INTRO TO RECURSION

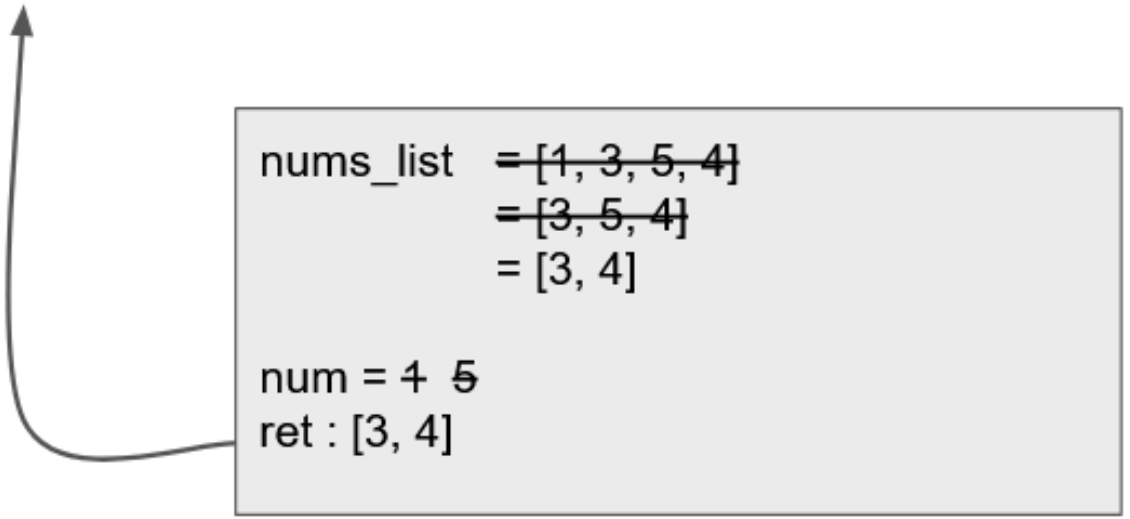
REMINDERS

- HW04 due 11/2 (Monday)
  - Sign up for presentations **now**
- Exam 2 11/9 (A week from monday)
  - Set up accomodated exams **now**

PREVIOUSLY:

```
def remove_odds(nums_list: list[int]):
    for num in nums_list:
        if num % 2 == 1:
            nums_list.remove(num)
    return nums_list
```

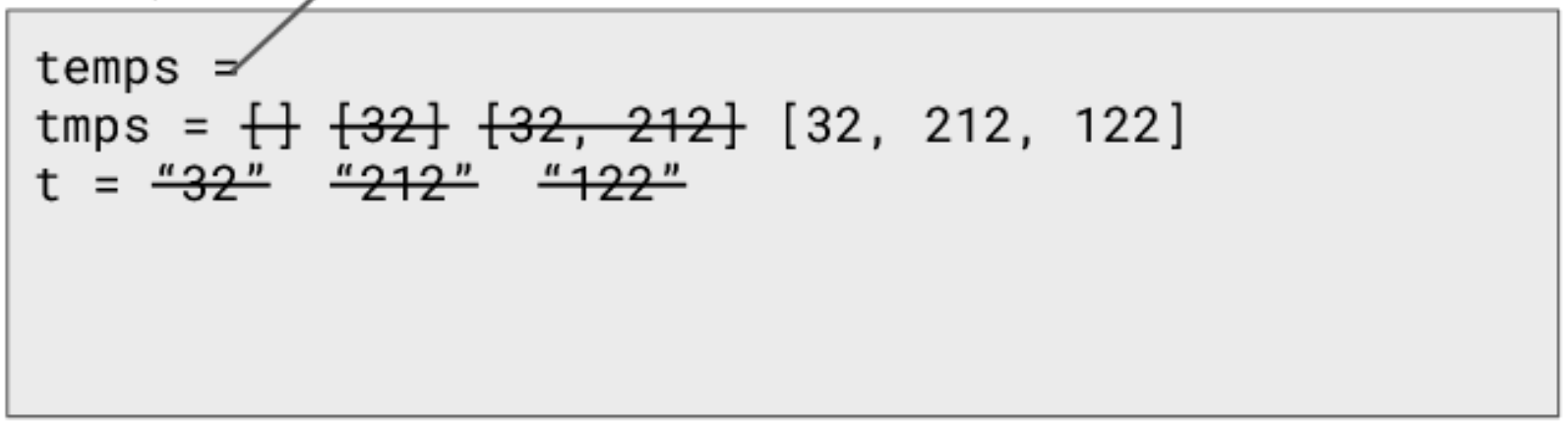
function call: remove_odds([1, 3, 5, 4])



ALSO PREVIOUSLY

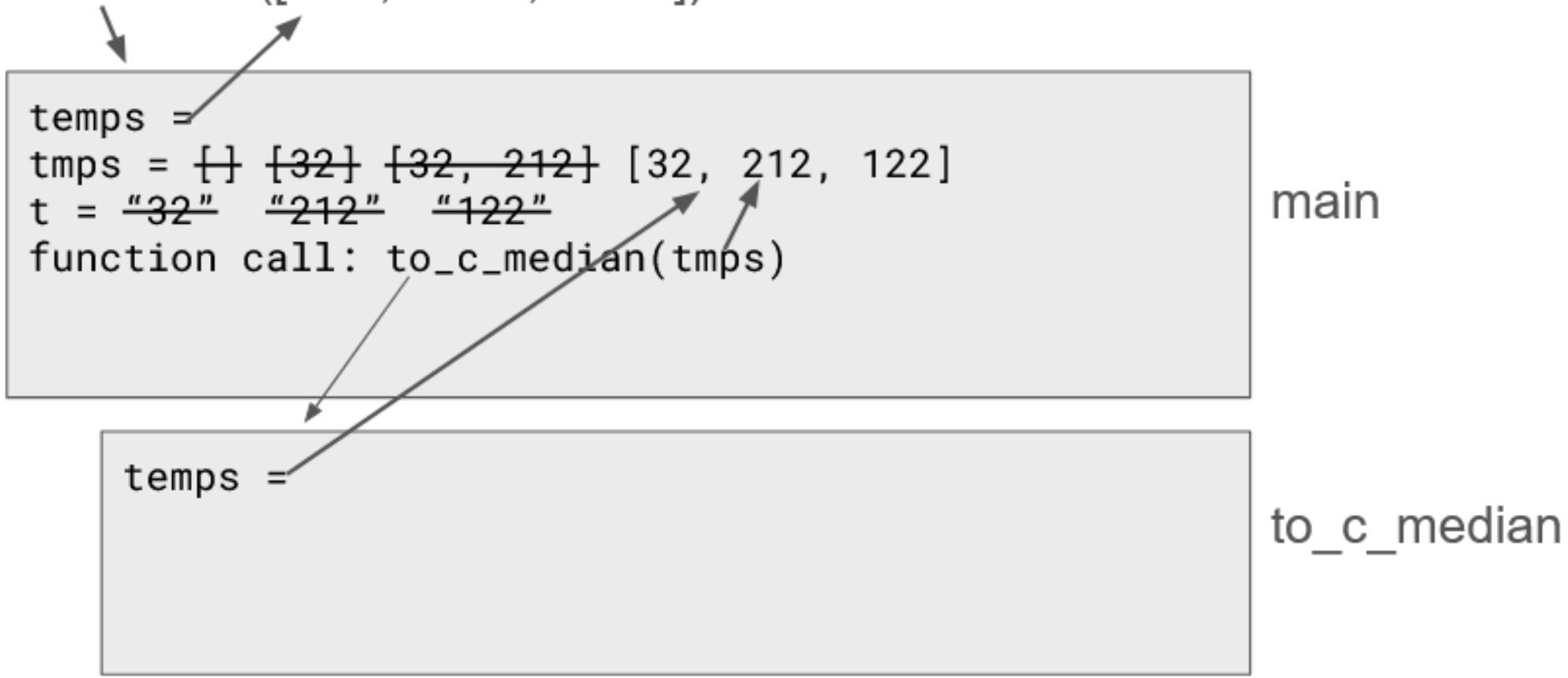
```
def main(temps):
    tmps = []
    for t in temps:
        tmps.append(int(t))
    print("full median:", to_c_median(tmps))

def to_c_median(temps):
    for i in range(len(temps)):
        t = temps[i]
        temps[i] = (t-32)*5/9
    temps = sorted(temps)
    if len(temps) % 2 == 0:
        return (temps[len(temps)//2] + temps[(len(temps)//2) - 1]) / 2
    return temps[len(temps) // 2]
```

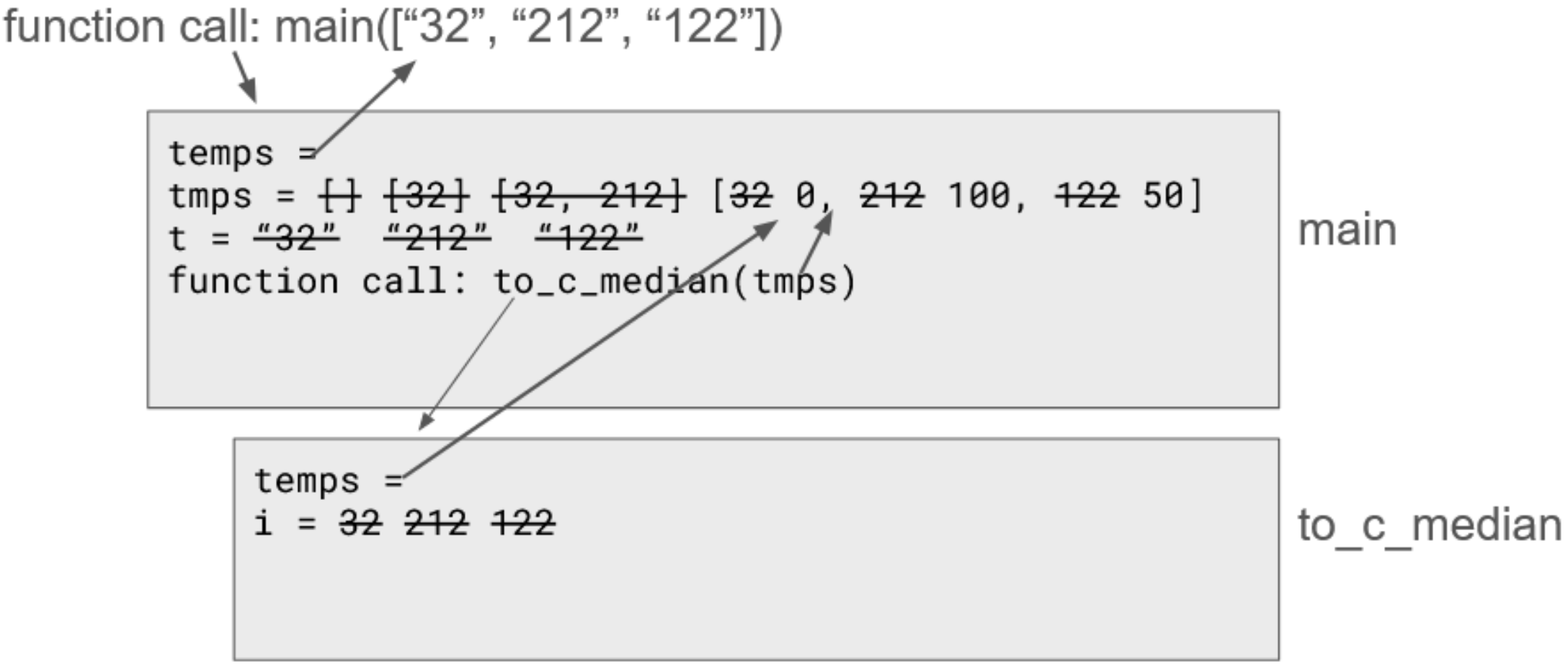
function call: main(["32", "212", "122"])

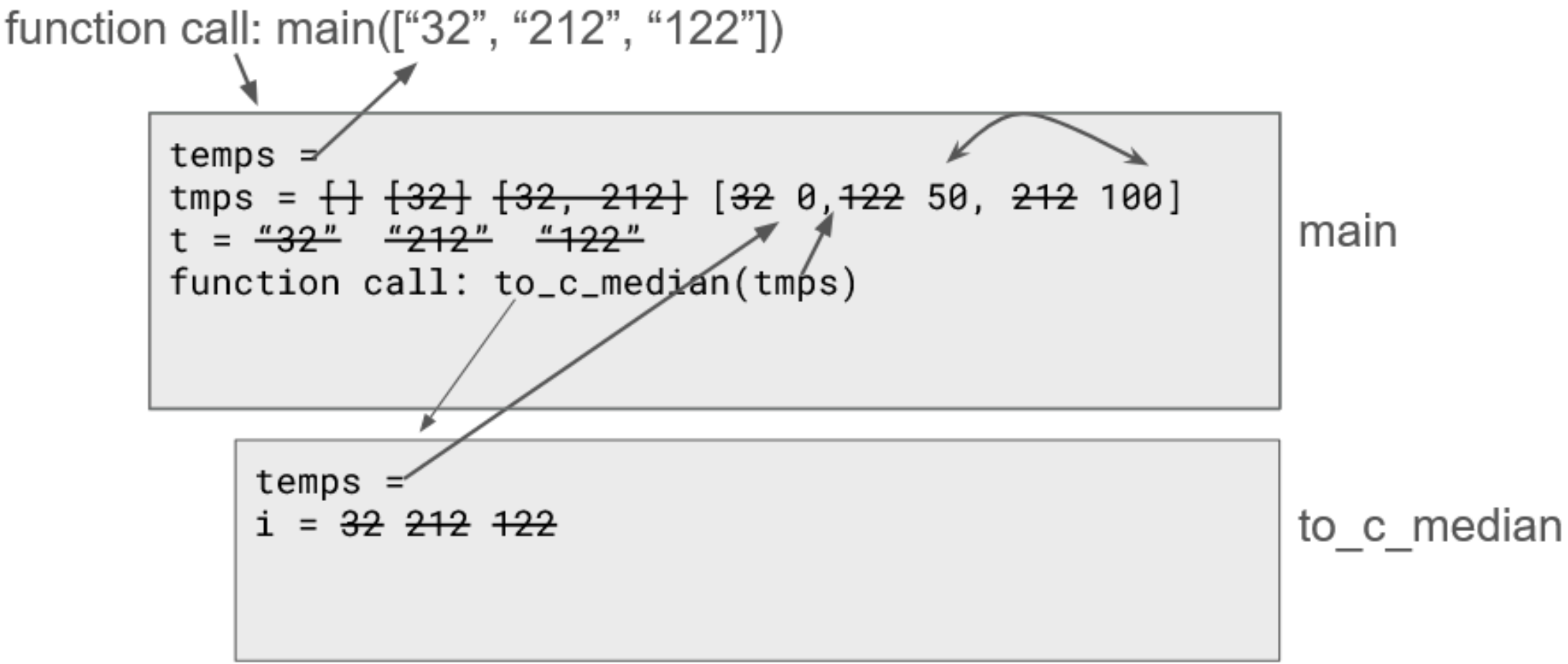


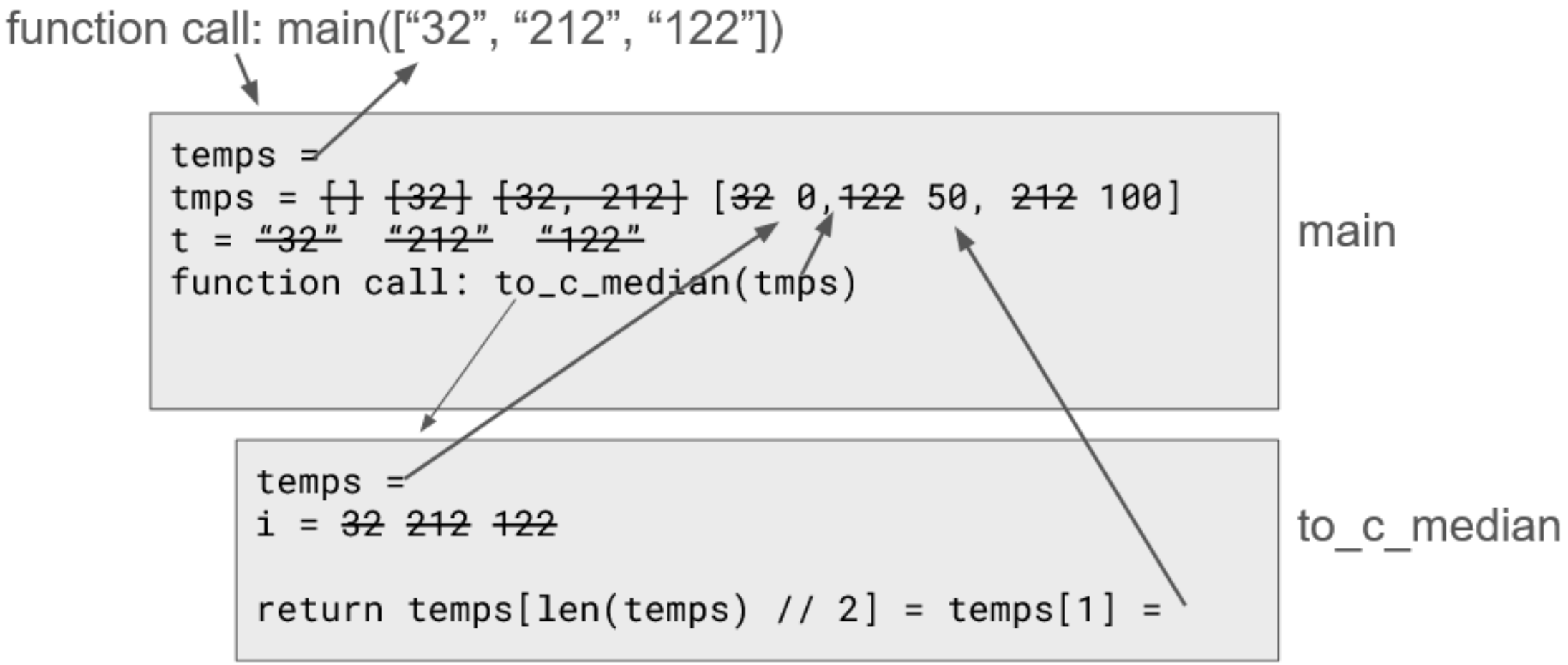
main

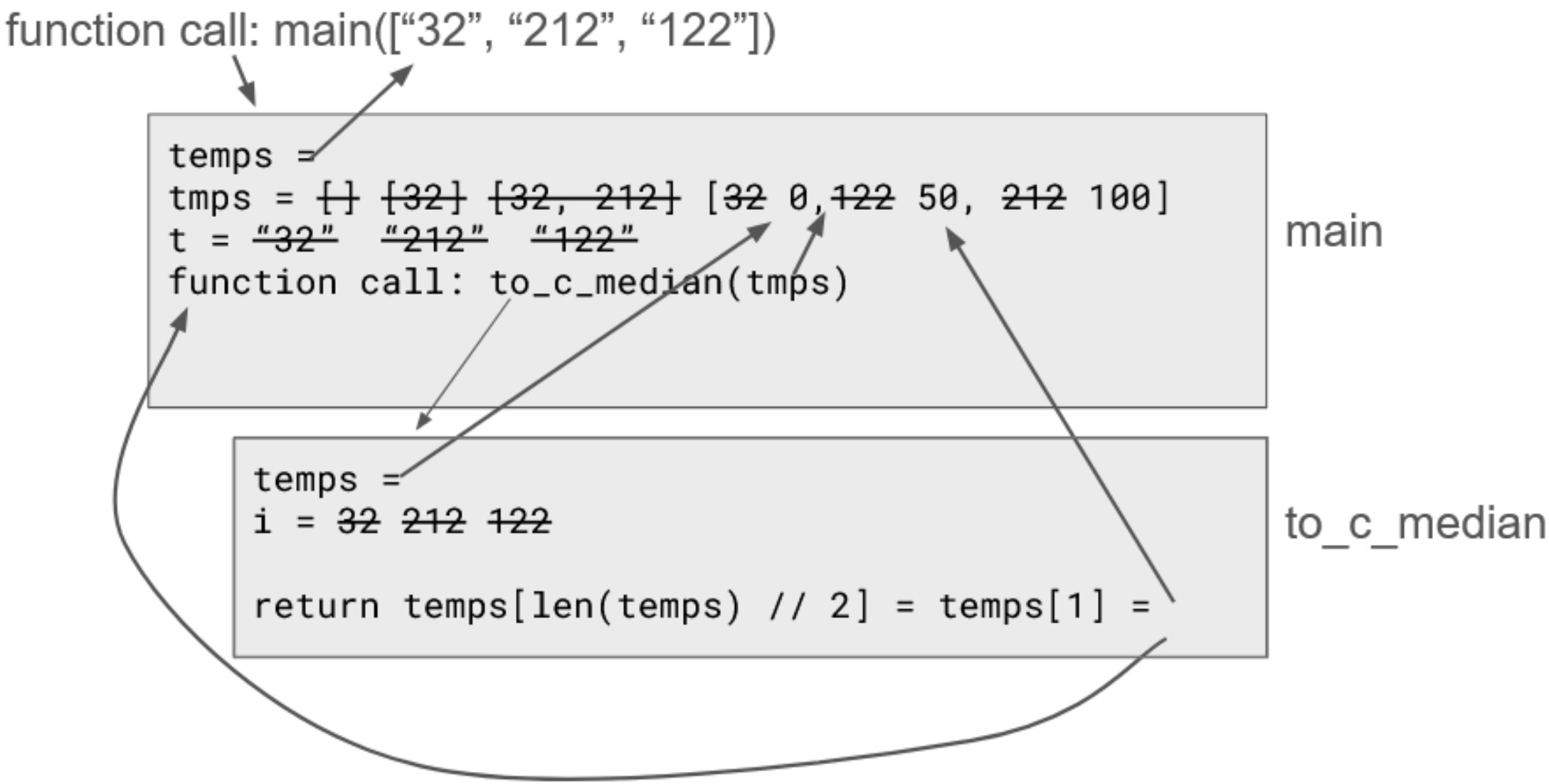
function call: main([“32”, “212”, “122”])




function call: main(["32", "212", "122"])
temps =
tmps = [] [32] [32, 212] [32 0, 212 100, 122 50]
t = "32" "212" "122"
function call: to_c_median(tmps)
main
temps =
i = 32 212 122
to_c_median


function call: main(["32", "212", "122"])
temps =
tmps = [] [32] [32, 212] [32 0, 122 50, 212 100]
t = "32" "212" "122"
function call: to_c_median(tmps)
main
temps =
i = 32 212 122
to_c_median


function call: main(["32", "212", "122"])
temps =
tmps = [] [32] [32, 212] [32 0, 122 50, 212 100]
t = "32" "212" "122"
function call: to_c_median(tmps)
main
temps =
i = 32 212 122
return temps[len(temps) // 2] = temps[1] =
to_c_median


function call: main(["32", "212", "122"])
temps =
tmps = [] [32] [32, 212] [32 0, 122 50, 212 100]
t = "32" "212" "122"
function call: to_c_median(tmps)
main
temps =
i = 32 212 122
return temps[len(temps) // 2] = temps[1] =
to_c_median

TODAY, CONSIDER:

```
def factorial(n : int) -> int:
    if n == 1:
        return 1
    return n * factorial(n-1)
```

TODAY, CONSIDER:

```
def factorial(n : int) -> int:
    if n == 1:
        return 1
    return n * factorial(n-1)
```

What is this doing? How do I trace this? Help!

RECURSION

```
def factorial(n : int) -> int:
    if n == 1:
        return 1
    return n * factorial(n-1)
```

- **Recursion** is when a function calls itself to do its work.
- A **recursive call** is what happens when a function calls itself; the input to the recursive call should be a "smaller" version of the problem.
  - e.g. `multiply(3, 4)` can be evaluated as `4 + multiply(2, 4)`
- A **base case** is an input to a recursive function that does not require another recursive call.

RECURSIVE PROBLEMS ARE PROBLEMS DEFINED IN TERMS OF THEMSELVES!

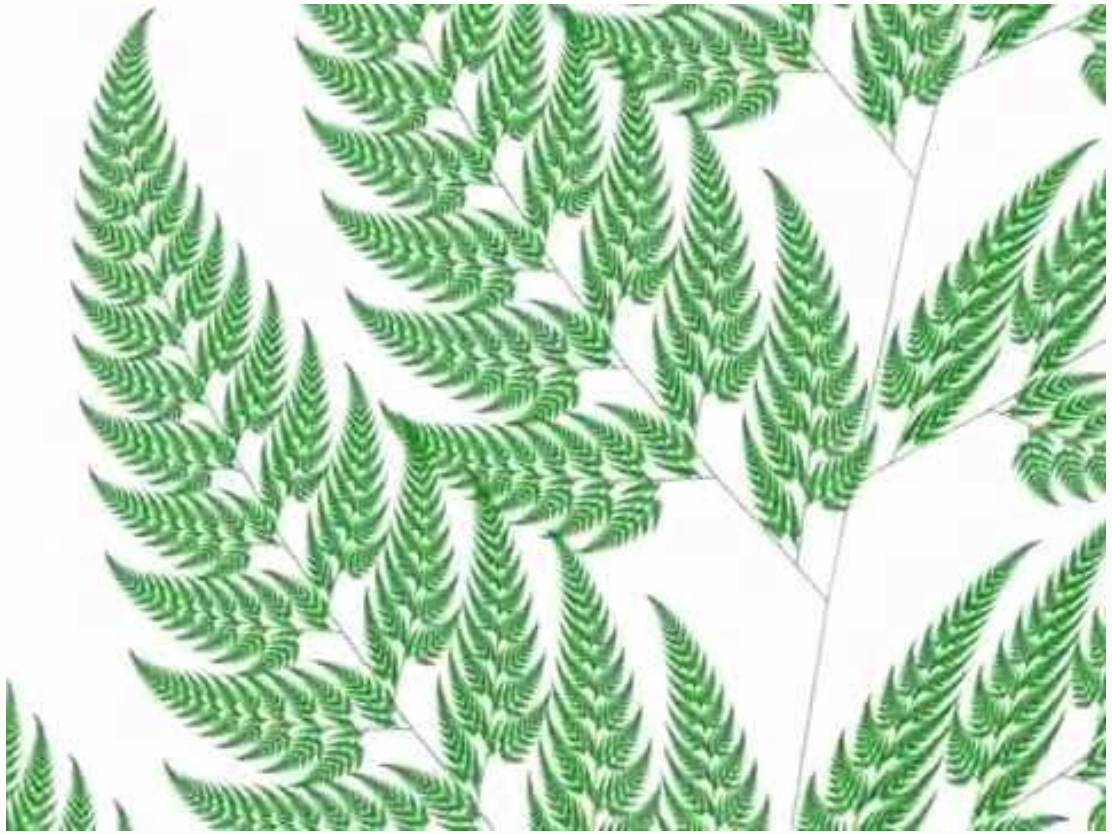
- Data representation:
  - **JSON** features dictionaries whose keys point to values that are dictionaries whose keys point to values that are dictionaries whose keys point to values that are dictionaries whose keys point to values that are...
  - **XML** is a format where we place tags inside of tags inside of tags inside of tags inside of...
- Algorithms like **searches** (linear, binary, depth-first, breadth-first, etc.)

Recursion is NOT just something Computer Scientists made up!

It crops up in...

- how we produce language
- how organisms grow
- (and yes...) math & logic & philosophy

RECURSION IN NATURE



DON’T LOSE SIGHT OF WHAT YOU KNOW!

Recursion is tricky, but it's manageable if you keep in mind everything you know about calling functions.

1. Functions are called by their names with inputs passed in
2. Executing a function call creates a new scope in which only **input variables** and **variables defined within the function** are accessible
3. The only way to have some information "escape" the body of a function is to **return it.**
4. Other than producing a value, `return` also stops the current function execution and brings us back to the place at which the function was called.

WITH THAT IN MIND, LET'S TRACE!

```
def factorial(n : int) -> int:
    if n == 1:
        return 1
    return n * factorial(n-1)
```

TRACING WITH TABLES

[C1] Try filling out this table, it will be similar to how we completed the trace, but without the frames

CALL	INPUT	RETURNS	STATUS
factorial(6)	6	6 * factorial(5)	waiting
factorial(5)	...		

TRACING WITH TABLES

[C1] Try filling out this table, it will be similar to how we completed the trace, but without the frames

CALL	INPUT	RETURNS	STATUS
factorial(6)	6	6 * factorial(5)	waiting
factorial(5)	5	5 * factorial(4)	waiting
factorial(4)	4	4 * factorial(3)	waiting
factorial(3)	3	3 * factorial(2)	waiting
factorial(2)	2	2 * factorial(1)	waiting
factorial(1)	1	1	returning 1

TRACING WITH TABLES

[C1] Try filling out this table, it will be similar to how we completed the trace, but without the frames

CALL	INPUT	RETURNS	STATUS
factorial(6)	6	6 * factorial(5)	waiting
factorial(5)	5	5 * factorial(4)	waiting
factorial(4)	4	4 * factorial(3)	waiting
factorial(3)	3	3 * factorial(2)	waiting
factorial(2)	2	2 * factorial(1)	returning 2
factorial(1)	1	1	returning 1

TRACING WITH TABLES

[C1] Try filling out this table, it will be similar to how we completed the trace, but without the frames

CALL	INPUT	RETURNS	STATUS
factorial(6)	6	6 * factorial(5)	returning 720
factorial(5)	5	5 * factorial(4)	returning 120
factorial(4)	4	4 * factorial(3)	returning 24
factorial(3)	3	3 * factorial(2)	returning 6
factorial(2)	2	2 * factorial(1)	returning 2
factorial(1)	1	1	returning 1

PRACTICE [C2]

Using whatever visualization makes the most sense to you, trace out this code for when `mystery("abcd")` is called.

Then write a one-sentence description of what it does

Tip: ord("A") = 65

```
def mystery(x : str):
    if len(x) <= 0:
        return 0
    value = ord(x[0].upper())
    return value + mystery(x[1:])
```

PRACTICE [C2]

This function accepts a string, then finds the sum of the ascii values for each character in the string (all letters in caps).

CALL	INPUT	RETURNS	STATUS
mystery("abcd")	"abcd"	65 + mystery("bcd")	waiting
mystery("bcd")	"bcd"	66 + mystery("cd")	waiting
mystery("cd")	"cd"	67 + mystery("d")	waiting
mystery("d")	"d"	68 + mystery("")	waiting
mystery("")	""	0	return 0

PRACTICE [C3] FIX THE FUNCTION

We want to write the function `remove_blanks(word)` that takes in a string and returns the same string without any spaces in it.

Try tracing this function, what is wrong with it? What do you need to change to fix it?

```
def remove_blanks(word : str) -> str:
    char = word[0]
    if char == " ":
        return remove_blanks(word[1:])
    return word[0] + remove_blanks(word[1:])
```

WHAT'S WRONG WITH THIS FUNCTION?

```
def remove_blanks(word : str) -> str:
    char = word[0]
    if char == " ":
        return remove_blanks(word[1:])
    return word[0] + remove_blanks(word[1:])
```

We're missing a **base case**. The function never knows when to stop!

Can you think of what a reasonable base case might be?

WHAT'S WRONG WITH THIS FUNCTION?

```
def remove_blanks(word : str) -> str:
    if len(word) == 0:
        return ""
    char = word[0]
    if char == " ":
        return remove_blanks(word[1:])
    return word[0] + remove_blanks(word[1:])
```

We're missing a **base case**. The function never knows when to stop!

Can you think of what a reasonable base case might be?

- usually "at zero return zero" or "at one return one"

EXTRA PRACTICE: TRACE ME!

Try tracing this function with the call `split_sum([4, 2, 5, 3, 6, 7])` using both the table method and the frames method.

There's no right answer, but see what you find easier to follow.

```python
def split_sum(items : list[int]) -> int:
    if len(items) <= 0:
        return 0
    if len(items) == 1:
        return items[0]

    # Split the array into two halves
    mid = len(items) // 2
    return split_sum(items[:mid]) + split_sum(items[mid:])
```

TALIE'S PERSONAL OPINION

```python
def split_sum(items : list[int]) -> int:
    if len(items) <= 0:
        return 0
    if len(items) == 1:
        return items[0]

    # Split the array into two halves
    mid = len(items) // 2
    return split_sum(items[:mid]) + split_sum(items[mid:])
```

Because if these cases where we might have multiple function calls in the recursive function, I personally prefer using frames when tracing. But it's preference and a lot of people like using tables 🤷‍♀️🤷‍♀️🤷‍♀️