursion

Exercises

Recursive Helper Functions

Recursion vs. Iteration

Learn to program

Make recursive function

No exit condition

Learn to program

Make recursive function

No exit condition
Recursion...

is a problem solving strategy where problems are solved via solving smaller or simpler instances of the same problem

A recursive function calls itself
Example - Building a Pyramid
Example - Building a Pyramid

Iteratively

1. Base
2. Second layer
3. Third layer
4. Top layer
To build a pyramid of width $n$:
  - For each width $w$ from $n$ down to 1 (decrementing by 2 each time):
    - Build a $w \times w$ layer of blocks on top
Example - Building a Pyramid Recursively

1. Build a 7 x 7 layer of blocks
2. Build a pyramid of width 5 on top!
To build a pyramid of width $n$:

1. Build an $n \times n$ layer
2. Then *build a pyramid of width* $n - 2$ on top
To build a pyramid of width $n$:

1. Build an $n \times n$ layer
2. Then *build a pyramid of width* $n - 2$ on top

What's wrong with this method?
To build a pyramid of width $n$:

1. If $n \leq 0$, do nothing
2. Otherwise:
   1. Build an $n \times n$ layer
   2. Then *build a pyramid of width* $n - 2$ on top
The factorial of \( n \) (where \( n \geq 0 \)) denoted by \( n! \) is the product of all positive integers less than or equal to \( n \).

\[
n! = n \times (n - 1) \times (n - 2) \times \cdots \times 2 \times 1
\]
Iterative method:

```c
int factorial(int n) {
    int res = 1;
    for (int i = 1; i <= n; i++) {
        res *= i;
    }
    return res;
}
```
Observation:

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

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

For example:

\[ 4! = 4 \times 3 \times 2 \times 1 \]

\[ = 4 \times 3! \]
Recursive method:

```c
int factorial(int n) {
    return n * factorial(n - 1);
}
```
Recursive method:

```c
int factorial(int n) {
    return n * factorial(n - 1);
}
```

What's wrong with this function?
Recursive method:

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

Example:

\[
\text{factorial}(3) = 3 \times \text{factorial}(2) \\
= 3 \times (2 \times \text{factorial}(1)) \\
= 3 \times (2 \times (1 \times \text{factorial}(0))) \\
= 3 \times (2 \times (1 \times 1)) \\
= 3 \times (2 \times 1) \\
= 3 \times 2 \\
= 6
\]
The Fibonacci sequence is a sequence where each number is the sum of the two previous numbers, and the first two numbers in the sequence are 0 and 1.

\[
F_0 = 0 \\
F_1 = 1 \\
F_n = F_{n-1} + F_{n-2}
\]
Recursive method:

```c
int fib(int n) {
    if (n == 0) {
        return 0;
    } else if (n == 1) {
        return 1;
    } else {
        return fib(n - 1) + fib(n - 2);
    }
}
```
A recursive function calls itself

This is possible because there is a difference between a function and a function call

Each function call creates a new mini-environment, called a stack frame, that holds all the local variables used by the function call
Consider this program (no recursion):

```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:

- `main()`
- `a(5)`
- `b(5)`
- `a()`
- `main()`
- `a()`
- `main()`
- `main()`

`main()` calls `a`

`a()` calls `b`

`b()` returns

`a()` returns
Now consider \texttt{factorial(2)}:

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

This is how the state of the stack changes:

fact(2) calls fact(1)
fact(1) calls fact(0)  \text{fact(0) returns 1}  \text{1 x 1 = 1}
fact(1) 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”
How Recursion Works

Pre-order operations
Operations before the recursive call occur during winding.

Post-order operations
Operations after the recursive call occur during unwinding.
Recall that recursion is a problem solving strategy where problems are solved via solving smaller or simpler instances of the same problem.
Recall that recursion is a problem solving strategy where problems are solved via solving smaller or simpler instances of the same problem.

How do we apply recursion to linked lists?
Example: summing values of 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
```
Recursive method:

```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);
    }
}
```
Example: append a value to a list

```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 solution?

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

Consider this list...

...and this function call:

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

```c
listAppend(myList, 5);
```
Example - List Append

1. \texttt{struct node *listAppend(struct node *list, int value) \{}
2. \quad \texttt{if (list == NULL) \{}
3. \quad \quad \texttt{return newNode(value);}
4. \quad \}} \texttt{else \{}
5. \quad \texttt{listAppend(list->next, value);}
6. \quad \texttt{return list;}
7. \}}

The recursive call on line 5 creates a new node and returns it...

...but this new node is not attached to the list!
The node containing 4 still points to NULL.
Correct 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;
    }
}
```
Why does this work?

list->next = listAppend(list->next, value);

Consider the following list:

Two cases to consider:
(1) The rest of the list is empty
(2) The rest of the list is not empty
Example - List Append

```c
list->next = listAppend(list->next, value);
```

Case 1: The rest of the list is empty

```
list
4
```

In this case, `listAppend(list->next, value)` will return a new node. `list->next = ...` causes `list->next` to point to this new node.
Example - List Append

```c
list->next = listAppend(list->next, value);
```

Case 1: The rest of the list is empty

In this case, `listAppend(list->next, value)` will return a new node.
Example - List Append

list->next = listAppend(list->next, value);

Case 1: The rest of the list is empty

In this case, listAppend(list->next, value) will return a new node
list->next = ... causes list->next to point to this new node
list->next = listAppend(list->next, value);

Case 2: The rest of the list is not empty

```
list 4 ...
```

In this case, `listAppend(...)` will append the value to the rest of the list and return a pointer to the (start of the) rest of the list. `list->next = ...` causes `list->next` to point to the start of the rest of the list (which it was already pointing to).
Example - List Append

```
list->next = listAppend(list->next, value);
```

**Case 2: The rest of the list is **not** empty**

In this case, `listAppend(…)` will append the value to the rest of the list and return a pointer to the (start of the) rest of the list.
Example - List Append

Case 2: The rest of the list is **not** empty

```c
list->next = listAppend(list->next, value);
```

In this case, `listAppend(...)` will append the value to the rest of the list and return a pointer to the (start of the) rest of the list

`list->next = ...` causes `list->next` to point to the start of the rest of the list (which it was already pointing to)
How to Write a Recursive Function

1. **Consider whether using recursion is appropriate**
   - Can the solution be expressed in terms of a smaller instance of the same problem?

2. **Identify the base case(s)**

3. **Identify the subproblem(s)**
   - **Assume** that the function works for the subproblem(s)
     - Like in mathematical induction!

4. **Think about how to relate the original problem to the subproblem(s)**
Exercise 1:
- Given a linked list, print the items in the list in reverse.

Exercise 2:
- Given a linked list and an index, return the value at that index. Index 0 corresponds to the first value, index 1 the second value, and so on.

Exercise 3:
- Given a linked list and a value, delete the first instance of the value from the list (if it exists), and return the updated 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)
Recursive Helper Functions

Wrapper structs

Wrapper struct for a linked list:

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

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

- Example - List Sum
- Example - List Append
- How to Use Recursion
- Recursive Helper Functions
- Recursion vs. Iteration
Example: Implement this function:

```c
void listAppend(struct list *list, int value);
```
Recursive Helper Functions
Wrapper structs

Definition
Example - Pyramid
Example - Factorial
Example - Fibonacci

How
Recursion
Works
Recursion on
Linked Lists
Example - List
Sum
Example - List
Append
How to Use
Recursion
Exercises
Recursive
Helper
Functions
Recursion vs.
Iteration

**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!
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.

```c
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);
}
```
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
Definition
Example - Pyramid
Example - Factorial
Example - Fibonacci

How Recursion Works
Recursion on Linked Lists
Example - List Sum
Example - List Append

How to Use Recursion
Exercises
Recursive Helper Functions

Recursion vs. Iteration

https://forms.office.com/r/5c0fb4tvMb