SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220427<A|D>

Replace <A|D> with this section's letter

Recursion II

CS 2124: Object Oriented Programming
Darryl Reeves, Ph.D.

Agenda

- Linked List Recursion (continued)
- Towers of Hanoi
- Recursive strategy

Linked lists and recursion

Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};

Node* build_list(const vector<int>& vals);
void print_list(const Node* ptr);

int main() {
    Node* my_list = build_list({1, 1, 2, 3, 5, 8, 13, 21, 34, 55});
    print_list(my_list);

    cout << endl;
    Node* other_list = dup_list(my_list);



}
```

Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


___ dup_list() {}
```

Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
_4_ dup_list() {}
```

TurningPoint

SRS Setup
Login: student.turningtechnologies.com
Session ID: 20220427<A|D>

Replace <A|D> with this section's letter

Which return type replaces blank #4 when duplicating a list?

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


_4_ dup_list() {}
```

Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list() {}
```

Linked lists as recursive data structures

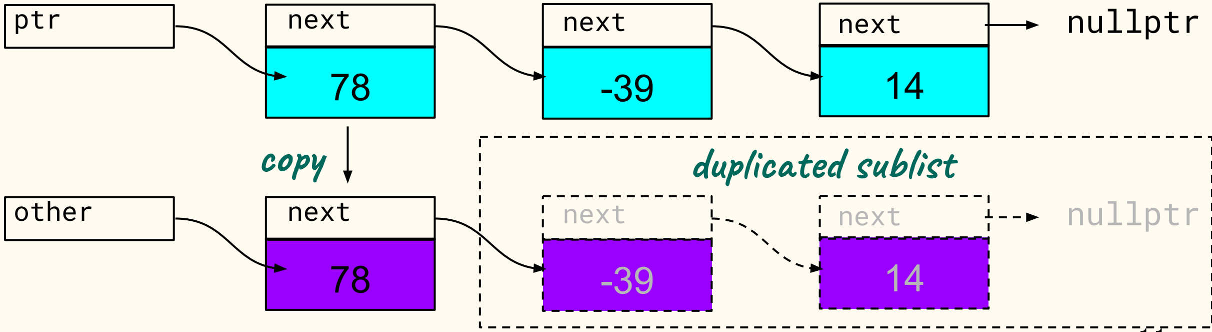
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {}
```

Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

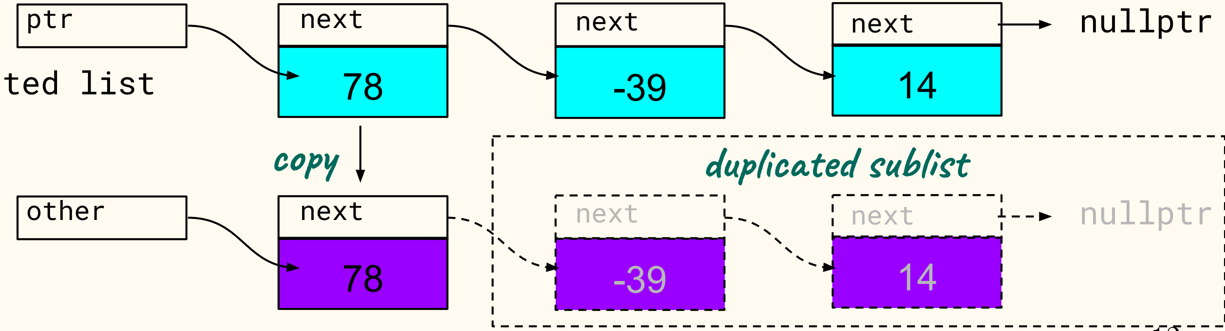
```
Node* dup_list(const Node* ptr) {
    // base case

    // recursive case
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

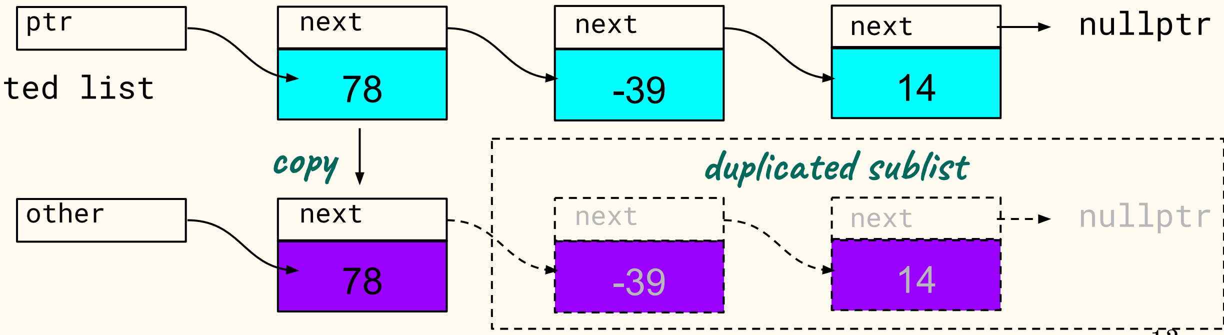
```
Node* dup_list(const Node* ptr) {
    // base case

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



When are we done? What will be true of the state of the problem after all `Node`s have been duplicated?

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

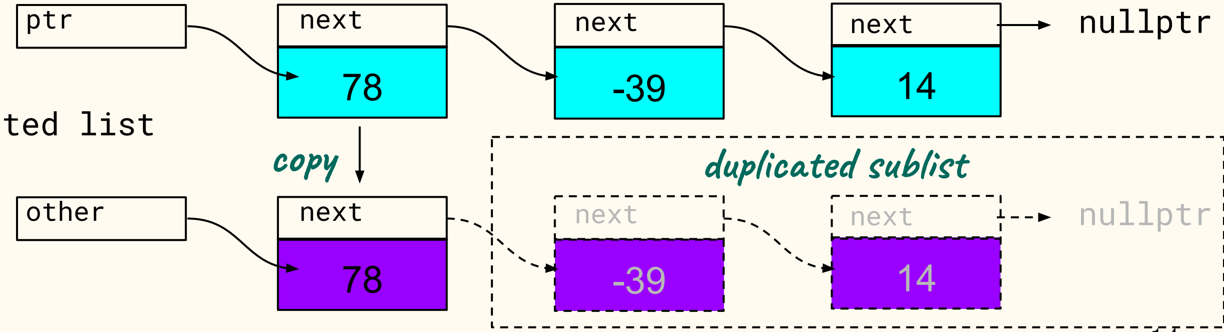
```
Node* dup_list(const Node* ptr) {
    // base case

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

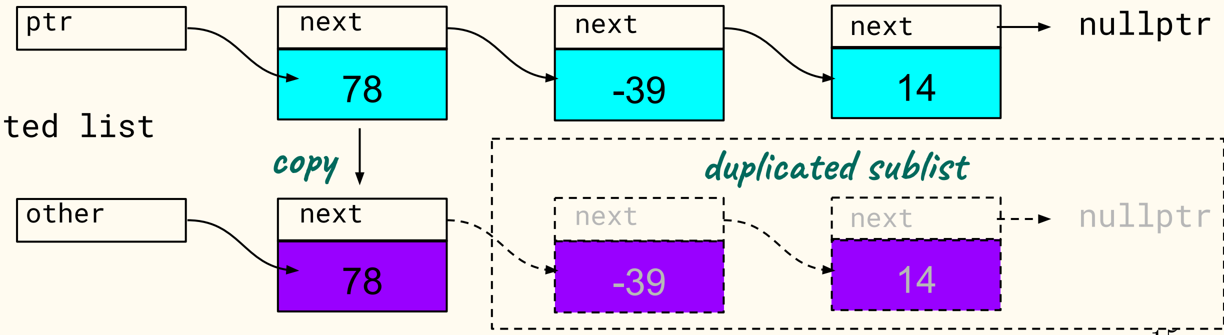
```
Node* dup_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) return ___;

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

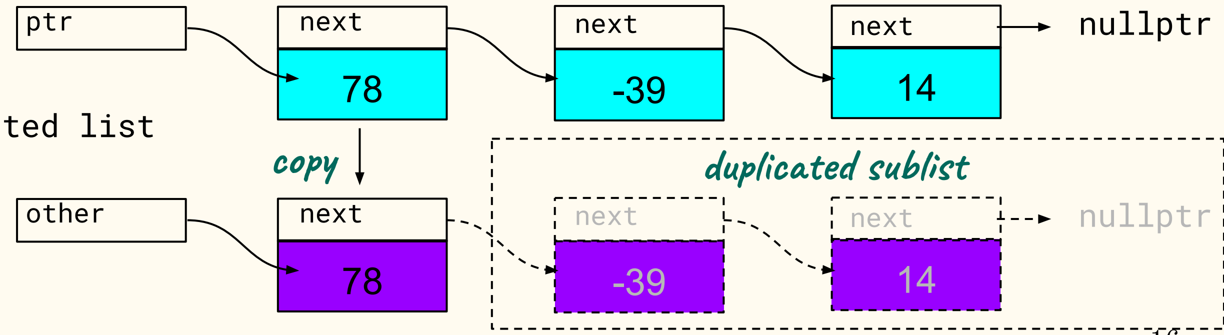
```
Node* dup_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) return _5_;

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



Which value do we return (replacing blank #5) when we want to duplicate an *empty* list?

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

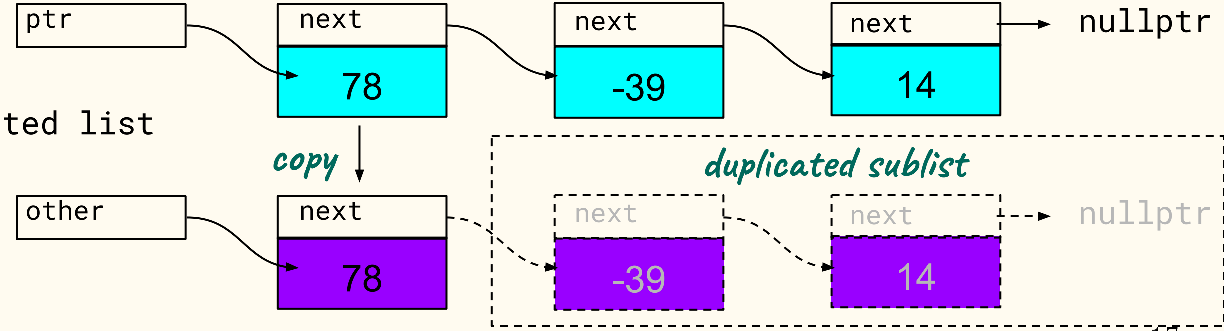
```
Node* dup_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) return _5_;

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



Linked lists as recursive data structures

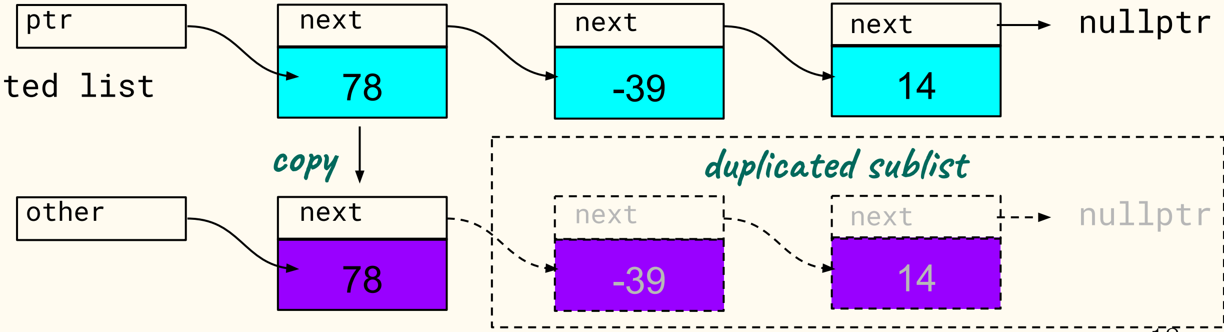
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {
    // base case
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



Linked lists as recursive data structures

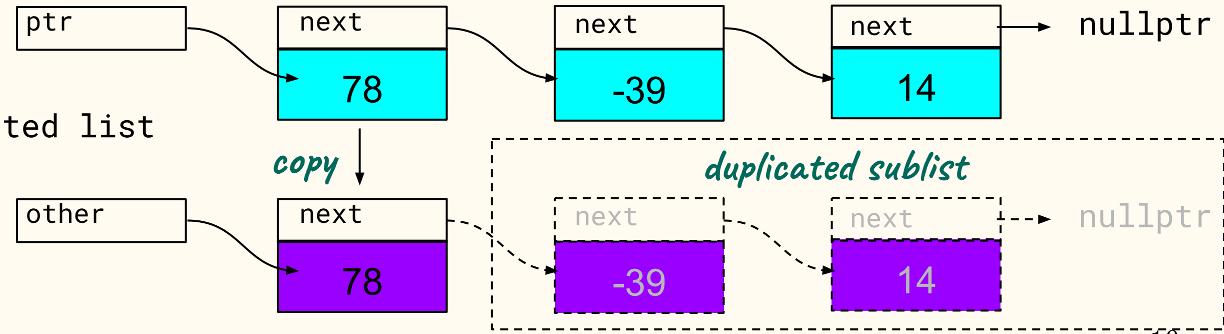
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    // make Node head of duplicated list
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

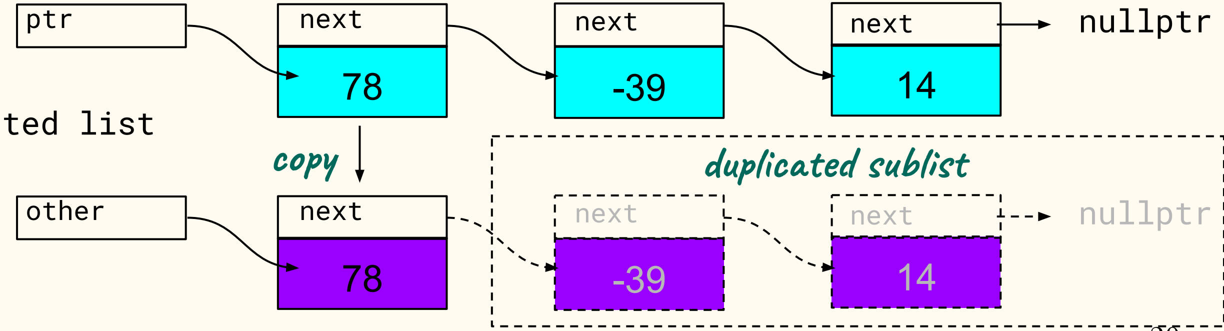
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    ___
    // make Node head of duplicated list
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

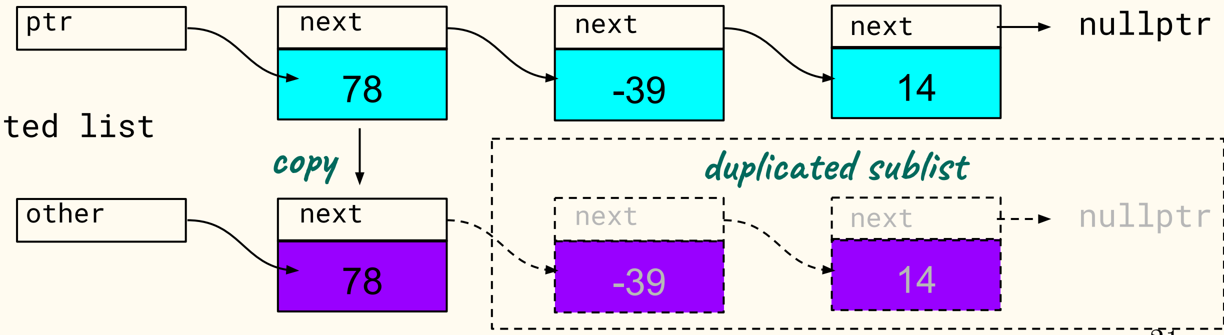
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = ___;
    // make Node head of duplicated list
}
```



Linked lists as recursive data structures

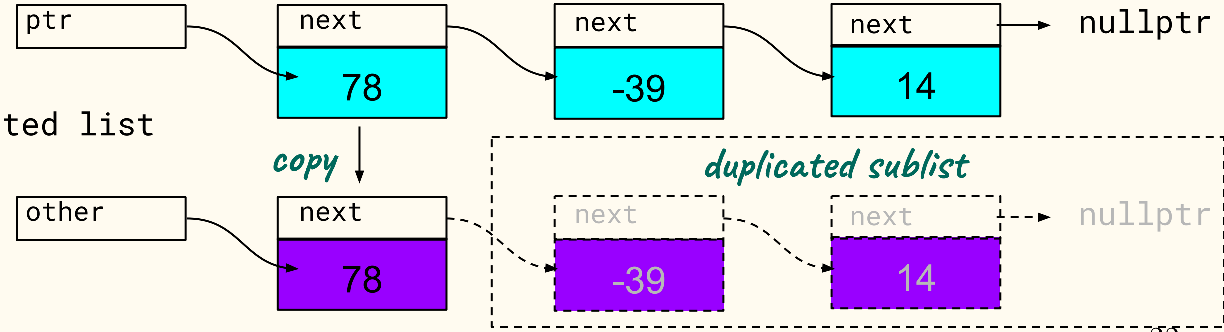
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = _6_;
    // make Node head of duplicated list
}
```



Which expression replaces blank #6 to instantiate a Node on the heap with the same `data` value as the `Node` pointed to by `ptr`?

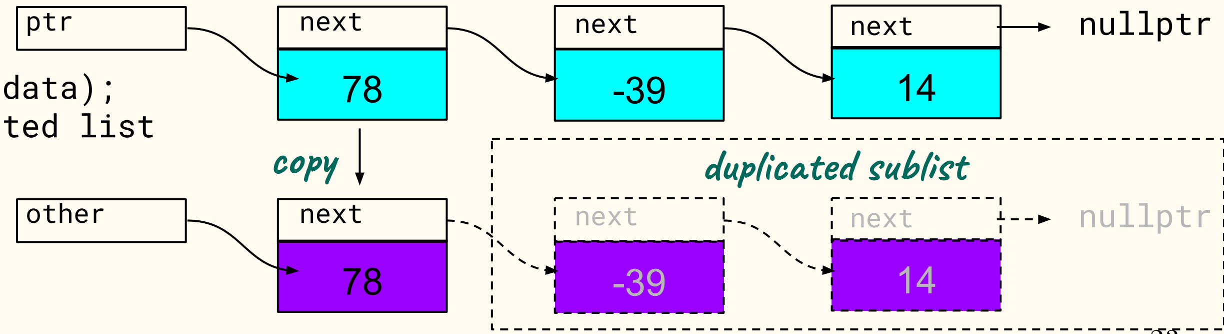
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};



Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = _6_;
    // make Node head of duplicated list
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

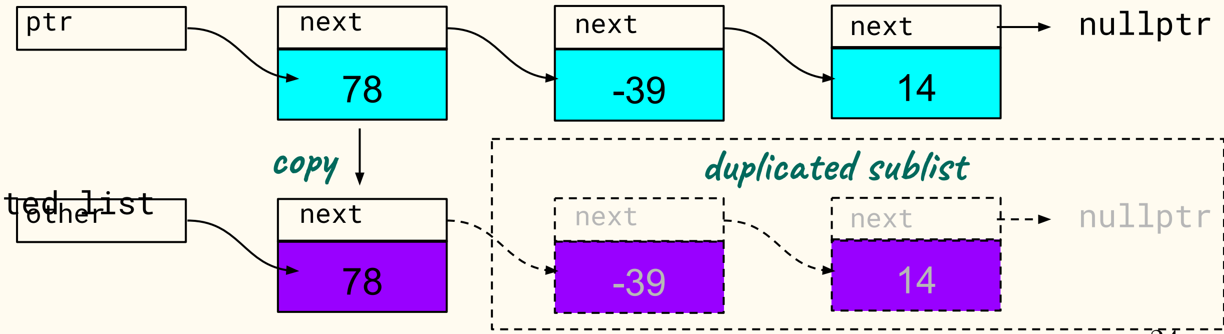
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(ptr->data);
    // make Node head of duplicated list
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

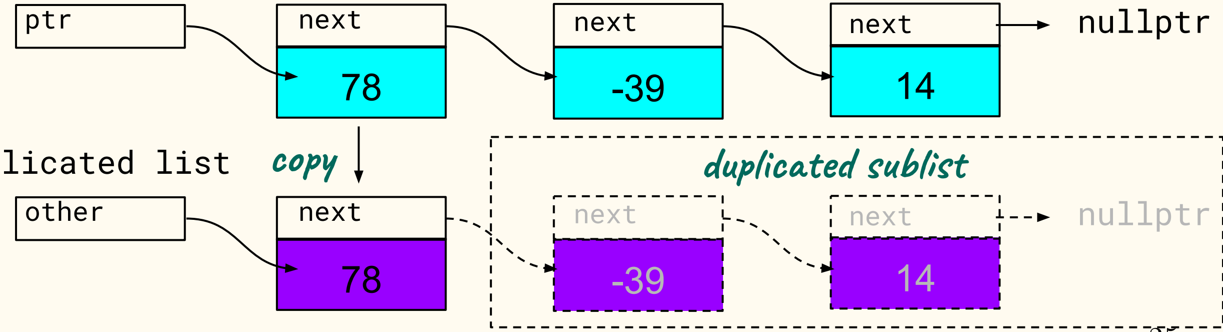
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data
    );
    // make Node head of duplicated list
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

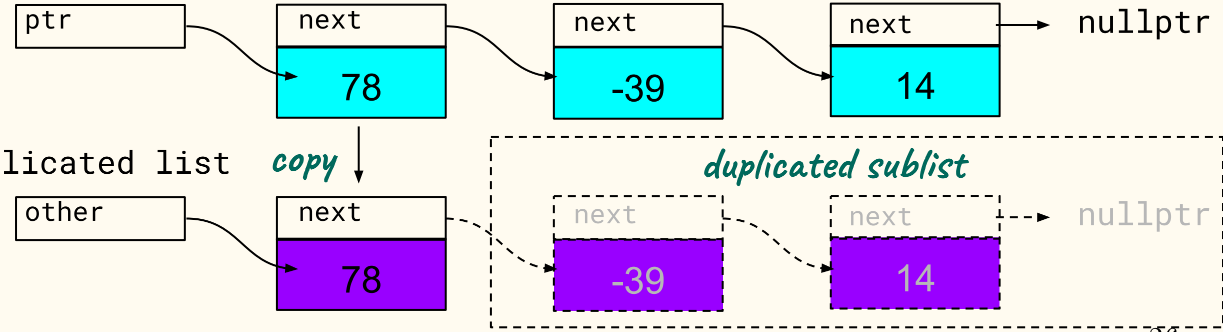
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        // make Node head of duplicated list
    );
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

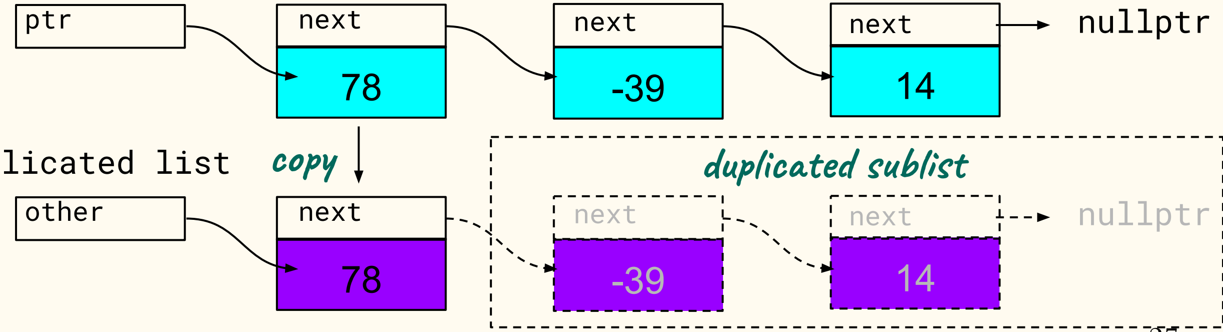
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        // make Node head of duplicated list
        ---
    );
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

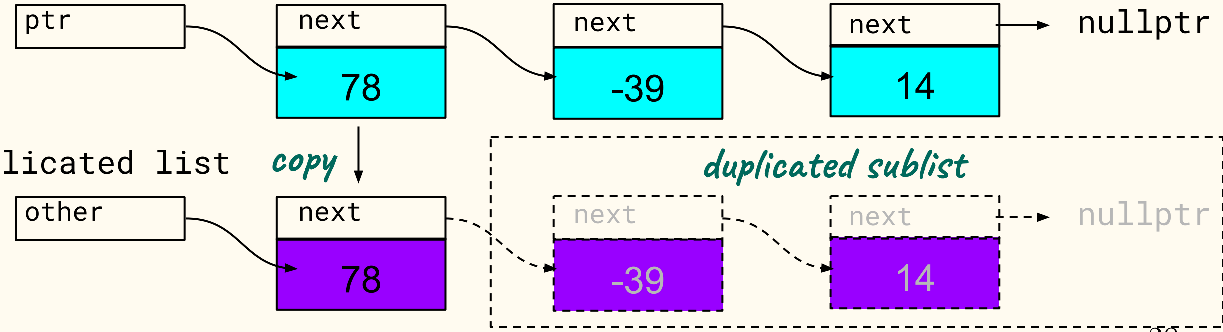
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        // make Node head of duplicated list
        _6_
    );
}
```



Which expression replaces blank #6 to create a duplicate list from the sublist following the current **Node** "pointed to" by **ptr**?

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

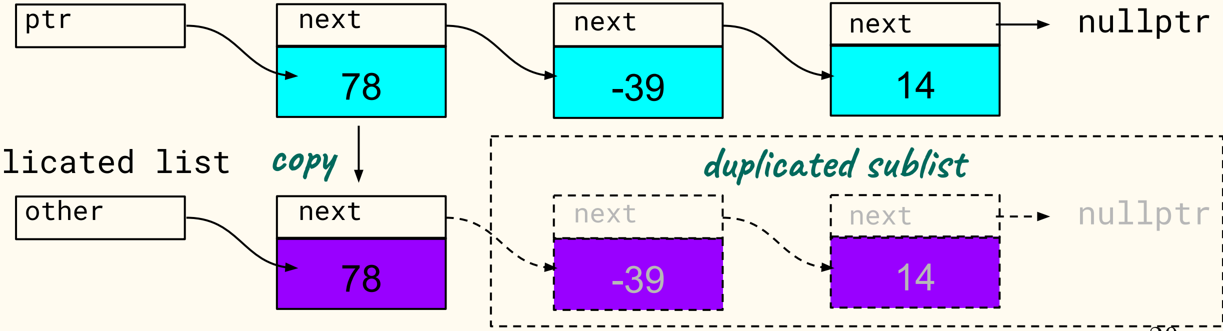
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        // make Node head of duplicated list
        _6_
    );
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

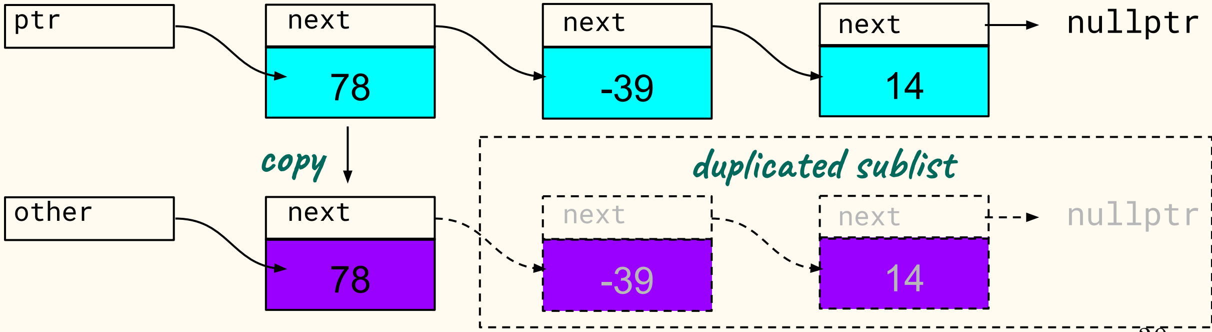
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        // make Node head of duplicated list
        dup_list(ptr->next)
    );
}
```



Linked lists as recursive data structures

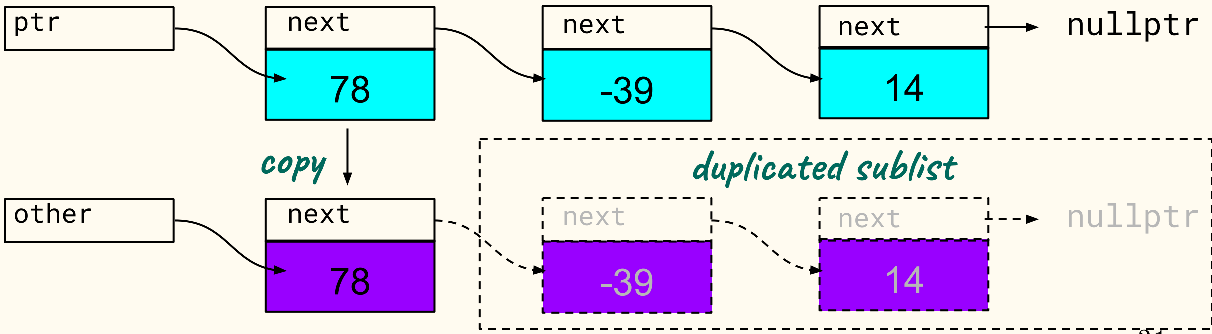
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    // create new Node
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
}
```



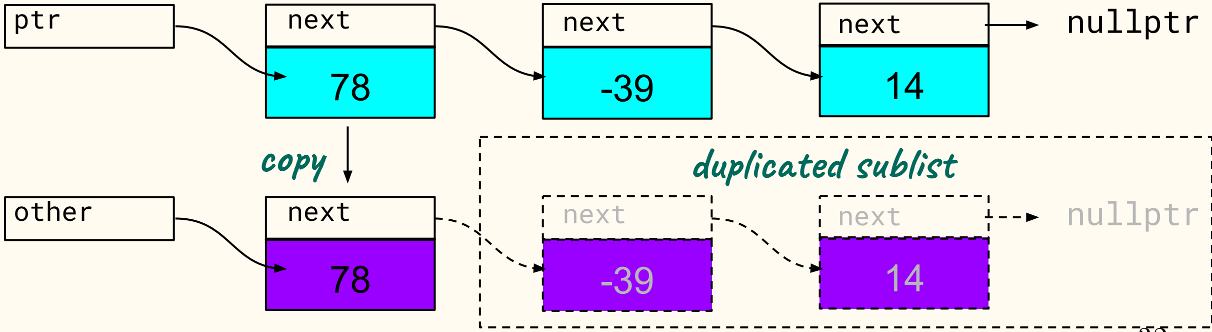
Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};



Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
}
```



Linked lists as recursive data structures

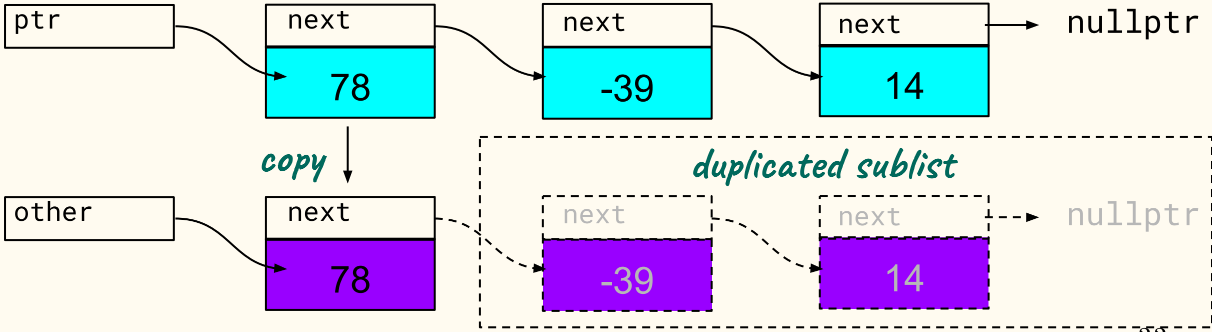
```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};



Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    ---
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

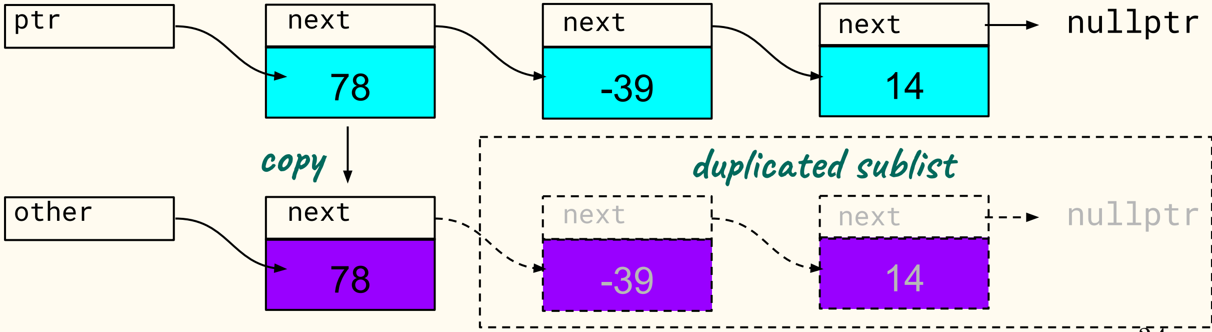
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return ___;
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

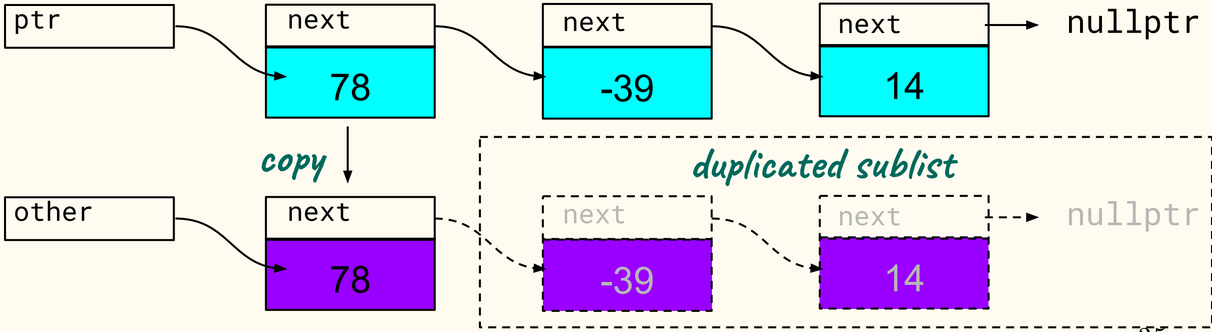
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return _7_;
}
```



What replaces blank #7 so that a `dup_list()` function call returns a pointer to the duplicated list?

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

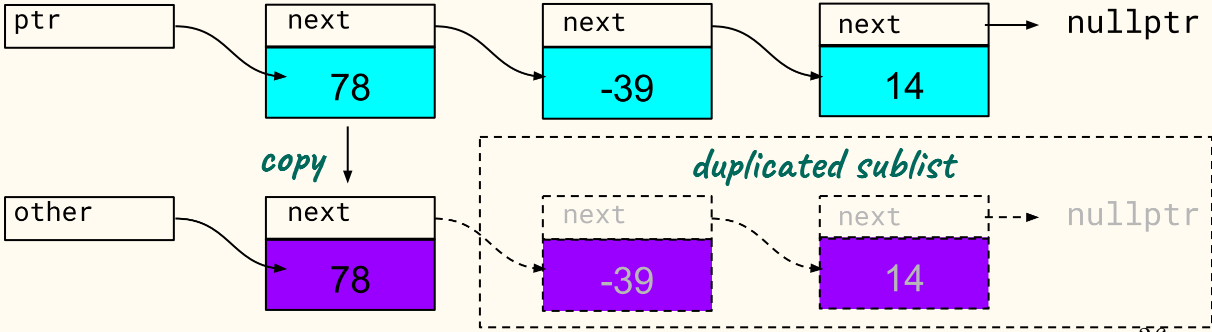
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return _7_;
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

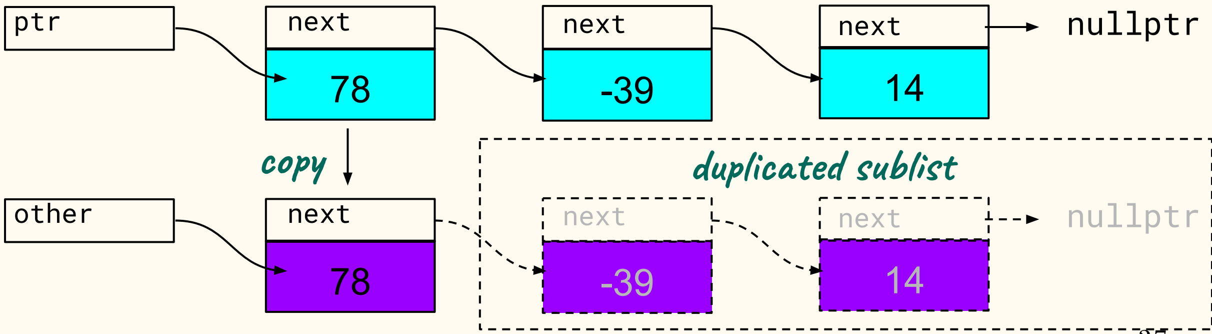
```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    // recursive case
    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return other;
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return other;
}
```



Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};


Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    Node* other = new Node(
        ptr->data,
        dup_list(ptr->next)
    );
    return other;
}
```

combine into single statement

Linked lists as recursive data structures

```
struct Node {
    Node(int data = 0, Node* next = nullptr) : data(data), next(next) {}
    int data;
    Node* next;
};
```

```
Node* dup_list(const Node* ptr) {
    if (ptr == nullptr) return nullptr;

    return new Node(
         ptr->data,
         dup_list(ptr->next)
    );
}
```

base case

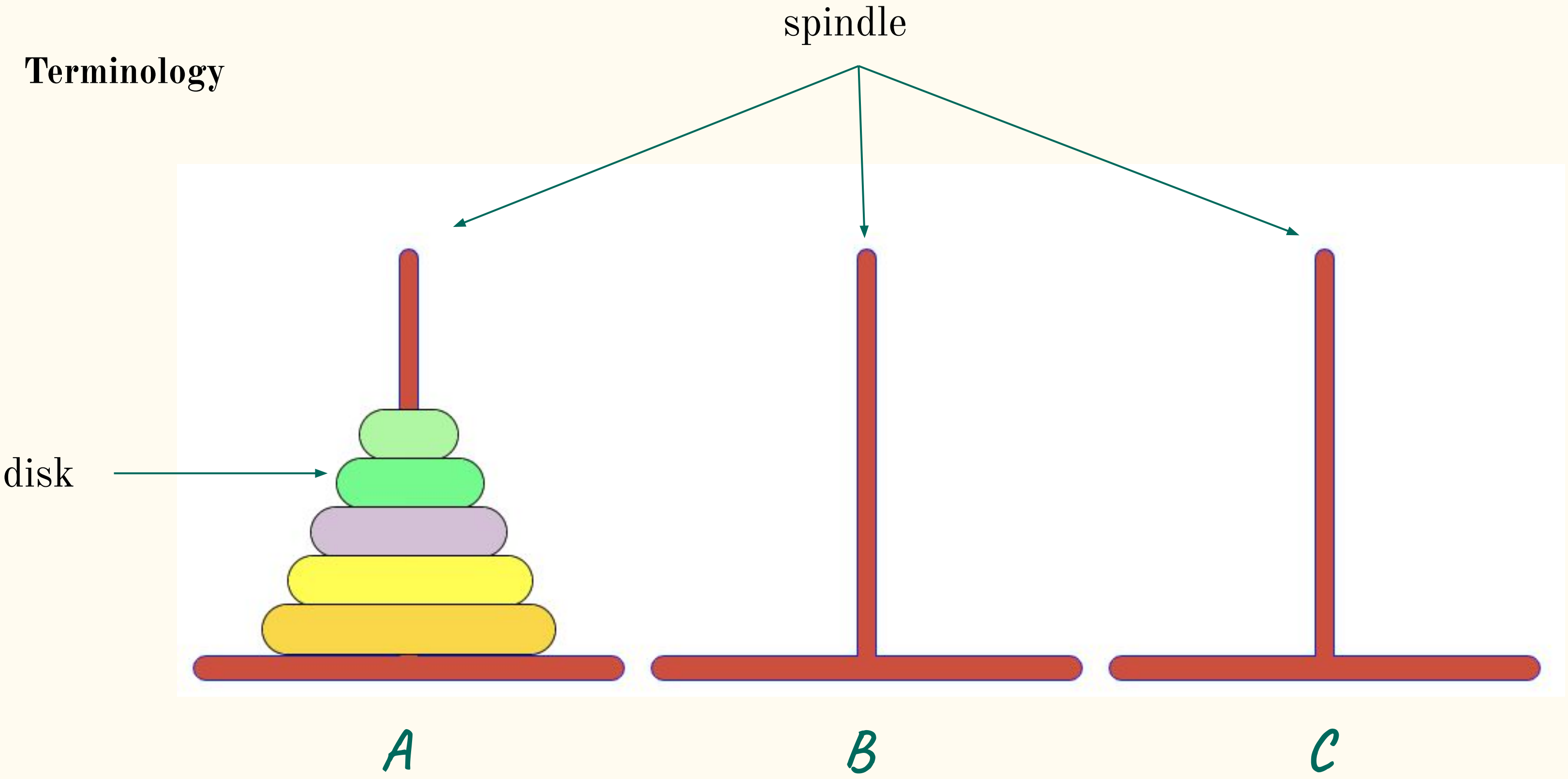
recursive case

Towers of Hanoi

A Towers of Hanoi visualization

Towers of Hanoi

Terminology

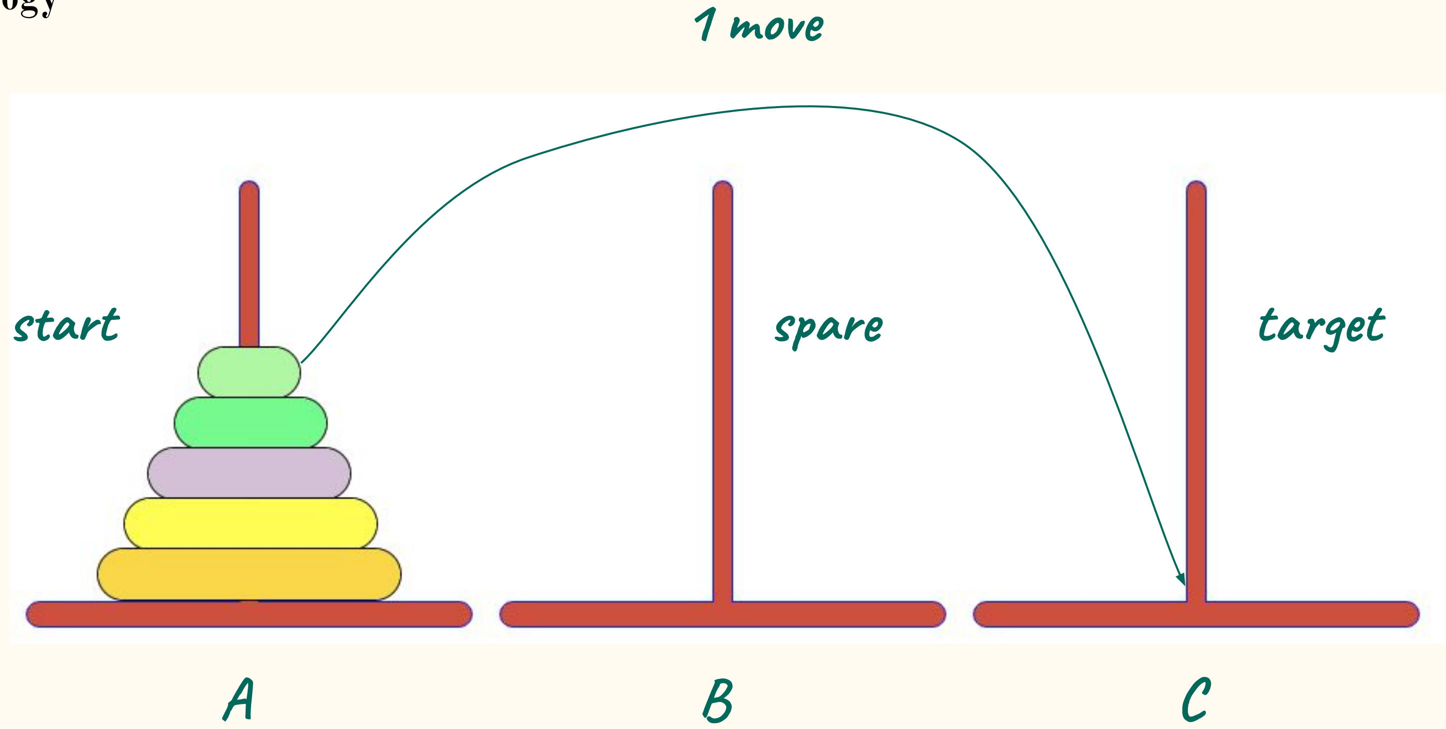


spindle

disk

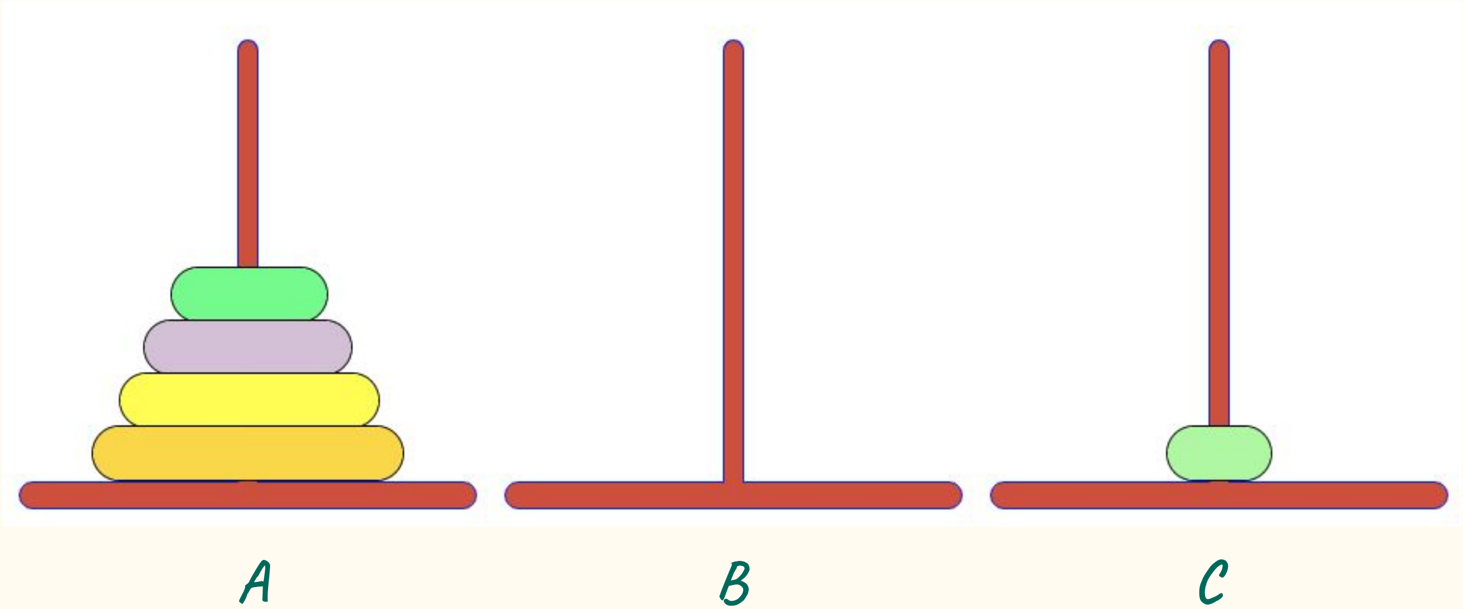
Towers of Hanoi

Terminology



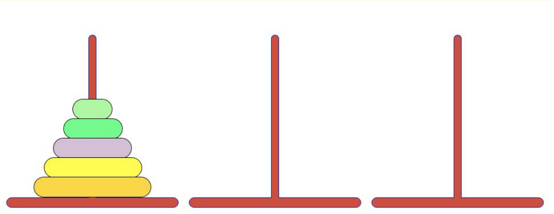
Towers of Hanoi

Terminology

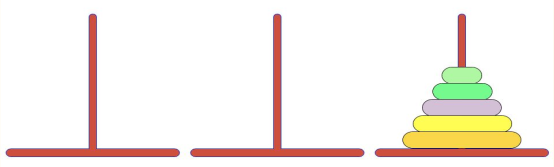


Towers of Hanoi

Terminology



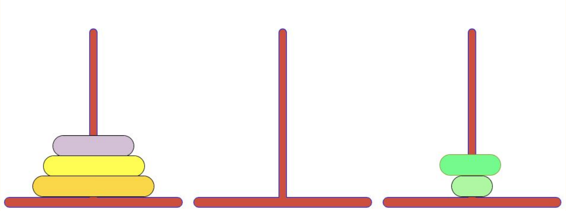
begin state



end state

Rules

- only one disk can be moved at a time
- no disk can rest on top of a smaller disk



illegal

Solving Towers of Hanoi

```
void towers() {}
```

Solving Towers of Hanoi

```
// n_disks: number of disks



void towers(int n_disks) {}
```

Solving Towers of Hanoi

```
// n_disks: number of disks
// start: start spindle


void towers(int n_disks, char start) {}
```

Solving Towers of Hanoi

```
// n_disks: number of disks
// start: start spindle
// target: target spindle (where disks are moving)

void towers(int n_disks, char start, char target) {}
```

Solving Towers of Hanoi

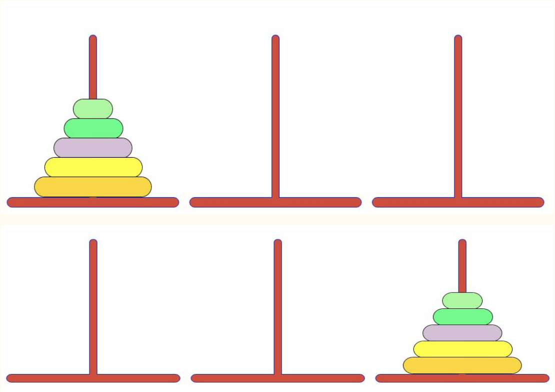
```
// n_disks: number of disks
// start: start spindle
// target: target spindle (where disks are moving)
// spare: "unused" spindle (not start/target)
void towers(int n_disks, char start, char target, char spare) {}







int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    // base case

    // recursive case

}







int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    // base case

    // recursive case
}
```



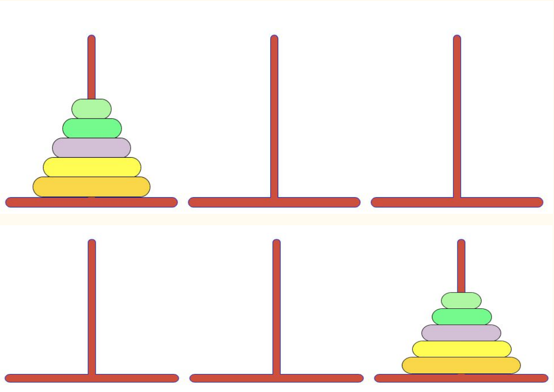
begin state

end state

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

What about the state of the start spindle differs between the begin state and the end state?

```
void towers(int n_disks, char start, char target, char spare) {
    // base case

    // recursive case
}
```



start spindle

begin state

end state

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    // base case
    if (n_disks == 0) ___;

    // recursive case
}









int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    // base case
    if (n_disks == 0) _8_;

    // recursive case
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Which statement replaces blank #8 when the base case is reached?

```
void towers(int n_disks, char start, char target, char spare) {
    // base case
    if (n_disks == 0) _8_;

    // recursive case
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    // base case
    if (n_disks == 0) return;

    // recursive case
}









int main() {

    towers(3, 'A', 'C', 'B');
}
```

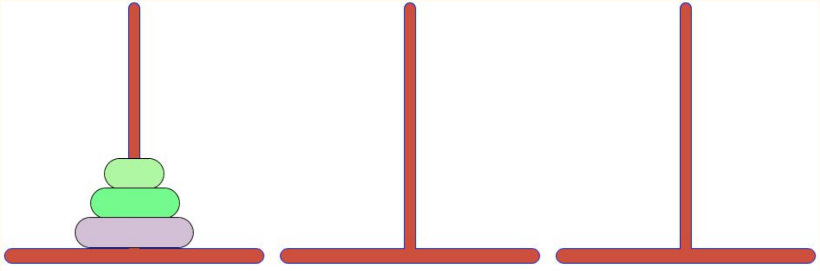
Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // recursive case
}
```

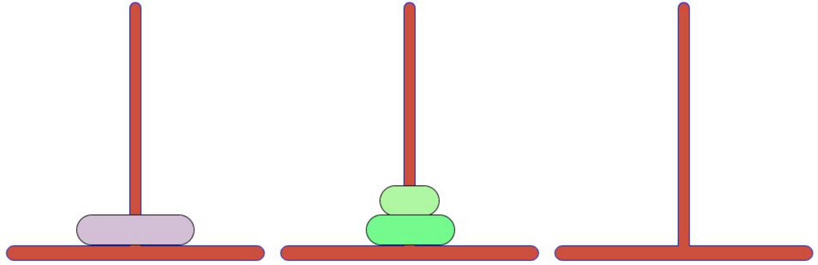
```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // recursive case
    // 1. move all but bottom disk to spare spindle
}
```



begin state

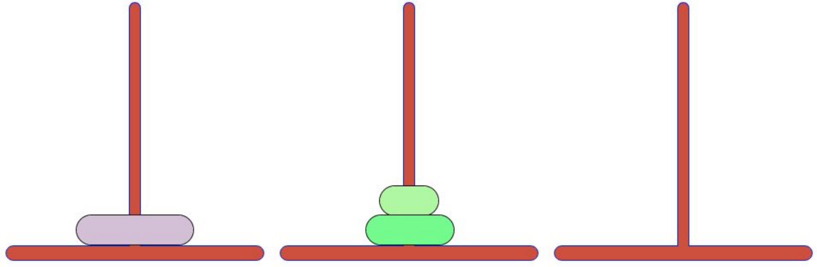


end state

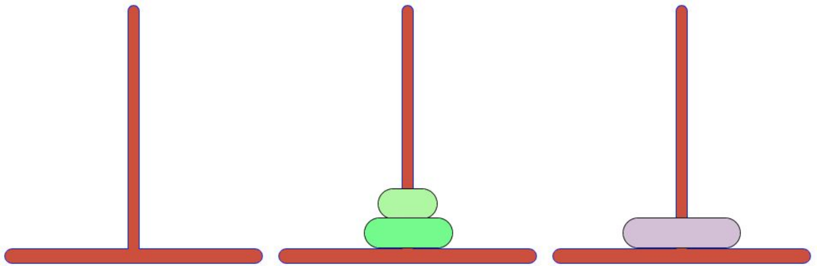
```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // recursive case
    // 1. move all but bottom disk to spare spindle
    // 2. move bottom disk to target spindle
}
```



begin state

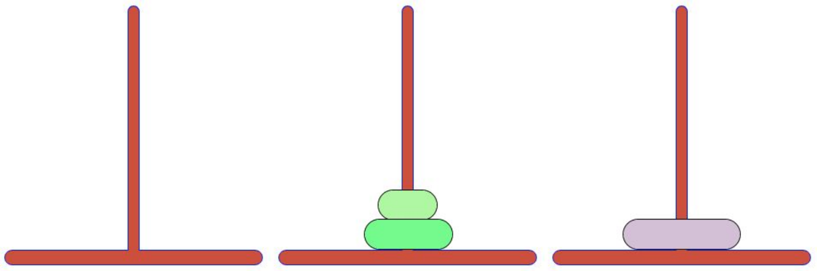


end state

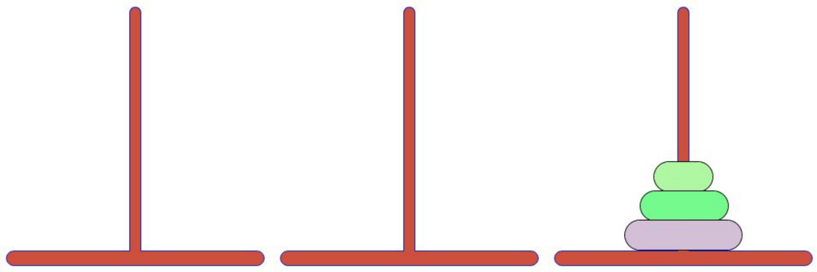
```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // recursive case
    // 1. move all but bottom disk to spare spindle
    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}
```



begin state



end state

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}








int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    ---

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}







int main() {

    towers(3, 'A', 'C', 'B');
}
```

What is the *name* of the function that will move the disks to the spare spindle?

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    ___

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}







int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers();

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}
```

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(___, ___, ___);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}
```

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(_9_, ___, ___, ___);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}






int main() {

    towers(3, 'A', 'C', 'B');
}
```

How many disks (in terms of `n_disks`) replaces blank #9 for the recursive call to the `towers()` function?

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(_9_, ___, ___, ___);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}






int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, ___, ___, ___);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}
```

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

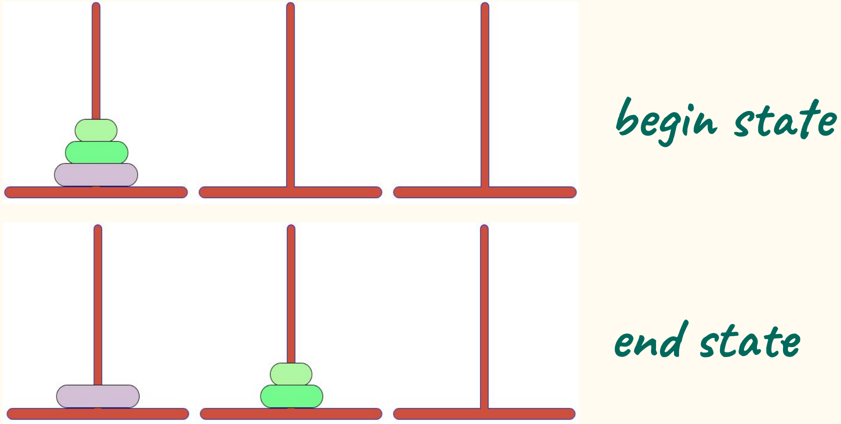
Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, _10_, ___, ___);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}
```

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Which spindle replaces blank #10 as the `start` spindle for the recursive function call?

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, _10_, ___, ___);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}
```



```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

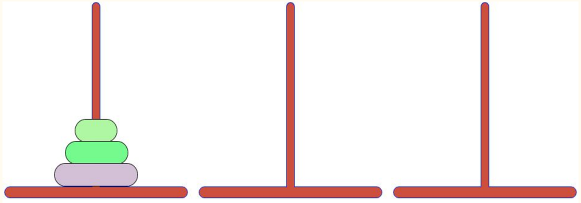
```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, ___, ___);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}







int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

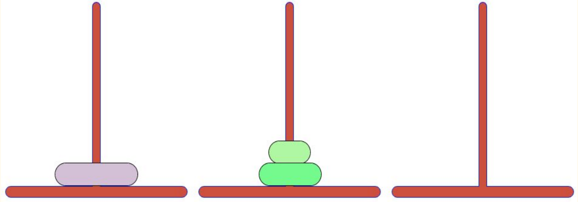
```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, _11_, ___);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}







int main() {

    towers(3, 'A', 'C', 'B');
}
```

Which spindle replaces blank #11 as the `target` spindle for the recursive function call?

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, _11_, ___);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}
```



begin state



end state

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, ___);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}


int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, _12_);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}
```

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Which spindle replaces blank #12 as the `spare` spindle for the recursive function call?

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, _12_);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}







int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle
    // 3. move all other disks on top of bottom disk
}
```

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
}


int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    ___
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

What is the *name* of the function that will move all of the disks on top of the bottom disk?

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    ___
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(___, ___, ___, ___);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(_13_, ___, ___, ___);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

How many disks (in terms of n_disks) must be moved on top of the bottom disk (replacing blank #13)?

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(_13_, ___, ___, ___);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

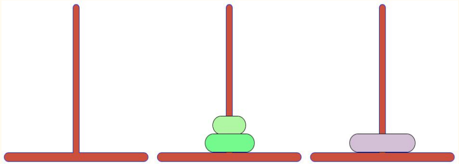
```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(n_disks - 1, ___, ___, ___);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

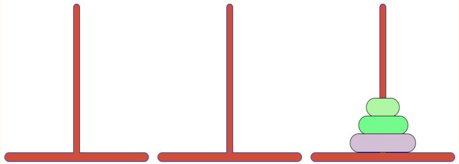
```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(n_disks - 1, _14_, ___, ___);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Which spindle replaces blank #14 as the `start` spindle for the recursive function call?

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(n_disks - 1, _14_, ___, ___);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```



begin state



end state

Solving Towers of Hanoi

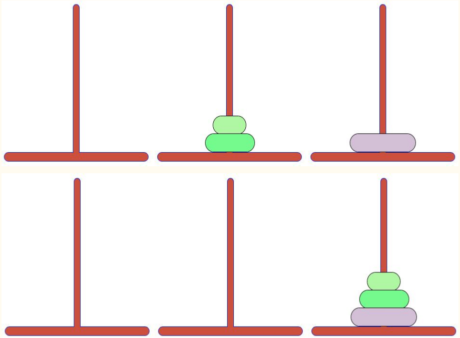
```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(n_disks - 1, spare, ___, ___);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(n_disks - 1, spare, _15_, ___);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Which spindle replaces blank #15 as the `target` spindle for the recursive function call?

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(n_disks - 1, spare, _15_, ___);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```



begin state

end state

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(n_disks - 1, spare, target, ___);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(n_disks - 1, spare, target, _16_);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Which spindle replaces blank #16 as the `spare` spindle for the recursive function call?

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(n_disks - 1, spare, target, _16_);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    // 1. move all but bottom disk to spare spindle
    towers(n_disks - 1, start, spare, target);

    // 2. move bottom disk to target spindle

    cout << "Moving disk: " << n_disks << " from spindle: " << start
         << " to spindle: " << target << endl;

    // 3. move all other disks on top of bottom disk
    towers(n_disks - 1, spare, target, start);
}

int main() {

    towers(3, 'A', 'C', 'B');
}
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    towers(n_disks - 1, start, spare, target);

    cout << "Moving disk: " << n << " from spindle: " << start
         << " to spindle: " << target << endl;

    towers(n_disks - 1, spare, target, start);
}
```

base case

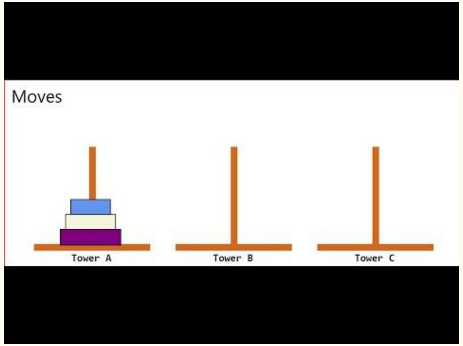
recursive case

```
int main() {

    towers(3, 'A', 'C', 'B');
}
```

```
% g++ --std=c++11 towers.cpp -o towers.o
% ./towers.o
Moving disk: 1 from spindle: A to spindle: C
Moving disk: 2 from spindle: A to spindle: B
Moving disk: 1 from spindle: C to spindle: B
Moving disk: 3 from spindle: A to spindle: C
Moving disk: 1 from spindle: B to spindle: A
Moving disk: 2 from spindle: B to spindle: C
Moving disk: 1 from spindle: A to spindle: C
```

Solving Towers of Hanoi

```
void towers(int n_disks, char start, char target, char spare) {
    if (n_disks == 0) return;

    towers(n_disks - 1, start, spare, target);

    cout << "Moving disk: " << n << " from spindle: " << start
         << " to spindle: " << target << endl;

    towers(n_disks - 1, spare, target, start);
}




int main() {

    towers(3, 'A', 'C', 'B');
}
```



Recursive strategy

Printing digits in an integer

an int

314159

How can each digit be output without string conversion??

Printing digits in an integer

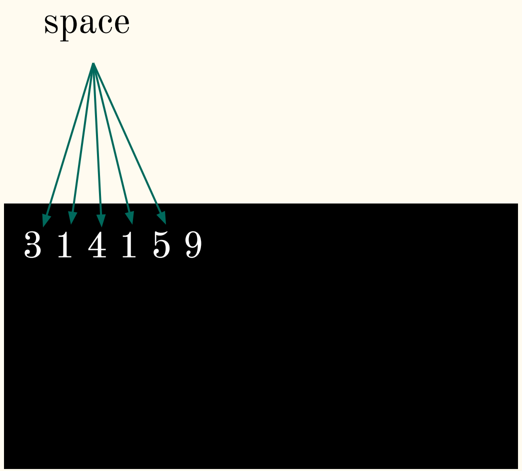
314159

Printing digits in an integer

314159

Challenge: int is atomic (not composite) type

Solution: separate into digits programmatically



space

```
3 1 4 1 5 9
```

Printing digits in an integer

$$\mathbf{314159} \xrightarrow{?} \mathbf{31415} \xrightarrow{?} \mathbf{3141} \xrightarrow{?} \mathbf{314} \xrightarrow{?} \mathbf{31} \xrightarrow{?} \mathbf{3}$$

Which operation can be used to "remove" one digit from the end of the integer?

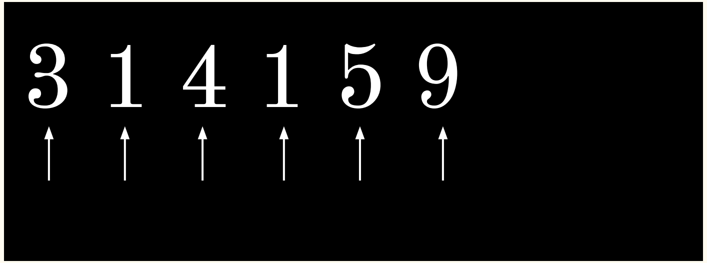
$$314159 \xrightarrow{?} 31415 \xrightarrow{?} 3141 \xrightarrow{?} 314 \xrightarrow{?} 31 \xrightarrow{?} 3$$

Printing digits in an integer

$$314159 \xrightarrow{?} 31415 \xrightarrow{?} 3141 \xrightarrow{?} 314 \xrightarrow{?} 31 \xrightarrow{?} 3$$

Printing digits in an integer

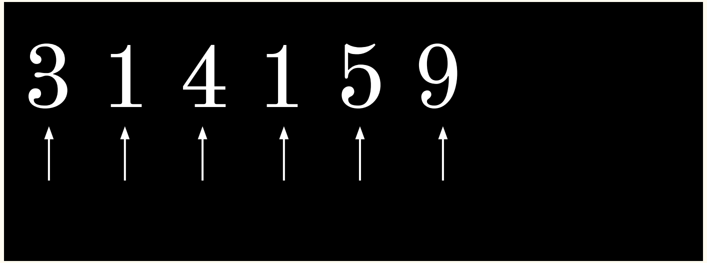
$$\begin{tikzcd}
314159 \arrow[r, "/10"] & 31415 \arrow[r, "/10"] & 3141 \arrow[r, "/10"] & 314 \arrow[r, "/10"] & 31 \arrow[r, "/10"] & 3 \\
? & ? & ? & ? & ? & ?
\end{tikzcd}$$



```
3 1 4 1 5 9
```

Which operation evaluates to the digit in the units position of each integer?

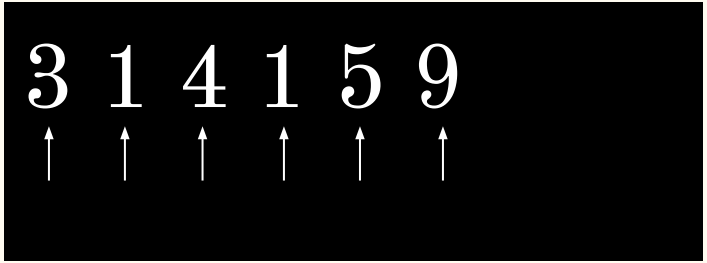
$$\begin{tikzcd}
314159 \arrow[r, "/\,10"] & 31415 \arrow[r, "/\,10"] & 3141 \arrow[r, "/\,10"] & 314 \arrow[r, "/\,10"] & 31 \arrow[r, "/\,10"] & 3 \\
? & ? & ? & ? & ? & ?
\end{tikzcd}$$



$$\begin{array}{cccccc} 3 & 1 & 4 & 1 & 5 & 9 \\ \uparrow & \uparrow & \uparrow & \uparrow & \uparrow & \uparrow \end{array}$$

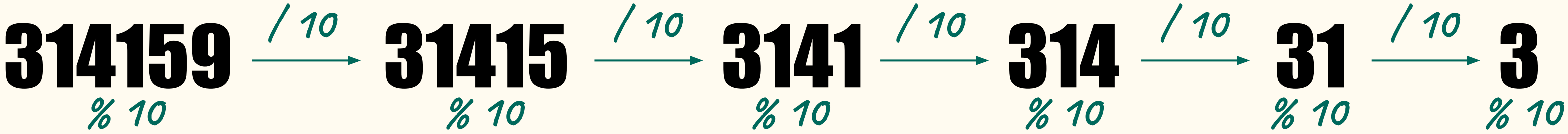
Printing digits in an integer

$$\underset{?}{314159} \xrightarrow{/\,10} \underset{?}{31415} \xrightarrow{/\,10} \underset{?}{3141} \xrightarrow{/\,10} \underset{?}{314} \xrightarrow{/\,10} \underset{?}{31} \xrightarrow{/\,10} \underset{?}{3}$$



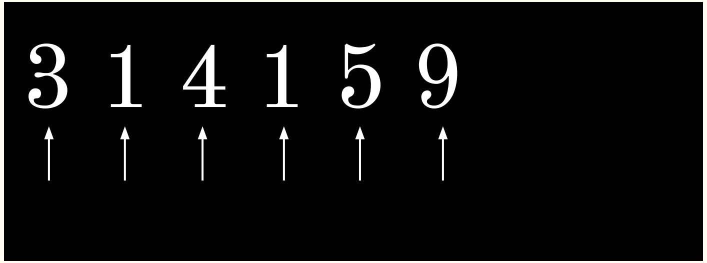
```
3 1 4 1 5 9
```

Printing digits in an integer



314159 \xrightarrow{/ 10} 31415 \xrightarrow{/ 10} 3141 \xrightarrow{/ 10} 314 \xrightarrow{/ 10} 31 \xrightarrow{/ 10} 3

% 10 % 10 % 10 % 10 % 10 % 10

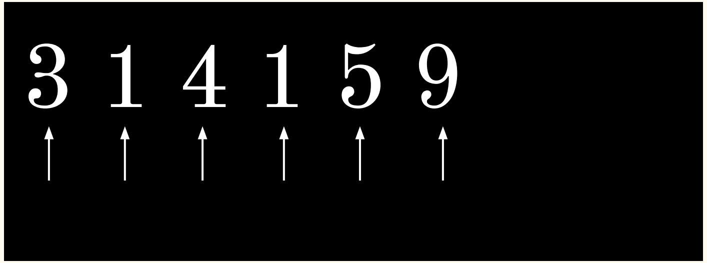


```
3 1 4 1 5 9
```

```
void print_digits(int num) {
    // base case

    // recursive case
}
```

What is the base case for the `print_digits()` function?

$$\underset{\%\,10}{314159} \xrightarrow{/\,10} \underset{\%\,10}{31415} \xrightarrow{/\,10} \underset{\%\,10}{3141} \xrightarrow{/\,10} \underset{\%\,10}{314} \xrightarrow{/\,10} \underset{\%\,10}{31} \xrightarrow{/\,10} \underset{\%\,10}{3}$$

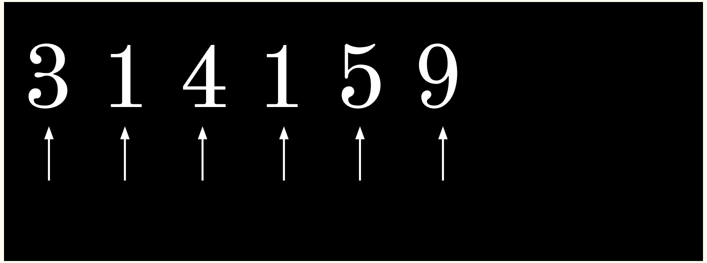


```
3 1 4 1 5 9
```

```
void print_digits(int num) {
    // base case

    // recursive case
}
```

Printing digits in an integer

$$\underset{\% \, 10}{314159} \xrightarrow{/\,10} \underset{\% \, 10}{31415} \xrightarrow{/\,10} \underset{\% \, 10}{3141} \xrightarrow{/\,10} \underset{\% \, 10}{314} \xrightarrow{/\,10} \underset{\% \, 10}{31} \xrightarrow{/\,10} \underset{\% \, 10}{3}$$

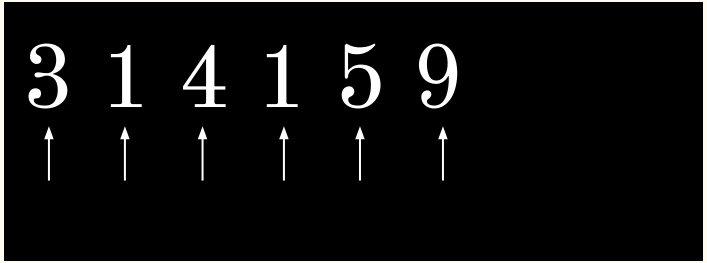


```
3 1 4 1 5 9
```

```
void print_digits(int num) {
    // base case
    ---

    // recursive case
}
```

Printing digits in an integer

$$\underset{\%\,10}{314159} \xrightarrow{/\,10} \underset{\%\,10}{31415} \xrightarrow{/\,10} \underset{\%\,10}{3141} \xrightarrow{/\,10} \underset{\%\,10}{314} \xrightarrow{/\,10} \underset{\%\,10}{31} \xrightarrow{/\,10} \underset{\%\,10}{3}$$



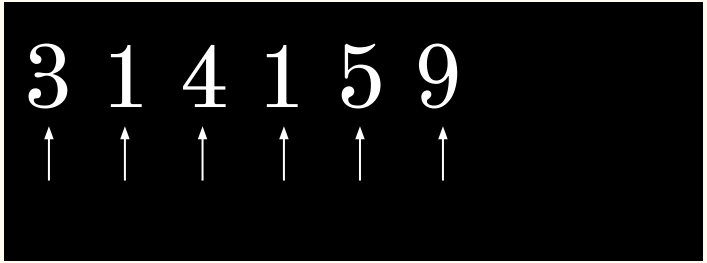
```
3 1 4 1 5 9
```

```
void print_digits(int num) {
    // base case
    if (___ < ___) cout << num % 10 << ' ';

    // recursive case
}
```

Printing digits in an integer

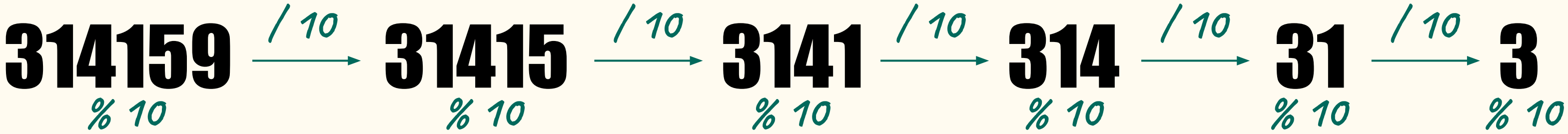
$$\underset{\% 10}{314159} \xrightarrow{/10} \underset{\% 10}{31415} \xrightarrow{/10} \underset{\% 10}{3141} \xrightarrow{/10} \underset{\% 10}{314} \xrightarrow{/10} \underset{\% 10}{31} \xrightarrow{/10} \underset{\% 10}{3}$$

```
void print_digits(int num) {
    // base case
    if (num < ___) cout << num % 10 << ' ';

    // recursive case
}
```

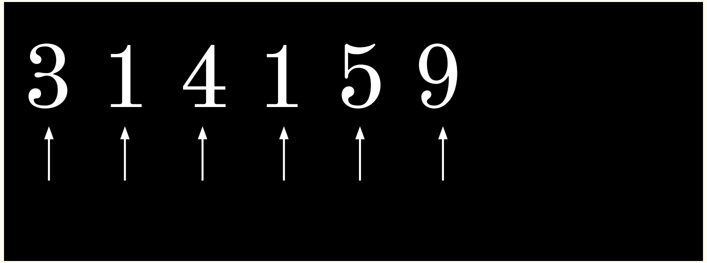


```
3 1 4 1 5 9
```

Printing digits in an integer



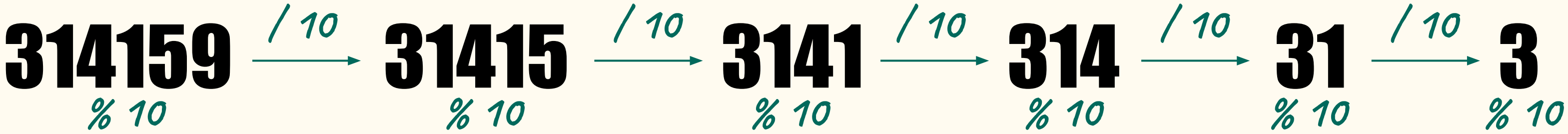
314159 → 31415 → 3141 → 314 → 31 → 3

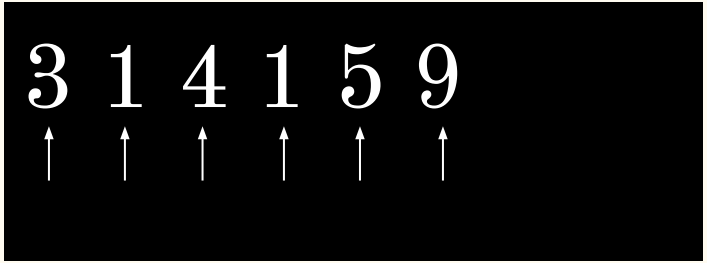


```
3 1 4 1 5 9
```

```
void print_digits(int num) {
    // base case
    if (num < _2_) cout << num % 10 << ' ';

    // recursive case
}
```

Which value replaces blank #2 for identifying the base case (when num is a single digit)?

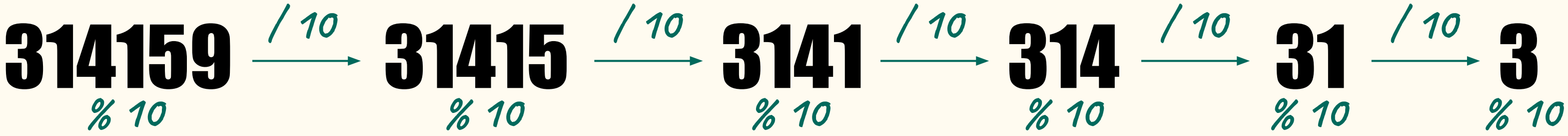




```
3 1 4 1 5 9
```

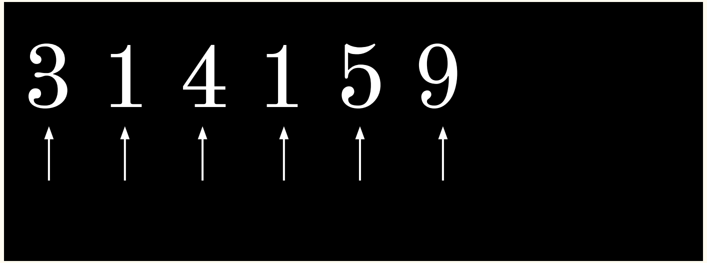
```
void print_digits(int num) {
    // base case
    if (num < _2_) cout << num % 10 << ' ';

    // recursive case
}
```

Printing digits in an integer



$$314159 \xrightarrow{/\ 10} 31415 \xrightarrow{/\ 10} 3141 \xrightarrow{/\ 10} 314 \xrightarrow{/\ 10} 31 \xrightarrow{/\ 10} 3$$

% 10 % 10 % 10 % 10 % 10 % 10

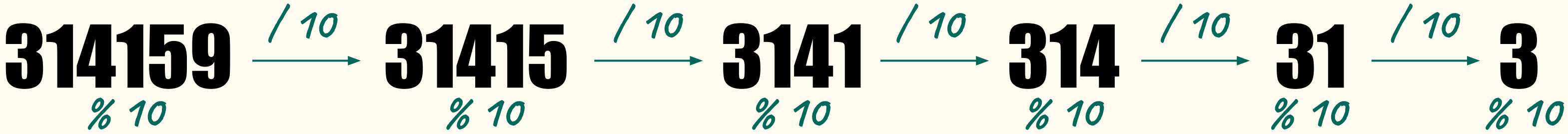


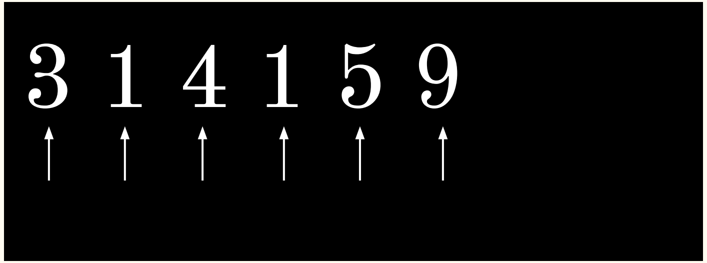
```
3 1 4 1 5 9
↑ ↑ ↑ ↑ ↑ ↑
```

```
void print_digits(int num) {
    // base case
    if (num < 10) cout << num % 10 << ' ';

    // recursive case
}
```

unnecessary

Printing digits in an integer



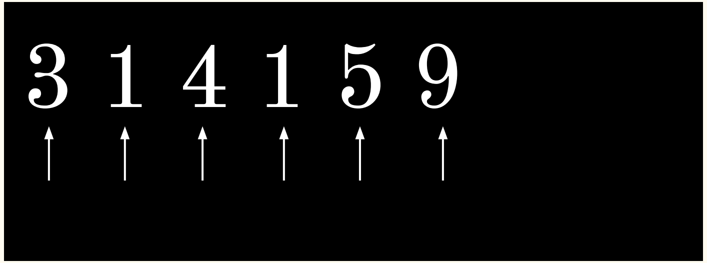


```
3 1 4 1 5 9
```

```
void print_digits(int num) {
    // base case
    if (num < 10) cout << num << ' ';

    // recursive case
}
```

Printing digits in an integer

$$\underset{\%\,10}{314159} \xrightarrow{/\,10} \underset{\%\,10}{31415} \xrightarrow{/\,10} \underset{\%\,10}{3141} \xrightarrow{/\,10} \underset{\%\,10}{314} \xrightarrow{/\,10} \underset{\%\,10}{31} \xrightarrow{/\,10} \underset{\%\,10}{3}$$



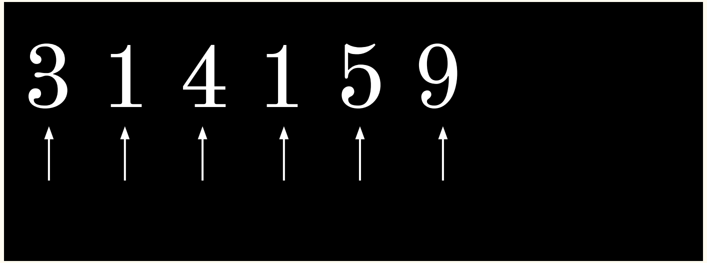
```
3 1 4 1 5 9
```

```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        // recursive case
    }
}
```

num is integer with 2 or more digits

Printing digits in an integer

$$\underset{\%\,10}{314159} \xrightarrow{/\,10} \underset{\%\,10}{31415} \xrightarrow{/\,10} \underset{\%\,10}{3141} \xrightarrow{/\,10} \underset{\%\,10}{314} \xrightarrow{/\,10} \underset{\%\,10}{31} \xrightarrow{/\,10} \underset{\%\,10}{3}$$

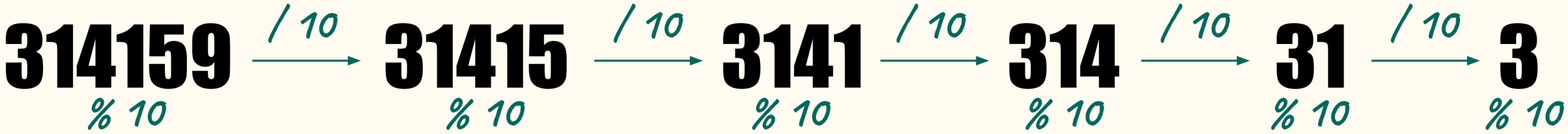


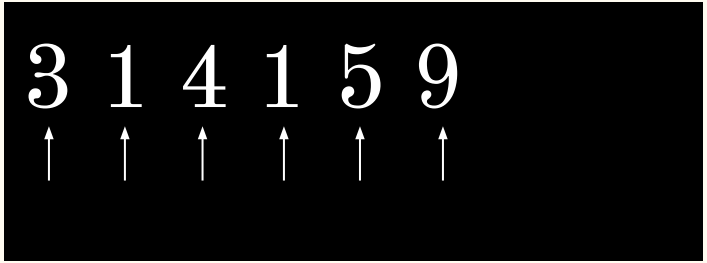
```
3 1 4 1 5 9
```

```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        // recursive case
        ---
    }
}
```

num is integer with 2 or more digits

Printing digits in an integer





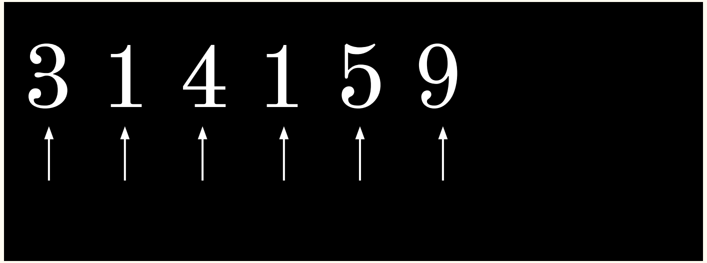
```
3 1 4 1 5 9
```

```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        // recursive case
        cout << ___ << ' ';
    }
}
```

num is integer with 2 or more digits

Printing digits in an integer

$$\underset{\% 10}{314159} \xrightarrow{/ 10} \underset{\% 10}{31415} \xrightarrow{/ 10} \underset{\% 10}{3141} \xrightarrow{/ 10} \underset{\% 10}{314} \xrightarrow{/ 10} \underset{\% 10}{31} \xrightarrow{/ 10} \underset{\% 10}{3}$$



```
3 1 4 1 5 9
```

```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        // recursive case
        cout << _3_ << ' ';
    }
}
```

num is integer with 2 or more digits

Which expression replaces blank #3 to output the digit in the unit (rightmost) position of the integer?

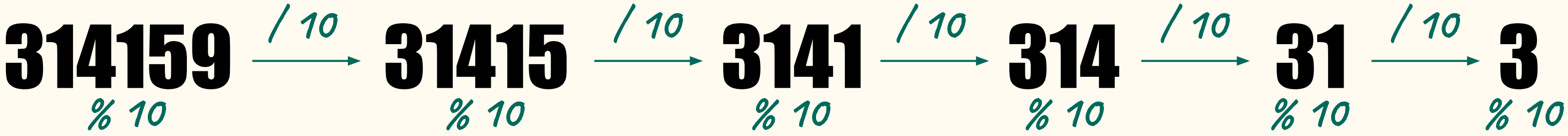
$$\underset{\%\,10}{314159} \xrightarrow{/\,10} \underset{\%\,10}{31415} \xrightarrow{/\,10} \underset{\%\,10}{3141} \xrightarrow{/\,10} \underset{\%\,10}{314} \xrightarrow{/\,10} \underset{\%\,10}{31} \xrightarrow{/\,10} \underset{\%\,10}{3}$$

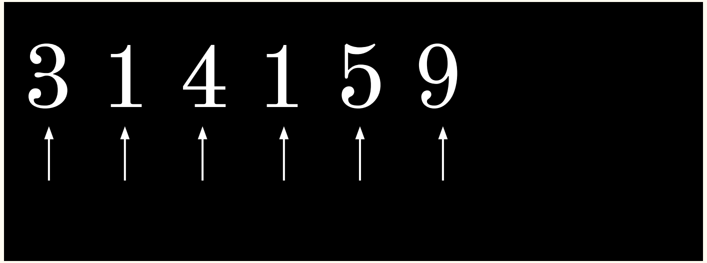
```
3 1 4 1 5 9
↑ ↑ ↑ ↑ ↑ ↑
```

```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        // recursive case
        cout << _3_ << ' ';
    }
}
```

num is integer with 2 or more digits

Printing digits in an integer



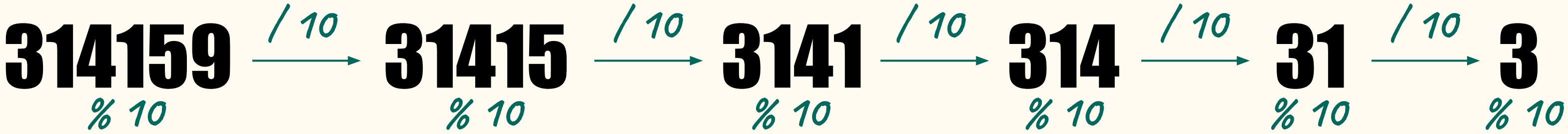


```
3 1 4 1 5 9
```

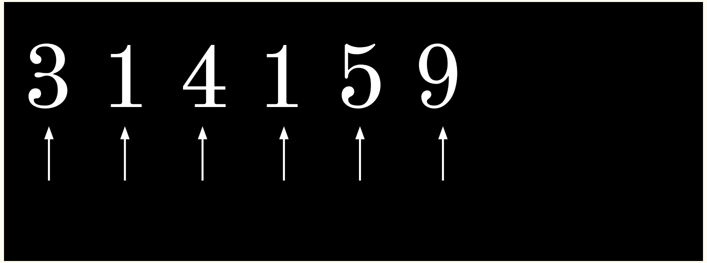
```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        // recursive case
        cout << num % 10 << ' ';
    }
}
```

need a recursive call on a smaller version of the problem

Which recursive function call will solve the print digits problem on a smaller version of the problem?



314159 → 31415 → 3141 → 314 → 31 → 3



```
3 1 4 1 5 9
↑ ↑ ↑ ↑ ↑ ↑
```

```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        // recursive case
        cout << num % 10 << ' ';
    }
}
```

need a recursive call on a smaller version of the problem

Printing digits in an integer

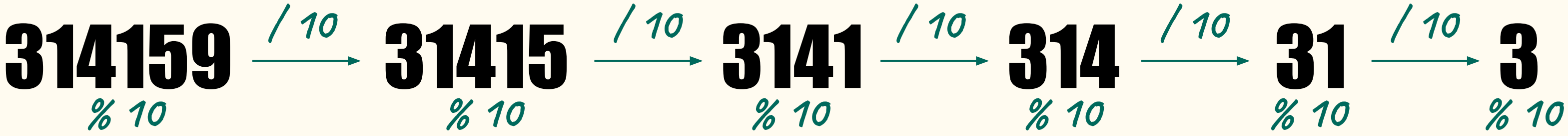
$$\underset{\%\ 10}{314159} \xrightarrow{/\ 10} \underset{\%\ 10}{31415} \xrightarrow{/\ 10} \underset{\%\ 10}{3141} \xrightarrow{/\ 10} \underset{\%\ 10}{314} \xrightarrow{/\ 10} \underset{\%\ 10}{31} \xrightarrow{/\ 10} \underset{\%\ 10}{3}$$

```
print_digits(num / 10);
```

```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        // recurse, then print
        cout << num % 10 << ' ';
        // print, then recurse
    }
}
```

```
3 1 4 1 5 9
↑ ↑ ↑ ↑ ↑ ↑
```

In which location do we place the recursive `print_digits()` function call?



314159 → 31415 → 3141 → 314 → 31 → 3

/ 10

% 10

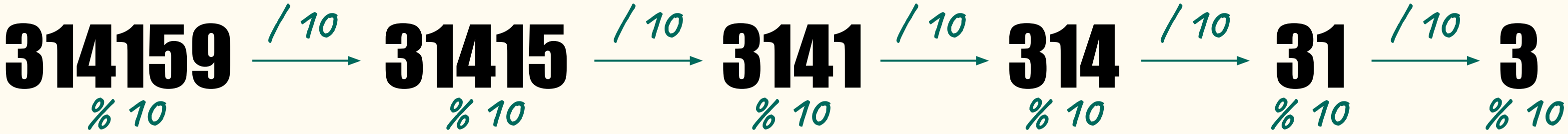
`print_digits(num / 10);`

```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        // recurse, then print
        cout << num % 10 << ' ';

        // print, then recurse
    }
}
```

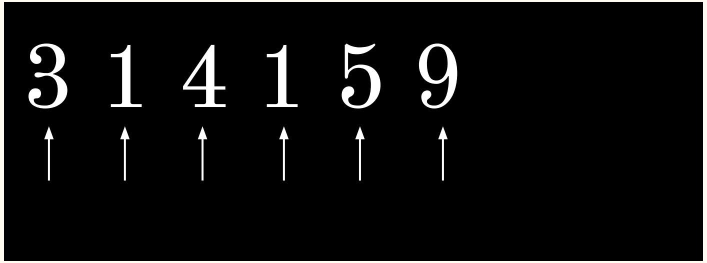
A. before outputting the units digit

B. after outputting the units digit

Printing digits in an integer

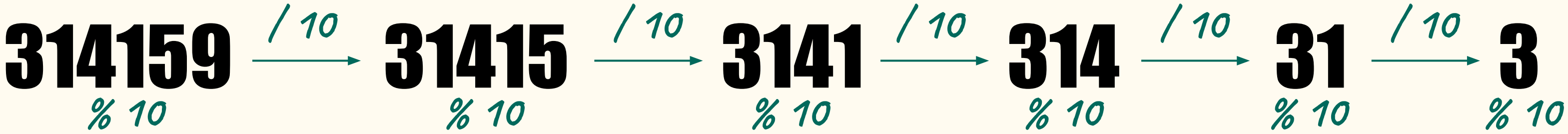


```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        print_digits(num / 10);
        cout << num % 10 << ' ';
    }
}
```



```
3 1 4 1 5 9
```

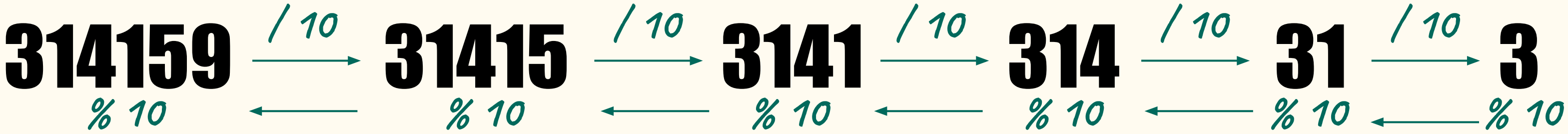
What would be output by this function in the case that the `print_digits()` recursive function call occurred after outputting the unit digit?

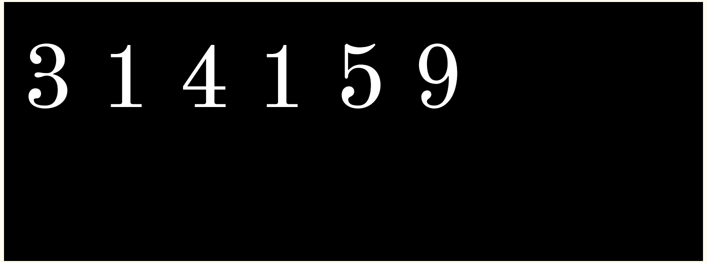


```
3 1 4 1 5 9
↑ ↑ ↑ ↑ ↑ ↑
```

```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        print_digits(num / 10);
        cout << num % 10 << ' ';
    }
}
```

Printing digits in an integer

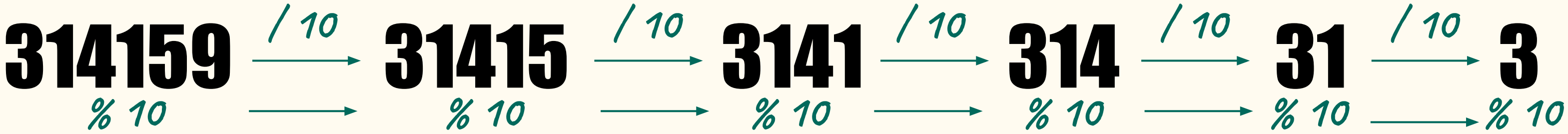




```
3 1 4 1 5 9
```

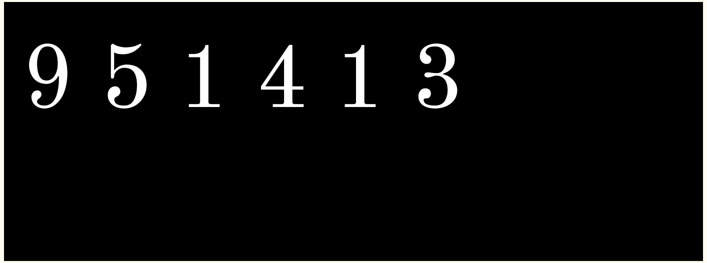
```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        print_digits(num / 10);
        cout << num % 10 << ' ';
    }
}
```

Printing digits in an integer



314159 / 10 → 31415 / 10 → 3141 / 10 → 314 / 10 → 31 / 10 → 3

% 10 → % 10 → % 10 → % 10 → % 10 → % 10

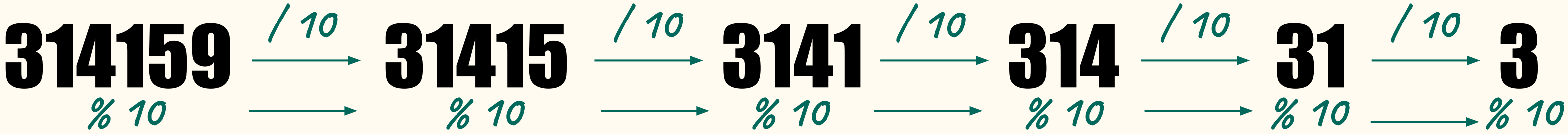


```
9 5 1 4 1 3
```

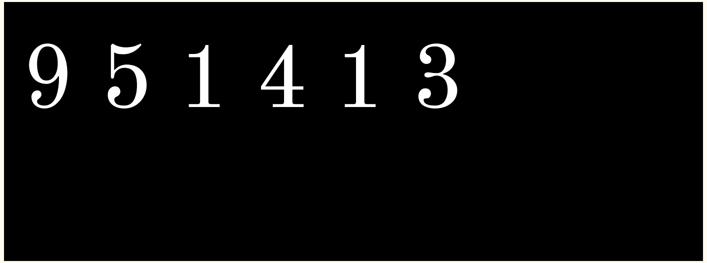
```
void print_digits(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        cout << num % 10 << ' ';
        print_digits(num / 10);
    }
}
```

flipped

Printing digits in an integer



Instruction order often matters

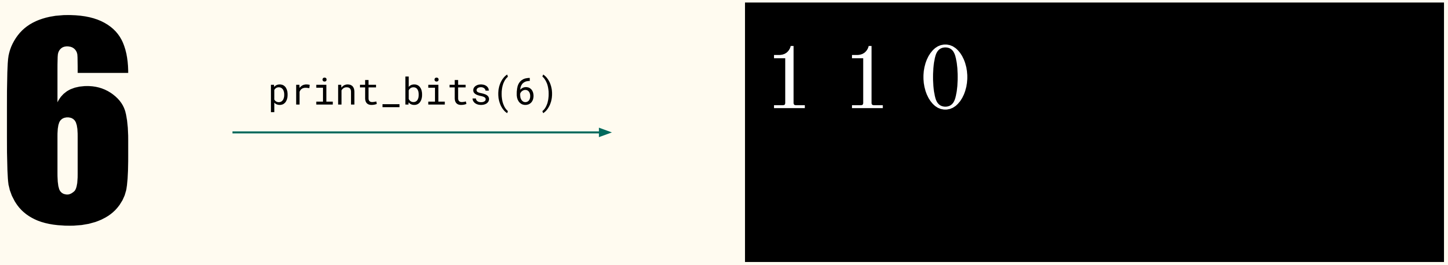


```
9 5 1 4 1 3
```

```
void print_digits_rev(int num) {
    if (num < 10) {
        cout << num << ' ';
    } else {
        cout << num % 10 << ' ';
        print_digits_rev(num / 10);
    }
}
```

tail recursion

Printing bits



6

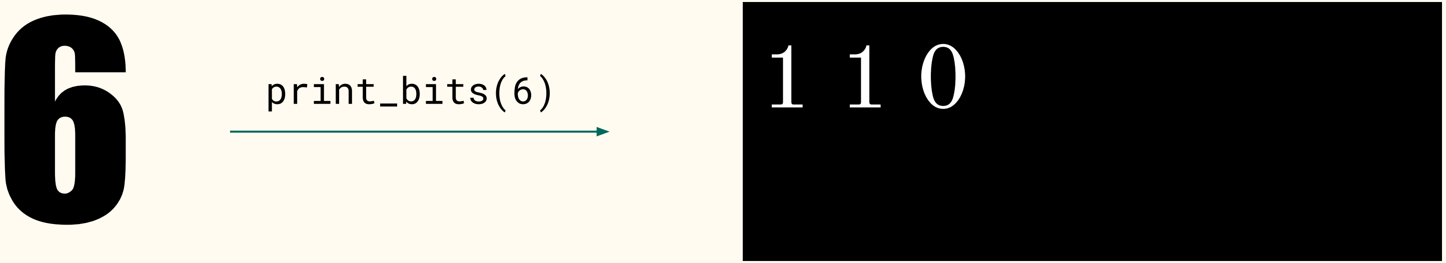
`print_bits(6)`

```
1 1 0
```

Convert base 10 to **any other base**

1. Divide number by **target** base
2. Set aside remainder
3. Divide remaining whole number by **target** base
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

What is the target base for this problem?



6

`print_bits(6)`

```
1 1 0
```

Convert base 10 to **any other base**

1. Divide number by **target** base
2. Set aside remainder
3. Divide remaining whole number by **target** base
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

Printing bits

think of remainder
as 0.5 "2s" left over

Convert base 10 to **any other base**

target base = 2

1. Divide number by **target** base
2. Set aside remainder
3. Divide remaining whole number by **target** base
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

6

0.5 * 2 = 1

6 / 2 = 3.0 remainders: 0

3 / 2 = 1.5 remainders: 0, 1

1 / 2 = 0.5 remainders: 0, 1, 1

1 1 0

Printing bits

Convert base 10 to **base 2**

```
void print_bits(int num) {
    // base case

    // recursive case
}
```

1. Divide number by **target** base
2. Set aside remainder
3. Divide remaining whole number by **target** base
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

For this algorithm, what is the base case?

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    // base case

    // recursive case
}
```

Under what conditions will `num` ÷ 2 result in a quotient with a whole number part that is 0?

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    // base case

    // recursive case
}
```

Printing bits

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    // base case
    ---

    // recursive case

}
```

Printing bits

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    // base case
    if (num < 2) {
        cout << num << ' ';
    } else {
        // recursive case
    }
}
```

Printing bits

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    if (num < 2) {
        cout << num << ' ';
    } else {
        // recursive case
        ___
    }
}
```

Printing bits

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    if (num < 2) {
        cout << num << ' ';
    } else {
        // recursive case
        cout << ___ << ' ';
    }
}
```

Printing bits

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    if (num < 2) {
        cout << num << ' ';
    } else {
        // recursive case
        cout << _4_ << ' ';
    }
}
```

Which value replaces blank #4 so that we output the remainder (0 or 1) that results from dividing the current value of `num` by 2?

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    if (num < 2) {
        cout << num << ' ';
    } else {
        // recursive case
        cout << _4_ << ' ';
    }
}
```

Printing bits

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    if (num < 2) {
        cout << num << ' ';
    } else {
        // recursive case
        cout << num % 2 << ' ';
    }
}
```

Which operation on `num` reduces the size of the integer so that there are fewer bits to print?

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

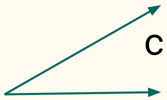
```
void print_bits(int num) {
    if (num < 2) {
        cout << num << ' ';
    } else {
        // recursive case
        cout << num % 2 << ' ';
    }
}
```

Printing bits

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    if (num < 2) {
        cout << num << ' ';
    } else {
        // recursive case

        cout << num % 2 << ' ';

    }
}
```



```
print_bits(num / 2);
```

Where do we place the recursive function call to `print_bits()` to output each remainder in the reverse order of its calculation?

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    if (num < 2) {
        cout << num << ' ';
    } else {
        // recursive case

        cout << num % 2 << ' ';

    }
}
```

`print_bits(num / 2);`

A. Above

B. Below

Printing bits

Convert base 10 to **base 2**

1. Divide number by **2**
2. Set aside remainder
3. Divide remaining whole number by **2**
4. When whole number > 0, go back to #2
5. When whole number = 0, set aside remainder
6. Write remainders in reverse

```
void print_bits(int num) {
    if (num < 2) {
        cout << num << ' ';
    } else {
        print_bits(num / 2);
        cout << num % 2 << ' ';
    }
}
```

Instruction order matters