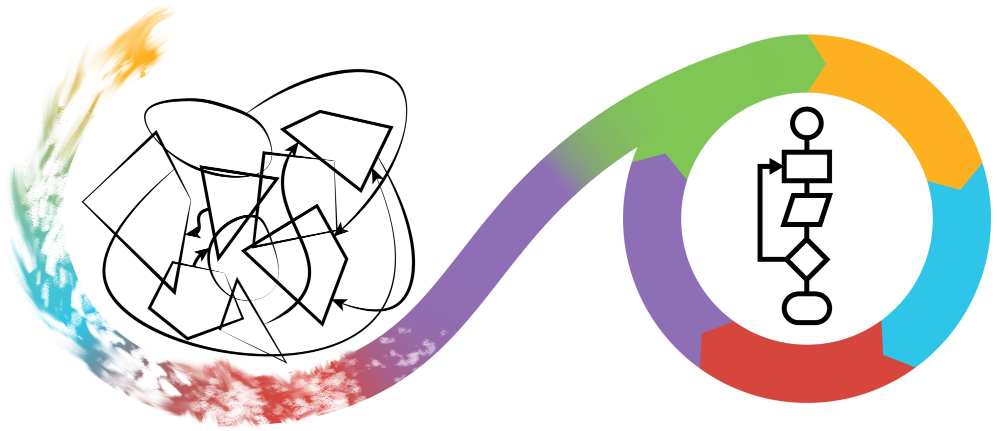
Algorithms

Richèl Bilderbeek

Algorithms

https://github.com/UPPMAX/programming_formalisms/blob/main/testing/testing_lecture/testing_lecture.qmd



Problem

How do I write functions [1] that are:

- easy to use
- correct

- fast [2]

References;

- [1] For now, we use `algorithm == function`, as the definition of an algorithm is 'a step-by-step procedure for solving a problem or accomplishing some end' [3],
- [2] pick any vague definition
- [3] https://www.merriam-webster.com/dictionary/algorithm

What is a good function?

A good function:

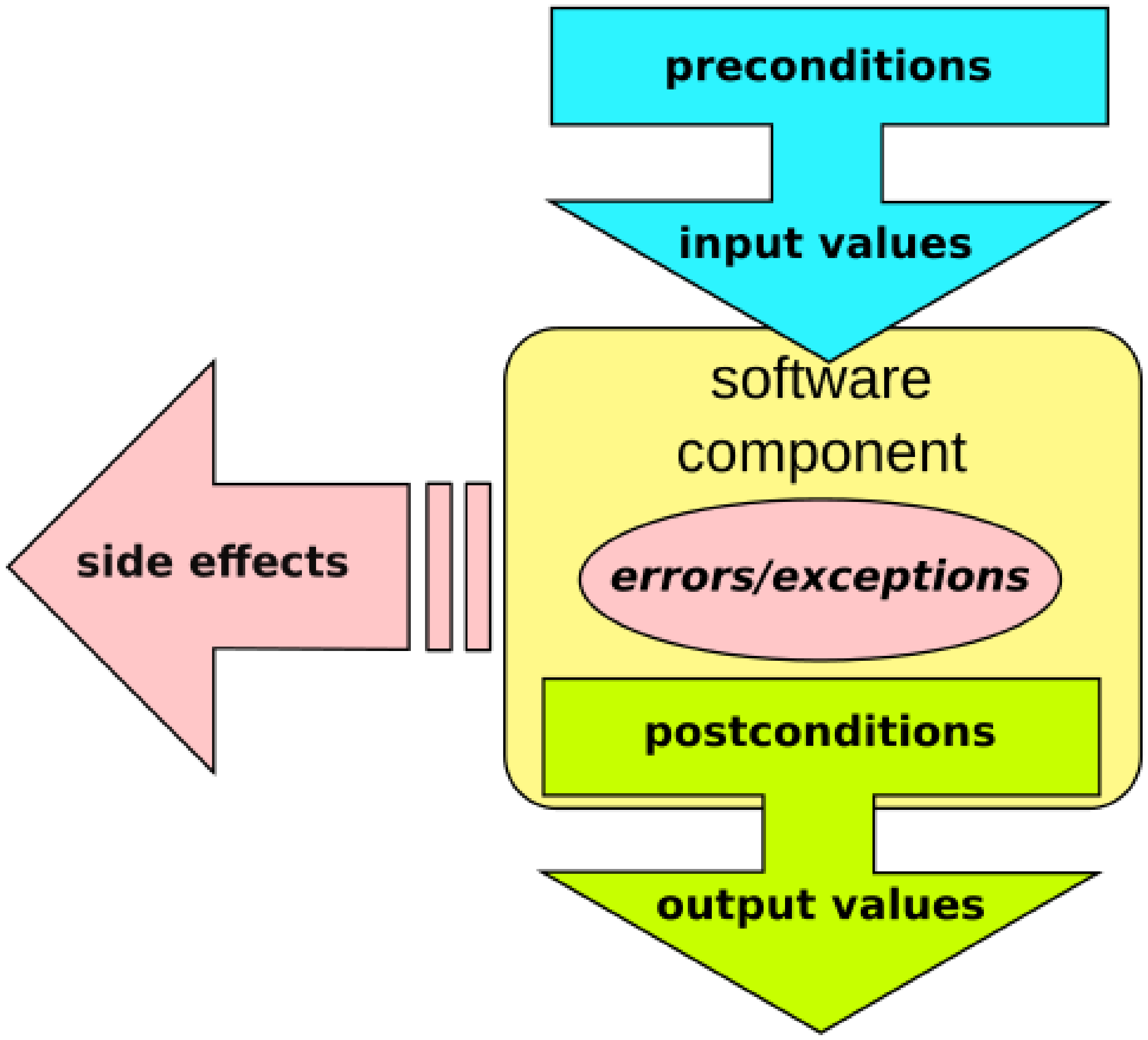
. . .

- Is documented
- Has a clear name
- Has a clear interface
- Does one thing correctly
- Is tested
- Gives clear error messages
- Fast iff needed

References

- C++ Core Guidelines on functions
- R Tidyverse style guidelines on functions
- The Hitchhiker's Guide to Python on general concepts
- (PEP 20: The Zen of Python)

Design by contract



Source: Wikipedia

A good function is documented

```
def sort_1(x):
  """Sort list `x` in-place.

  Returns nothing
  """

def sort_2(x):
  """Sort list `x`.

  Returns the sorted list.
  """

assert sort_1.__doc__ is not None
assert sort_2.__doc__ is not None
```

A good function has a clear name

> There are only two hard things in Computer Science: cache invalidation and naming things
>
> Phil Karlton

Examples of bad function names?

Could you give examples of bad function names?

. . .

- `calculate`: calculates what?

. . .

- `calc_bfgt`: calculates what??

. . .

- `prime`: a prime number is a data type. What does this function do?

. . .

- `needleman_wunch`: this is a technique to get a DNA alignment.

Figure 1: https://www.karlton.org/karlton/images/with-fish.jpg

…

Example 1

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCCGGGTAT
 *         x
```

How would you call the algorithm that detects the location of the `*` (but not of the `x`, as the `*` comes earlier)?

Solutions 1

- `get_first_mismatch_pos`, `get_first_mismatch_locus`, `find_first_mismatch_pos`, `find_first_mismatch_locus`
- you answer that …
  1. starts with a verb
  2. is as English as possible
  3. only uses common abbreviations

Example 2

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCCGGGTAT
 *         *
```

How would you call the algorithm that detects all the locations of the `*`s?

Solutions 2

- `find_mismatch_positions`, `find_mismatch_loci`, `get_mismatch_positions`, `get_mismatch_loci`
- you answer that ...
  1. starts with a verb
  2. is as English as possible
  3. only uses common abbreviations

Example 3

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCGGGTTT
```

How would you call the algorithm that makes the sequences have as much similarities as possible, by possibly inserting a `-`

```
AAACCCGGGTTT
ATACC-GGGTTT
```

Solutions 3

- `align_seqs`, `align_sequences`, `align_dna_sequences`, `align_dna_seqs`, `calc_aligned_seqs`, `get_aligned_seqs`
- you answer that ...
  1. starts with a verb
  2. is as English as possible
  3. only uses common abbreviations

A good function has a clear name

- F.1: "Package" meaningful operations as carefully named functions
- Use verbs, strive for names that are concise and meaningful

A function has a clear interface 1/3

Comment on this function from Pythonpool:

```
i=2

def Prime(no, i):
    if no == i:
        return True
    elif no % i == 0:
        return False
    return Prime(no, i + 1)
```

. . .

Function names start with lowercase character, name does not start with a verb, input is not checked, clumsy interface:

```
assert Prime(2, 2)
assert Prime(3, 2)
assert Prime(3, 3) # Nothing stops me!
assert not Prime(4, 2)
assert Prime(5, 2)
```

The classic on refactoring is (1).

A function has a clear interface 2/3

Comment on this function again:

```
def is_prime(no, i = 2):
    assert isinstance(no, int)
    assert isinstance(i, int)
    if no == i:
        return True
    elif no % i == 0:
        return False
    return is_prime(no, i + 1)
```

. . .

- Clumsy interface:

```
assert is_prime(2)
assert is_prime(2, 2) # Nothing stops me!
assert is_prime(3)
assert not is_prime(4)
assert is_prime(5)
```

A function has a clear interface 3/3

Comment on this function again:

```
def is_prime(no):
    if not isinstance(no, int):
        raise TypeError("'no' must be integer")
    return is_prime_internal(no)

def is_prime_internal(no, i = 2):
    assert isinstance(no, int)
    assert isinstance(i, int)
    if no == i:
        return True
    elif no % i == 0:
        return False
    return is_prime_internal(no, i + 1)

assert is_prime(2)
assert is_prime(3)
assert not is_prime(4)
assert is_prime(5)
```

. . .

I think it is OK, please correct me :-)

A function does one thing correctly

Example 1

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCGGGTTT
```

How would you call the algorithm that aligns the sequences and returns the mismatches?

```
AAACCCGGGTTT
ATACC-GGGTTT
 *   *
```

Solution 1

- ~~`align_and_collect_mismatched_loci`~~

You should not write a function like this.

- F.2: A function should perform a single logical operation

You rarely need `and` in a function name.

What is a good function?

A good function:

- **Has a clear name**
- **Does one thing correctly**
- Is tested
- Gives clear error messages
- Is documented
- Fast iff needed

A good function is tested

- F.2: A function should perform a single logical operation: A function that performs a single operation is simpler to understand, test, and reuse.
- Joint Strike Fighter Coding Standards, section 3: Testability: Source code should be written to facilitate testability

Example 1

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCCGGGTAT
 *        x
```

The function `get_first_mismatch_locus` detects the first mismatch locus. Which tests would you write?

Solutions 1

```
assert get_first_mismatch_locus(
    "AAACCCGGGTTT",
    "ATACCCGGGTAT"
  ) == 1
assert get_first_mismatch_locus(
    {
      "AAACCCGGGTTT",
      "ATACCCGGGTAT"
    }
  ) == 1
```

```
expect_equal(
  1,
  get_first_mismatch_locus(
    "AAACCCGGGTTT",
    "ATACCCGGGTAT"
  )
)
expect_equal(
  1,
  get_first_mismatch_locus(
    c(
      "AAACCCGGGTTT",
      "ATACCCGGGTAT"
    )
  )
)
```

The different tests *signal* different expectations. This should end up in the documentation and/or error messages.

Example 2

Imagine two DNA sequences:

```
AAACCCGGGTTT
```

```
ATACCCGGGTAT
 *        *
```

The function `get_mismatch_loci` detects the mismatched loci. Which tests would you write?

Solutions 2

```
assert get_mismatch_loci(
  "AAACCCGGGTTT",
  "ATACCCGGGTAT"
  ) == {1, 10}
assert get_mismatch_loci(
  {
    "AAACCCGGGTTT",
    "ATACCCGGGTAT"
  }
) == {1, 10}
```

```
expect_equal(
  get_mismatch_loci(
    "AAACCCGGGTTT",
    "ATACCCGGGTAT"
  ),
  c(1, 10)
)
expect_equal(
  get_mismatch_loci(
    c(
      "AAACCCGGGTTT",
      "ATACCCGGGTAT"
    )
  ),
  c(1, 10)
)
```

The different tests *signal* different expectations. This should end up in the documentation and/or error messages.

Example 3

Imagine two DNA sequences:

```
AAACCCGGGTTT
ATACCGGGTTT
```

The function `align_dna_seqs` aligns two DNA sequences to this:

```
AAACCCGGGTTT
ATACC-GGGTTT
```

Which tests would you write?

Solutions 3

```
assert align_dna_seqs(
  "AAACCCGGGTTT",
  "ATACCCGGGTAT"
  ) == {
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  }
assert align_dna_seqs(
  {
    "AAACCCGGGTTT",
    "ATACCCGGGTAT"
  }
) ==
  {
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  }
```

```
expect_equal(
  align_dna_seqs(
    "AAACCCGGGTTT",
    "ATACCCGGGTAT"
  ),
  c(
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  )
)
expect_equal(
  align_dna_seqs(
    c(
      "AAACCCGGGTTT",
      "ATACCCGGGTAT"
    )
  ),
  c(
    "AAACCCGGGTTT",
    "ATACC-GGGTTT"
  )
)
```

What is a good function?

A good function:

- **Has a clear name**
- **Does one thing correctly**
- **Is tested**
- Gives clear error messages
- Is documented
- Fast iff needed

The role of `assert` within functions

Within functions, `assert` is used for:

- as a stub
- to do things in debug mode only
- to document assumptions a developer makes

`assert` differs between debug and release

```
$ cat assert.py
assert 1 == 2

$ python -O assert.py

$ python assert.py
Traceback (most recent call last):
  File "/home/richel/assert.py", line 1, in <module>
    assert 1 == 2
AssertionError
```

`assert` as a stub

```python
def align(dna_sequences):
    """Align the DNA sequences"""
    assert len(dna_sequences) == 2 # TODO
```

`assert` in debug mode

```python
def align_two_dna_sequences(dna_sequences):
    """Internal function to align two DNA sequences"""
    assert len(dna_sequences) == 2 # TODO
```

Superior to documentation, as it cannot be ignored.

‘Assert liberally to document internal assumptions and invariants’ (2) chapter 68.

`assert` to document assumptions a developer makes

```python
def align_two_dna_sequences(dna_sequences):
    """Align the DNA sequences"""
    # ....
    results = ["AAAA", "AAC-"] # Mock
    assert len(results[1]) == len(results[2])
```

Clear error messages

Overview

- **Types of algorithms**
- Common data structures
- Big O complexity
- Parallelisms

Types of algorithms

- Divide and conquer algorithms
- Brute force algorithms
- Randomized algorithms
- Greedy algorithms
- Recursive algorithms
- Backtracking algorithms
- Dynamic programming algorithms

- Iterative versus recursive
- Exact or approximate
- Deterministic or non-deterministic
- Single-thread, multi-threaded, distributed
- (Digital or quantum)

Recursive versus iterative

- Iterative: use a for-loop
- Recursive: a function that calls itself

Example 1: factorial

`n`	`n!`
0	1
1	1
2	2 * 1 = 2
3	3 * 2 * 1 = 6
4	4 * 3 * 2 * 1 = 24
5	5 * `4!`
`n`	`n` * `(n-1)!`

Exercise 1: factorial

- Develop a function to get the factorial of a number
- If the function used a for-loop, create another function that uses recursion, or the other way around

```
assert calc_factorial_iterative(13) ==
  calc_factorial_recursive(13)
```

```
expect_equal(
  calc_factorial_iterative(13),
  calc_factorial_recursive(13)
)
```

Example 2

Fibonacci sequence:

N	0	1	2	3	4	5	6	7	8	9	10
`Fn`	0	1	1	2	3	5	8	13	21	34	55

Example 2

N	`Fn`
0	0
1	1
2	1
3	`Fn(1) + Fn(2)`
n	`Fn(n - 1) + Fn(n - 2)`

Exercise

- Develop a function to get the nth value in the Fibonacci sequence
- If the function used a for-loop, create another function that uses recursion, or vise versa

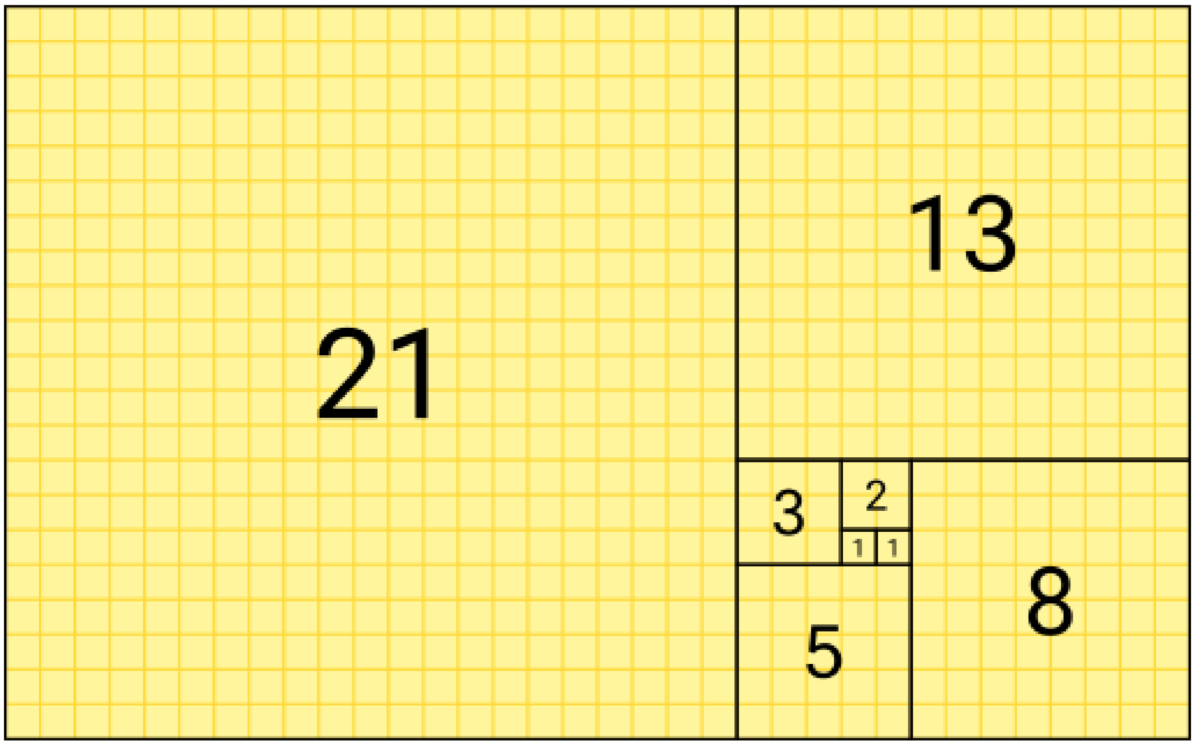


Figure 2: https://en.wikipedia.org/wiki/File:Fibonacci_Squares.svg

```
assert get_nth_fibonacci_iterative(13) ==
  get_nth_fibonacci_recursive(13)
```

```
expect_equal(
  get_nth_fibonacci_iterative(13),
  get_nth_fibonacci_recursive(13)
)
```

Divide-and-conquer algorithms

- Divide a complex problem in many simpler problems
- Solve the simpler problems
- Combine the results

Appendix

Ruthless copy-paste from here, thanks Quarto for **not** copy-pasting formatted HTML :-/

Introduction to algorithms & datastructures

Computer science is the study of how computers operate on information. Both information and operations can be structured in different ways, which gives them different properties and makes them more or less useful for different purposes.

Lesson plan

Storing things Searching for things Sorting lists BLAST parallelism

Why these five topics? The first three exemplify how much your choice of data structure matters, and they're simple and general enough that they're accessible for a brief course such as this. BLAST is included as an introduction to a practical algorithm which has become a theme for countless variations. Parallel algorithms require special consideration beyond the usual.

But first some fundamentals What is an algorithm and what is a data structure?

An algorithm is not a program. A program implements algorithms on a practical level. Algorithms are mathematics. As such, they are discovered and not invented, and therefore not patentable unless you're a patent trolling megacorporation.

Moore's Law, which famously says that the number of transistors on a chip will double every other year, is usually touted as the explanation for why modern computing is such a powerful tool. This is a misrepresentation

Links to an external site..

Algorithmic improvements superimposed on Moore's Law Classification of algorithms Iterative or recursive

Algorithms predicated on repeating steps either through recursion (self-referencing functions) or iteration (loops). Most algorithms naturally fall in one category, but both categories are computationally equivalent (i.e. there is always a recursive variant of an iterative algorithm and vice-versa).

See exercise: converting a recursive algorithm to an iterative Serial, parallel, distributed

Traditionally, algorithms are expressed as a single thread of logical steps. This works perfectly well for designing programs for single-core computers, but modern machines have multiple cores and full performance is only reached through parallelization. Unless otherwise specified, parallel algorithms assume that all data is available equally everywhere. Distributed algorithms are capable of working with data that is… distributed… on different machines. Deterministic or non-deterministic

For a given input, the execution of deterministic algorithms always yield the same result via the same steps. Non-deterministic algorithms include an element of randomness and will not, in general, generate exact repeatable results. An example of a non-deterministic step in an algorithm is randomly distributing points in a space to generate a mesh, or choosing random elements from a set.

Because of the possibility of race conditions, parallel algorithms may be inadvertently non-deterministic. In computer science jargon, this is known as "fun". Exact or approximate

Algorithms that never reach an exact result are called approximate algorithms. These can be far faster than an exact algorithm for the same problem, yet yield an equally useful result. Much of the advances in computational methods in the last 20 years have come about due to the discovery of better approximate algorithms. Digital or quantum

Quantum algorithms rely on qubits and are well beyond the scope of this instructor's skill set. Let's just stick with ordinary digital computer science, shall we? More classification!

```
Divide and conquer algorithms - divide the problem into smaller subproblems of the same type
Brute force algorithms - try all possible solutions until a satisfactory solution is found.
Randomized algorithms - use a random number at least once during the computation to find a so
Greedy algorithms - find an optimal solution at the local level with the intent of finding an
Recursive algorithms - solve the lowest and simplest version of a problem to then solve incre
Backtracking algorithms - divide the problem into subproblems, each which can be attempted to
Dynamic programming algorithms - break a complex problem into a collection of simpler subprob
```

Algorithms

How do computer scientists develop and analyse algorithms systematically?

One of their main tools is complexity analysis — simply: how does the work required to solve a problem change with the size of the problem asymptotically?

Asymptotically means "for very big problems". This also means that, in practice, computer science can lead you astray.

An example: the recursive Fibonacci function

F(n): if n=1 return 1 if n=2 return 1 return F(n-1) + F(n-2)

For N=1 or 2, the function returns after just one or two instructions. But for N=3, it needs like seven instructions. For N=4, it'll need twelve. Here is how the work grows: n work 3 7 4 12 5 22 6 37 7 62 8 102

The required amount of work for that algorithm grows faster than the Fibonacci sequence! As the plot below shows, it's an exponential increase! This is awful!

cost of recursive fibonacci

Here is another example: finding the largest number in a random list

findLargest(list): largest = -1 for each item in list: if item > largest: set largest=item return largest

Here, we clearly see that we iterate over each item in the list. The number of instructions executed for a list of size N is 2 + 2*N+k, for some number k<N that depends on how unordered the list is. For big N, the constant term (2) is completely insignificant.

In the study of algorithms, "big O notation" is often used to describe the asymptotical behaviour of an algorithm. In big O notation, the k is also dropped, because it is less than N, and the factor of 2 is dropped as well. The complexity of the findLargest algorithm is O(N).

Actually, this is just the time complexity of the algorithm. But the number of execution steps may not be the only value of interest! Suppose you're interested in how *big* a problem becomes in memory?

Welcome to space complexity. Instead of the number of steps, just count the number of variables, size of arrays, etc. But also consider the overhead of a function call. Here is a great resource for complexity of common algorithms

- many algorithms

1. Fowler M. Refactoring. Addison-Wesley Professional; 2018.

2. Sutter H, Alexandrescu A. C++ coding standards: 101 rules, guidelines, and best practices. Pearson Education; 2004.