Recursion

+You **Web** Images Videos Maps News Gmail More ▾

Google

recursion

Search

About 6,370,000 results (0.07 seconds)

Everything

Images

Maps

Videos

News

Shopping

More

Golden, CO

Change location

Any time

Past hour

Past 24 hours

Past week

Past month

Past 2 months

Past year

Custom range...

More search tools

Did you mean: ***recursion***

Recursion - Wikipedia, the free encyclopedia

en.wikipedia.org/wiki/**Recursion**

Recursion is the process of repeating items in a self-similar way. For instance, when the surfaces of two mirrors are exactly parallel with each other the nested ...

Formal definitions of recursion - Recursion in language - Recursion in mathematics

Recursion (computer science) - Wikipedia, the free encyclopedia

en.wikipedia.org/wiki/**Recursion**_(computer_science)

Recursion in computer engineering is a method where the solution to a ...

Recursive data types - Recursive algorithms - Structural versus generative ...

Show more results from wikipedia.org

Recursion -- from Wolfram MathWorld

mathworld.wolfram.com › ... › Algorithms › Recursion

5 days ago – A **recursive** process is one in which objects are defined in terms of other objects of the same type. Using some sort of recurrence relation, the ...

Recursion in C and C++ - Cprogramming.com

www.cprogramming.com/tutorial/lesson16.html

Learn how to use **recursion** in C and C++, with example **recursive** programs.

Java **Recursion** with examples

www.danzig.us/java_class/**recursion**.html

Simply put, **recursion** is when a function calls itself. That is, in the course of the function definition there is a call to that very same function. At first this may seem ...

2.3 **Recursion** - Introduction to Programming in Java

introcs.cs.princeton.edu/23**recursion**

Recursion is a powerful general-purpose programming technique, and is the key to ... The HelloWorld for **recursion** is to implement the factorial function, which is ...

Recursion

recursion (rɪˈkɜːʃən)
—***n***

1*. see "recursion"*

Recursion

Think of recursion as a function calling itself:

```
// a recursive function
void recursive_fn() {
    recursive_fn(); // calls itself
}
```

This particular recursive function never stops (infinite recursion)

- you'll probably never intentionally write a function like this!

Recursion

<u>Useful</u> recursive functions will consist of two parts:

The recursive step

- if the problem can be decomposed into a smaller version of the same task, the function should call itself on the smaller version

The base case

- when the problem is simple enough, it can be solved *without* further recursive calls
- this is called the "base case" or "stopping case"
- to prevent infinite recursion, a recursive function should <u>always</u> test for the base case <u>before</u> making the recursive call

Recursion

Recursive tasks are common

- a recursive task is one that can be broken into smaller versions of itself
- the smaller versions of the problem are much easier to solve than the entire larger problem (an idea called "divide and conquer")

Example: binary search

- searching for a value in a sorted array (think like using a phone book) can be thought of as a recursive task
- at each step, find the middle of the array, determine which half contains the target value, and then repeat the process on that half of the array (the recursive step)
- when the array consists of only one item, the problem is trivially easy; either the value is the one you're looking for, or it's not (the base case)

Recursion

Recursive structures are also common

- a recursive structure is one that can be broken into smaller versions of itself
- for any such structure, there is very likely a recursive approach to utilizing it (traversing, emptying, sorting, etc...)

Which of the following data structures can treated as recursive?

- linked lists
- strings
- arrays
- stacks
- queues
- trees

Recursion

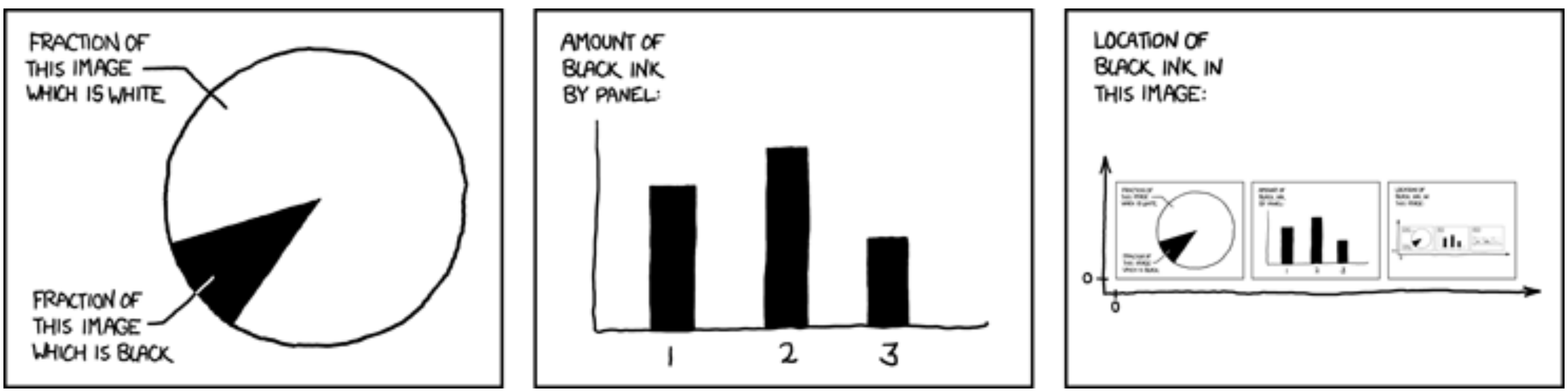
How about all of them?

- linked lists:
  - if you remove a node from a linked list, the result is a shorter linked list
- strings:
  - if you remove a character from a string, the result is a shorter string
- arrays:
  - if you consider only a subset of an array, that subset is essentially just a shorter array
- stacks:
  - remove an item, the result is just a smaller stack
- queues:
  - remove an item, the result is just a shorter queue
- trees:
  - the chapter on recursion proceeds the chapter on trees in your book for a good reason =)

Recursion

Recursion isn't magic, and it won't solve all your problems...

- it can be an extremely powerful tool, though
- many experienced programmers start expecting to find simpler versions of a larger problem during the design process
- in such a case, recursion can potentially be used for a solution (one which tends to be both simple and elegant)



A Simple Example

Recursion Examples

Here's a recursive function for printing numbers, one digit per line:

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

What is the recursive step? How is it a smaller problem?

```
print_number(num / 10); // number with 1 fewer digit
```

Recursion Examples

Here's a recursive function for printing numbers, one digit per line:

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

What is the base case and why was it chosen?

```
if (num < 10) { ... } // only 1 digit
```

Recursion Examples

Here's a recursive function for printing numbers, one digit per line:

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

What output does this function call generate?

```
print_number(12345);
```

Recursion Examples

Here's a recursive function for printing numbers, one digit per line:

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

How could you reverse the order in which the digits are printed?

Recursion Examples

Printing the digits in the reverse order:

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

How could you reverse the order in which the digits are printed?

- just do the recursive call *after* printing `(num % 10)`

Another Example

Recursion Examples

What is the recursive step here? How is it a smaller problem?

```
void recurse_to_zero(size_t num) {
    cout << num << endl;

    if (num == 0) {
        cout << "Zero!" << endl;
    } else {
        recurse_to_zero(num - 1);
    }

    cout << num << endl;
}
```

Recursion Examples

What is the base case?

```
void recurse_to_zero(size_t num) {
    cout << num << endl;

    if (num == 0) {
        cout << "Zero!" << endl;
    } else {
        recurse_to_zero(num - 1);
    }

    cout << num << endl;
}
```

Recursion Examples

What does this function output when called with 3 as the input?

```
void recurse_to_zero(size_t num) {
    cout << num << endl;

    if (num == 0) {
        cout << "Zero!" << endl;
    } else {
        recurse_to_zero(num - 1);
    }

    cout << num << endl;
}
```

How Recursion Works

A closer look

The Call Stack

Computers keep track of function calls using a stack (the “call stack”)

- every time a function is called, an entry (called an activation record) gets added onto the call stack
- execution immediately stops in the original function and jumps to the start of the called function

Each activation record contains:

- the location to which execution should return when the function has finished
- the current value of arguments
- the current value of local variables

The Call Stack

Here's a simple program. We're going to trace it. =)

```
int recurse_to_zero(int n) {
    cout << num << endl;
    if (num != 0) {
        recurse_to_zero(num - 1);
    }
    cout << num << endl;
}

int main() {
    recurse_to_zero(3);
    return 0;
}
```

Recursion Pitfalls

Recursion Pitfalls

What is wrong with this recursive function?

```
void print_number(int n) {
    print_number(n / 10);
    cout << n % 10 << endl;
}
```

No base case!

- infinite recursion =(

Recursion Pitfalls

What is wrong with this recursive function?

```
void print_again(int n) {
    if (n < 10) {
        cout << n << endl;
    } else {
        print_number(n);
        cout << n % 10 << endl;
    }
}
```

Recursive step doesn't use a smaller problem (uses SAME problem)

- infinite recursion =(

Recursion Pitfalls

What is wrong with this recursive setup?

```
bool fn_1() {
    return fn_2();
}

bool fn_2() {
    return fn_1();
}
```

Crazy cyclic recursion!

- infinite recursion =(

Recursion Pitfalls

When writing recursive functions:

- always test for the base case
- make sure the recursive step works on a smaller problem (or on a problem that is somehow closer to the base case)

Recursion Practice!

Recursion Practice

Write a recursive function to test if an input string is a palindrome

- what is the recursive step?
- what is the base case?

Recursion Practice

An example implementation:

```
bool is_palindrome(const string& s, int beg, int end) {
    if (beg >= end) {
        return true;
    } else if (s[beg] == s[end]) {
        return is_palindrome(s, beg+1, end-1);
    }

    return false;
}
```

Recursion Practice

Write a recursive function to calculate n-factorial

- what is the recursive step?
- what is the base case?

Recursion Practice

An example implementation:

```
// returns n-factorial (n!)
int factorial(int n) {
    if (n == 1) {
        return 1;
    }

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

Recursion Practice

Write a recursive function to calculate the nth Fibonacci number

- what is the recursive step?
- what is the base case?

Recursion Practice

An example implementation:

```
// returns the nth fibonacci number (n > 0)
unsigned fibonacci(unsigned n) {
    if (n <= 2) {
        return 1;
    }

    return fibonacci(n - 1) + fibonacci(n - 2);
}
```

This last function is hugely inefficient...

Make sure you understand why!