Quiz 0 Review!

Part 0


eclectech.co.uk

Logistics

Bits ‘n’ Bytes

- A bit = 0 or 1
- A byte = 8 bits
- 0001101101111100
  - How many bits?
  - How many bytes?
  - Convert it to hexadecimal!

C and Compilers

- The reason we don’t have to write code using 0’s and 1’s!
- We use the GNU Compilers Collection (GCC)
- gcc vs. make
  - make hello
  - gcc -ggdb -std=c99 -Wall -Werror hello.c -lcs50 -lm -o hello
- Where did we specify the options listed above?

Data Types

- int
- char
- float
- double
- long
- long long
- string (same as char *)

Casting

- float age = 6.75;
  float new_age = (int) age + 1.5;
  printf("%.2f", new_age);

Casting

- float age = 6.75;
  float new_age = (int) age + 1.5;
  printf(“%.2f”, new_age);

- Output: 7.50

Arithmetic operators

- +, -, *, /, %
- x = x+1 is the same as x += 1 is the same as x++
- Integer division: why does (n/10) * 10 not always equal n?
- What does the mod operator do?

Ye Olde ASCII Table

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

ASCII Math

- ‘P’ + 1?

- ‘5’ ≠ 5
  - How would we transform one to the other?

ASCII Math

- ‘P’ + 1?
  - ‘Q’

- ‘5’ ≠ 5
  - How would we transform one to the other?
    - ‘5’ – ‘0’ = 5
    - 5 + ‘0’ = ‘5’

A simple C program…

```
#include <stdio.h>
#include <cs50.h>

#define LIMIT 100

int
main(int argc, char* argv[])
{
    int x = GetInt();

    if (x >= LIMIT)
    {
        printf("That number is too big!\n");
        return 1;
    }
    else
    {
        int y = x*x;
        printf("The square of %d is %d\n", x, y);
    }

    return 0;
}
```

…with lots of elements!

library

constant

return type

command line arguments

local variable

statement of assignment

boolean expression

“if” condition

call to library function in stdio.h

```
#include <stdio.h>
#include <cs50.h>

#define LIMIT 100

int
main(int argc, char* argv[])
{
    int x;
    x = GetInt();

    if (x >= LIMIT)
    {
        printf(“That number is too big!\n”);
        return 1;
    }

    printf(“The square of %d is %d\n”, x, x*x);

    return 0;
}
```

arguments to printf

Loops

When would we use each of the following?

- for?

- while?

- do while?

Loops

When would we use each of the following?

- for?
  - We already know how many times we want to iterate through our loop (could also use while)
- while?
  - We're not sure how **many** times we want our loop to run, but there is some condition that needs to be true for our loop to keep running
- do while?
  - Similar to while, but we want the code in our loop to run **at least once**

Loops

Each loop needs an initialization, a condition, and an update.

- for?
  - ```
    for (initialization; condition; update)
    {
        // do this
    }
    ```
- while?
  - ```
    initialization
    while (condition)
    {
        // do this
        // update
    }
    ```
- do while?
  - ```
    initialization
    do
    {
        // do this
        // update
    }
    while (condition);
    ```
    ```

Look, a function!

```
void
double_arrary(int nums[], int length)
{
   for (int i = 0; i < length; i++)
       nums[i] *= 2;
   printf(“Your array has been doubled!\n”);
}
```


MyFunnyPics.net

quiz0

Tommy MacWilliam

tmacwilliam@cs50.net

October 9, 2011

Today

- scope
- arrays
- command-line arguments
- searching
- sorting
- asymptotic notation
- recursion

Variable scope

- global variables: accessible by all functions
  - defined outside of `main`
- local variables: accessible by a single block
  - defined within a block, only accessible in that block

Variable scope


```
int x = 5;
int f() {
    int y = 6;
    x++;
}
int g() {
    int y = 8;
    x--;
}
```

Arrays

- list of elements of the same type
- elements accessed by their **index** (aka position)
  - index starts at 0!
- `int array[3] = {1, 2, 3};`
- `array[1] = 4;`

Multi-dimensional Arrays

- can also have arrays of arrays!
- multi-dimensional array creates a grid instead of a list
- needs multiple indices: `int grid[3][5];`
  - 3 rows, 5 columns

Multi-dimensional Arrays


```
int grid[2][3] = {{1, 2, 3}, {4, 5, 6}};
grid[1][2] = 6;
```

		Columns		
		0	1	2
Rows	0	1	2	3
	1	4	5	6

Passing Arrays to Functions

- `int`s, `char`s, `float`s, etc. are passed by **value**
  - contents CANNOT be changed by the function they're passed to (unless we use pointers!)
- arrays (of any type) are passed by **reference**
  - contents CAN be changed by the function they're passed to

main

- main is a **function** that can take 2 arguments
  - `argc`: number of arguments given
  - `argv[]`: array of arguments

Arguments

- `./this is cs 50`
  - `argc == 4`
  - `argv[0] == ‘‘./this’’`
  - `argv[1] == ‘‘is’’`
  - `argv[2] == ‘‘cs’’`
  - `argv[3] == ‘‘50’’`
- `‘‘50’’ != 50;`
  - `atoi(‘‘50’’) == 50;`

Big O

- O: worst-case running time
  - given the worst possible scenario, how fast can we solve a problem?
    - e.g. array is in descending order, we want it in ascending order
  - upper bound on runtime

Omega

- Ω: best-case running time
  - given the best possible scenario, how fast can we solve a problem?
    - e.g. array is already sorted
  - lower bound on runtime

Common Running Times

- in ascending order:
  - $O(1)$: constant
  - $O(\log n)$: logarithmic
  - $O(n)$: linear
  - $O(n \log n)$: linearithmic
  - $O(n^c)$: polynomial
  - $O(c^n)$: exponential
  - $O(n!)$: factorial

Comparing Running Times

- $O(n)$, $O(2n)$, and $O(5n+3)$ are all asymptotically equivalent: $O(n)$
  - constants drop out, because n dominates
- similarly, $O(n^3+2n^2) = O(n^3)$
  - n^3 dominates n^2
- however, $O(n^3) > O(n^2)$
  - 2 and 3 are not constants here, they're exponents

Linear Search

- implementation: iterate through each element of the list, looking for it
- runtime: $O(n)$, $\Omega(1)$
- does not require list to be sorted

Binary Search

- implementation: keep looking at middle elements
  - start at middle of list
  - if too high, forget right half and look at middle of left half
  - if too low, forget left half and look at middle of right half
- runtime: $O(\log n)$, $\Omega(1)$
- requires list to be sorted

Binary Search


```
while length of list > 0
  look at middle of list
  if number found, return true
  else if number is too high, only consider
  left half of list
  else if number is too low, only consider
  right half of list
return false
```

Binary Search

50	61	121	124	143	161	164	171	175	182

Binary Search

164 171 175 182

Binary Search

164 171

Binary Search

164

Bubble Sort

- implementation: if adjacent elements are out of place, switch them
  - repeat until no swaps are made
- runtime: $O(n^2)$, $\Omega(n)$

Bubble Sort


```
do
  swapped = false
  for i = 0 to n - 2
    if array[i] > array[i + 1]
      swap array[i] and array[i + 1]
      swapped = true
while elements have been swapped
```

Bubble Sort

5 0 1 6 4

Bubble Sort

0 5 1 6 4

Bubble Sort

0 1 5 6 4

Bubble Sort

0 1 5 6 4

Bubble Sort

0 1 5 4 6

Bubble Sort

0 1 5 4 6

Bubble Sort

0 1 5 4 6

Bubble Sort

0 1 4 5 6

Bubble Sort

0 1 4 5 6

Selection Sort

- implementation: start at beginning of list, find smallest element
  - swap first element with smallest element
  - go to second element, treat that as the new first element, continue
    - because everything to the left is already sorted
- runtime: $O(n^2)$, $\Omega(n^2)$, $\Theta(n^2)$

Selection Sort


```
for i = 0 to n - 1
  min = i
  for j = i + 1 to n
    if array[j] < array[min]
      min = j
  if array[min] != array[i]
    swap array[min] and array[i]
```

Selection Sort

5 0 1 6 4

Selection Sort

0 5 1 6 4

Selection Sort

0 1 5 6 4

Selection Sort

0 1 4 6 5

Selection Sort

0 1 4 5 6

Recursion

- base case: when function should stop calling itself
  - without a base case, function would call itself forever!
- recursive case: function calls itself, probably using different arguments

Recursion


```
int factorial(int n) {
    if (n <= 1)
        return 1;

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

Recursion and the Stack

`factorial(4)`
`main`

Recursion and the Stack

`factorial(3)`
`factorial(4)`
`main`

Recursion and the Stack

`factorial(2)`
`factorial(3)`
`factorial(4)`
`main`

Recursion and the Stack

`factorial(1)`
`factorial(2)`
`factorial(3)`
`factorial(4)`
`main`

CS50

This is CS50. (Quiz 0 Review)

o hai!

Joseph Ong

CS50: Quiz 0

Merge Sort

Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```

50	3	42	1337	15

Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```

50	3	42	1337	15

Merge Sort

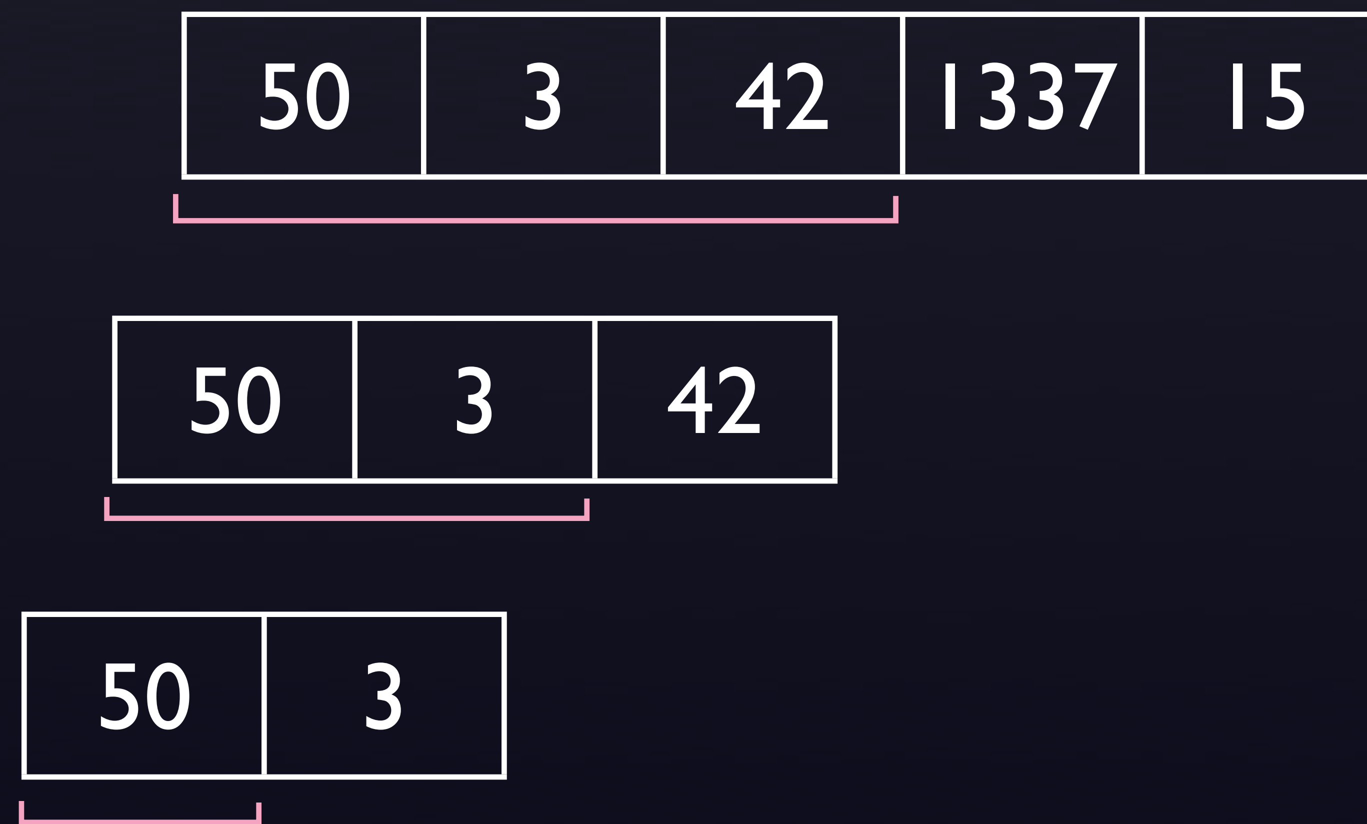
```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```

50	3	42	1337	15

50	3	42

Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



Merge Sort

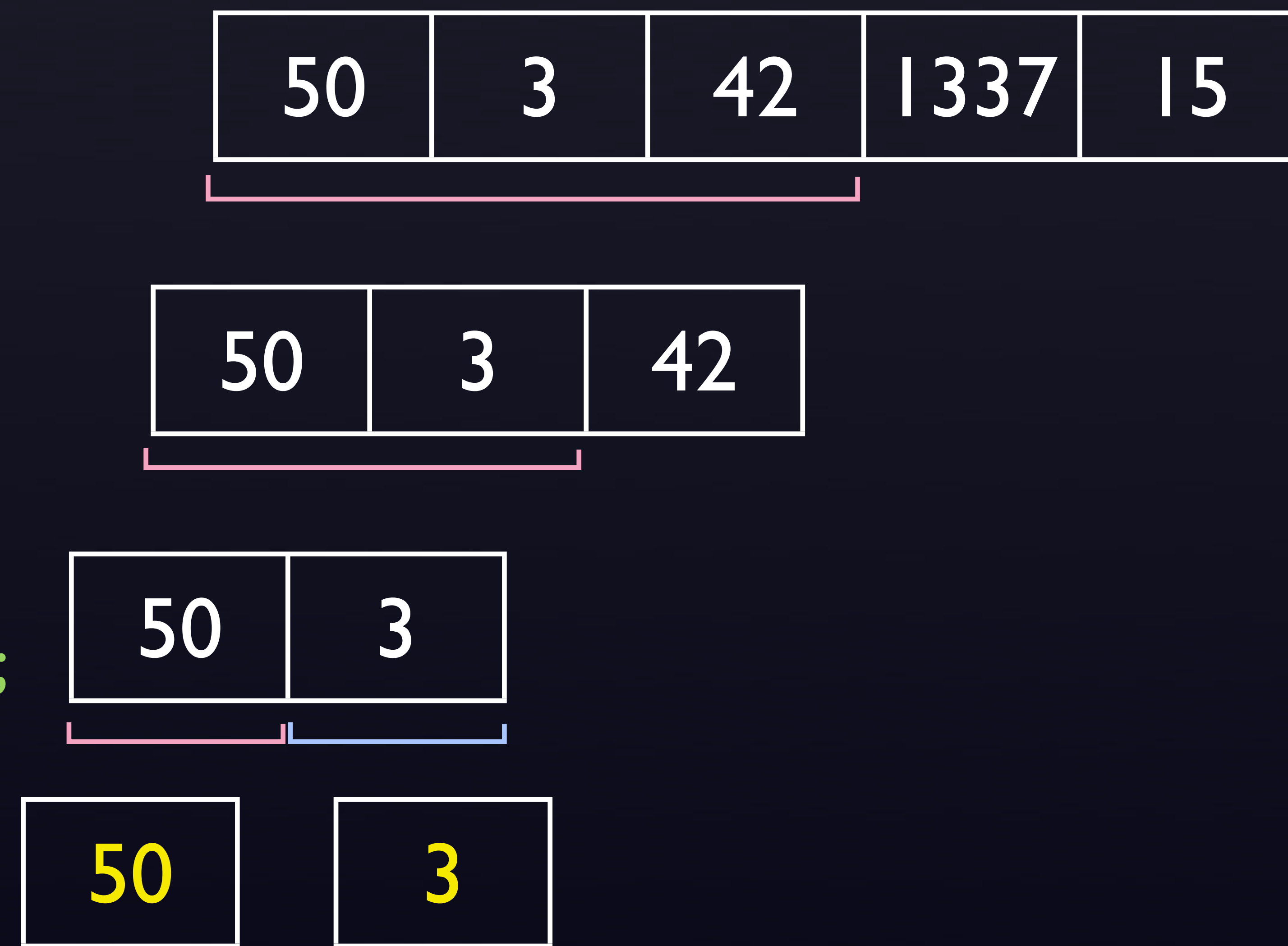
```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```

Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```

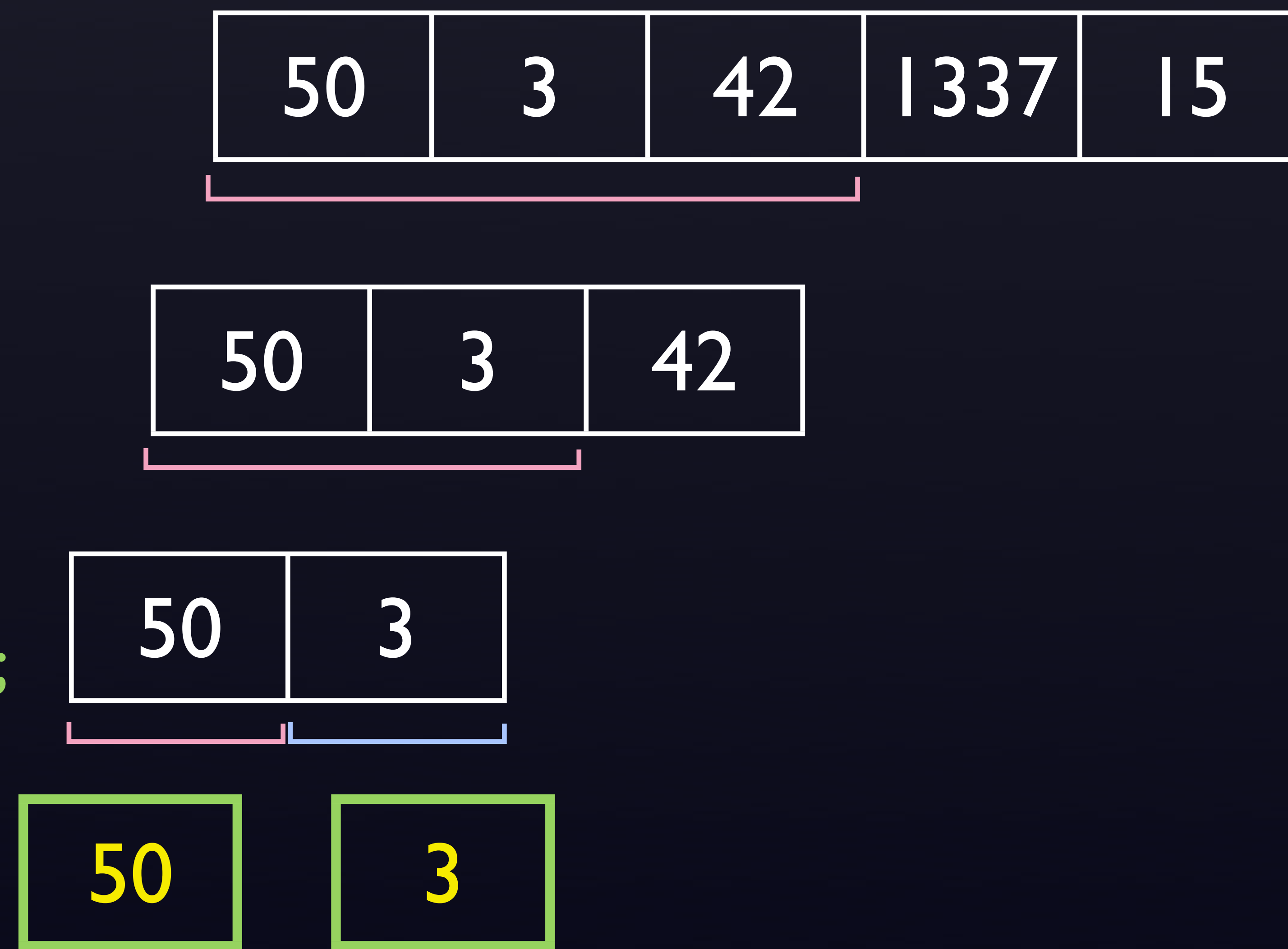
Merge Sort

```
mSort (list of n numbers)
    if n < 2
  →     return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



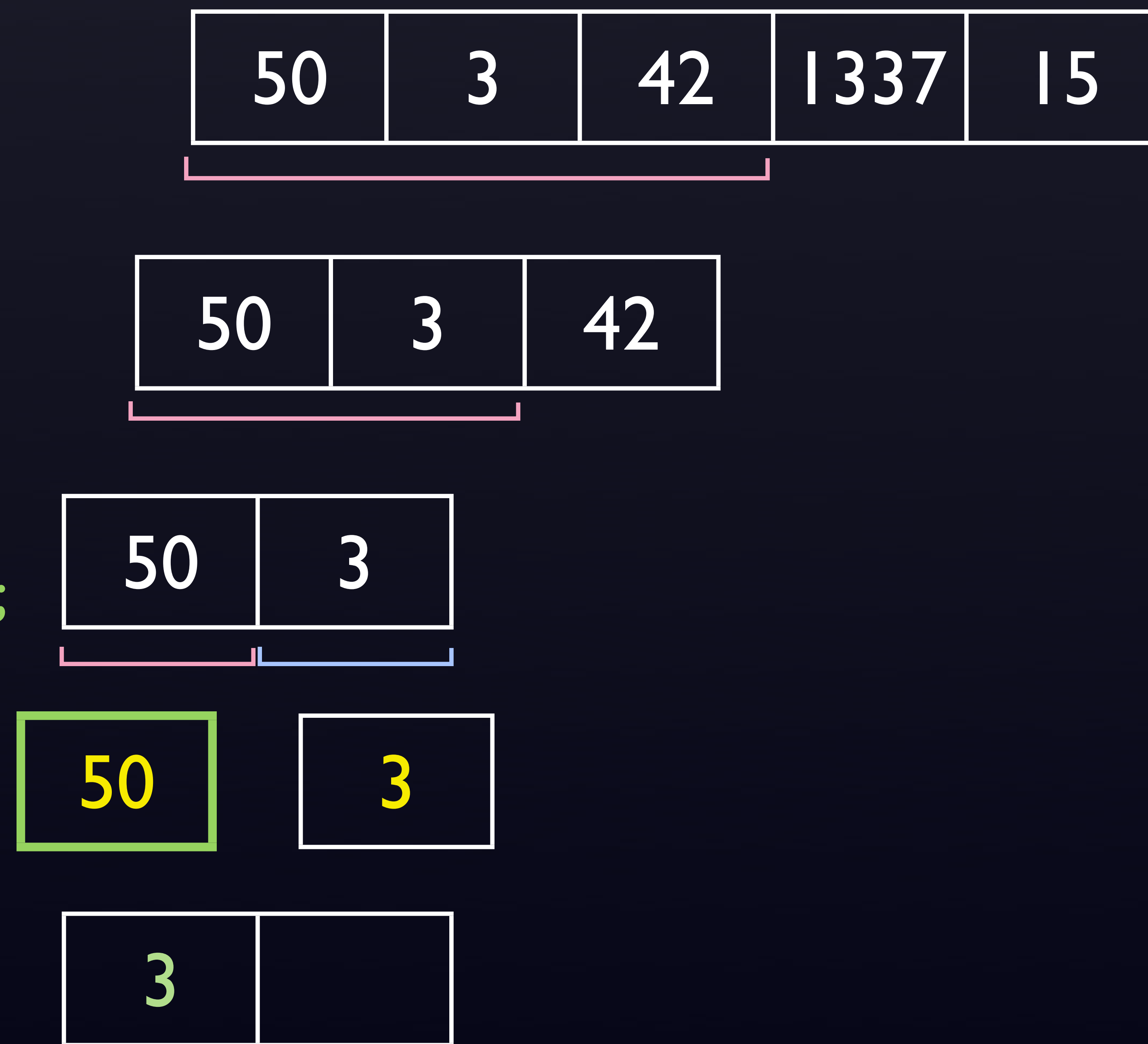
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



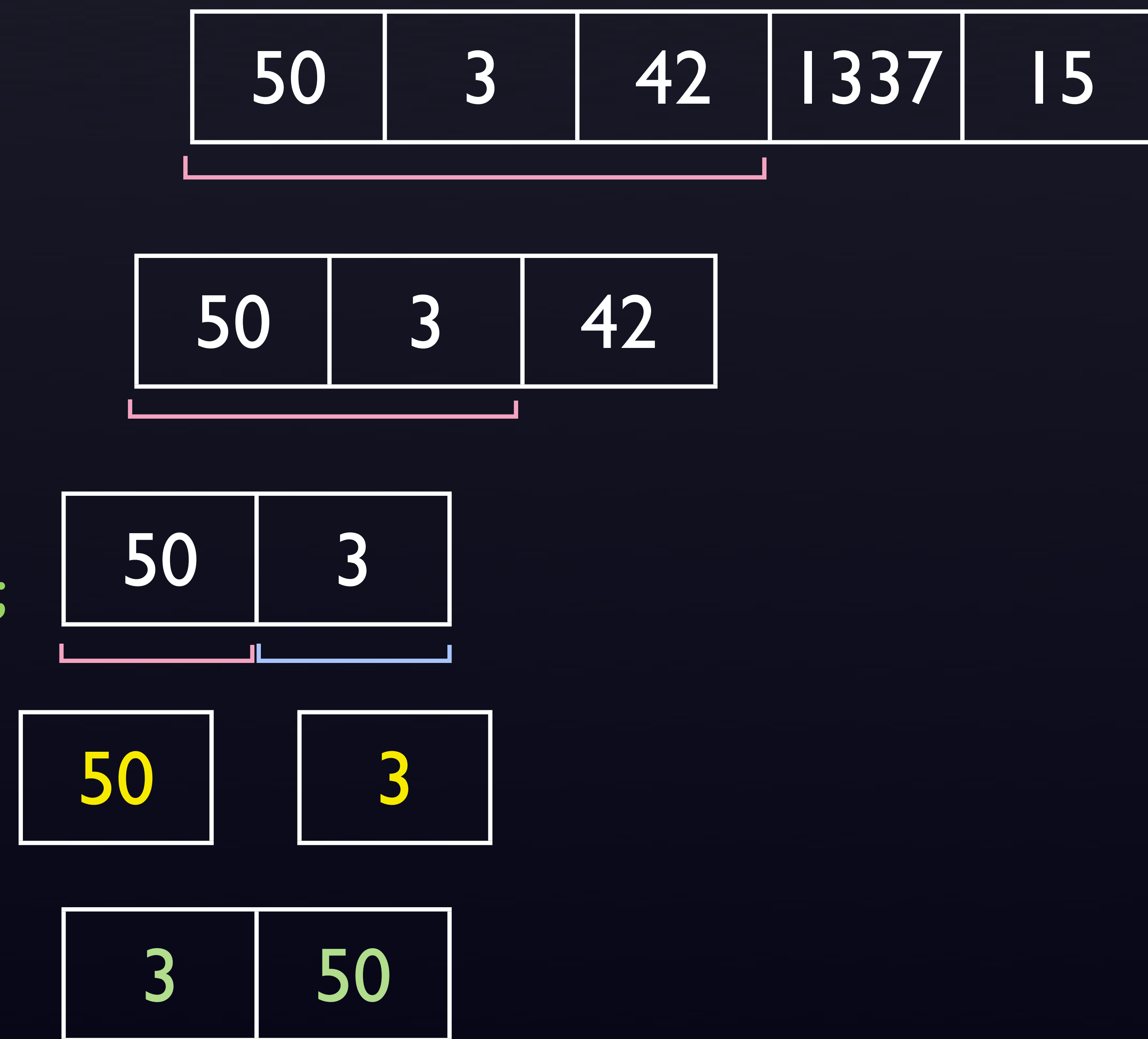
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



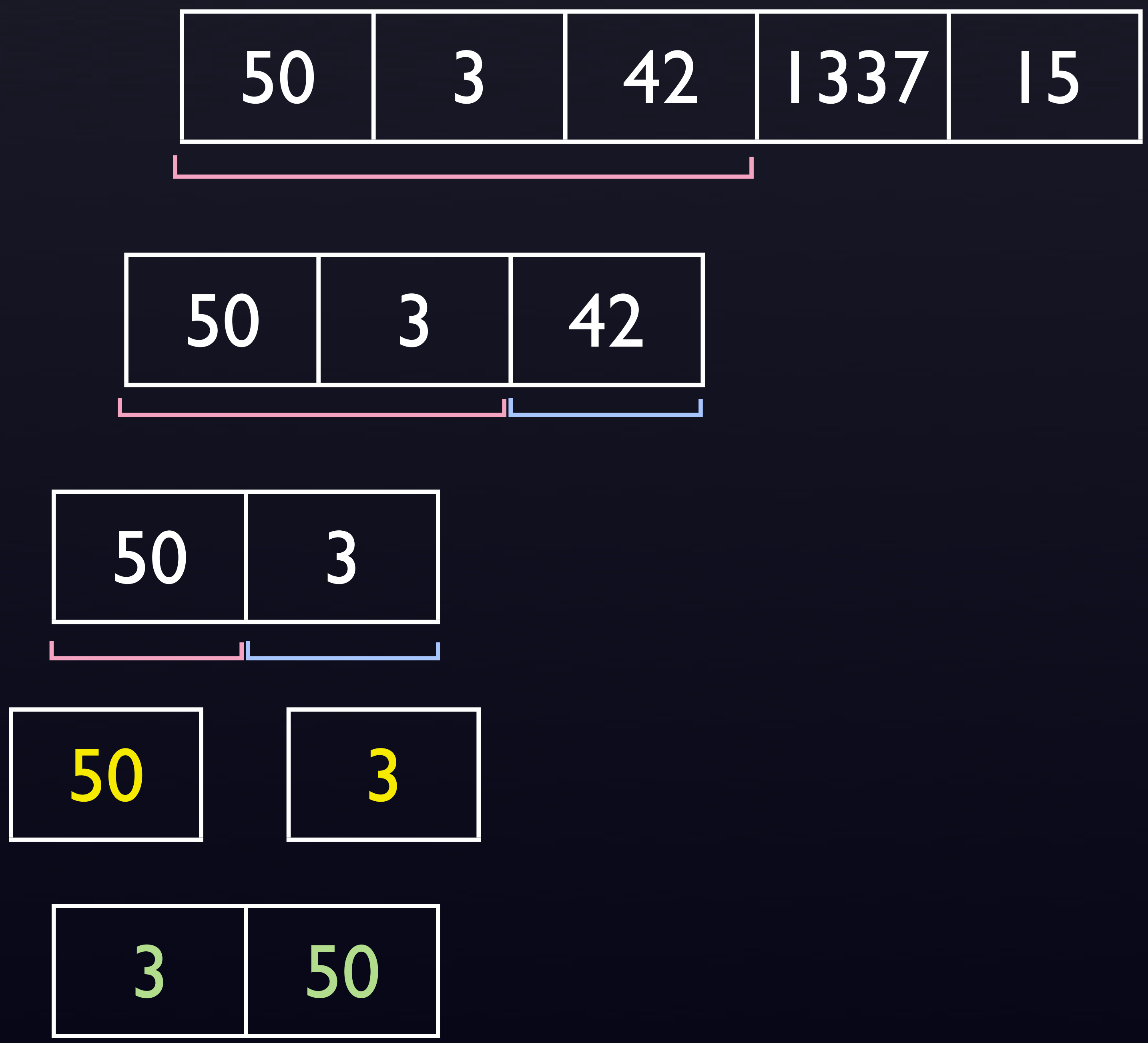
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



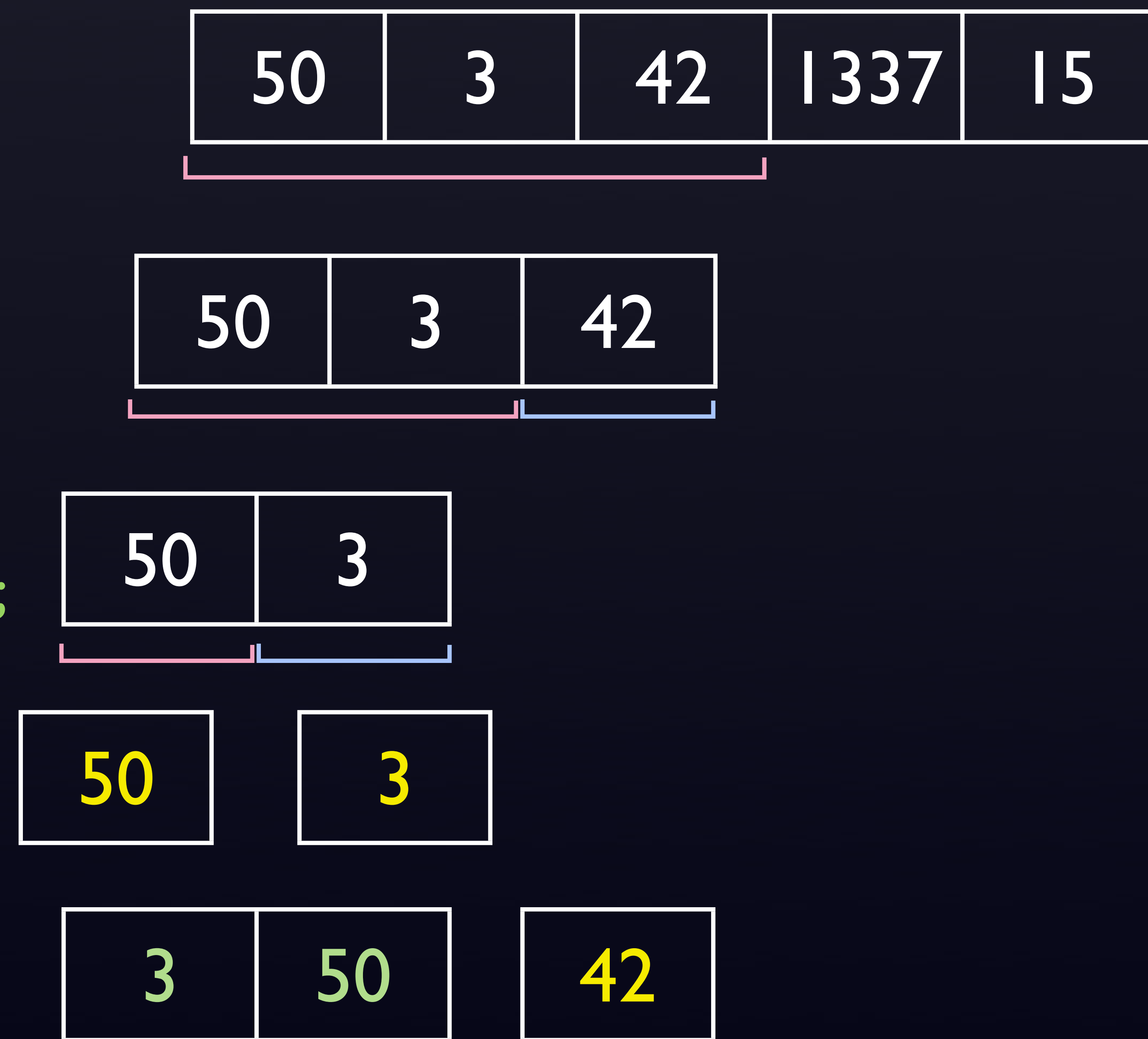
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



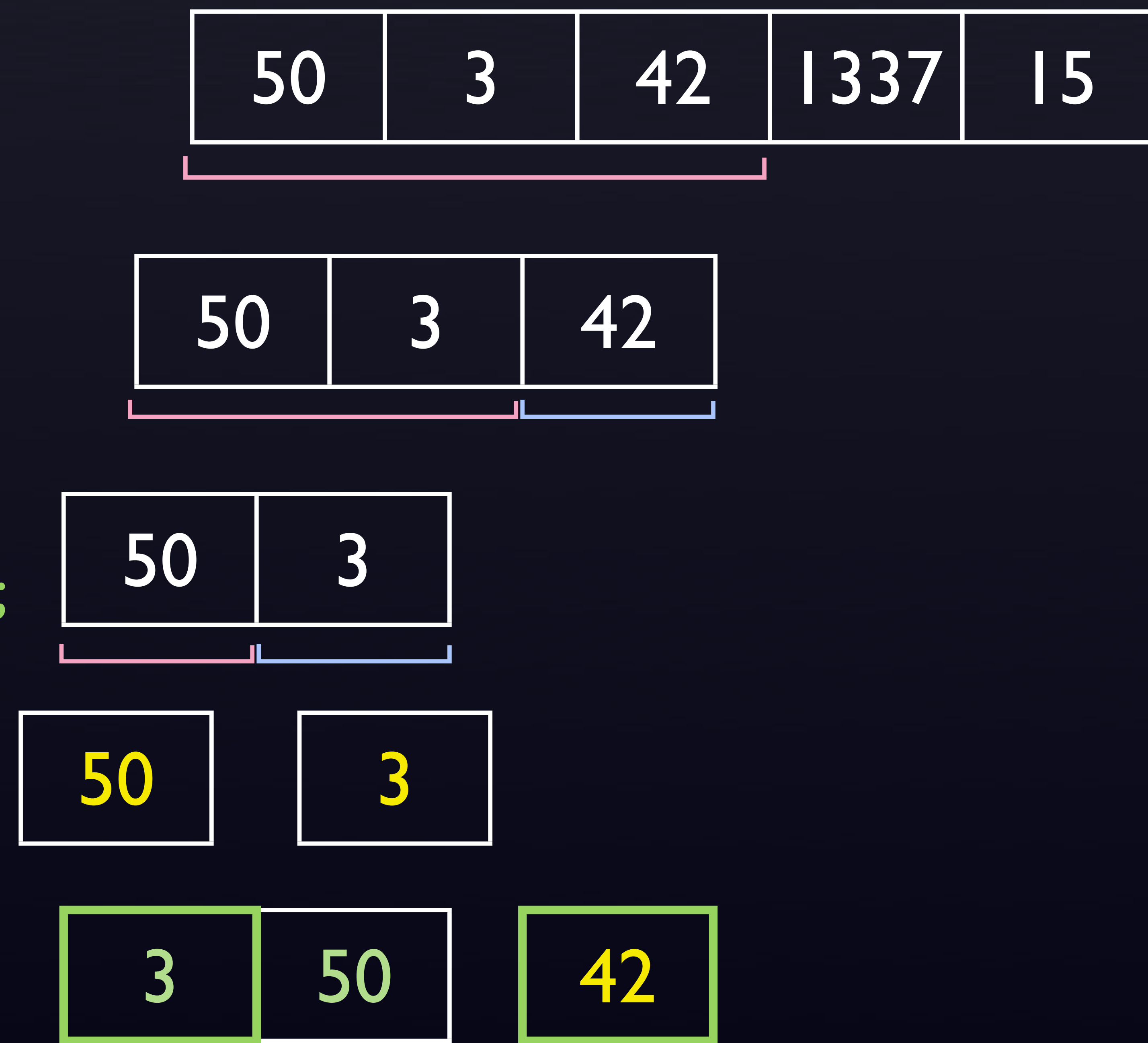
Merge Sort

```
mSort (list of n numbers)
    if n < 2
  →     return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



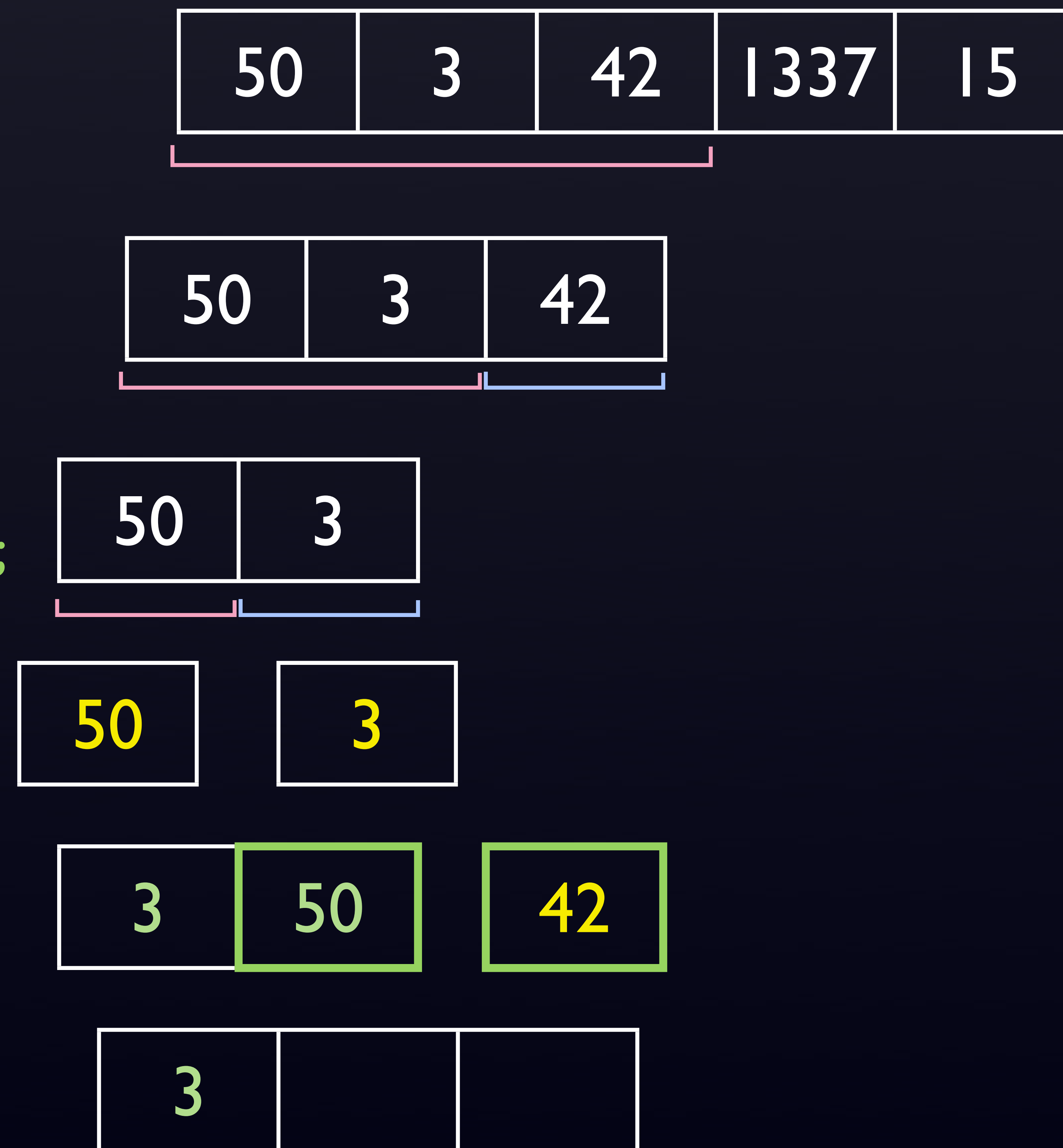
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
→       merge sorted halves;
```



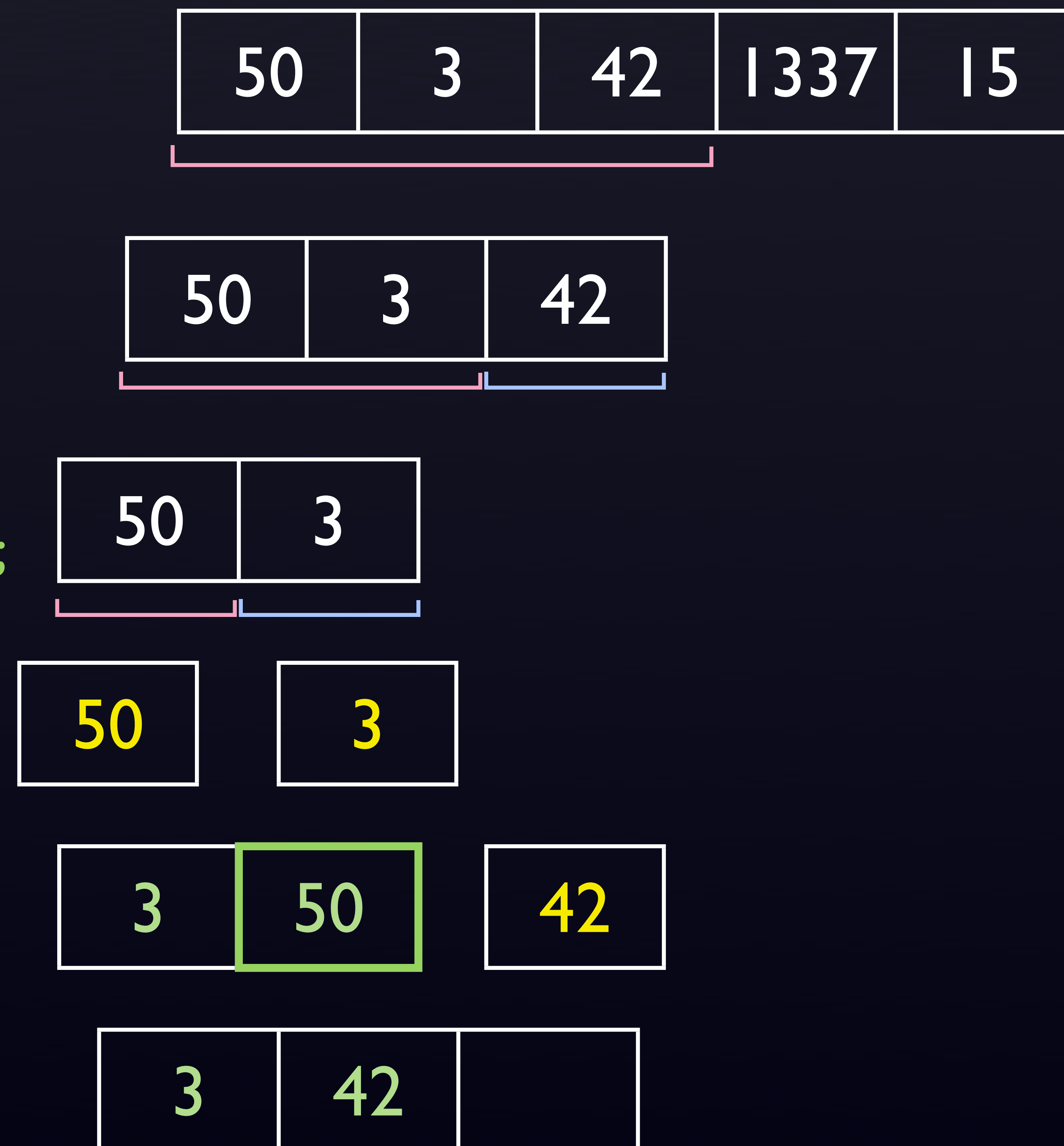
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



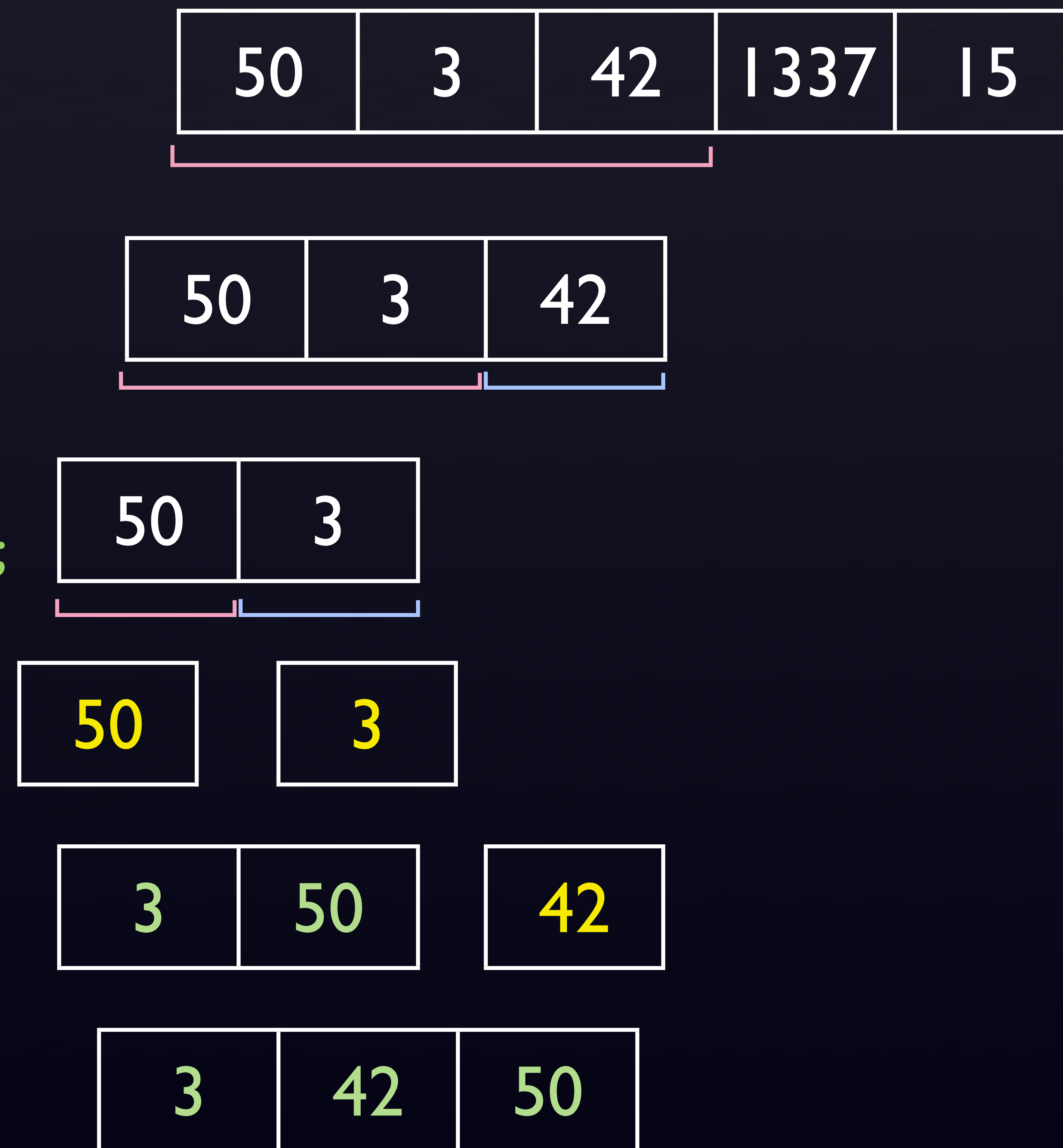
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



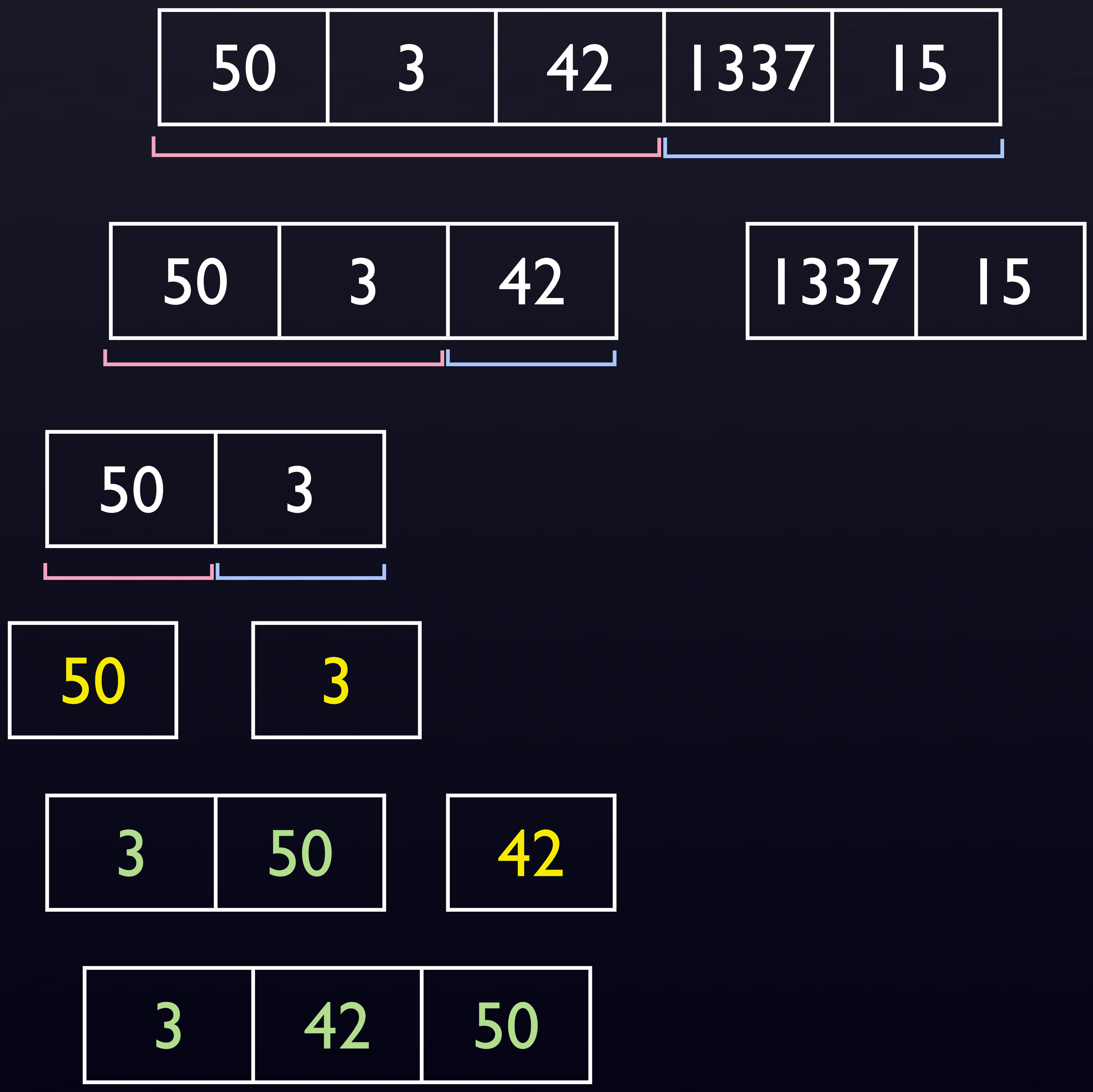
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



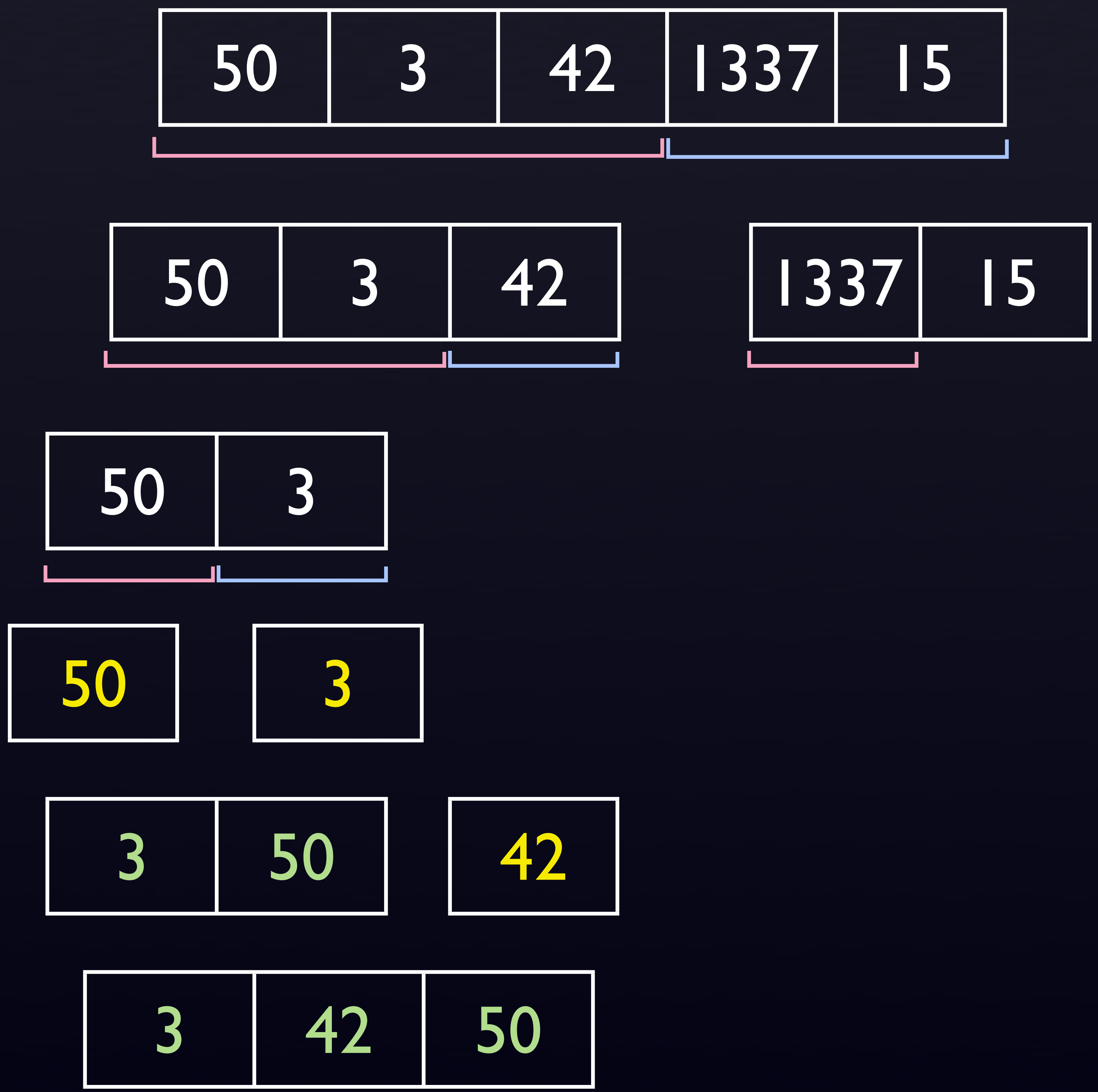
Merge Sort

mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;



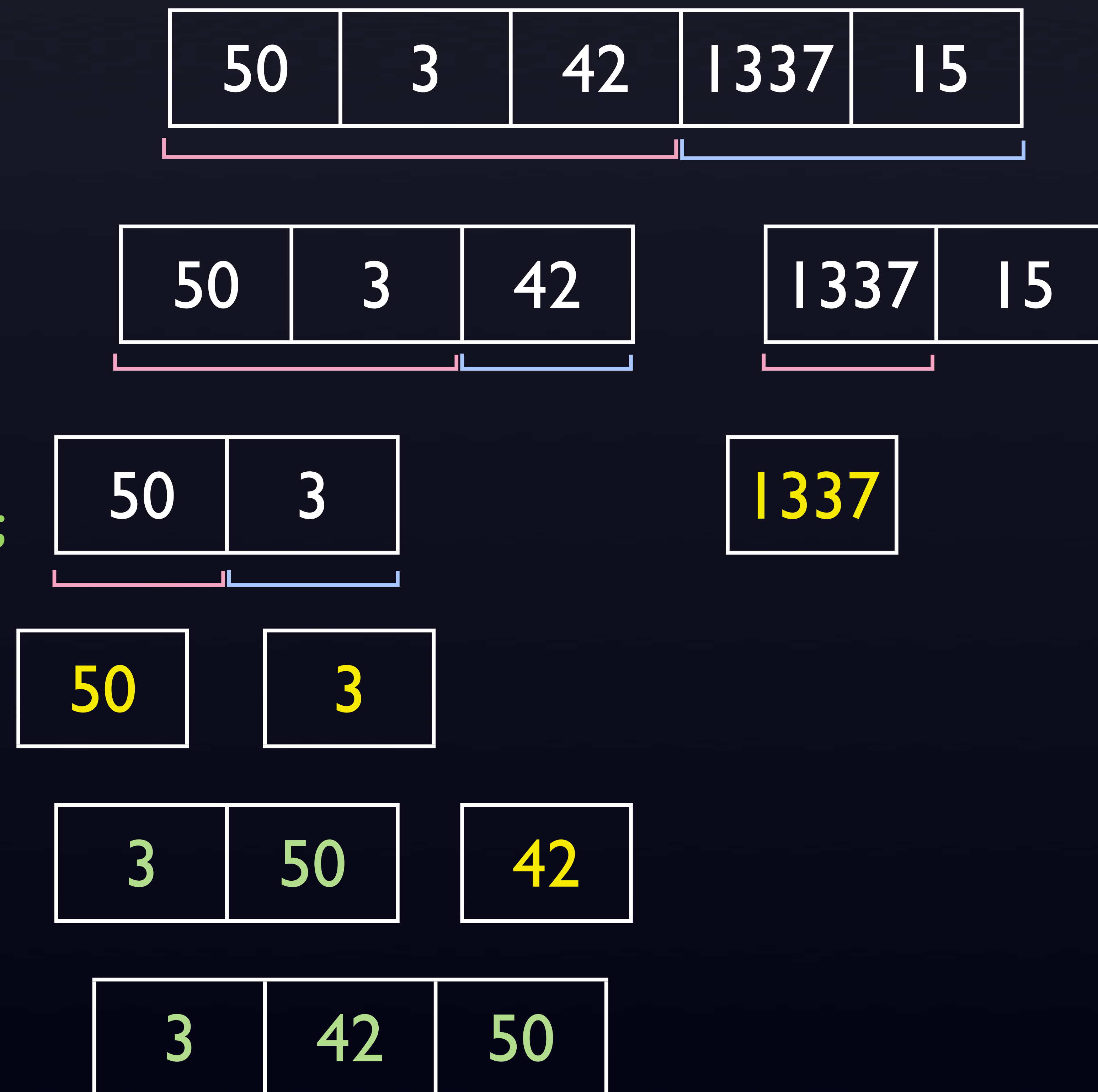
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



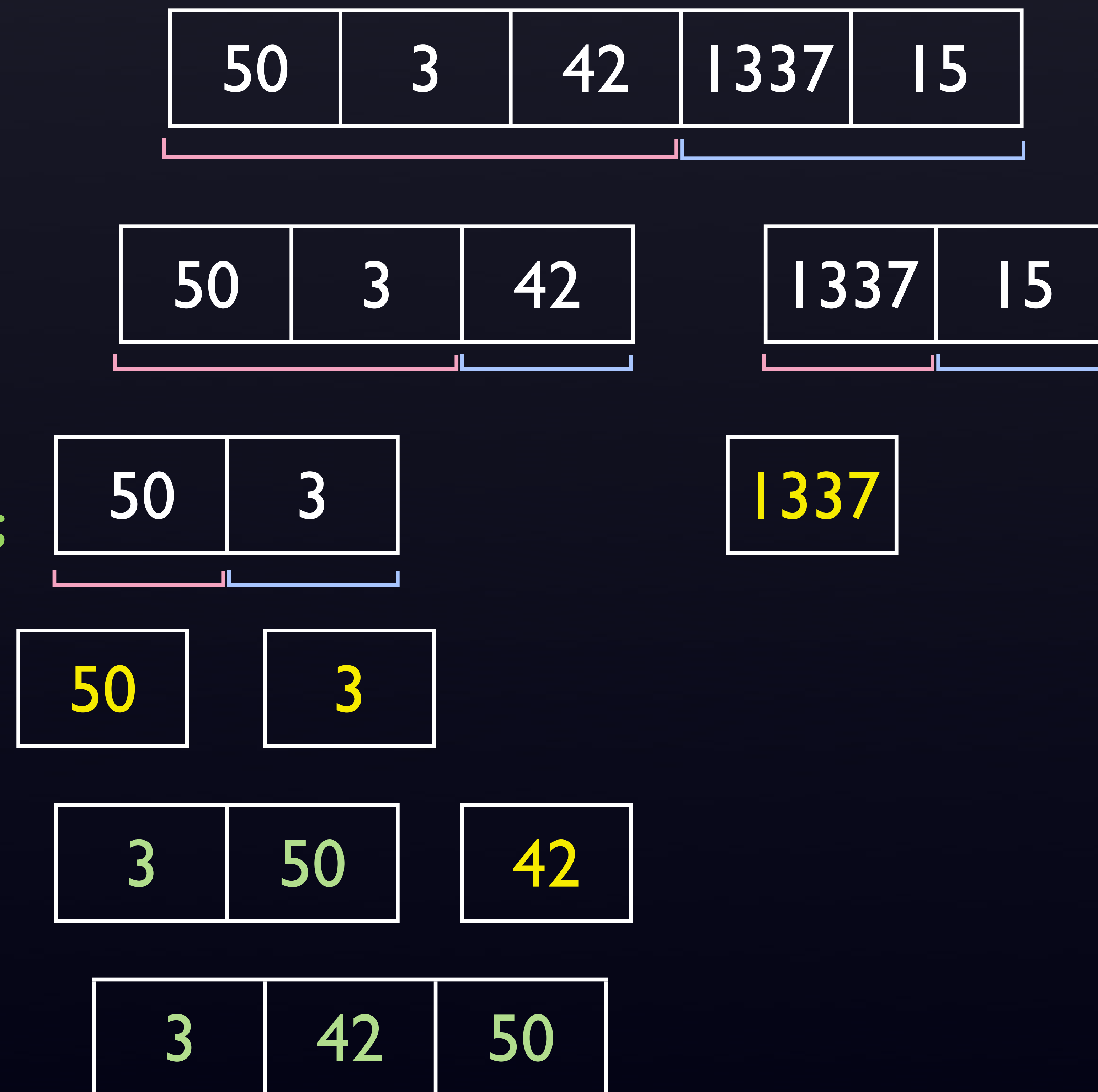
Merge Sort

mSort (list of n numbers)
if n < 2
→ return;
else
mSort left half;
mSort right half;
merge sorted halves;



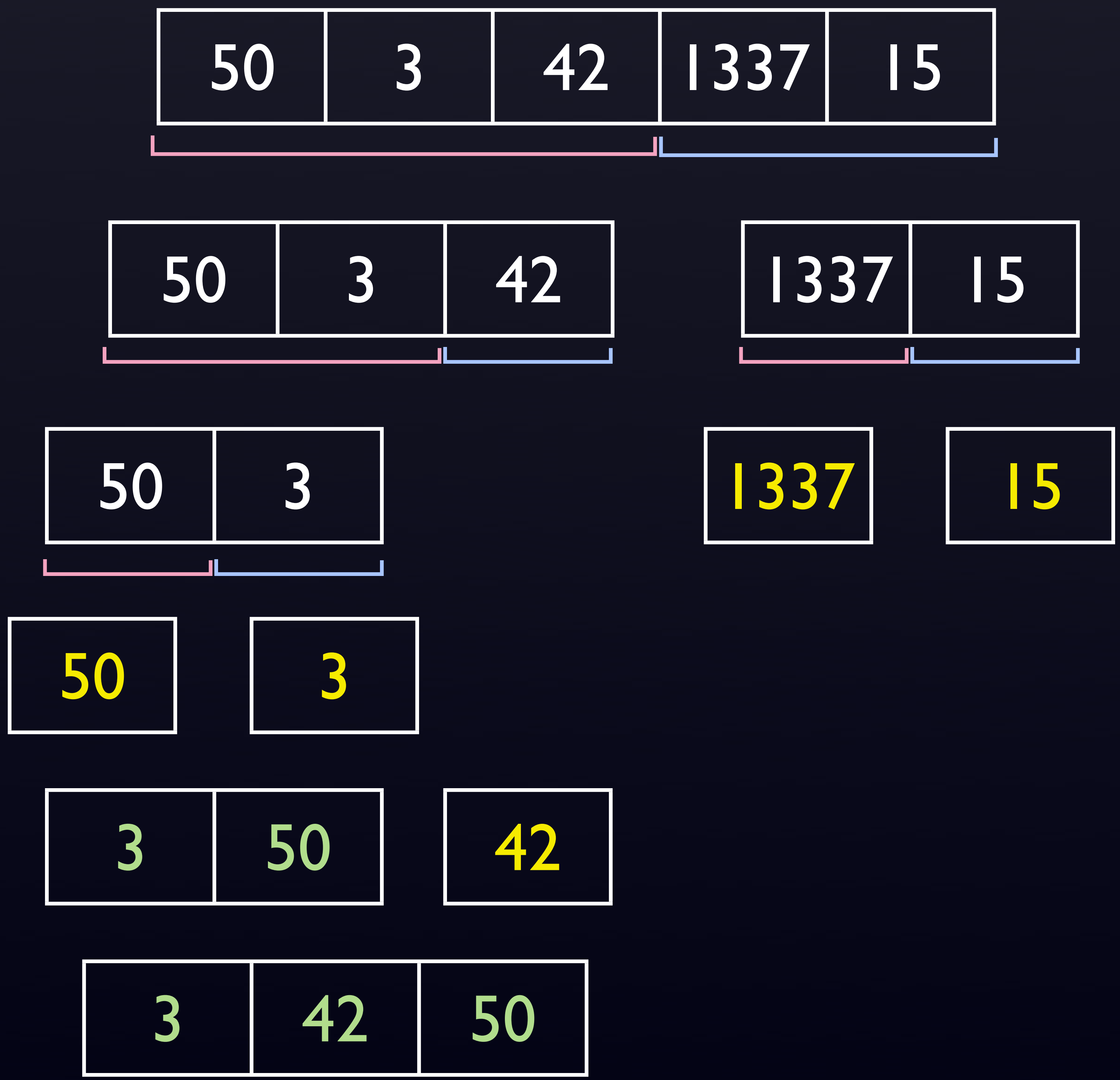
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
  →     mSort right half;
        merge sorted halves;
```



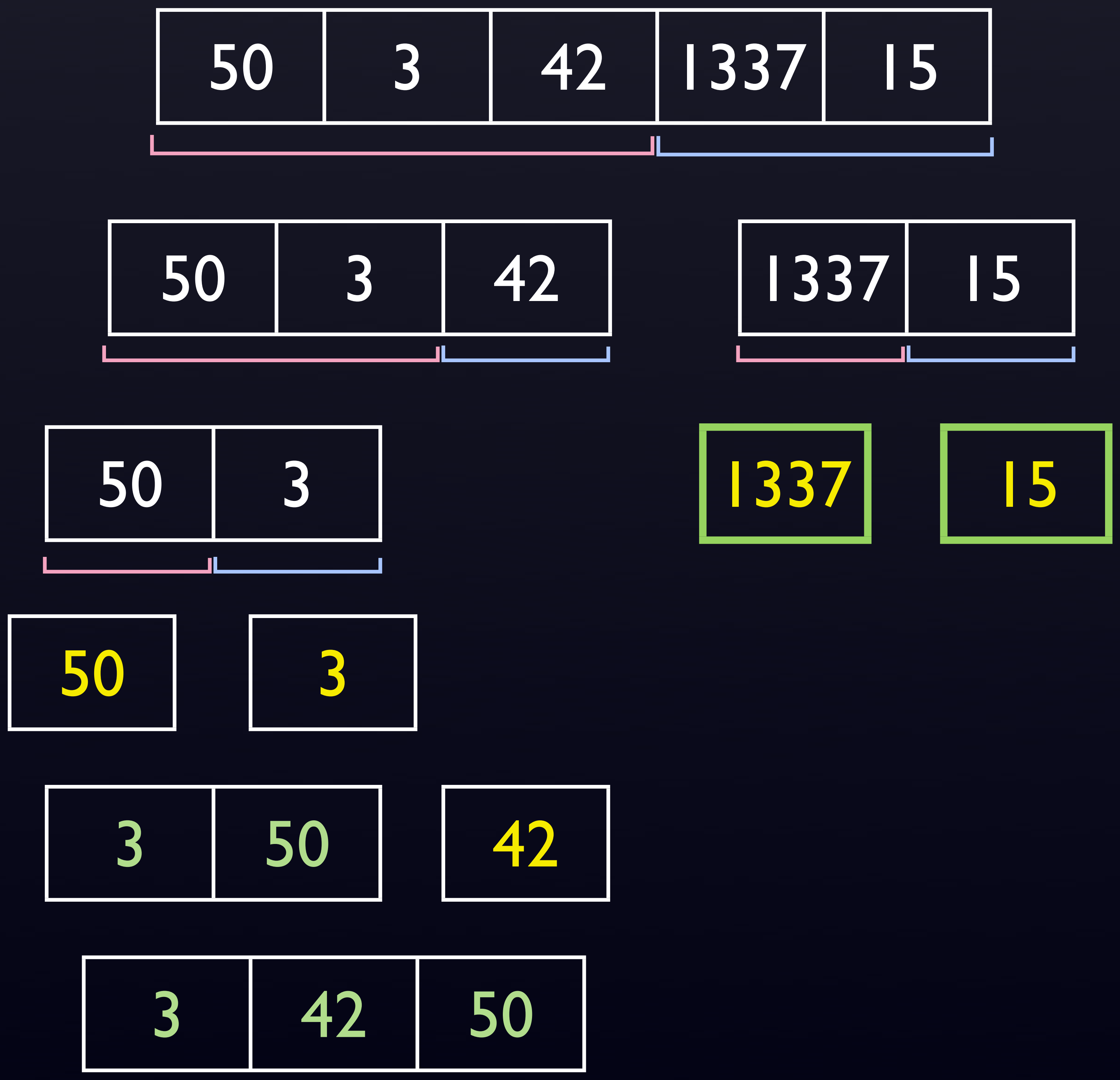
Merge Sort

mSort (list of n numbers)
    if n < 2
        → return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;



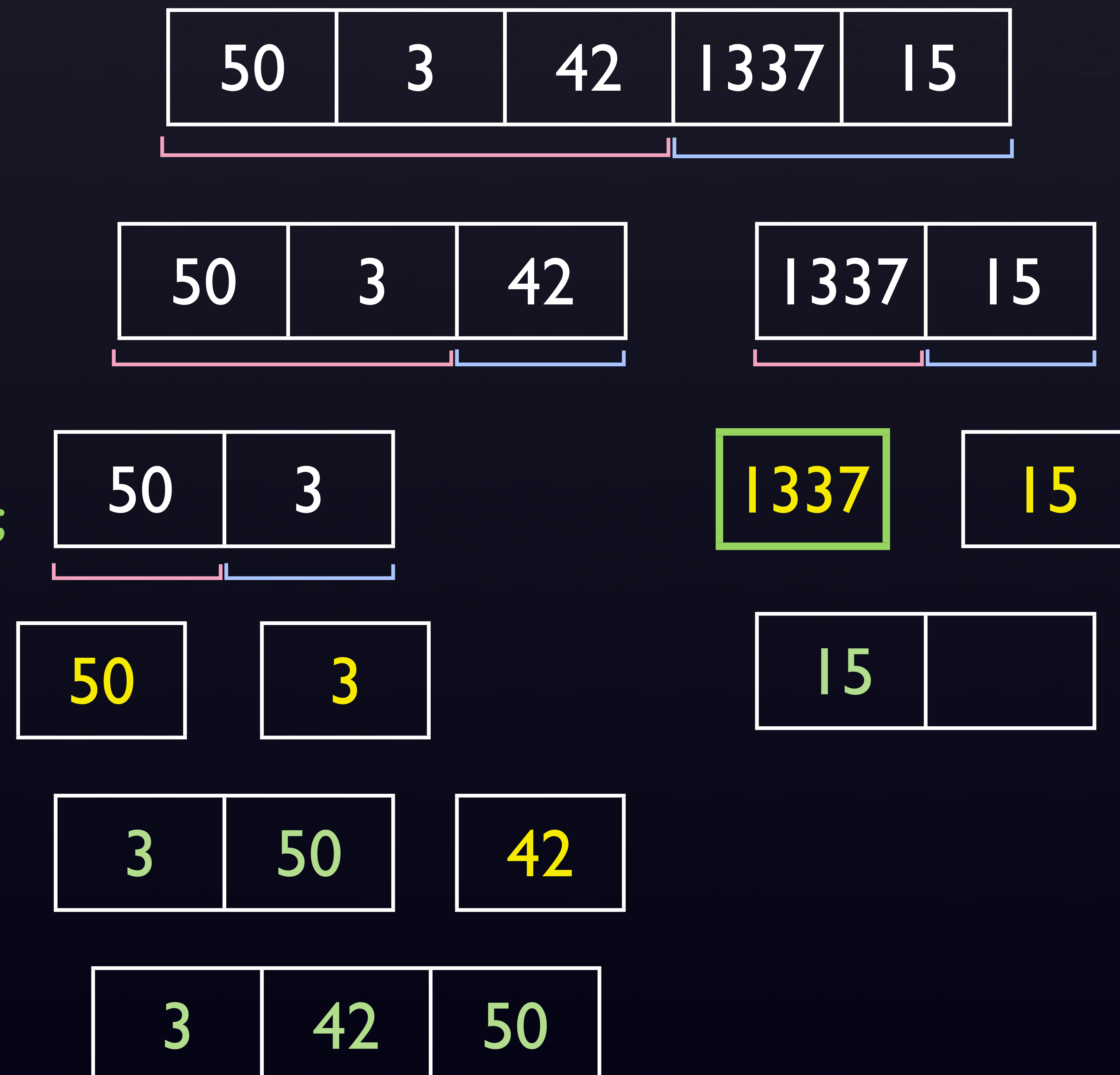
Merge Sort

mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;



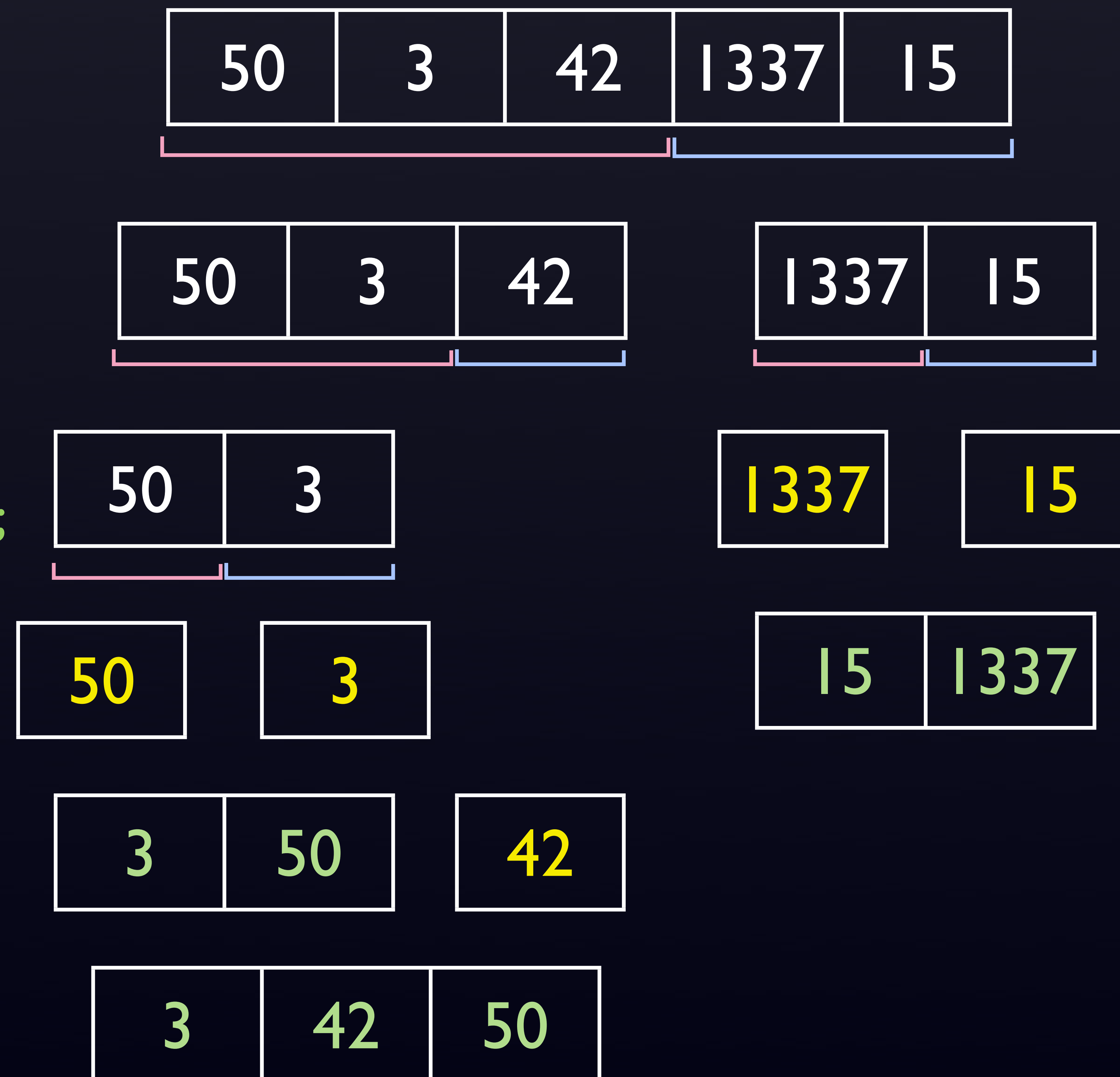
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



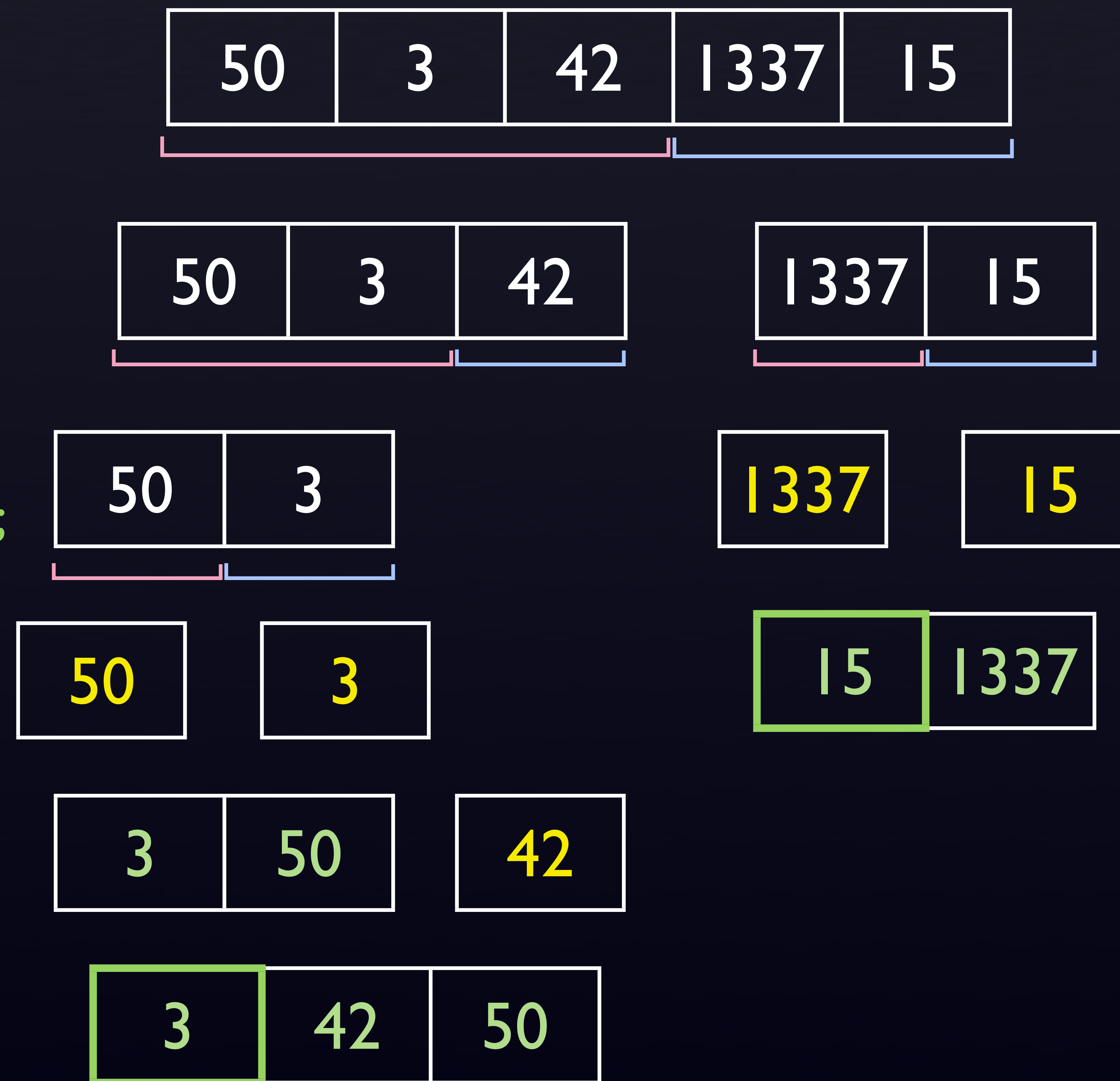
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
 →      merge sorted halves;
```



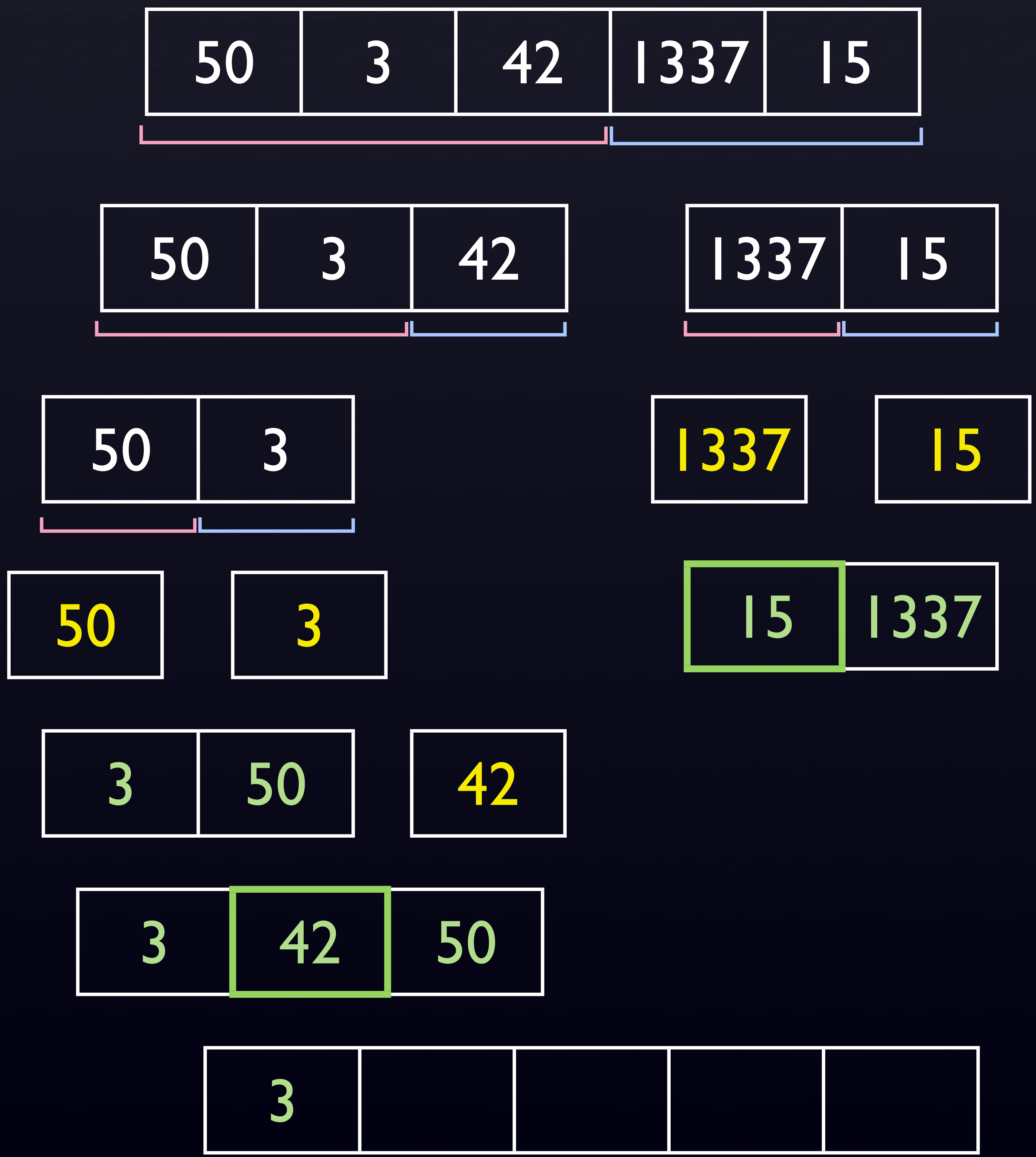
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```



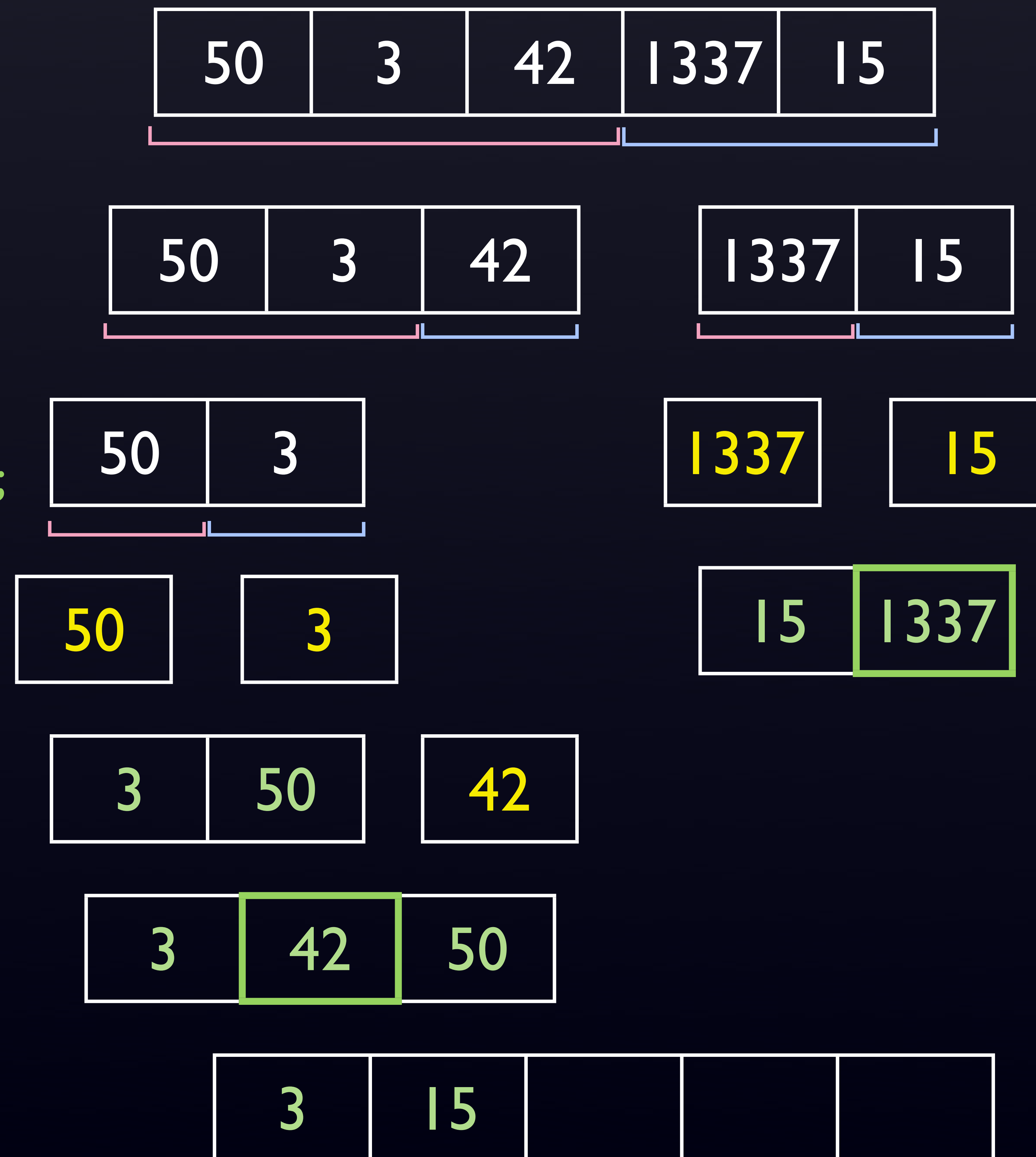
Merge Sort

mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
    → merge sorted halves;



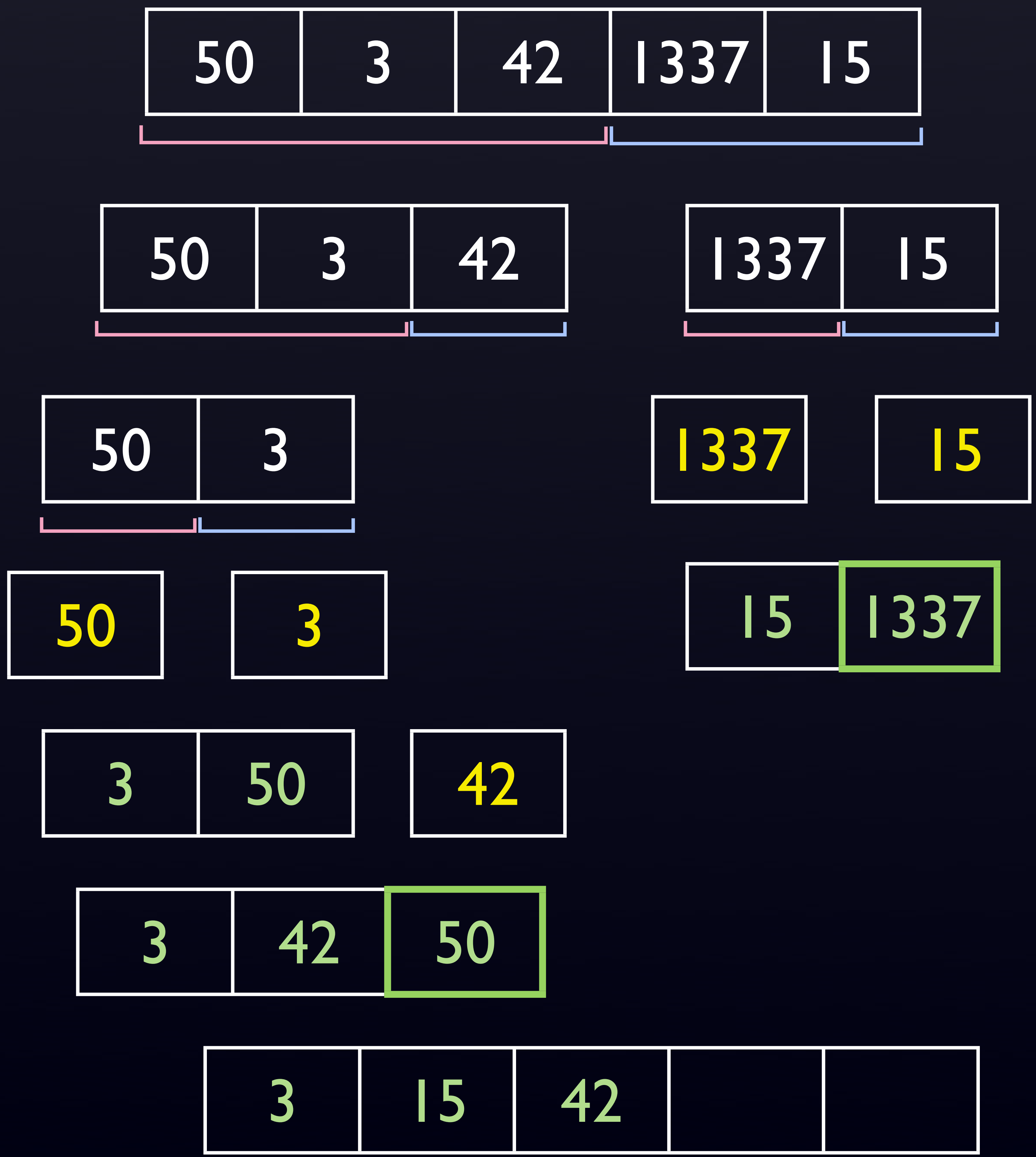
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



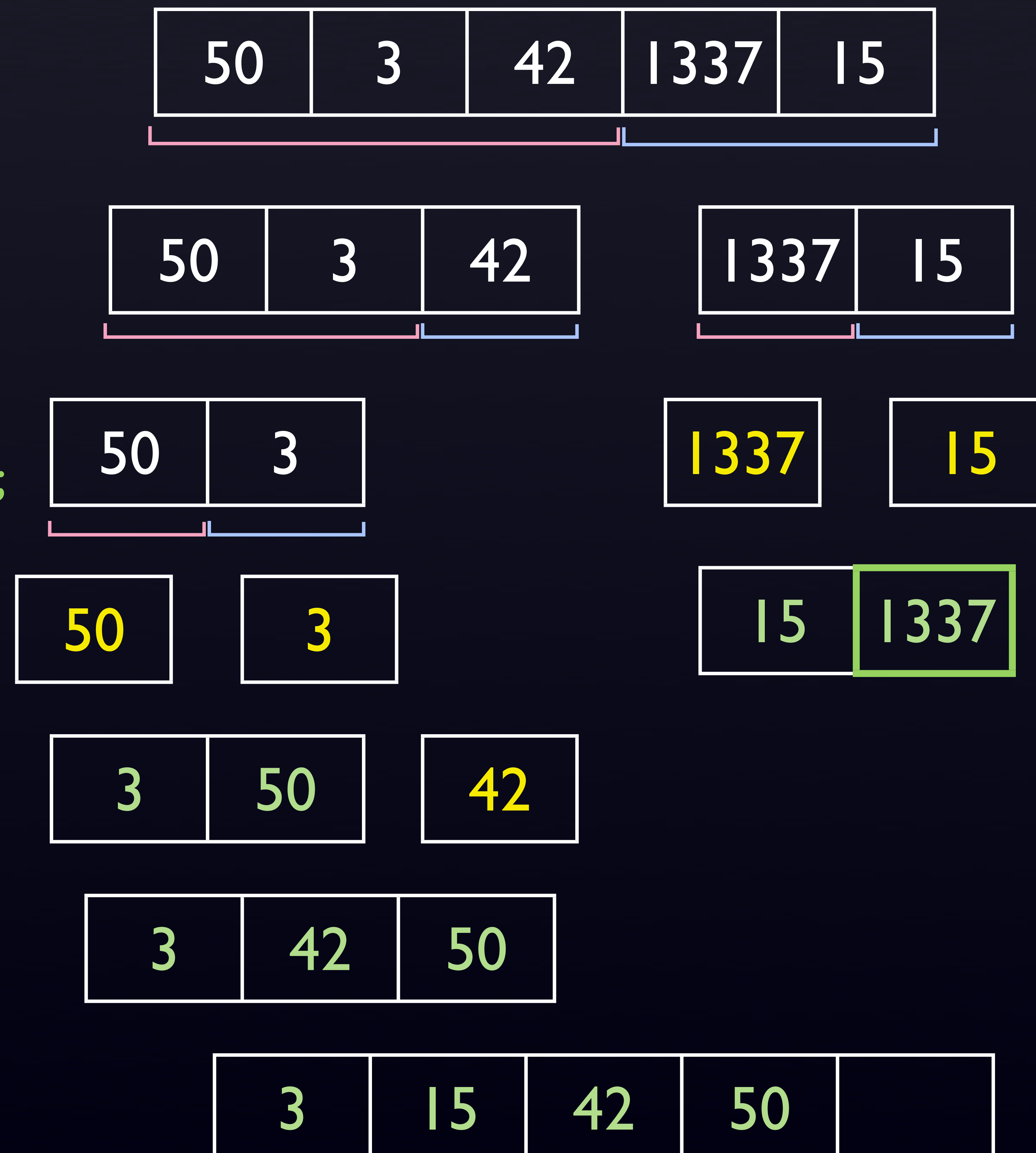
Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
  →     merge sorted halves;
```



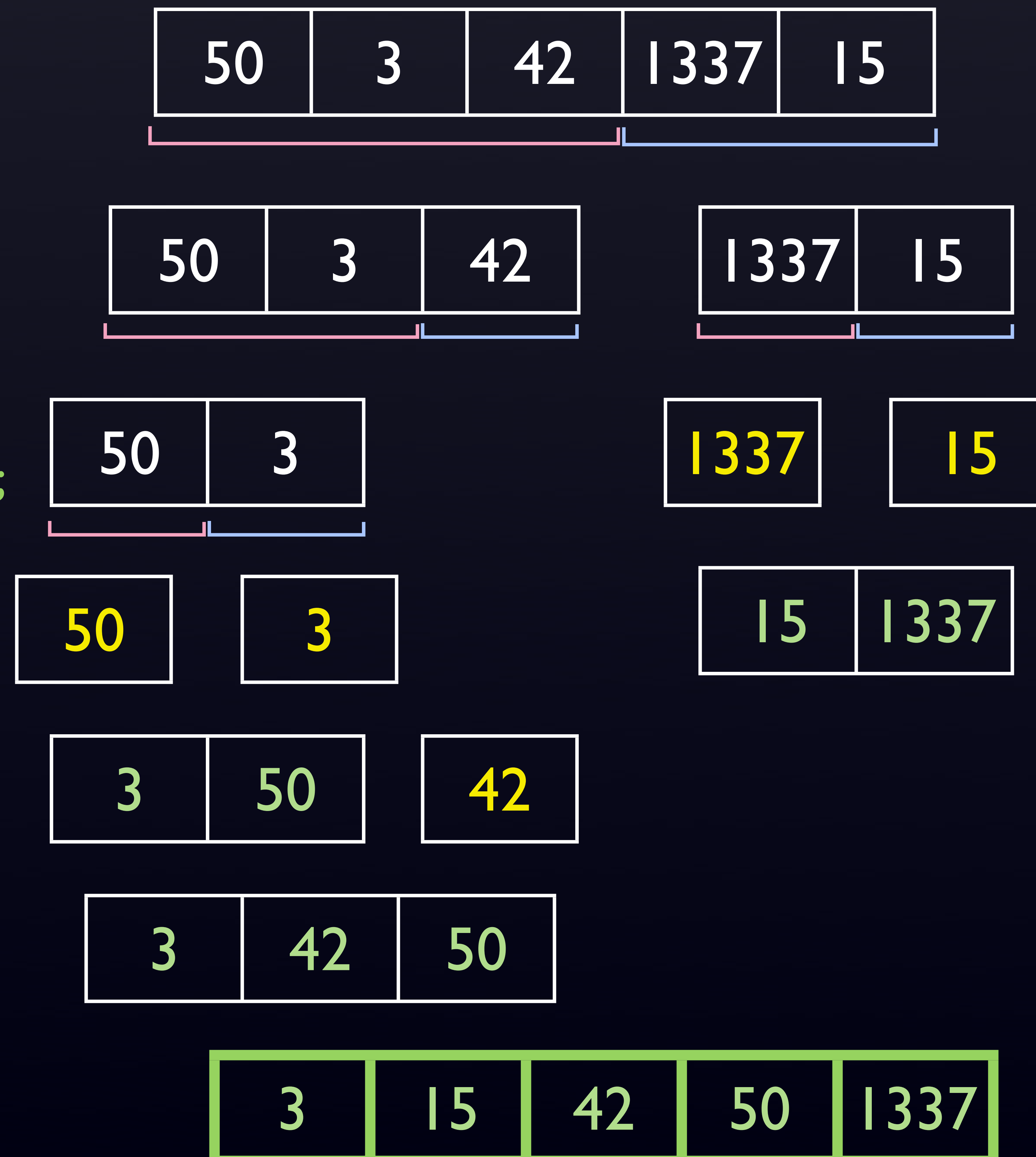
Merge Sort

mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;



Merge Sort

mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
→   merge sorted halves;

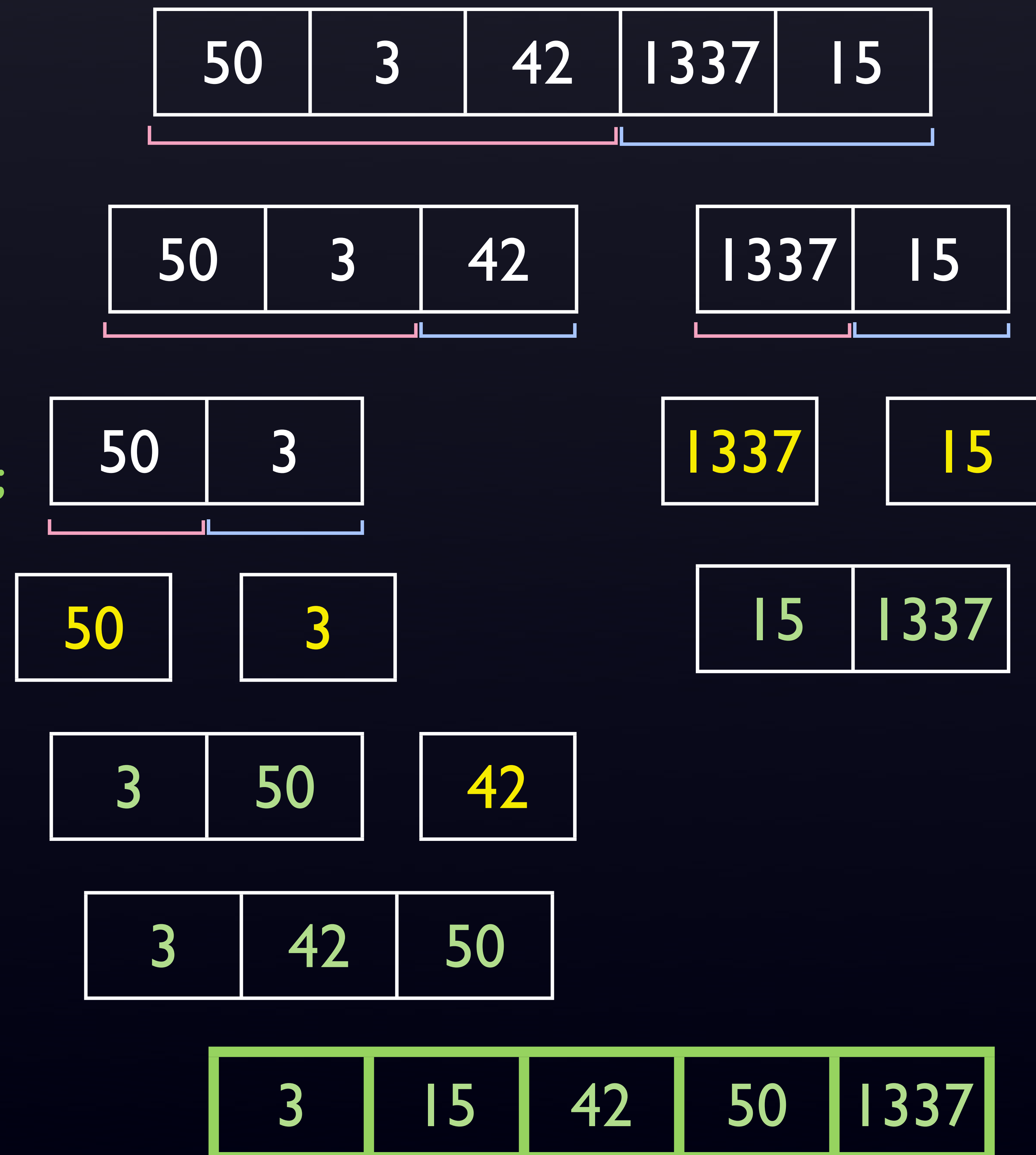


Merge Sort

```
mSort (list of n numbers)
    if n < 2
        return;
    else
        mSort left half;
        mSort right half;
        merge sorted halves;
```

$O(n \log n)$

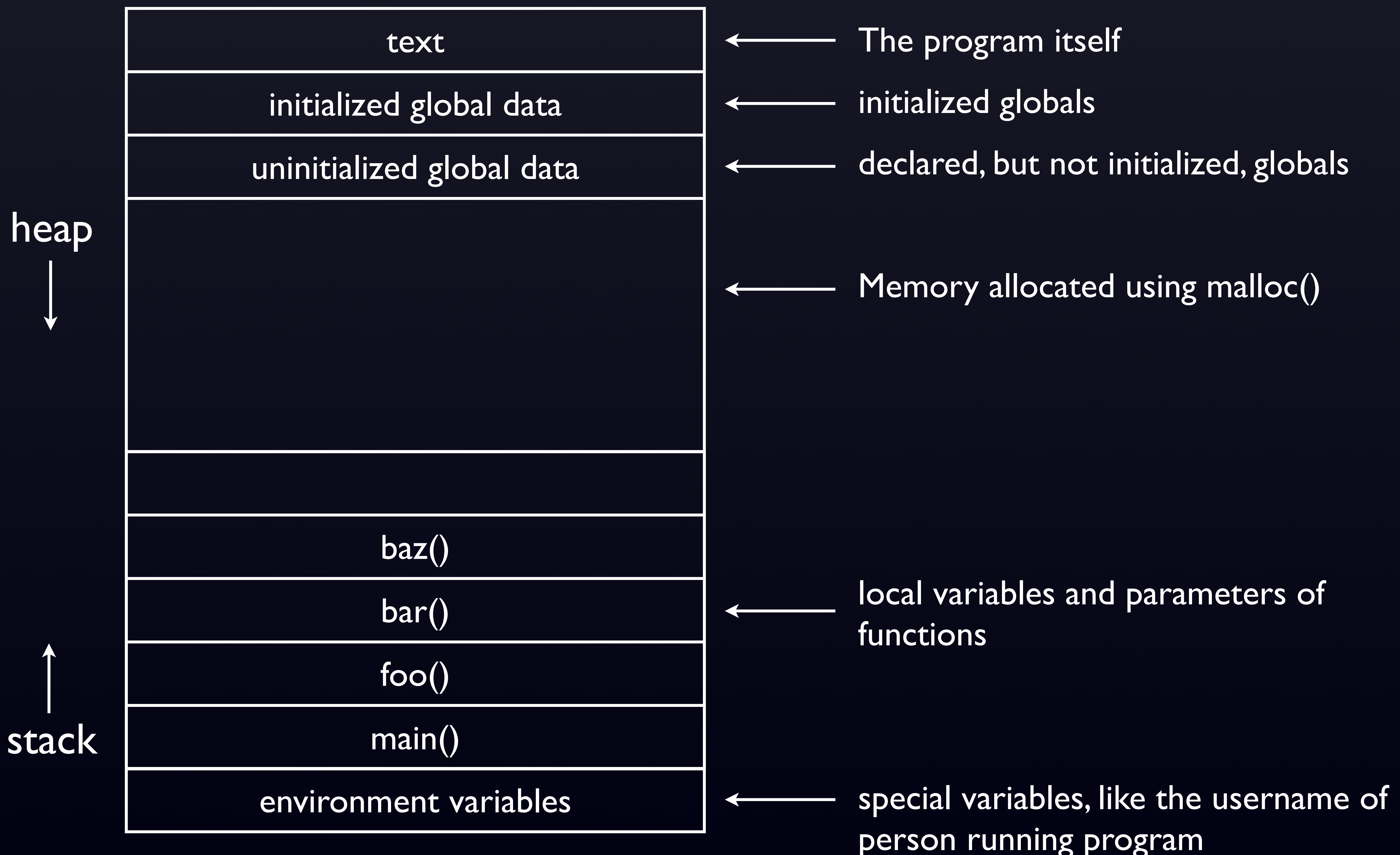
$\Omega(n \log n)$



CS50: Quiz 0

Memory, Stack, Heap

Memory Layout



What are pointers?

They are data types that refer to another location in memory, where other data is stored.

In this case, ptr "references" 50.

Just fyi, on 32-bit systems, pointers take up 32 bits, or 4 bytes, of space, just like an int does.

Dynamic Memory Allocation

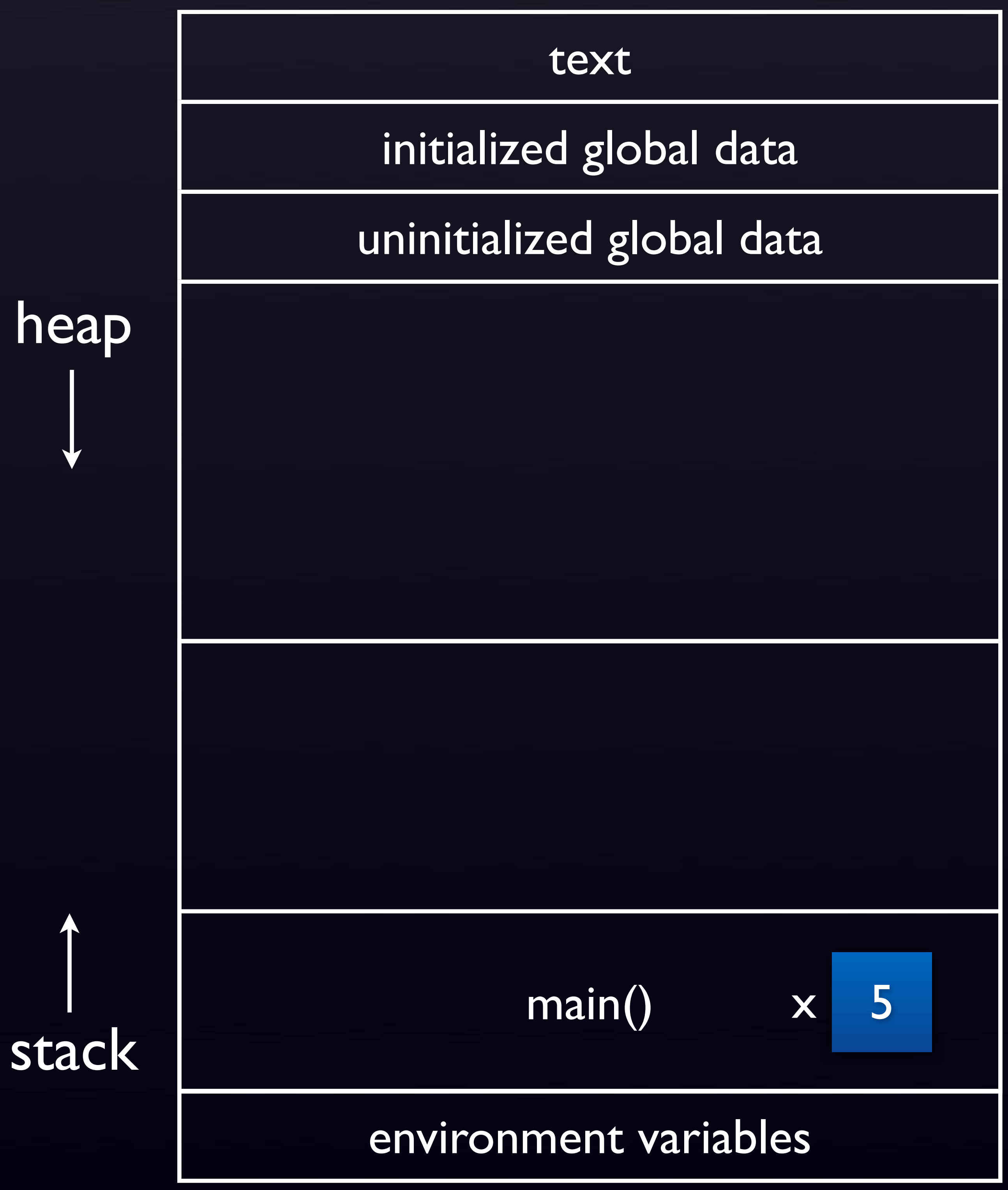
Recall, local variables are allocated on the stack, and we can't access them outside the scope of the functions or loops they belong to.

So, what dynamic memory allocation lets us do is hold on to data for the entire duration of the program.

This is done by:

1) Allocating that data in a permanent space on the heap.
2) Giving us a pointer to that location in memory.

malloc()




```
int
main(void)
{
        int x = 5;
        int *ptr = giveMeThreeInts();

        ptr[0] = 1;
        ptr[1] = 2;
        ptr[2] = 3;
}

int *
giveMeThreeInts(void)
{
        int *temp = malloc(sizeof(int) * 3);

        return temp;
}
```

malloc()

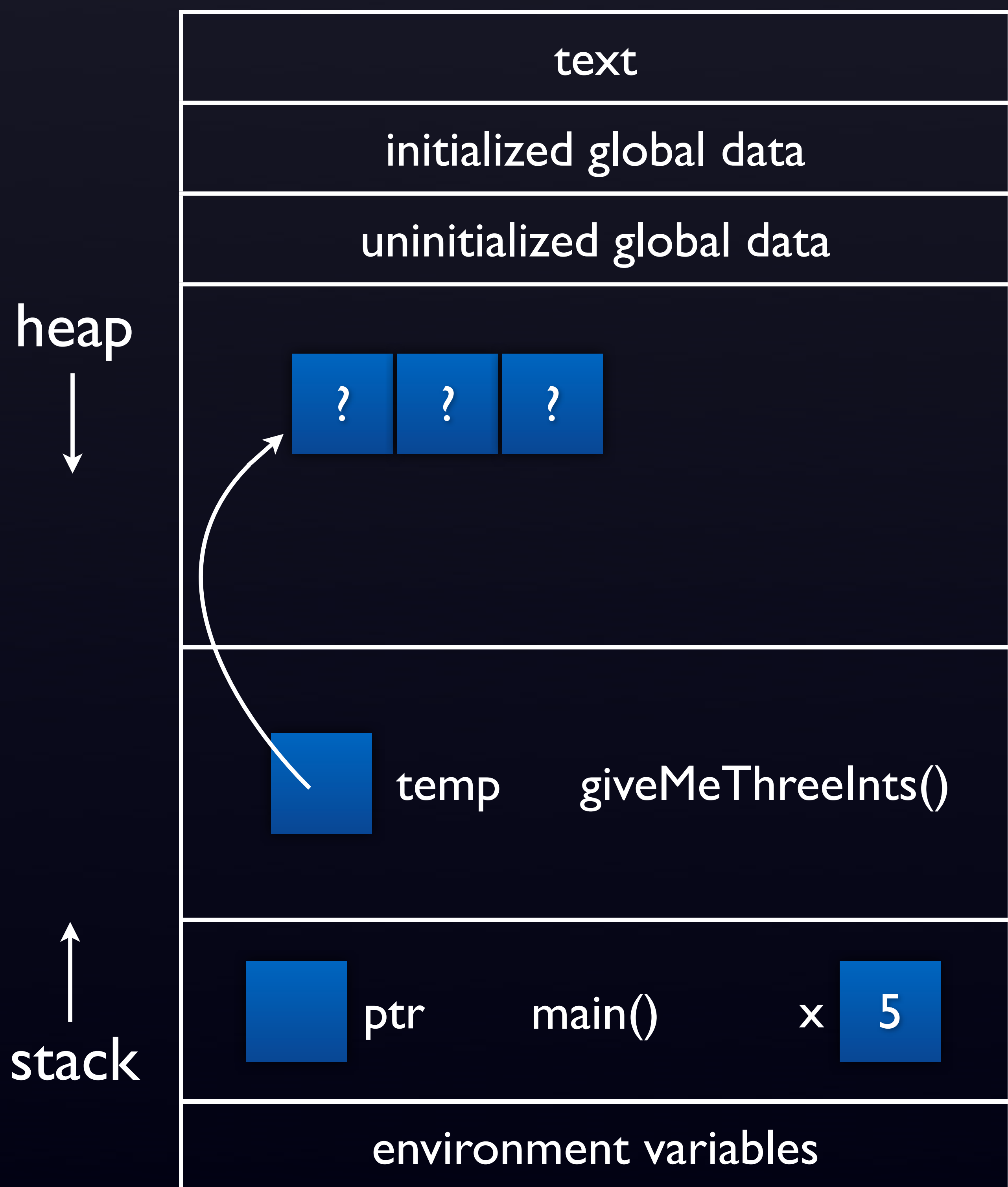

```
int
main(void)
{
        int x = 5;
        int *ptr = giveMeThreeInts();

        ptr[0] = 1;
        ptr[1] = 2;
        ptr[2] = 3;
}

int *
giveMeThreeInts(void)
{
        int *temp = malloc(sizeof(int) * 3);

        return temp;
}
```

malloc()



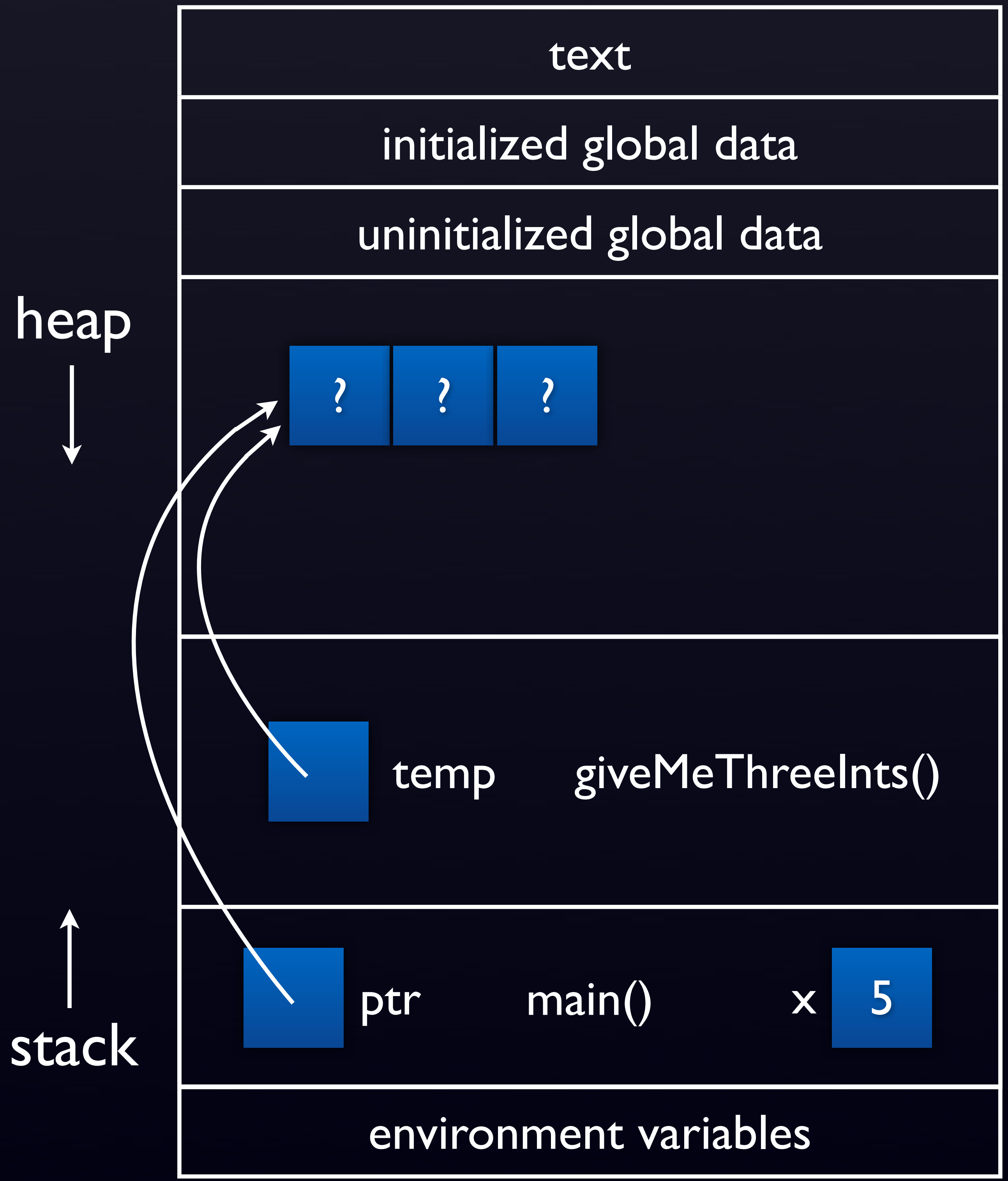

```
int
main(void)
{
        int x = 5;
        int *ptr = giveMeThreeInts();

        ptr[0] = 1;
        ptr[1] = 2;
        ptr[2] = 3;
}

int *
giveMeThreeInts(void)
{
        int *temp = malloc(sizeof(int) * 3);

        return temp;
}
```

malloc()



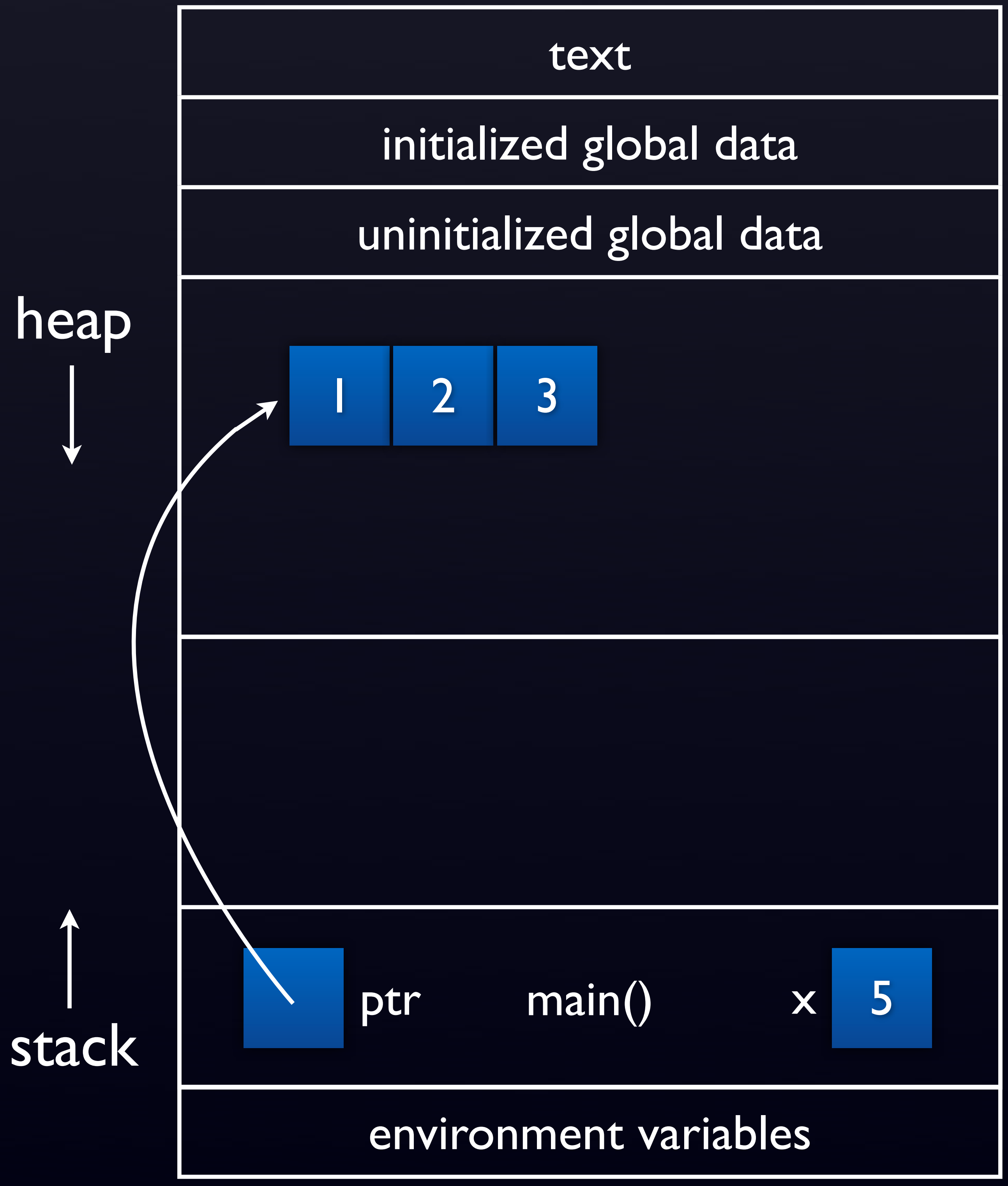

```
int
main(void)
{
        int x = 5;
        int *ptr = giveMeThreeInts();

        ptr[0] = 1;
        ptr[1] = 2;
        ptr[2] = 3;
}

int *
giveMeThreeInts(void)
{
        int *temp = malloc(sizeof(int) * 3);

        return temp;
}
```

malloc()




```
int
main(void)
{
        int x = 5;
        int *ptr = giveMeThreeInts();

        ptr[0] = 1;
        ptr[1] = 2;
        ptr[2] = 3;
}

int *
giveMeThreeInts(void)
{
        int *temp = malloc(sizeof(int) * 3);

        return temp;
}
```

CS50: Quiz 0

Pointers

Recall Binky

```
int
main(void)
{
    // usually done in same step
--> int *ptr;
    ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    int x = 5;
    ptr = &x;

    return 0;
}
```

Recall Binky

```
int
main(void)
{
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    int x = 5;
    ptr = &x;

    return 0;
}
```

Recall Binky

```
int
main(void)
{
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));

    if (ptr == NULL)    // did malloc work?
        return 1;

    *ptr = 1;
    free(ptr);

    int x = 5;
    ptr = &x;

    return 0;
}
```

Recall Binky

```
int
main(void)
{
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

→   *ptr = 1;     // sets contents of memory to 1
    free(ptr);

    int x = 5;
    ptr = &x;

    return 0;
}
```

Recall Binky

```
int
main(void)
{
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);      // frees up memory from heap

    int x = 5;
    ptr = &x;

    return 0;
}
```

Recall Binky

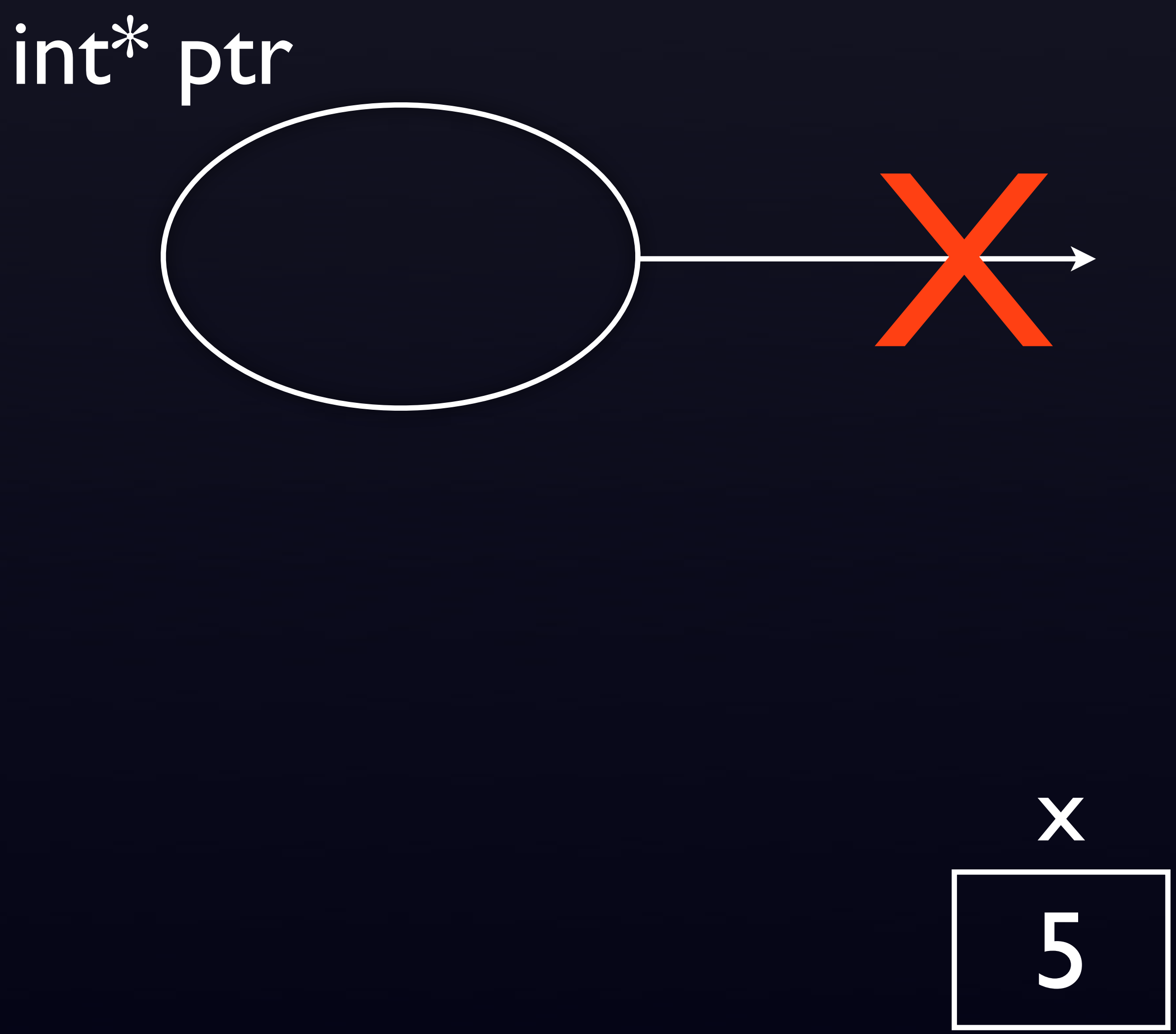
```
int
main(void)
{
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    int x = 5;
    ptr = &x;

    return 0;
}
```



Recall Binky

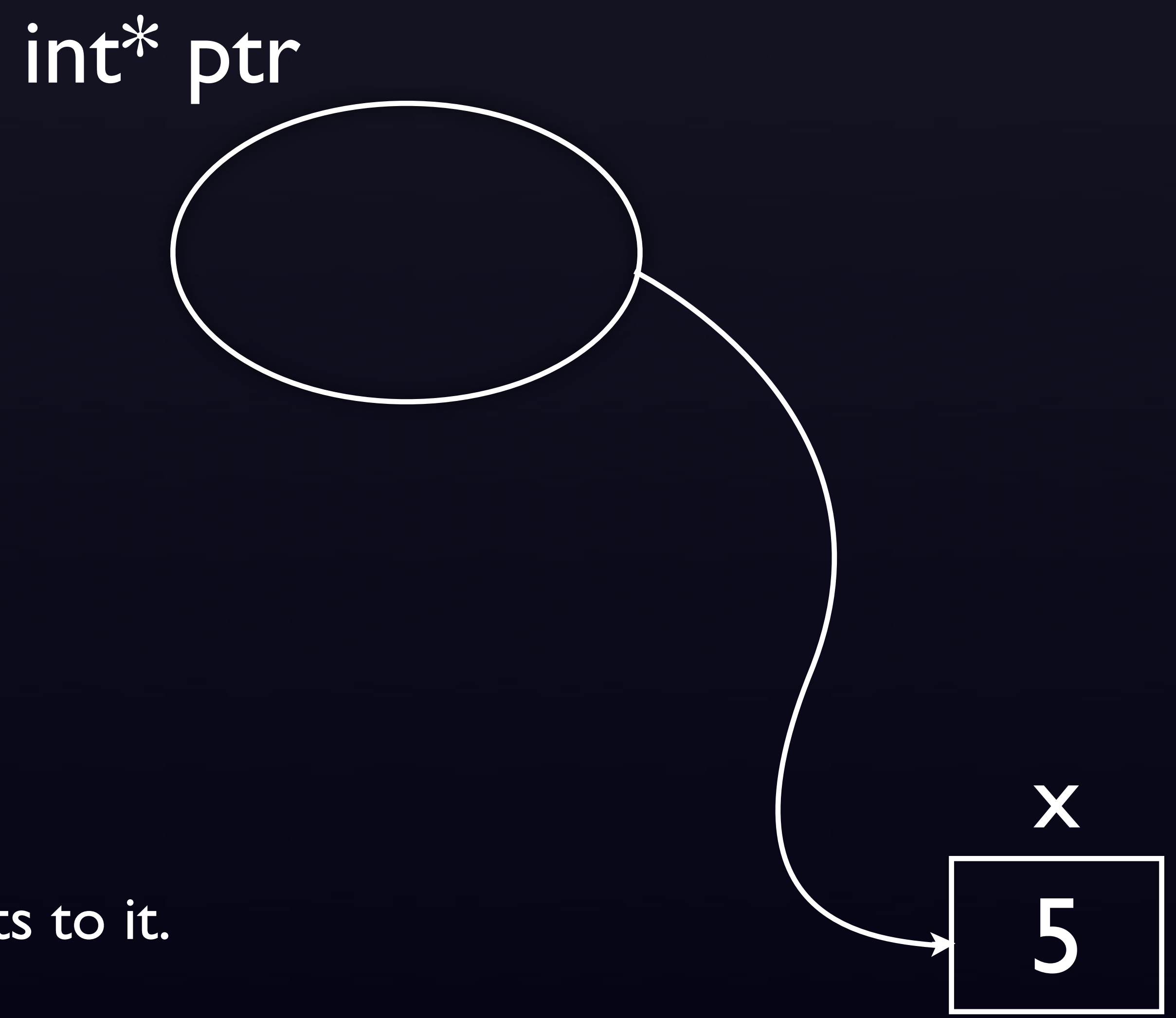
```
int
main(void)
{
    // usually done in same step
    int *ptr;
    ptr = malloc(sizeof(int));

    if (ptr == NULL)
        return 1;

    *ptr = 1;
    free(ptr);

    int x = 5;
    ptr = &x;        // gets address of x, points to it.

    return 0;
}
```



Pointer Arithmetic

```
int
main(void)
{
    int *ptr = malloc(sizeof(int) * 3);

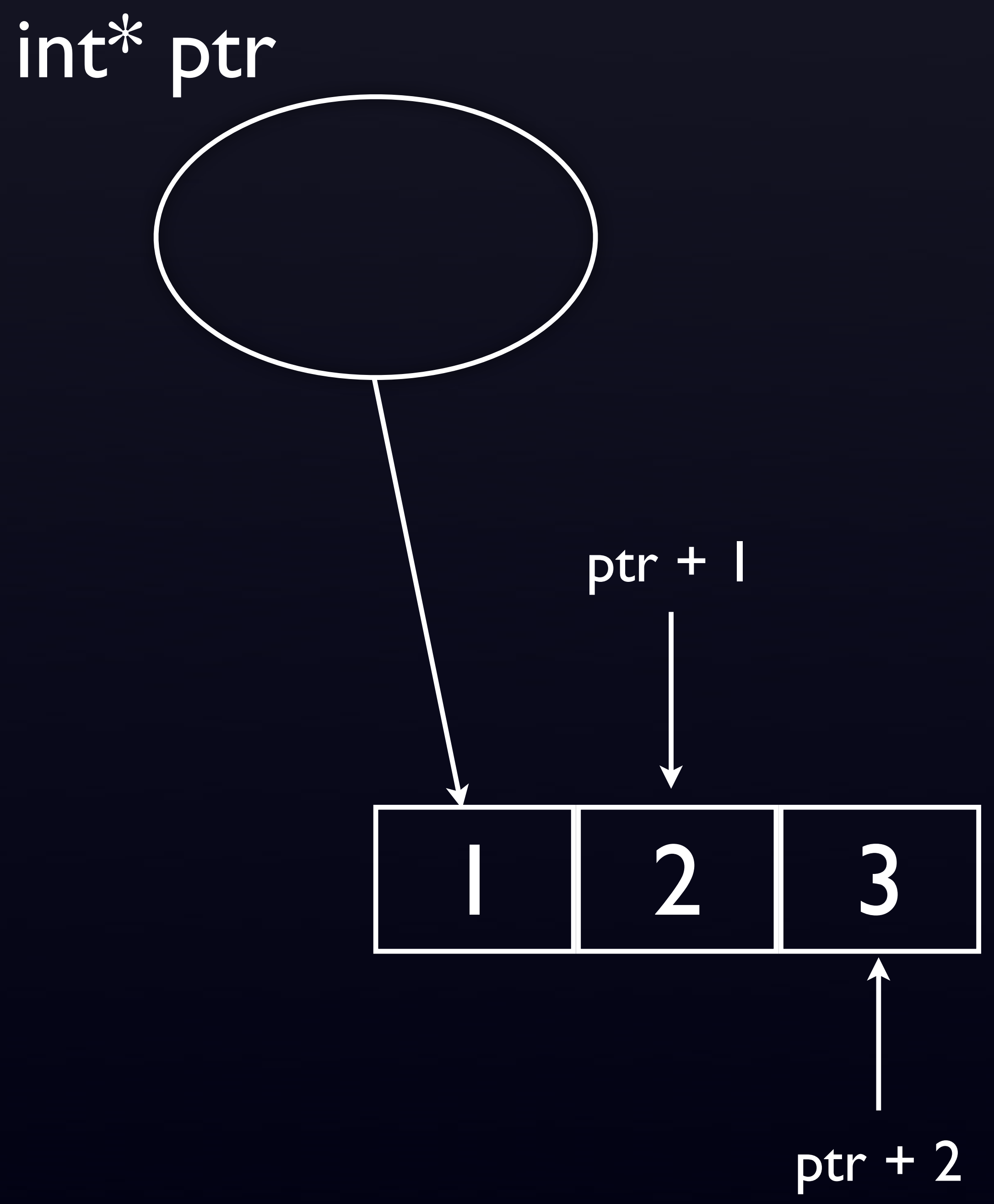
    *ptr = 1;
    *(ptr + 1) = 2;    // one int down from ptr
    *(ptr + 2) = 3;

    printf("%d", *(ptr + 1));

    ptr++;

    printf("%d", *(ptr + 1));

    ptr--;
    free(ptr);
}
```



Pointer Arithmetic

```
int
main(void)
{
    int *ptr = malloc(sizeof(int) * 3);

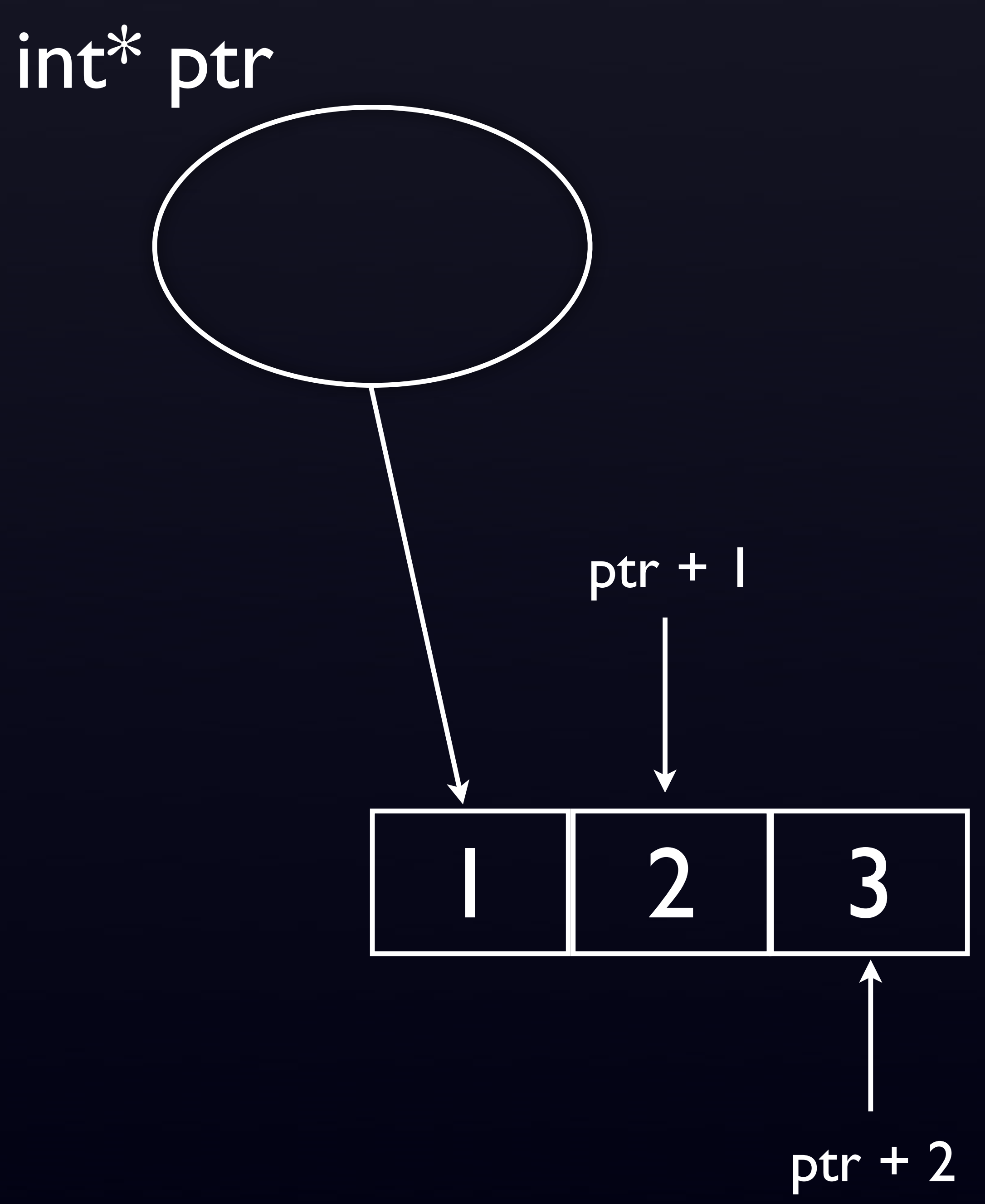
    *ptr = 1;
    *(ptr + 1) = 2;
    *(ptr + 2) = 3;

    printf("%d", *(ptr + 1));    // prints out 2

    ptr++;

    printf("%d", *(ptr + 1));

    ptr--;
    free(ptr);
}
```



Pointer Arithmetic

```
int
main(void)
{
    int *ptr = malloc(sizeof(int) * 3);

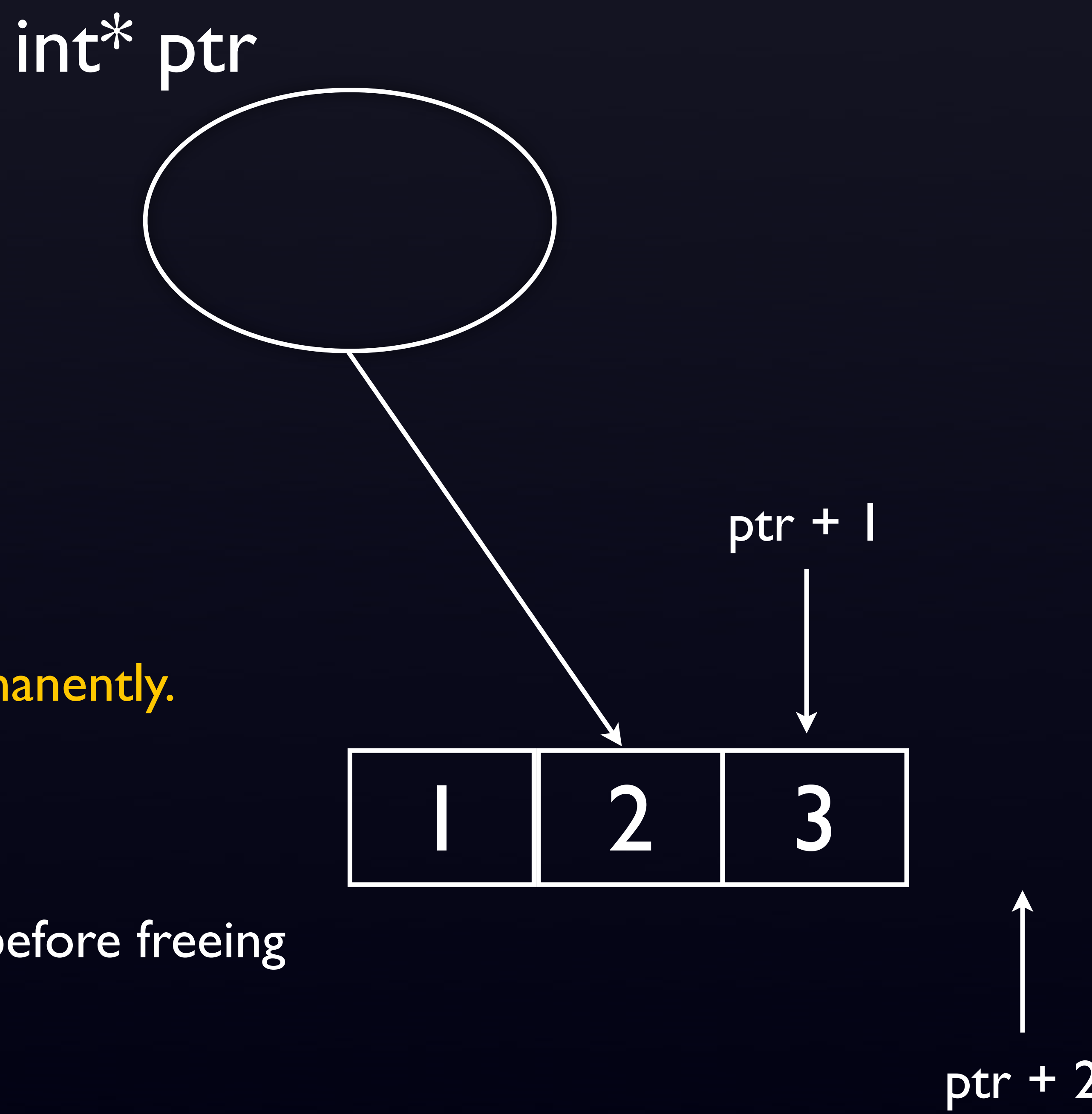
    *ptr = 1;
    *(ptr + 1) = 2;
    *(ptr + 2) = 3;

    printf("%d", *(ptr + 1));

    ptr++;                    // changes ptr permanently.

    printf("%d", *(ptr + 1));   // now prints out 3

    ptr--;   //move back to original ptr location before freeing
    free(ptr);
}
```



Pointer Arithmetic with Strings

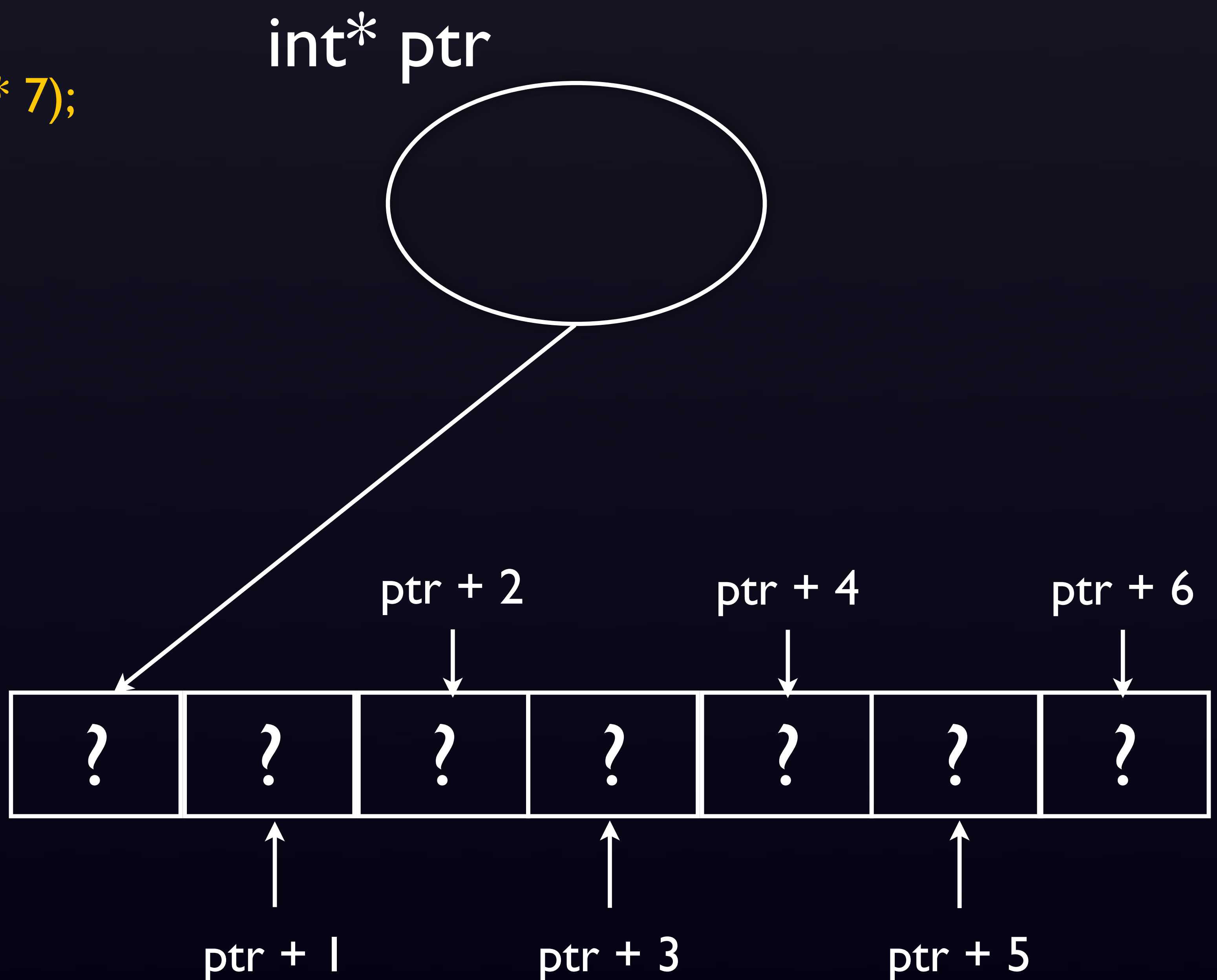
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



Pointer Arithmetic with Strings

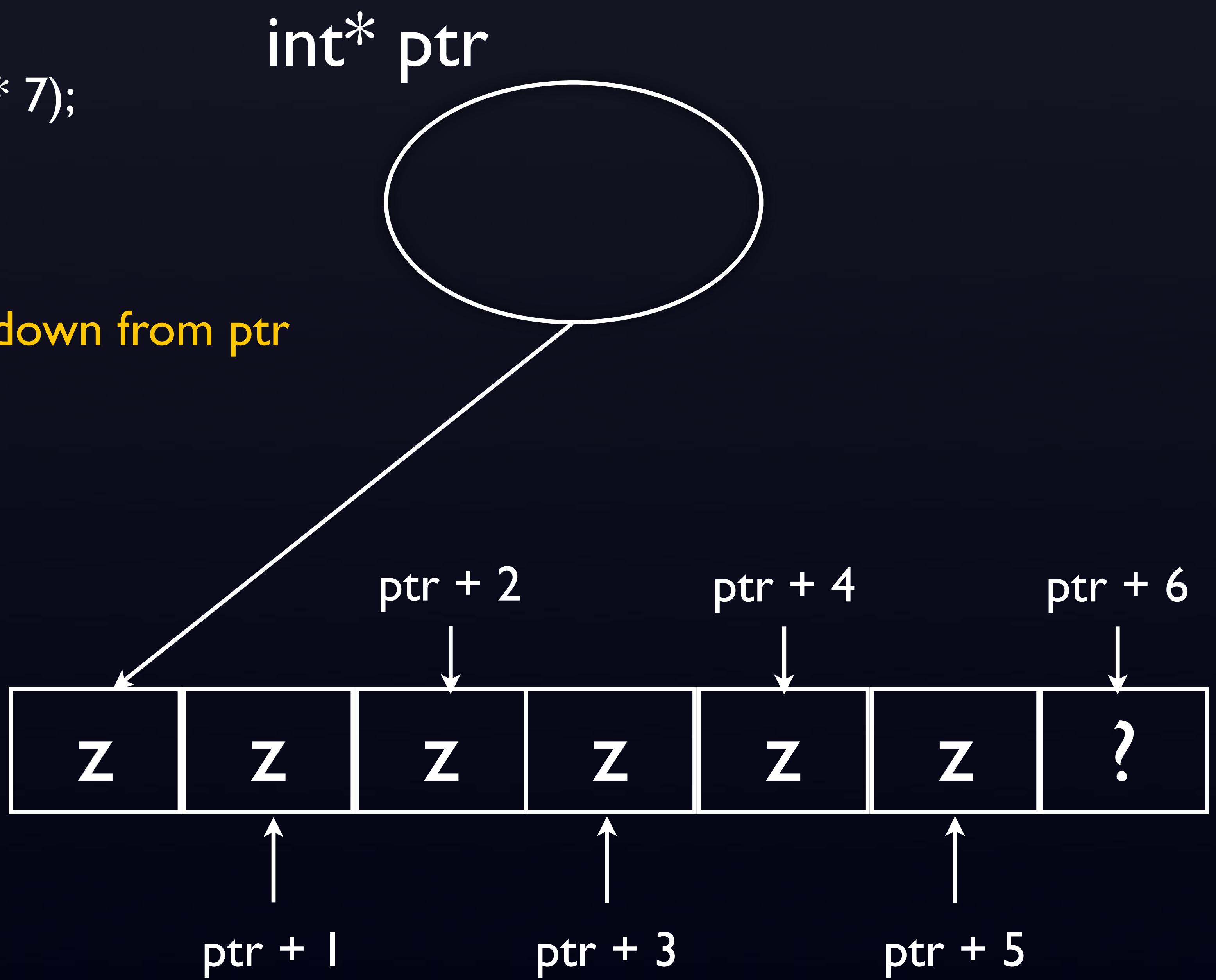
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = 'z';   //  i chars down from ptr
    }

    ptr[6] = '\0';

    while (*ptr != '\0')
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



Pointer Arithmetic with Strings

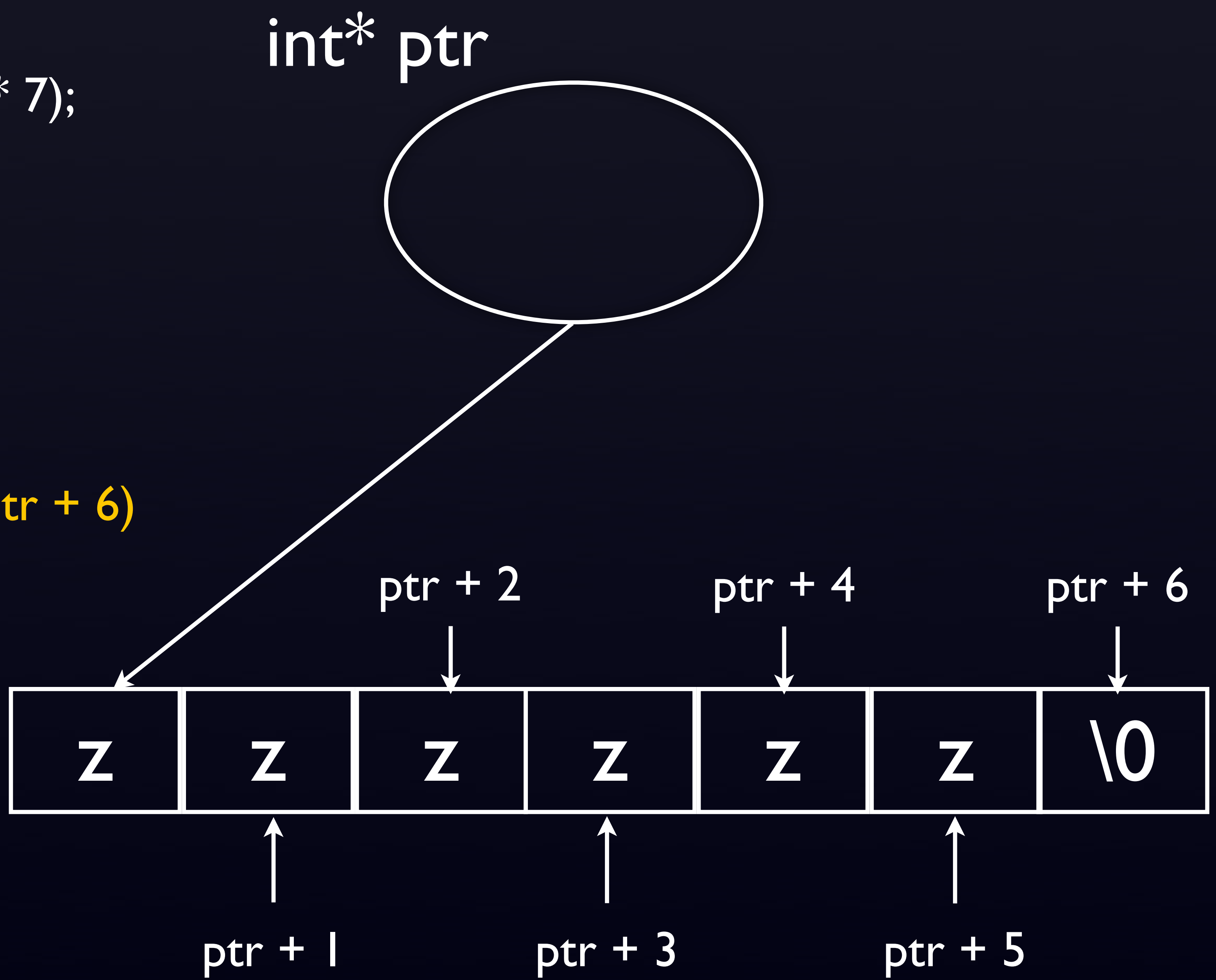
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;  //shorthand for *(ptr + 6)

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



Pointer Arithmetic with Strings

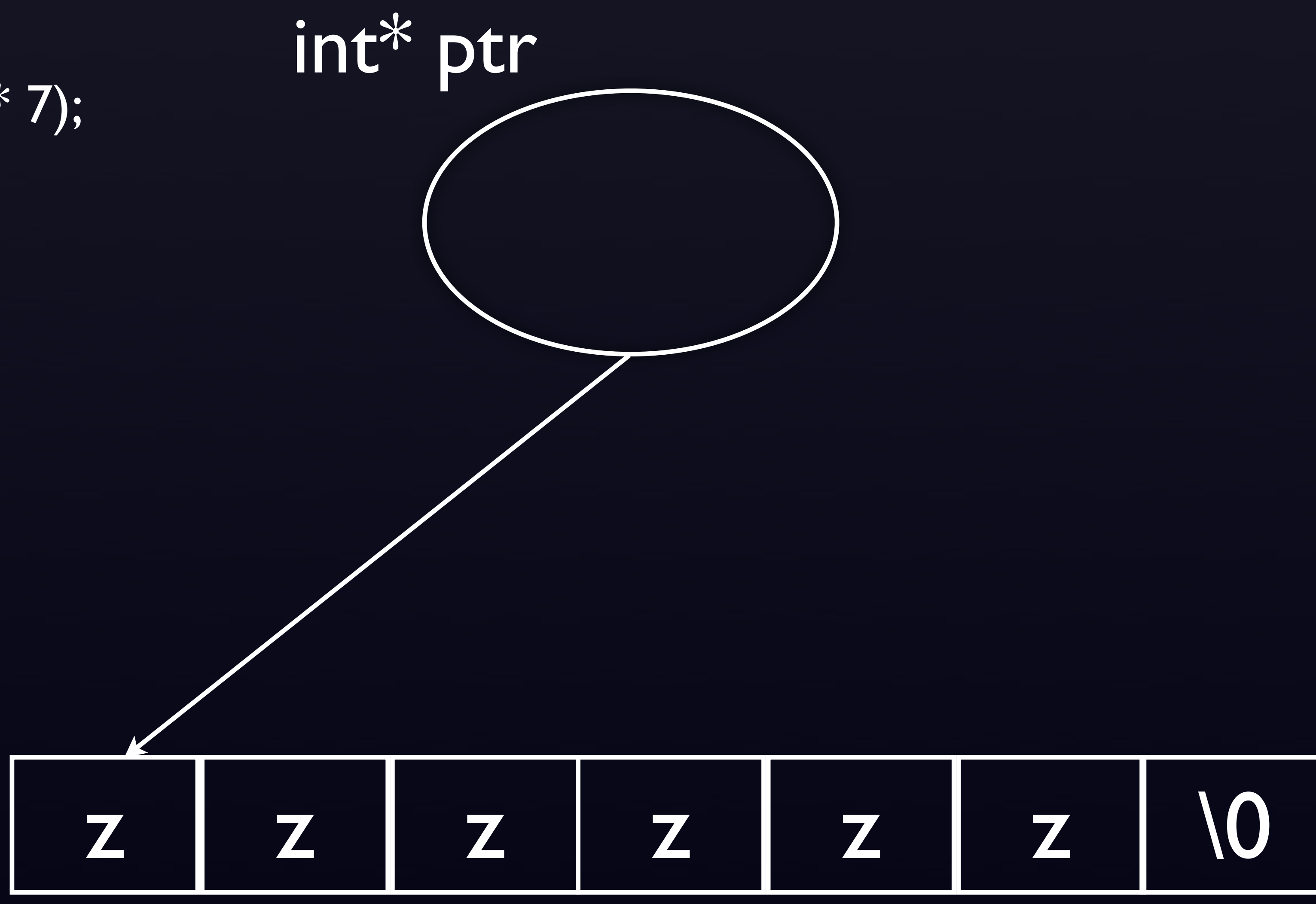
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



Pointer Arithmetic with Strings

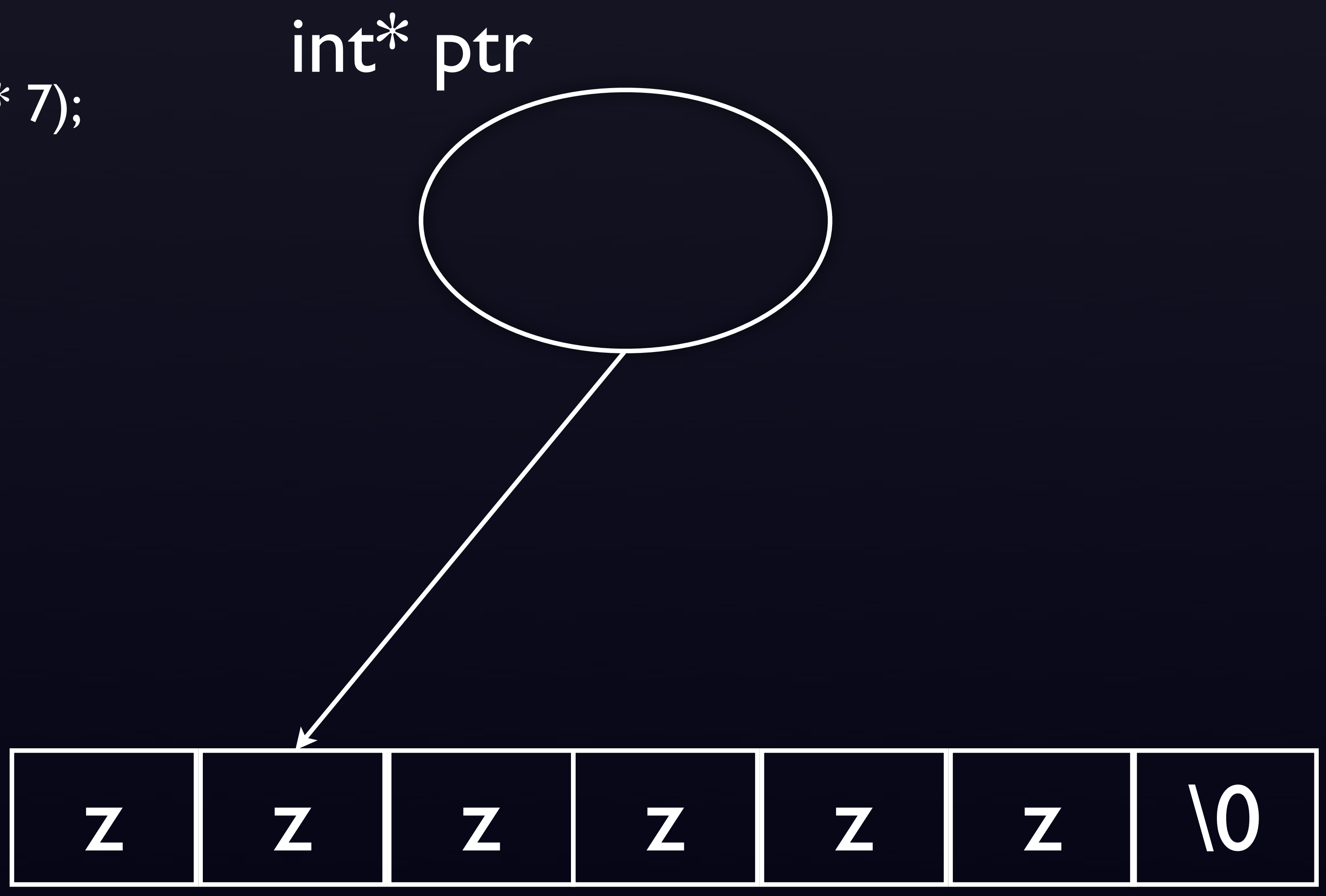
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    while (*ptr != '\0')
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



Pointer Arithmetic with Strings

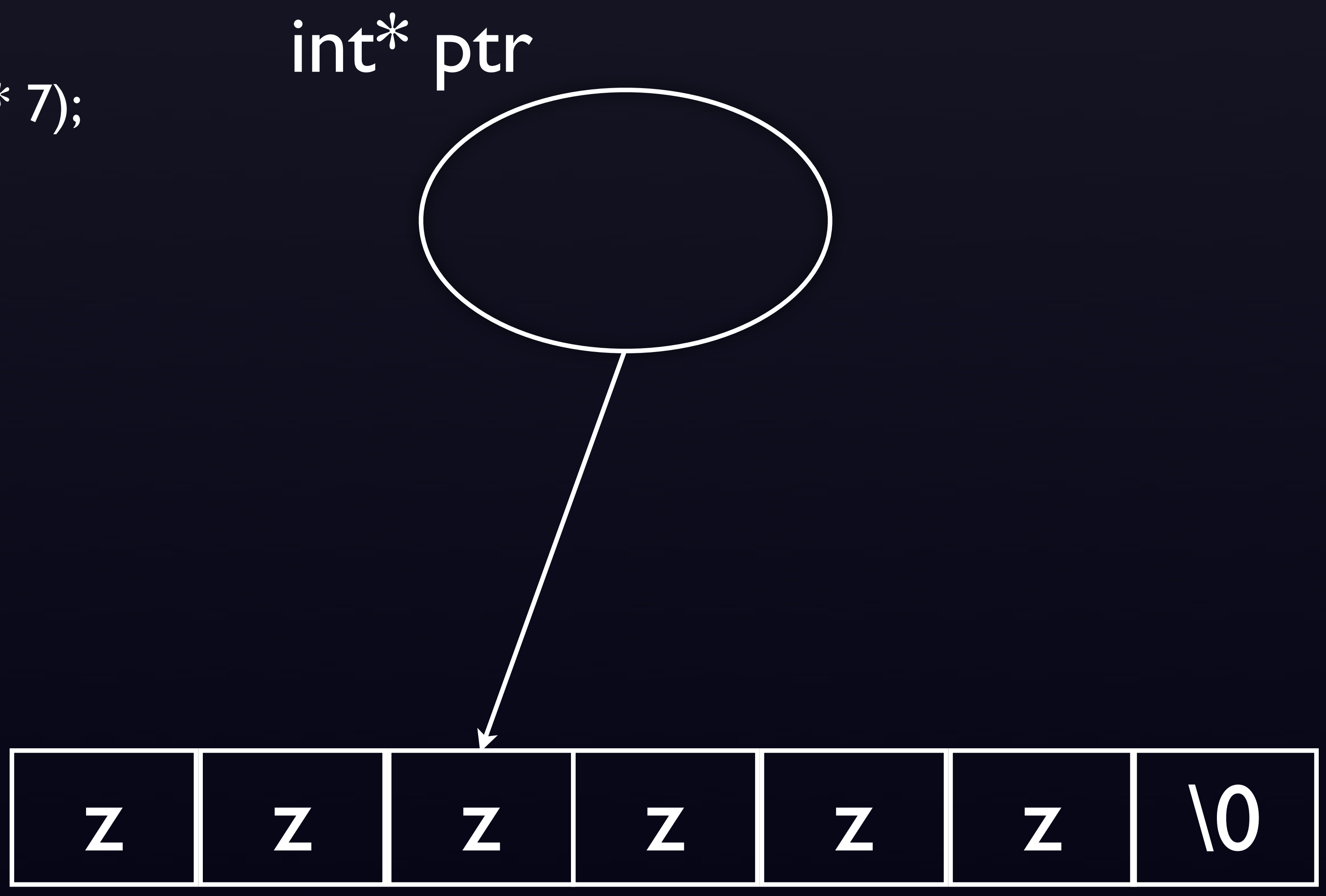
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



Pointer Arithmetic with Strings

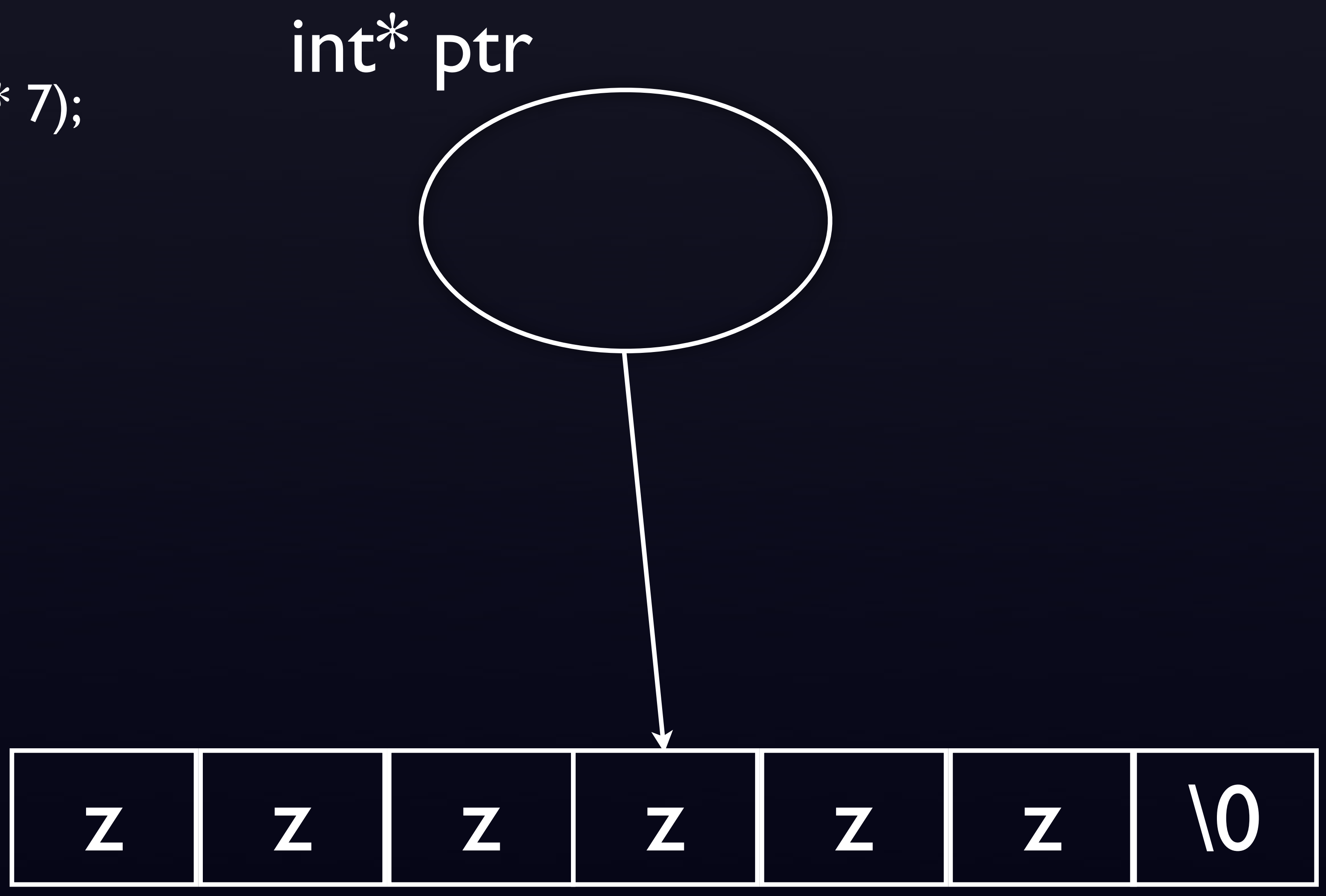
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



Pointer Arithmetic with Strings

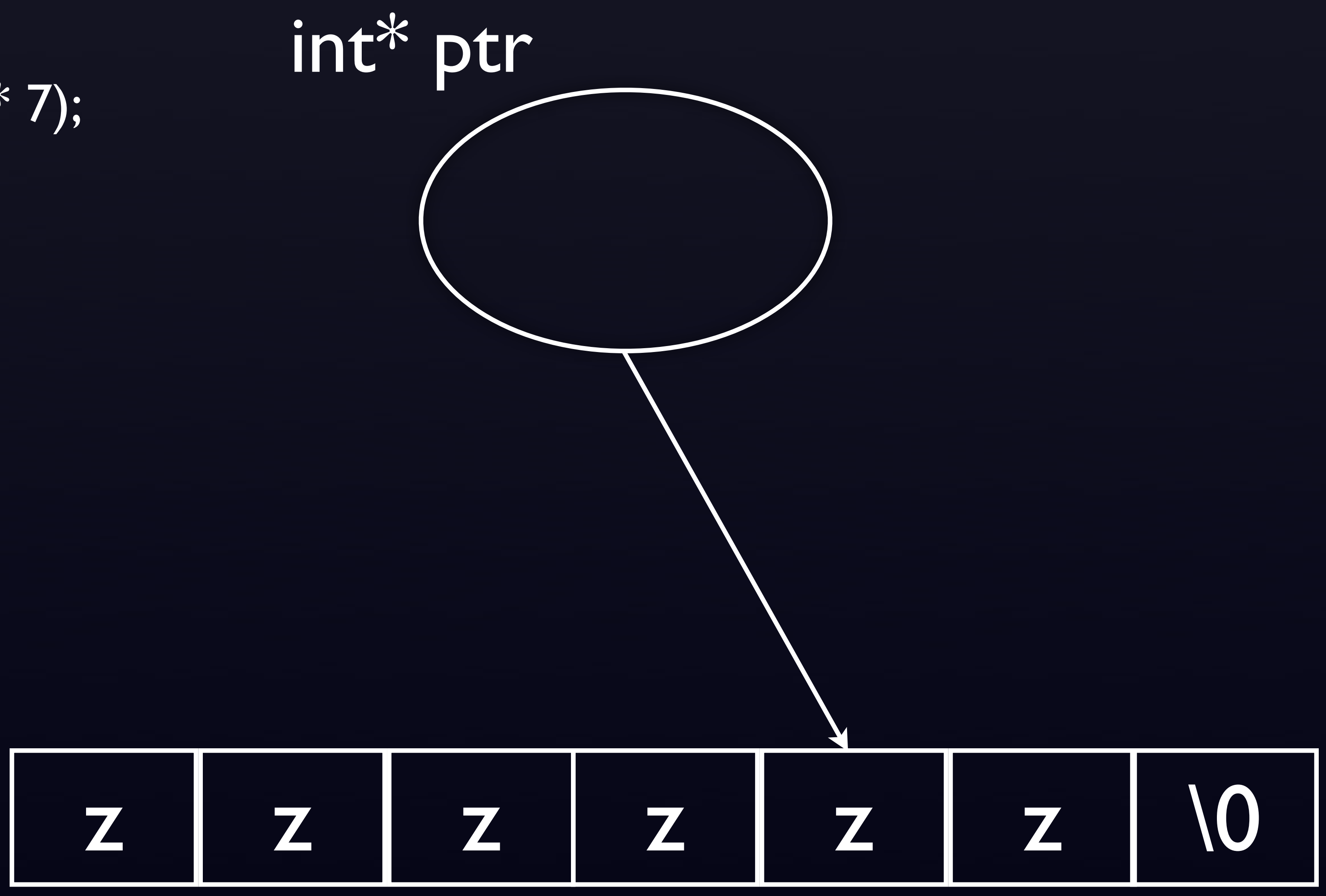
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    while (*ptr != '\0')
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



Pointer Arithmetic with Strings

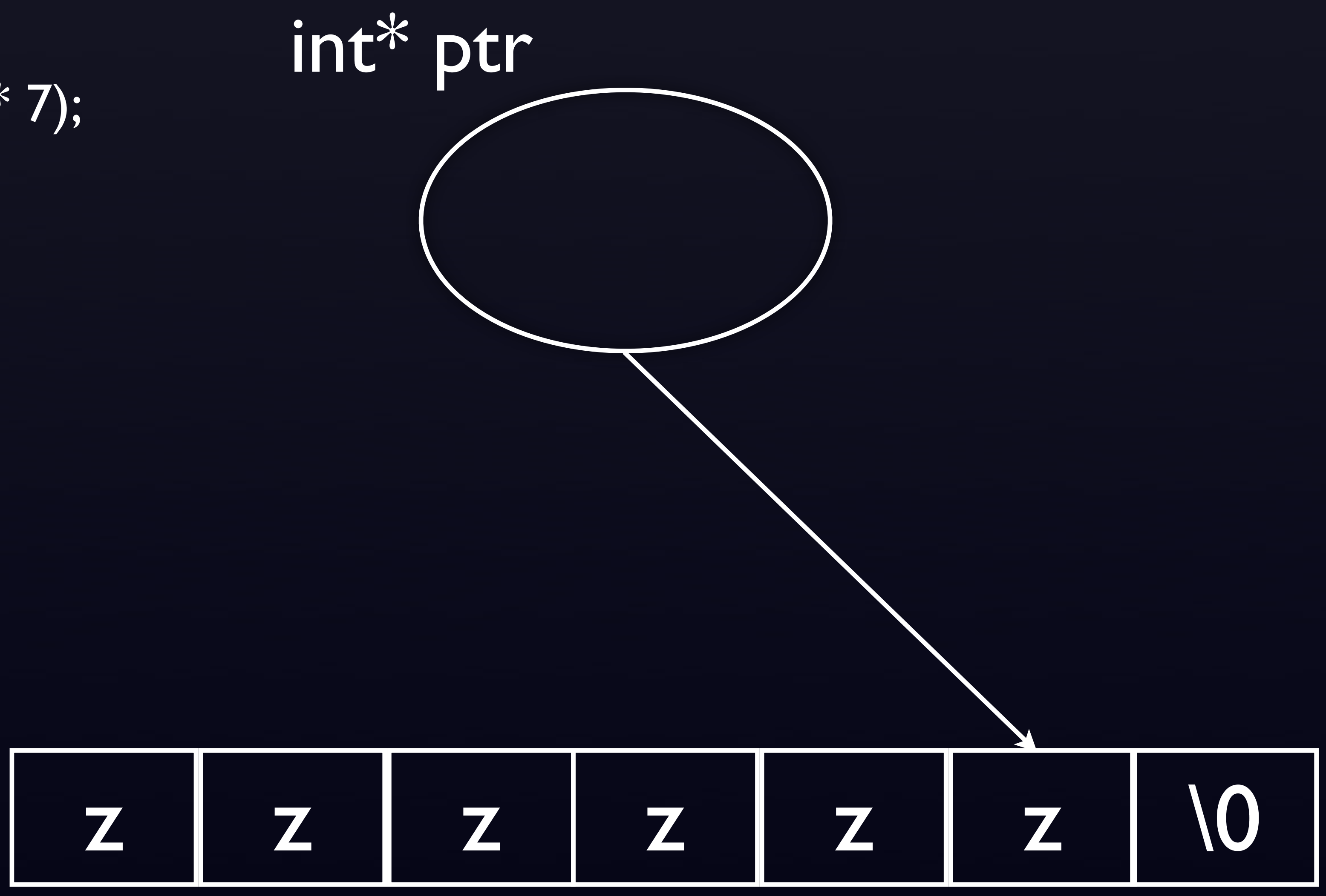
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    while (*ptr != '\0')
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;
    free(ptr);
}
```



Pointer Arithmetic with Strings

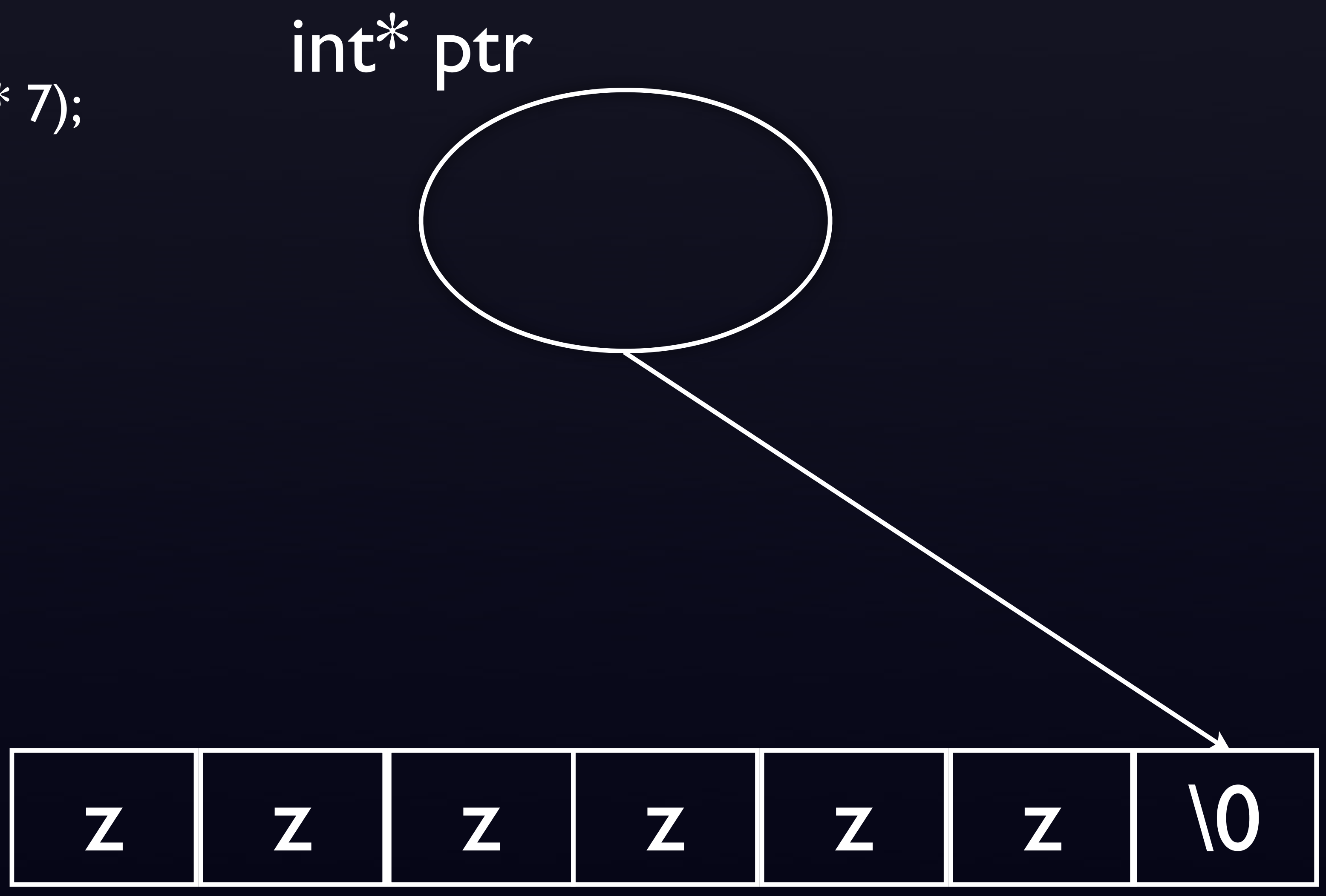
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = ‘z’;
    }

    ptr[6] = ‘\0’;

    while (*ptr != ‘\0’)    // !!!
    {
        printf("%c", *ptr);
        ptr++;
    }

    ptr -= 6;   // move back to original memory location before freeing
    free(ptr);
}
```



Pointer Arithmetic with Strings

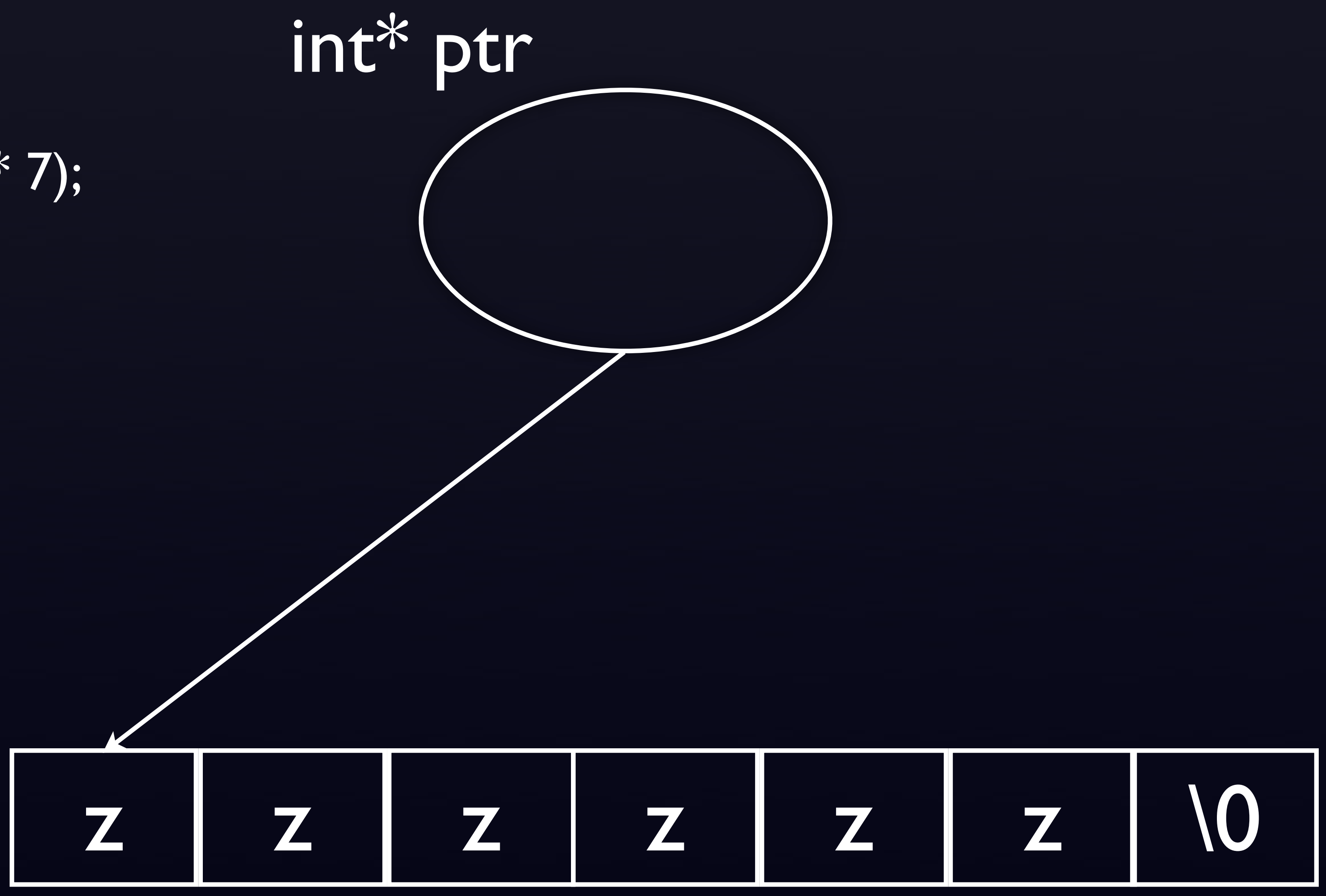
```
int
main(void)
{
    char *ptr = malloc(sizeof(char) * 7);

    for (int i = 0; i < 6 i++)
    {
        *(ptr + i) = 'z';
    }

    ptr[6] = '\0';

    printf("%s", ptr);    // no *

    free(ptr);
}
```



DANGER BAD THINGS D:

```
int
main(void)
{
    // oops, pretty sure we don't have that much memory
    // malloc will fail, returning a NULL pointer
    int *ptr = malloc(sizeof(int) * 2147483647);

    // oops, we forgot to check if it was NULL

    *ptr = 1;

    return 0;
}
```

int* ptr

Null Pointer Dereference

```
int
main(void)
{
    // oops, pretty sure we don't have that much memory
    // malloc will fail, returning a NULL pointer
    int *ptr = malloc(sizeof(int) * 2147483647);

    // oops, we forgot to check if it was NULL

    *ptr = 1;      // oops, we just died x.x,  aka  "dereferencing a null pointer"

    return 0;
}
```

Null Pointer Dereference

```
int
main(void)
{
    // oops, pretty sure we don't have that much memory
    // malloc will fail, returning a NULL pointer
    int *ptr = malloc(sizeof(int) * 2147483647);

    // solution, check if null, and exit the program
    if (ptr == NULL)
        return 1;

    *ptr = 1;     // no longer dereferenced if ptr is NULL

    return 0;
}
```

:D


OKAY, HUMAN.
HUH?
BEFORE YOU HIT 'COMPILE,' LISTEN UP.
YOU KNOW WHEN YOU'RE FALLING ASLEEP, AND YOU IMAGINE YOURSELF
WALKING OR SOMETHING,
AND SUDDENLY YOU MISSTEP, STUMBLE, AND JOLT AWAKE?
YEAH!
WELL, THAT'S WHAT A SEGFAULT FEELS LIKE.
DOUBLE-CHECK YOUR DAMN POINTERS, OKAY?

Memory Leaks


```
#define cs50isAwesome 1

int
main(void)
{
    while (cs50isAwesome)
    {
        int *ptr = malloc(sizeof(int));

        if (ptr == NULL)
            return 1;

        *ptr = 1;      // oops, we forgot to free memory, we'll get a memory leak!
    }

    return 0;
}
```

Memory Leaks

```
#define cs50isAwesome 1

int
main(void)
{
    while (cs50isAwesome)
    {
        int *ptr = malloc(sizeof(int));

        if (ptr == NULL)
            return 1;

        *ptr = 1;       // oops, we forgot to free memory, we'll get a memory leak!
    }

    return 0;
}
```

Image Name	User Name	CPU	Mem Usage
csrss.exe	SYSTEM	00	3,064 K
ctfmon.exe	[redacted]	00	1,712 K
ddmserv.exe	SYSTEM	00	1,064 K
explorer.exe	[redacted]	00	[illegible]
firefox.exe	[redacted]	00	1,532,804 K
GoogleToolbarNot…	[redacted]	00	[illegible]

Memory Leaks

```
#define cs50isAwesome 1

int
main(void)
{
    while (cs50isAwesome)
    {
        int *ptr = malloc(sizeof(int));

        if (ptr == NULL)
            return 1;

        *ptr = 1;
        free(ptr);     // fix't!
    }

    return 0;
}
```

Freeing Twice (or n > 1 times)

```
int
main(void)
{
	int *ptr = malloc(sizeof(int));

	if (ptr == NULL)
		return 1;

	*ptr = 1;
	free(ptr);

	free(ptr);	// oops, we freed something we already freed earlier.

	return 0;
}
```

Freeing Twice (or n > 1 times)

```
int
main(void)
{
	int *ptr = malloc(sizeof(int));

	if (ptr == NULL)
		return 1;

	*ptr = 1;
	free(ptr);

	// fix’t!

	return 0;
}
```

Failure to use sizeof()

```
int
main(void)
{
	// wants to malloc 2 ints. 8 bytes? Right?
	int *ptr = malloc(8);

	if (ptr == NULL)
		return 1;

	*ptr = 1;
	free(ptr);

	return 0;
}
```

Failure to use sizeof()

```
int
main(void)
{
	// actually, an int isn't necessarily 4 bytes on all systems.
	// this is safer and is more compatible with different architectures.
	int *ptr = malloc(sizeof(int) * 2);

	if (ptr == NULL)
		return 1;

	*ptr = 1;
	free(ptr);

	return 0;
}
```

CS50: Quiz 0

Structs

Structs

A struct is a container that can hold and organize meaningfully related variables of different types.

For example, let's say we want to make a collection of variables to represent a Sudoku board! (this is good for Pokemon too)

```
typedef struct
{
	int board[9][9];

	char *level;
	int x, y;
	int timeSpent;
	int totalMoves;
} sudokuBoard;
```

```
int
main(void)
{
	sudokuBoard board;

	board.board = {{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, ...};
	board.level = "n00b";
	board.x = 0;
	board.y = 0;
	board.timeSpent = 0;
	int totalMoves = 0;

	// do stuff with board in rest of program
};
```

Structs

You can also initialize all the variables inside a struct at declaration time using the curly brace syntax.

The variables have to come in the same order as in the struct!

```
typedef struct
{
        int board[9][9];

        char *level;
        int x, y;
        int timeSpent;
        int totalMoves;
} sBoard;
```

```
int
main(void)
{
        sBoard boardA = {{{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}, ...},
                         "n00b",  0, 0, 0, 0};

        sBoard boardB = {{{0, 0, 0, 0, 0, 0, 0, 0, 0, 0}, ...},
                         "l33t", 0, 0, 0, 0};

        // do stuff with boards in rest of program
};
```

Structs

Finally, if we have a pointer to a struct, we can access its fields with the −> operator.

```
typedef struct
{
        int board[9][9];

        char *level;
        int x, y;
        int timeSpent;
        int totalMoves;
} sBoard;
```

```
int
main(void)
{
        sBoard* board = malloc(sizeof(sBoard));

        (*board).x = 0;    // deref board, then change x field

        board->x = 0;    // does same thing as above
};
```

CS50: Quiz 0

GDB

GDB Commands

These are the ones you'll need to know!

run arg1, arg2, ...	runs program with command line arguments
print x	prints out the value of a variable named x in stack frame
break function_name	sets a breakpoint at a function called function_name
break line_number	sets a breakpoint at a line of your code
frame n	gives you information about the nth stack frame
backtrace n	tells you the last n stack frames prior to current point in program
next	moves forward one line in the current execution of code
step	moves forward one line, stepping into a function where applicable
continue	moves forward in the program until the next breakpoint

Structs

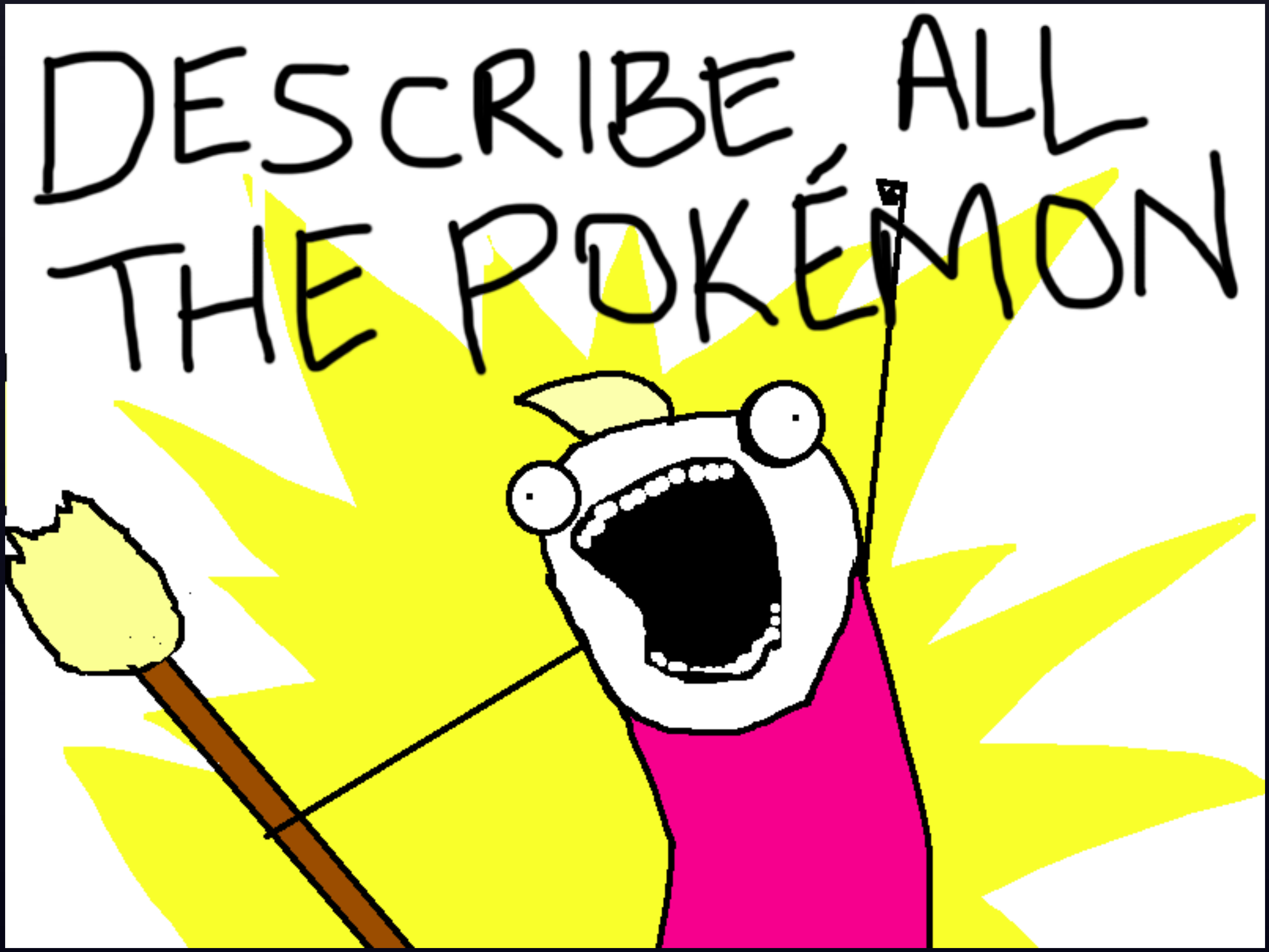
```
typedef struct
{
        int pokedexNo;
        int level;

        char* owner;
        char* pokemonType;
        char* nickName;

        int stats[6];
        char* moveset[4];

        ...

} pokemon;
```



Questions?