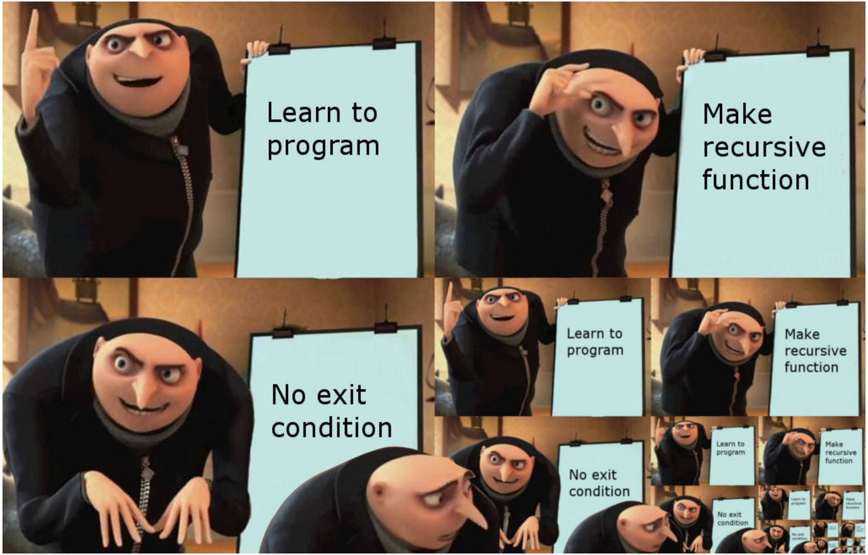
COMP2521 25T2
Recursion

Sim Mautner

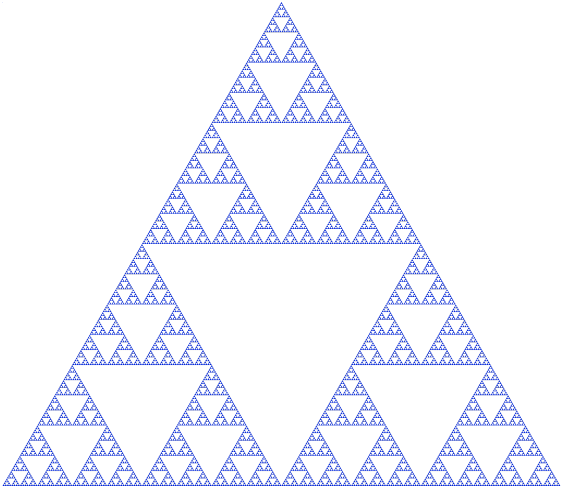
cs2521@cse.unsw.edu.au


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

Definition

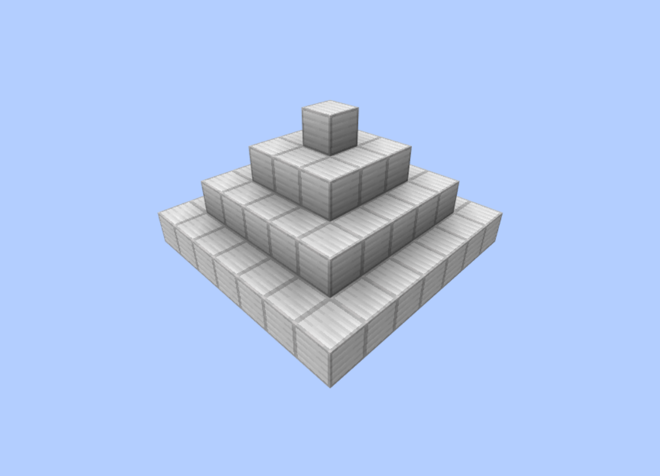
Recursion is a problem solving strategy
where problems are solved via solving **subproblems**
(smaller or simpler instances of the same problem)

In programming, we solve problems recursively by
using functions that call themselves



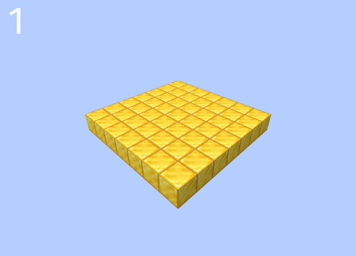
The Sierpinski triangle

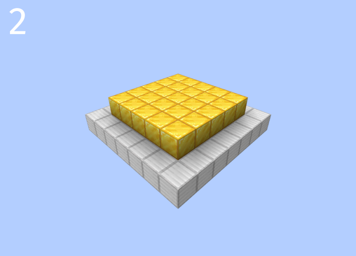
Example - Building a Pyramid

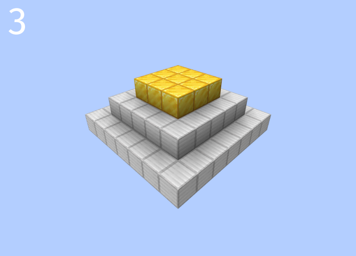


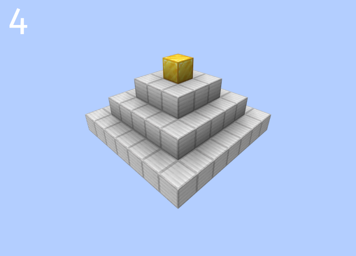
Example - Building a Pyramid

Iteratively









Example - Building a Pyramid

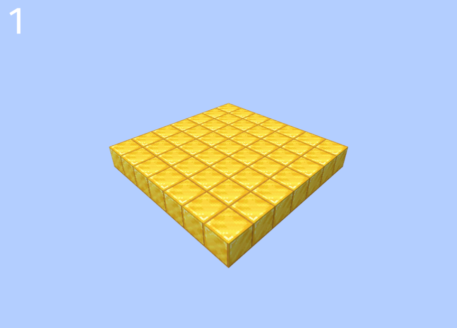
Iteratively

To build a pyramid of width n:

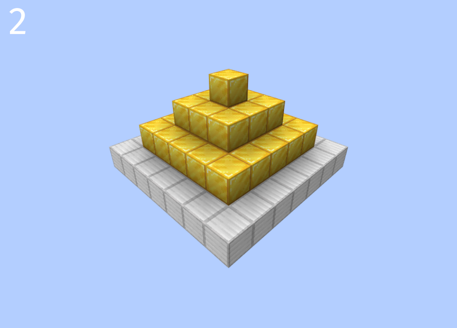
- For each width w from n down to 1 (decrementing by 2 each time):
  - Build a $w \times w$ layer of blocks on top

Example - Building a Pyramid

Recursively



Build a 7 x 7 layer of blocks



Build a pyramid of width 5 on top!

Example - Building a Pyramid
Recursively

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

Example - Building a Pyramid

Recursively

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?

Example - Building a Pyramid

Recursively

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

Example - Factorial

The factorial of n (where $n \geq 0$)
denoted by $n!$
is the product of all positive integers
less than or equal to n.

$$n! = n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1$$

Example - Factorial

Iteratively

Iterative method:

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

Example - Factorial

Observation:

$$
\begin{aligned}
n! &= n \times (n-1) \times (n-2) \times \cdots \times 2 \times 1 \\
&= n \times (n-1)!
\end{aligned}
$$

For example:

$$
\begin{aligned}
4! &= 4 \times 3 \times 2 \times 1 \\
&= 4 \times 3!
\end{aligned}
$$

Example - Factorial

Recursively

Recursive method:

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

Example - Factorial

Recursively

Recursive method:

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

What's wrong with this function?

Example - Factorial

Recursively

Recursive method:

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

How Recursion Works

- 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

How Recursion Works

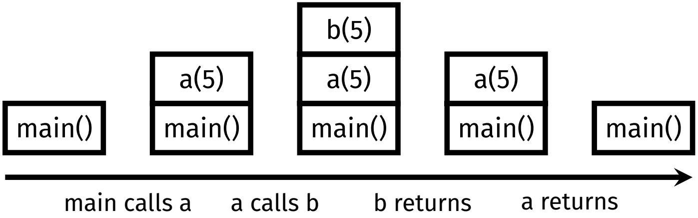
Consider this program (no recursion):

```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:

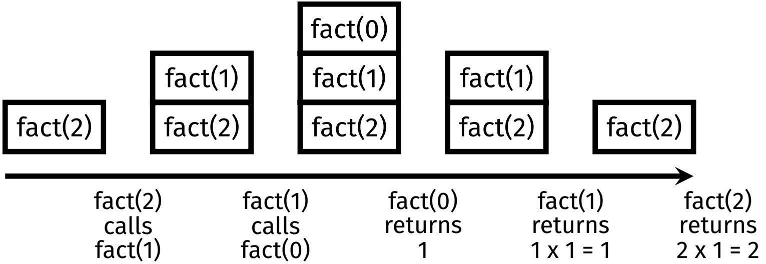


How Recursion Works

Now consider `factorial(2)`:

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

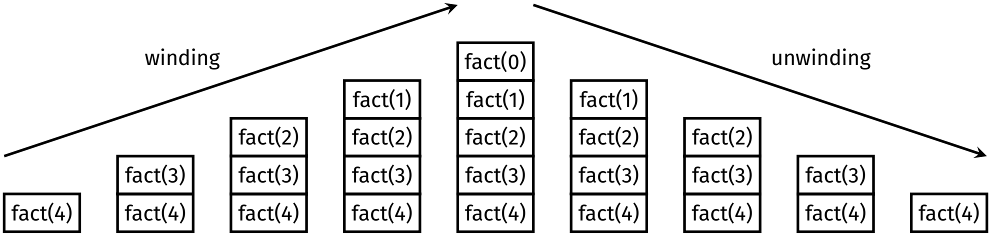
This is how the state of the stack changes:



How Recursion Works

When the function recurses, that is called “winding”

When recursive calls return, that is called “unwinding”



How Recursion Works

Pre-order operations

Operations before the recursive call occur during winding.

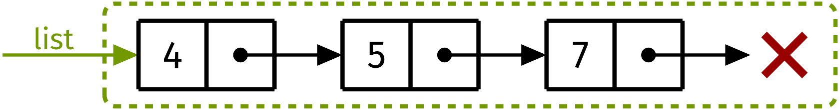
Post-order operations

Operations after the recursive call occur during unwinding.

Recursion on Linked Lists

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

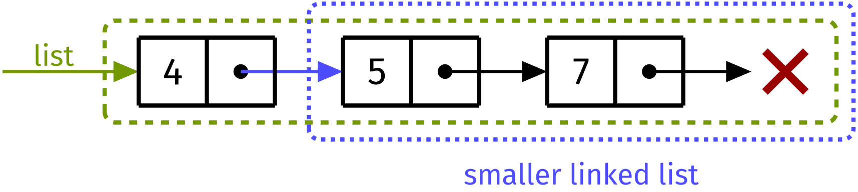
How do we apply recursion to linked lists?



Recursion on Linked Lists

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 a List

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
```

Example - Summing a List

Recursive method:

```
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

First, **think**:

- How can the solution be expressed in terms of subproblems?
- What would the subproblem(s) be?
- How can you relate the original problem to the subproblem(s)?
- What are the base cases?

Then, **implement**:

- Implement base case(s) first
- Then implement recursive cases
- Each subproblem corresponds to a recursive call
  - **Assume** that the function works for the subproblem(s)
    - Like in Mathematical Induction!

Exercises

Exercise 1:

- Given a linked list, print the items in the list in reverse.

Exercise 2:

- Given a linked list, print every second item.

Exercise 3:

- 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.

Example - List Append

Example: append a value to a list

```
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

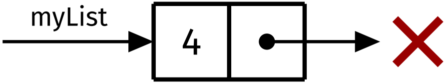
What's wrong with this solution?

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

Example - List Append

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

Consider this list...



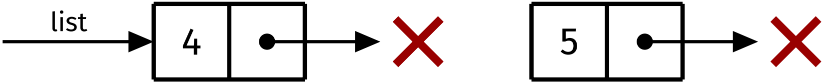
...and this function call:

```c
listAppend(myList, 5);
```

Example - List Append

```
1  struct node *listAppend(struct node *list, int value) {
2      if (list == NULL) {
3          return newNode(value);
4      } else {
5          listAppend(list->next, value);
6          return list;
7      }
8  }
```

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.

Example - List Append

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;
    }
}
```

Example - List Append

Why does this work?

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

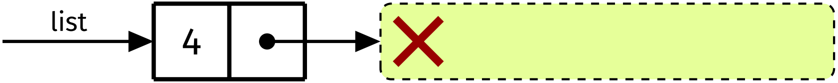
Consider the following list:

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

Example - List Append


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

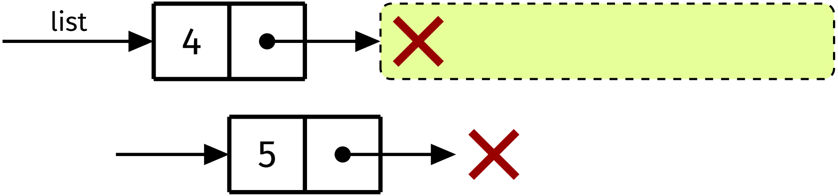
Case 1: The rest of the list is empty



Example - List Append

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

Case 1: The rest of the list is empty

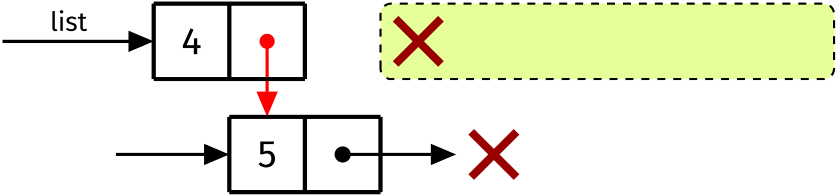


In this case, `listAppend(list->next, value)` will return a new node

Example - List Append

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

Case 1: The rest of the list is empty



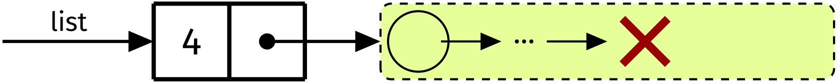
In this case, `listAppend(list->next, value)` will return a new node

`list->next = ...` causes `list->next` to point to this new node

Example - List Append


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

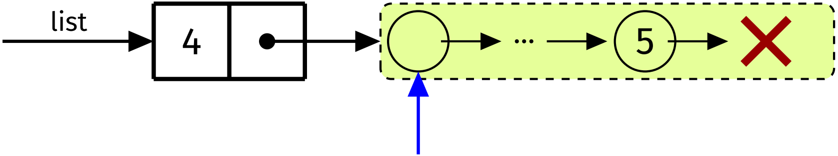
Case 2: The rest of the list is **not** empty



Example - List Append

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

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

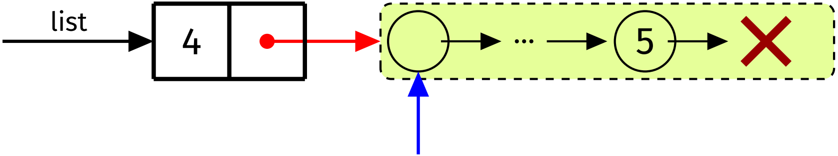


In this case, `listAppend(...)` will append the value to the rest of the list and return a pointer to the (start of the) rest of the list

Example - List Append

```
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

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

More Exercises

Exercise 1:

- Given a linked list, return a copy of the linked list.

Exercise 2:

- 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.

Recursive Helper Functions

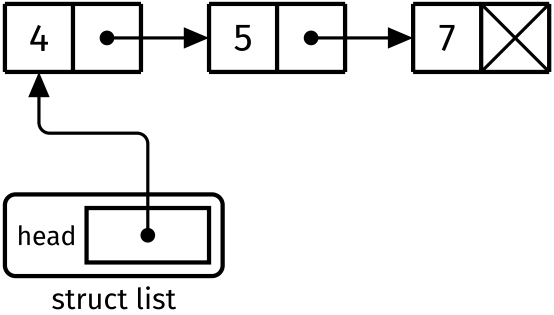
Sometimes, recursive solutions require recursive helper functions

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

Recursive Helper Functions

Wrapper structs

Wrapper struct for a linked list:




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

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

Recursive Helper Functions

Wrapper structs

Example: Implement this function:

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

Recursive Helper Functions

Wrapper structs

```
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

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

void doPrintNumberedList(struct node *list, int num) {
    if (list == NULL) return;

    printf("%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