Australian National University

COMP1730/COMP6730
Programming for Scientists

Code Quality

Announcements

∗ You can discuss your homework 1 with tutor in lab this week.

∗ Homework 2 marks may be available soon.

Australian National University

What is code quality and why should we care?

- Writing code is easy – writing code so that you (and others) can be confident that it is correct is not.
- You will always spend more time finding and fixing the errors that you made (“bugs”) than writing code in the first place.
- Good code is not only correct, but helps people (including yourself) *understand* what it does and why it is correct.

(Extreme) example

* What does this function do? Is it correct?

```
def AbC(ABc):
    ABC = len(ABc)
    ABc = ABc[ABC-1:-ABC-1:-1]
    if ABC == 0:
        return 0
    abC = AbC(ABc[-ABC:ABC-1:])
    if ABc[-ABC] < 0:
        abC += ABc[len(ABc)-ABC]
    return abC
```

(Extreme) example – continued

* What does this function do? Is it correct?

```
def sum_negative(input_list):
    """Return sum of all negative numbers in input_list."""
    total = 0
    i = 0
    while i < len(input_list):
        if input_list[i] < 0:
            total = total + input_list[i]
        i = i+1
    return total
```

Aspects of code quality

1. Commenting and documentation.
2. Variable and function naming.
3. Code organisation (for large programs).
4. Code efficiency (somewhat).

What makes a good comment?

- Raises the level of abstraction: *what* the code does and *why*, not *how*.
  - Except when “how” is especially complex.
- Describe parameters and assumptions
  – python is not a typed language.

```
def sum_negative(input_list):
    """Return sum of negative numbers in input_list.
    Assumes input_list contains only numbers."""
```

- Up-to-date and in a relevant place.
- Good commenting is most important when learning to program and when working with other people.

How <u>not</u> to comment

* Don't use comments to make up for poor quality in other aspects (organisation, naming, etc.).

```
x = 0 # Set the total to 0.
```

* Wrong, or in the wrong place.

```
# loop over list to compute sum
avg = sum(the_list) / len(the_list)
```

* Stating the obvious.

```
x = 5 # Sets x to 5.
```

* Assume the reader knows python.

Function docstring

- A (triple-quoted) string as the first statement inside a function (module, class) definition.
- State the *purpose* and *limitations* of the function, parameters and return value.

```python
def solve(f, y, lower, upper):
    """Returns x such that f(x) = y (approximately).
    Assumes f is monotone and that a solution lies in the interval
    [lower, upper] (and may recurse infinitely if not)."""
```

- Can be read by python’s `help` function.

∗ Some format conventions for parameters and return value:

```
def solve(f, y, lower, upper):
    """Find x such that f(x) = y (approximately).

    :param f: a monotonic function with one numeric parameter and
    return value.
    :param y: integer or float, the value of f to solve for.

    ...

    :returns: float, value of x such that f(x) within +/− 1e−6 of y.
    """
```

Good naming practice

* The name of a function or variable should tell you what it does / is used for.
* Variable names should not *shadow* a names of standard types, functions, or significant names in an outer scope.

```
def a_fun_fun(int):
    a_fun_fun = 2 * int
    max = max(a_fun_fun, int)
    return max < int
```

(more about scopes in a coming lecture).

* Names can be long (within reason).
  - A good IDE will autocomplete them for you.
* Short names are not always bad:
  - `i` (`j`, `k`) are often used for loop indices.
  - `n` (`m`, `k`) are often used for counts.
  - `x`, `y` and `z` are often used for coordinates.
* Don’t use names that are confusingly similar in the same context.
  - E.g., `sum_of_negative_numbers` vs. `sum_of_all_negative_numbers` – what’s the difference?

Code organisation

- Good code organisation
  - avoids repetition;
  - fights complexity by isolating subproblems and encapsulating their solutions;
  - raises the level of abstraction; and
  - helps you find what you’re looking for.
- python constructs that support good code organisation are functions, classes (not covered in this course) and modules (later).

Functions

* Functions promote abstraction, i.e. they separate *what* from *how*.
* A good function (usually) does *one* thing.
* Functions reduce code repetition.
  - Helps isolate errors (bugs).
  - Makes code easier to maintain.

* A function should be as general as it can be without making it more complex.

```
def solve(lower, upper):
    """Returns x such that x ** 2 * pi ˜= 1. Assumes ..."""
```

vs.

```
def solve(f, y, lower, upper):
    """Returns x such that f(x) ˜= y. Assumes ..."""
```

Problem: Below is a function that returns the position of an element in a sequence:

```
def my_func(sq, y):
    x = 0 # index
    # this handles the case when y is not in the sequence
    if len(sq) == 0:
        return 0
    while sq[x] != y:
        x = x + 1
        # np.max returns the maximum number in an array
        if x < len(sq):
            x = x + 1 - 1 # don't want to change i again
        else:
            return x
    # this is the end of the loop
    return x
```

Can the quality of this code be improved?

Efficiency

Premature optimisation is the root of all evil in programming.

C.A.R. Hoare

- Modern computers usually have enough power to solve your problem, even if the code is not perfectly efficient.
- Programmer time is far more expensive than computer time.
- Code correctness, readability and clarity is more important than optimisation.

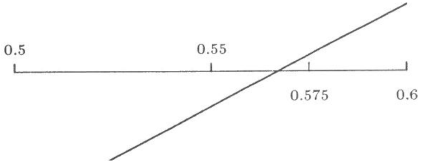
When should you consider efficiency?

- For code that is going to run *very* frequently.
- If your program is too slow to run at all.
  A poor choice of algorithm or data structure may prevent your program from finishing, even on small inputs.
- When the efficient solution is just as simple and readable as the inefficient one.

Interval halving algorithm revisited

- Assumption: $f(x)$ is monotone increasing and crosses 0 in the interval $[lower, upper]$.
- Idea:
  - Find the middle of the interval, $m = (lower + upper)/2$:
  - if $f(m) \approx 0$, return m;
  - if $f(m) < 0$, the solution lies between m and $upper$;
  - if $f(m) > 0$, the solution lies between $lower$ and m.

- *Don't compare `floats` with ==.*



Australian National University

Solution using iteration (reminded)

```python
import math
def f(x):
    return x**2 * math.pi - 1

def interval_halving(lower, upper):
    while upper - lower > 1e-10:
        middle = (lower+upper)/2
        value = f(middle)
        if abs(value) < 1e-6:
            return middle
        elif value < 0:
            lower = middle
        else:
            upper = middle
    return "No value found"
```

Another algorithm - exhaustive search

```
def solve_exhaustive(lower, upper):
    x = lower
    while x <= upper:
        value = f(x)
        if abs(value) < 1e-6:
            return x
        x = x + 1e-6

    return "No solution found"
```

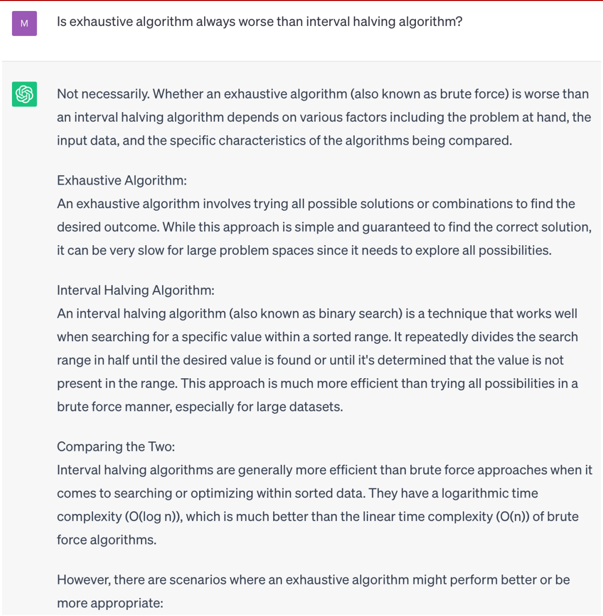
Is this algorithm considered always worse than the previous algorithm?

Question: Is exhaustive algorithm always worse than interval halving algorithm?

(This slide is added after lecture)

Results (correct answer in bold): 75 votes in total

- Yes: 8 votes
- **No**: 61 votes
- I don't know: 6 votes


M

Is exhaustive algorithm always worse than interval halving algorithm?

Not necessarily. Whether an exhaustive algorithm (also known as brute force) is worse than an interval halving algorithm depends on various factors including the problem at hand, the input data, and the specific characteristics of the algorithms being compared.

Exhaustive Algorithm:
An exhaustive algorithm involves trying all possible solutions or combinations to find the desired outcome. While this approach is simple and guaranteed to find the correct solution, it can be very slow for large problem spaces since it needs to explore all possibilities.

Interval Halving Algorithm:
An interval halving algorithm (also known as binary search) is a technique that works well when searching for a specific value within a sorted range. It repeatedly divides the search range in half until the desired value is found or until it's determined that the value is not present in the range. This approach is much more efficient than trying all possibilities in a brute force manner, especially for large datasets.

Comparing the Two:
Interval halving algorithms are generally more efficient than brute force approaches when it comes to searching or optimizing within sorted data. They have a logarithmic time complexity (O(log n)), which is much better than the linear time complexity (O(n)) of brute force algorithms.

However, there are scenarios where an exhaustive algorithm might perform better or be more appropriate:

Source: ChatGPT

Take-away

* Good code organisation and documentation is important:
  - For others to understand your code.
  - For you to understand what you have done wrong.
* Efficiency, generality and compactness are also good qualities of code, but secondary to clarity.