Recursion is a simple idea - functions that call themselves

- if you already know how functions work - you know how recursion works
- good opportunity to check your understanding of functions is correct
- the patterns of program execution it creates are complex & fascinating

It's easy if you already understand it
But we haven't learnt it?
Loop through and add them up . . . we can already do this

```c
// return sum of data fields of list
int sum_list(struct node *head) {
    struct node *n = head;
    int total = 0;
    while (n != NULL) {
        total += n->data;
        n = n->next;
    }
    return total;
}
```

source code for sum_list.c
What about a different way?

Let’s look at what might happen if we have a function that can call itself.

A function that says:

// sum_list calls itself again, but on a different // part of the list
int sum_list(struct node *head) {
    int total = head->data + sum_list(head->next);
    return total;
}

The function can call itself? What happens here?
Functions calling themselves

\[
\text{sum\_list}(\text{head}) = \text{head}\rightarrow \text{data} + \text{sum\_list}(\text{head}\rightarrow \text{next});
\]

The total is equal to the value of the head added to the \text{sum\_list} function called on the rest of the list

10 + whatever the rest of the list happens to add up to . . .
The second function

\[
\text{sum\_list}(\text{head}) = \text{head}\rightarrow\text{data} + \text{sum\_list}(\text{head}\rightarrow\text{next});
\]

\[
\text{sum\_list}(\text{head}) = \text{head}\rightarrow\text{data} + \text{head2}\rightarrow\text{data} + \text{sum\_list}(\text{head2}\rightarrow\text{next});
\]

10 + 2 + whatever the rest of the list adds up to
It keeps going . . .

\[
\text{sum\_list}(\text{head}) = \text{head}\rightarrow\text{data} + \text{sum\_list}(\text{head}\rightarrow\text{next}) ;
\]

\[
\text{sum\_list}(\text{head}) = \text{head}\rightarrow\text{data} + \text{head2}\rightarrow\text{data} + \text{sum\_list}(\text{head2}\rightarrow\text{next}) ;
\]

\[
\text{sum\_list}(\text{head}) = \text{head}\rightarrow\text{data} + \text{head2}\rightarrow\text{data} + \text{head3}\rightarrow\text{data} + \text{sum\_list}(\text{head3}\rightarrow\text{next}) ;
\]

10 + 2 + 5 + whatever the rest of the list adds up to
Is this endless?

Like loops, Recursive function calls still need to know when to stop

- In the previous example:
- What happens if we reach the end of the list?
- What happens if the list was empty to begin with?

We need a “stopping case” where the function won’t call itself again
Two Cases

Keep going or stop

- We’ve already got the “keep going” case
- How do we stop?
- Let’s test for the situation where we wouldn’t want to add more elements

```c
// sum_list calls itself again, but stops if there's
// nothing to add
int sum_list(struct node *head) {
    if (head == NULL) {
        return 0;
    } else {
        int total = head->data + sum_list(head->next);
        return total;
    }
}
```
Simplify `sum_list` a little

We actually don’t need the variable `total`

- this code is exactly the same

```c
// return sum of data fields of list
int sum_list(struct node *head) {
    if (head == NULL) {
        return 0;
    } else {
        return head->data + sum_list(head->next);
    }
}
```

source code for `sum_list_recursive.c`
The Function Call Stack during recursion

Initially we have a main function that calls `sum_list()`
The Call Stack as Recursion continues

As the function “recurses”, it adds more function calls

running
sum_list() again

adding another
sum_list()
Reaching a stopping case

Returning from a recursive function to the previous call

- `sum_list()`
- `sum_list()`
- `sum_list()`
- `main()`

Returning because of the stopping case

- `sum_list()`
- `sum_list()`
- `main()`

Return to previous function call

- `sum_list()`
- `sum_list()`
- `main()`

This also returns
Completion of a recursive function

Eventually the chain of returns will finish

Functions return one after the other until . . .

We're back to the main
Here is a loop which calls `free` for every node in a list

```c
// free all the node in a list
void free_list(struct node *head) {
    struct node *p = head;
    while (p != NULL) {
        struct node *next_node = p->next;
        free(p);
        p = next_node;
    }
}
```

What does a recursive version look like?
A recursive version is shorter and simpler

```c
// free all the node in a list
void free_list(struct node *head) {
    if (head != NULL) {
        free_list(head->next);
        free(head);
    }
}
```

source code for sum_list_recursive.c
create_list - procedural version

Here is a loop which creates a list of integers

```c
// create a list of the integers start..finish
// return pointer to head
struct node *create_list(int start, int finish) {
    struct node *head = NULL;
    // create list starting from last node
    int i = finish;
    while (i >= start) {
        struct node *new_node = malloc(sizeof (struct node));
        new_node->data = i;
        new_node->next = head;
        head = new_node;
        i--;
    }
    return head;
}
```

source code for sum_list.c
Again recursive version is shorter and simpler

```c
// create a list of the integers start..finish
// return pointer to head
struct node *create_list(int start, int finish) {
    if (start > finish) {
        return NULL;
    } else {
        struct node *new_node = malloc(sizeof (struct node));
        new_node->data = start;
        new_node->next = create_list(start + 1, finish);
        return new_node;
    }
}
```

source code for sum_list_recursive.c
Returning to our Battle Royale Example

Say we had a list of people who had been knocked out of the game and we want to “replay” the order they were knocked out?

- We have a linked list of names
- It’s currently in the reverse order of when they were knocked out
- So we want to print out their names in the opposite of their order
We have a standard linked list node

Contains a name and a pointer

```c
struct node {
    char name[MAX_NAME_LENGTH];
    struct node *next;
};
```
How it was created

Let’s say that during our game, we built a list of players

- Each time a player is knocked out, we add them to the head of a list

We want to be able to print this out in the order that they were knocked out
How do we do this without Recursion?

A “procedural” implementation

Loop to end of the list and print out the name

Have some way of remembering which player we’ve already printed from

Start a new loop, going until just before the one we printed previously

Print out that name

Keep repeating until there are no names left
struct player *print_before(struct player *player_list,
    struct player *after) {

    // loop until you see the after pointer
    struct player *curr = player_list;
    struct player *prev = NULL;
    while (curr != after) {
        prev = curr;
        curr = curr->next;
    }

    if (prev != NULL) {
        // element exists, print its name
        printf("%s\n", prev->name);
    }

    return prev;
}

source code for battle_royale.c
Code for a procedural reverse_print()

// Print out the names stored in the list in reverse order
// This is a procedural programming implementation

void reverse_print(struct player *player_list) {
    struct player *end = NULL;
    int finished = 0;

    // Loop once for each name in the list
    while (!finished) {
        end = print_before(player_list, end);
        if (end == NULL) {
            finished = 1;
        }
    }
}

source code for battle_royale.c
What did we need to do?

**Outer Loop**
- Loops once for each element of the list
- Keeps track of the last element printed

**print_before() function**
- Loops until the given element pointer
- prints out the one before that (if it exists)
- returns a pointer to the element that was printed
Let’s try this recursive and see how it works

Stopping case

- there are no elements, so print out nothing

Otherwise

- printReverse() the rest of the list
- After that print out the current head.
// Print out the names stored in the list in reverse order
// This is a recursive programming implementation

void rev_print_rec(struct player *player_list) {
    if (player_list == NULL) {
        // stopping case (there are no elements)
        return;
    } else {
        // there are element(s)
        rev_print_rec(player_list->next);
        fputs(player_list->name, stdout);
        putchar('\n');
    }
}
Even simpler code for reverse_print_recursive()

// Print out the names stored in the list in reverse order
// This is a recursive programming implementation
void rev_print_rec(struct player *player_list) {
    if (player_list != NULL) {
        // there are element(s)
        rev_print_rec(player_list->next);
        printf("%s\n", player_list->name);
    }
}

source code for battle_royale.c
Yes.

- Recursion often takes a lot of thinking and not much code

Still, let’s look deeper to get a stronger understanding

- What order are things happening?
- What happens if we change the order?
What’s the order of execution?

A single call of our recursive function:

- Check if we’re stopping, if so return
- Otherwise, call the function again with the tail (all remaining elements)
- Then print the name of the current head of the list
Order of execution

- **More recursive function calls**
  - Check if we’re stopping, if so return
  - Otherwise, call the function again with the tail (all remaining elements)
    - Check if we’re stopping, if so return
    - Otherwise, call the function again with the tail (all remaining elements)
      - Check if we’re stopping, if so return
      - Otherwise, call the function again with the tail (all remaining elements)
        - Then print the name of the current head of the list
        - Then print the name of the current head of the list
  - Then print the name of the current head of the list
What happens if we change the order in a recursive function?

- Check if we’re stopping, if so return
- Then print the name of the current head of the list
- Otherwise, call the function again with the tail (all remaining elements)
- Having swapped 2 and 3, will the function behave differently?
// Changing the order of operations in a recursive function
// This is a recursive programming implementation
void rev_print_rec(struct player *player_list) {
    if (player_list != NULL) {
        // there are element(s)
        // the recursion is now after the print
        printf("%s\n", player_list->name);
        rev_print_rec(player_list->next);
    }
}
Interesting results

We’re now printing in order . . .

How did this happen?

Let’s look at the order of execution again
Order of execution

- If we change when the recursive function call is made . . .
- Check if we’re stopping, if so return
- Otherwise, print the name of the current head of the list
- Then call the function again with the tail (all remaining elements)
  - Check if we’re stopping, if so return
  - Otherwise, print the name of the current head of the list
  - Then call the function again with the tail (all remaining elements)