What have we learned?

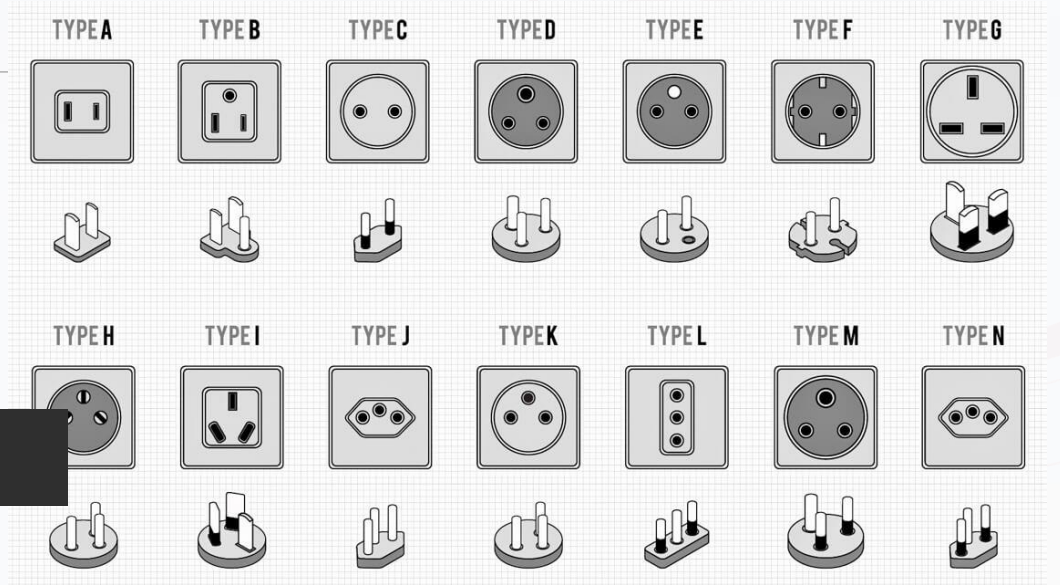
Functions

- Declaration (interface) *before usage*

```
double average(double a, double b);
```

- Definition (implementation) *anywhere*

```
double average(double a, double b)
{
    return (a + b) / 2;
}
```



What have we learned?

Array/pointer arguments

C doesn't pass array length automatically!

```
int f(int a[])
{
        …
}
```

```
int f(int *a)
{
    …
}
```

Pass in the length as another argument

```
int sum_array(int a[], int n)
{
    …
}
```

Variable-length array (C99+)

```
int sum_array(int n, int a[n])
{
    …
}
```

What have we learned?

Multidimensional array

Must specify all inner dimensions

```
int sum_3D_array(int a[][10][5])
{
    …
}
```

Variable-length array (C99+)

```
int sum_3D_array(int k, int m, int n, int a[k][m][n])
{
    …
}
```

Implicit conversion

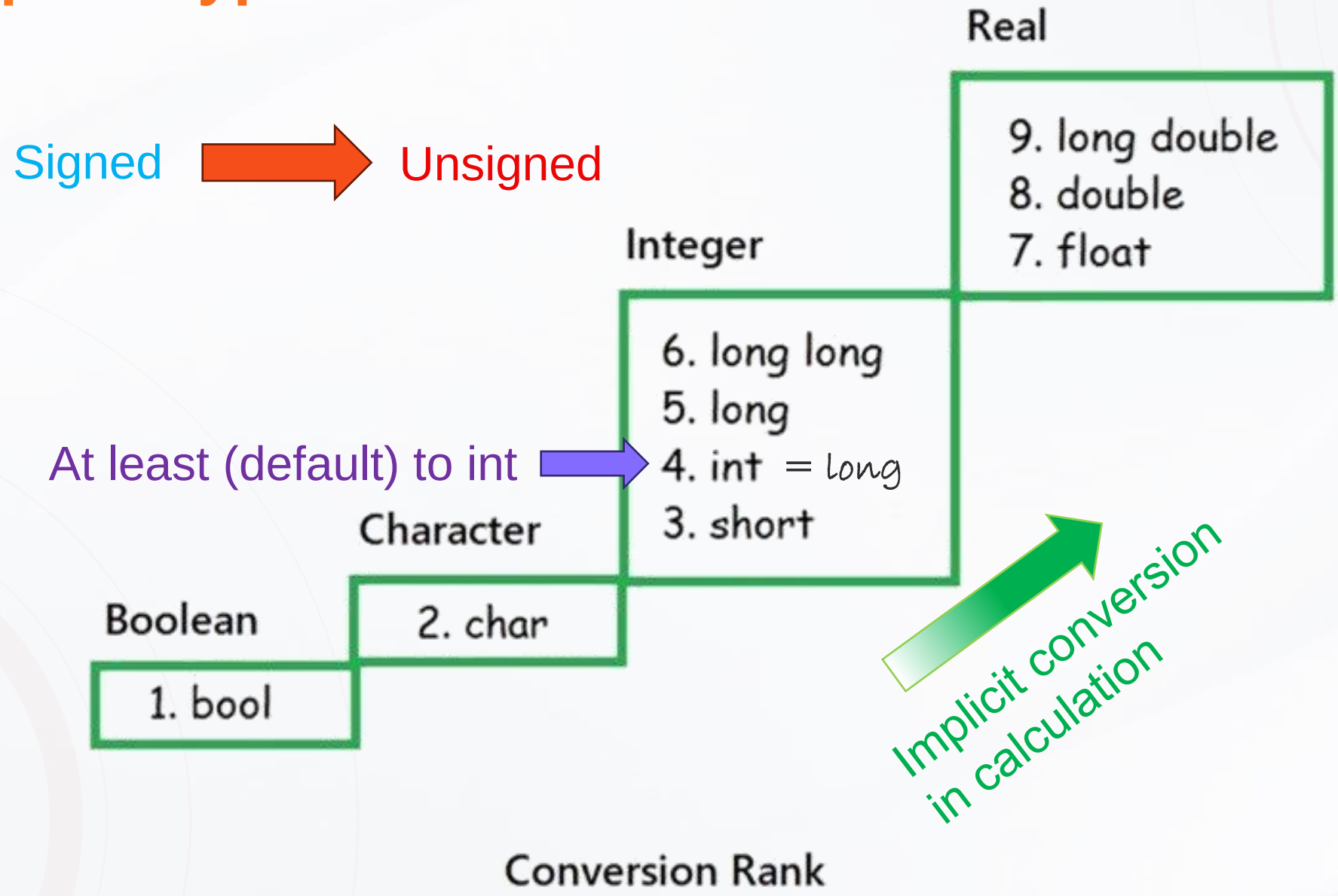
- When the operands in an arithmetic or logical expression don't have the same type.
- When the right side of an assignment doesn't match the type of the variable on the left side.
- When the type of an argument in a function call doesn't match the type in the definition/declaration.
- When the type of the expression in a `return` statement doesn't match the function's return type.

Implicit conversion

```c
#include <stdio.h>

int main(void)
{
    unsigned int x = 123;

    if (x>-10) printf("%d is larger than %d\n", x, -10);
    else printf("%d is less than or equal to %d\n", x, -10);

    return 0;
}
```

```
123 is less than or equal to -10
```

Implicit type conversion



The magic of unsigned

```c
#include <stdio.h>

int main(void)
{
    unsigned int x = 123;

    printf("-10 + %d = %d\n", x, -10 + x);

    return 0;
}
```

```
-10 + 123 = 113
```

```c
#include <stdbool.h>
#include <stdio.h>

int main(void)
{
    bool b = true;
    char c = 'X';
    float d = 1234.5;
    int i = 123;
    short s = 98;

    printf("bool + char:   %c\n", b+c);
    printf("int * short:   %d\n", i*s);
    printf("float * char:  %f\n", d*c);

    // bool promoted to char
    c = c + b;
    // char promoted to float
    d = d + c;
    // float demoted to bool
    b = -d;

    printf("\nAfter execution \n");
    printf("char = char + true:   %c\n", c);
    printf("float = float + char: %f\n", d);
    printf("bool = -float:        %d\n", b);

    return 0;
}
```

To control your types - casting

(*type-name*) *expression* Casts the *expression* to the specified *type*

```c
#include <stdio.h>

int main(void)
{
    unsigned int x = 123;

    if ((int)x>-10)
        printf("%d is larger than %d\n", x, -10);
    else
        printf("%d is less than or equal to %d\n", x, -10);

    return 0;
}
```

Casting

- To help calculation

```
float f, frac_part;
frac_part = f - (int) f;
```

- To explicitly document type conversion

```
i = (int) f;  /* f is converted to int */
```

- To avoid truncation error

```
float quotient;
int dividend, divisor;
quotient = (float) dividend / divisor;
```

Unary operators have higher precedence

Casting

- To avoid overflow

```
int i;
short int j = 1000;
i = j * j;    /* overflow may occur */
```

```
i = (int) j * j;
```

return and exit()

- return statement can be used to escape from a function

```
void print_int(int i)
{
  if (i < 0)
    return;
  printf("%d", i);
}
```

- exit() function (declared <stdlib.h>) terminates a program

```
exit(0);   /* normal termination */
```

- A return statement in the main function is equivalent to exit()

Recursion

Functions in C can be recursive: *it may call itself.*

```
int fact(int n)
{
    if (n <= 1)
        return 1;
    else
        return n * fact(n - 1);
}
```

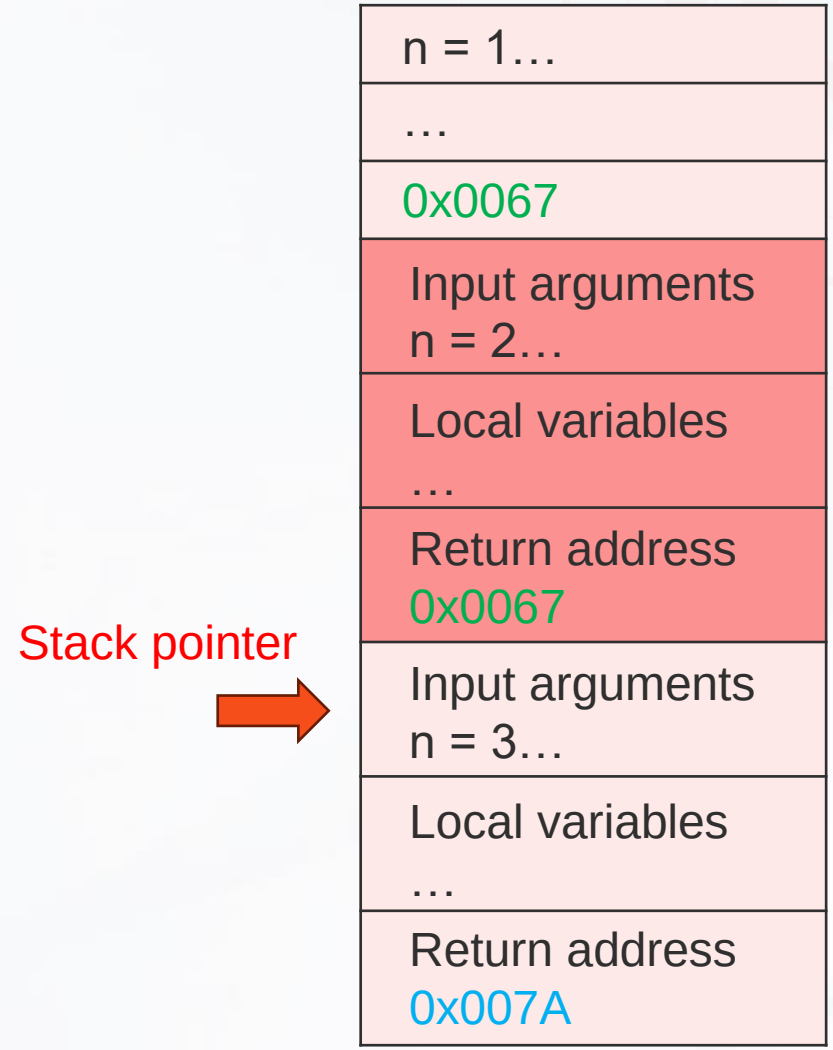
Factorial: n! = 1*2*…*n = n*(n-1)!

What happens when calling a function

Code

```
0x0064  int fact(int n)
0x0065  {
0x0066      if (n <= 1)
0x0067          return 1;
            else
                return n * fact(n - 1);
        }

        int main(void)
        {
            int f;
0x0079      ...
0x007A      f = fact(3);
0x007B      ...
        }
```

Stack

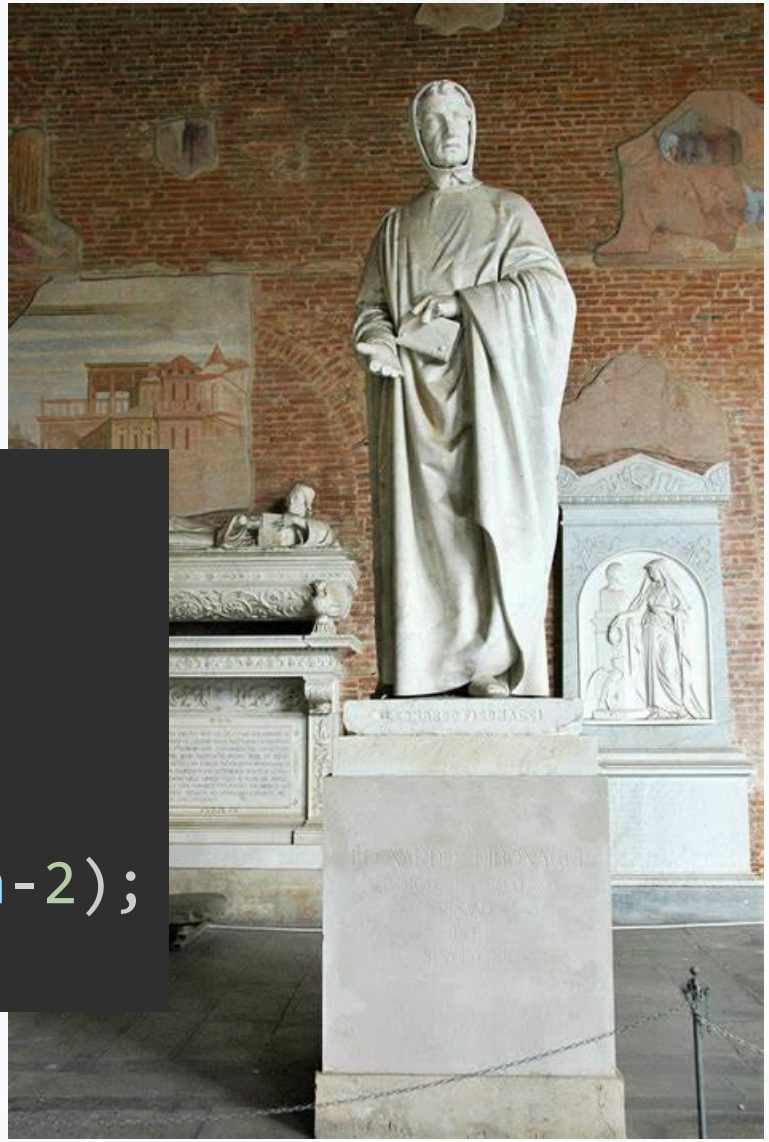


Stack pointer

Example: Fibonacci sequence

1, 1, 2, 3, 5, 8, 13, 21, …

$$F_n = F_{n-1} + F_{n-2}$$



```
unsigned int Fibonacci(unsigned int n)
{
    if (n <= 2)
        return 1;
    else
        return Fibonacci(n-1) + Fibonacci(n-2);
}
```

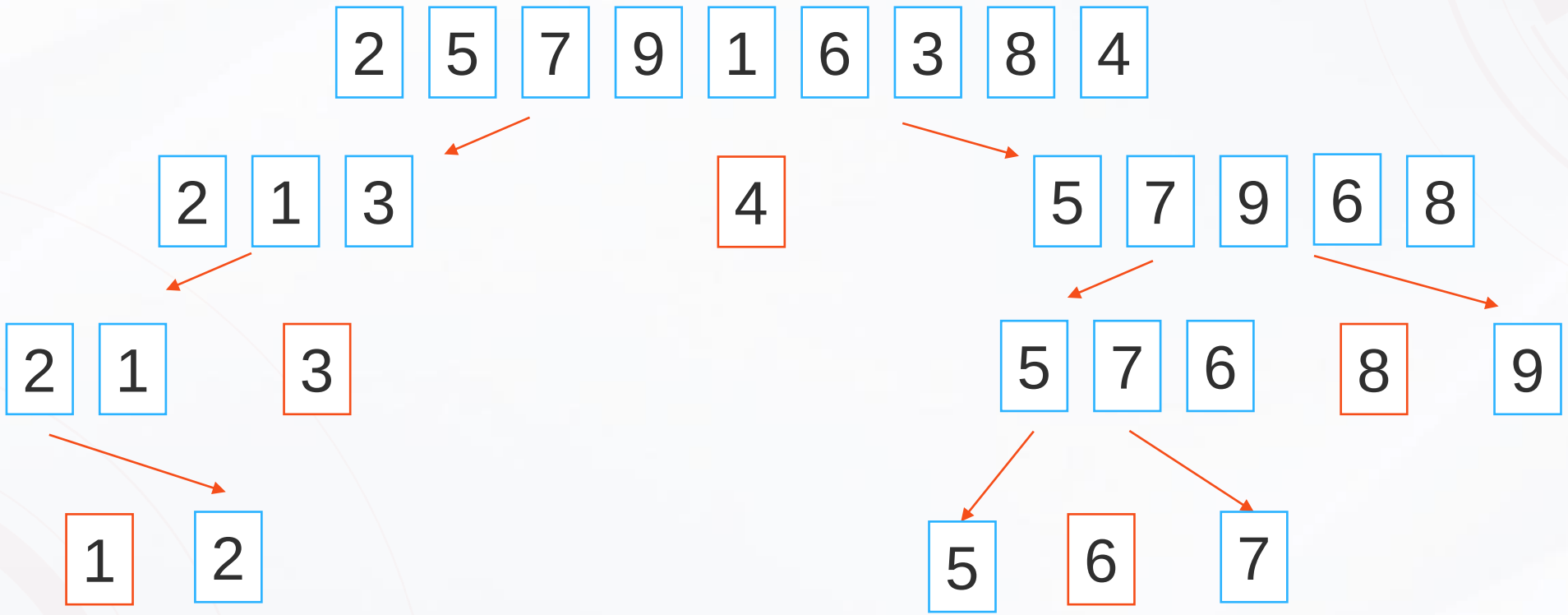
Any recursive function needs a termination condition to prevent infinite recursion

The quicksort algorithm

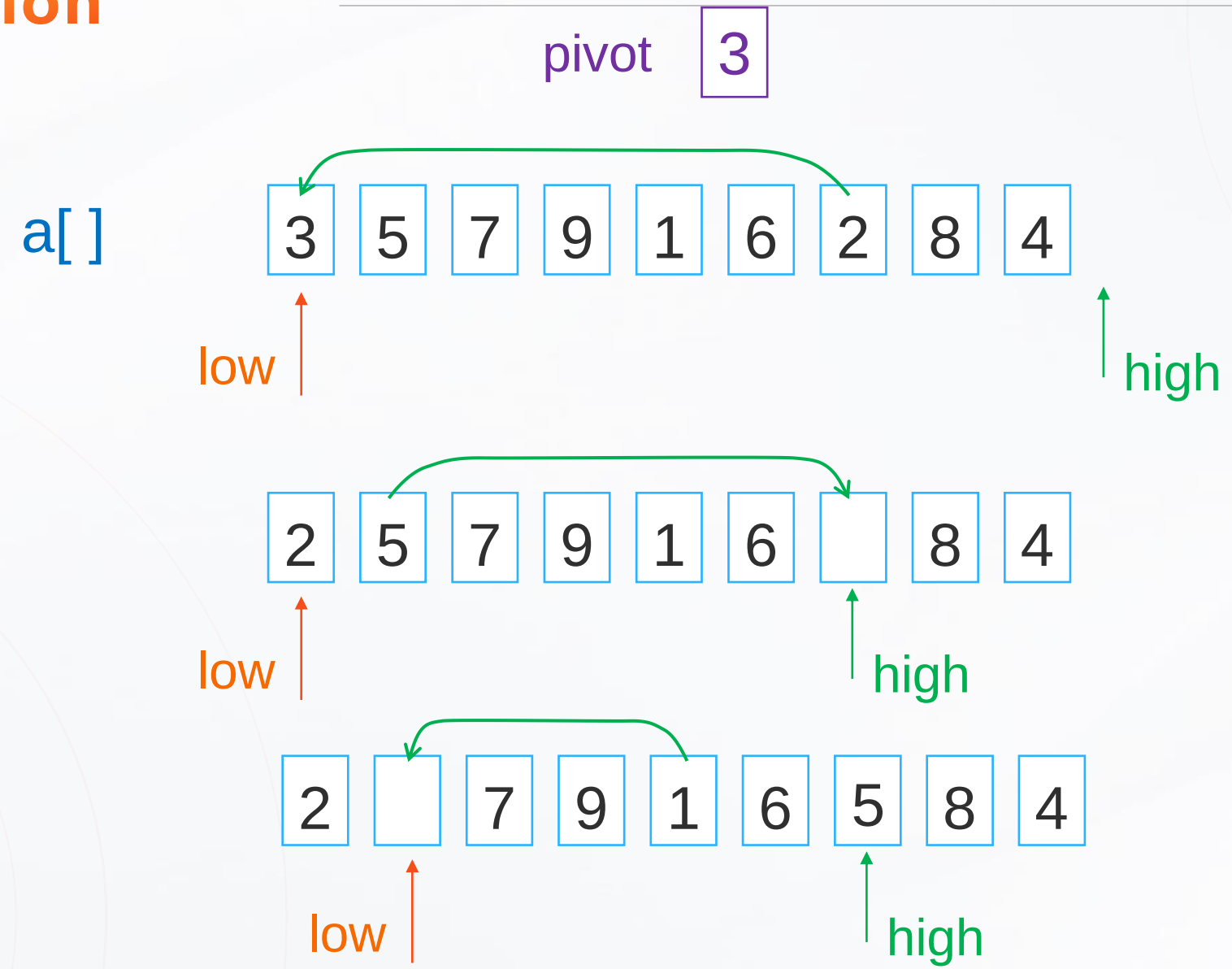
Divide and Conquer!

1. Choose an array element *e* (the "partitioning element"), then rearrange the array so that elements 1, …, *i* – 1 are less than or equal to *e*, element *i* contains *e*, and elements *i* + 1, …, *n* are greater than or equal to *e*.
2. Sort elements 1, …, *i* – 1 by using Quicksort recursively.
3. Sort elements *i* + 1, …, *n* by using Quicksort recursively.

Quicksort



Partition


pivot
3
a[ ]
3 5 7 9 1 6 2 8 4
low
high
2 5 7 9 1 6 8 4
low
high
2 7 9 1 6 5 8 4
low
high

Partition (continued)