(Un)helper functions

Kevin Smith[1] Bernhard Egger[2*]

[1] Magic Institute of Technology
[2] Fantastic-Amazing-University Erlangen-Nürnberg (FAU)
k2smith@mit.edu bernhard.egger@fau.de
[*] this author was (un)helpful

Abstract

Here [4] we provide a number of functions to solve common programming problems, in pseudo-code such that they are independent of any single programming language. We provide too few details to implement any of these algorithms without knowing how to solve the problems independently, yet just enough that we can respond with a disdainful "look at the paper" if ever asked for help. We further ignore glaring logical errors and simply assume that the algorithms will work as intended. Finally, we end with grandiose claims about the uniqueness of our work, actively ignoring contributions from vast areas of the field.

1. Introduction

Programming is a time-intensive activity that often keeps us from doing a number of important other tasks, like writing up submissions to comedic conferences. However, as OpenAI's Codex has demonstrated, most programs can be written by copying code written by others.[1] Thus we propose that by providing example algorithms we can speed up the pace of science, and pad our citation counts by yelling at anyone who uses a vaguely related function without citing us.

In this submission to the Sigbovik "algorithms and 17^{th} century poetry" track (which we just made up), we propose a number of non-standard but efficient solutions to common programming tasks. We present these algorithms in pseudo-code in order to allow readers to implement them in a language-agnostic way. Also, we were too lazy to code them up ourselves, so no Github link will be provided.

2. Algorithms

2.1 do_this

Often, it is important to call a function and return its output. Algorithm 1 demonstrates how to do so, regardless of the specifics of any programming language.

Algorithm 1 An algorithm for running a function

```
function DO_THIS(f)
    r ← call(f)
    return r
end function
```

2.2 is_even

Introductory programming courses often have students write the function is_even, but in of itself this function definition is underspecified: should this be interpreted as a question ("*is* this number even?") or an assertion ("this number *is* even")? In order to cover all of our bases, we provide Algorithm 2 which does both.

[1] And of course changing variable names to avoid charges of plagiarism.

Algorithm 2 The is_even algorithm

```
function IS_EVEN(x)
    if x%2 ≠ 0 then
        raise Error "x is not even"
    end if
    return TRUE
end function
```

2.3 self_sort

Sorting lists is one of the most fundamental tasks in programming, used in almost every advanced computer algorithm. However, to date even the best sorting functions are slow, running at best in $O(n\ log\ n)$ time for a list of length n. Furthermore, in untyped languages comparisons can be ambiguous and lead to user confusion – for instance, is the string "3" less than the int 4?

In Algorithm 3, we present a novel approach that solves both of these problems by offloading computation onto the user. We ask the user at each point in time to pick the smallest item, until no items remain. This algorithm is guaranteed to run in $O(n)$ time, and ambiguous comparisons are always resolved in the way the user expects. Any violations of these guarantees can be trivially attributed to user error.

Algorithm 3 An algorithm for sorting a list

```
function INNER_SORT(L_unsorted, L_sorted)
    if empty(L_unsorted) then
        return L_sorted
    end if
    print L_unsorted
    print What is the index of the smallest item?
    idx ← user input
    L_sorted ← append(L_sorted, L_unsorted[idx])
    remove L_unsorted[idx]
    return inner_sort(L_unsorted, L_sorted)
end function
function SELF_SORT(L)
    return inner_sort(L, [])
end function
```

2.4 real_rand_int

Many algorithms require stochastic input to function, and thus rely on random number generators. However, these are typically *pseudo*random number generators, and anyone who has taken Latin or Greek or something knows that *pseudo* means *false*. To improve the quality of our code, we want *real* random numbers, and provide them using the following algorithm that leverages real humans to generate random numbers.

This process, shown in Algorithm 4, uses Amazon's Mechanical Turk, a website used to hire workers to answer simple surveys. This algorithm creates a simple webpage with a query for a random

Figure 1. Example survey for $real\_rand\_int(1, 42)$.

integer (see Figure 1), posts this query to Mechanical Turk, and waits until a human answers and provides an appropriately random number.[2]

Algorithm 4 An algorithm to provide *real* random integers

```
function REAL_RAND_INT(a, b)
    page ← create_site(a, b)
    posting ← post_to_mturk(page)
    while TRUE do
        result ← query(posting)
        if posting is not NULL then
            return result
        end if
    end while
end function
```

2.5 aggressive_ping

A standard way to test whether one can connect to a web server is to 'ping' it: sending an intermittent packet to an IP address and waiting for a response from that server. However, each individual ping can be unreliable, as packets can be dropped either on the way out or as they return, and the time to respond can vary depending on the route the packet takes. We solve these problems by sending out a number of pings from a large number of servers, and aggregating results over all of them, as described in Algorithm 5.

Algorithm 5 An algorithm to aggregate a large number of pings

```
function AGGRESSIVE_PING(ip)
    B ← activate_botnet(N = 10,000)
    while TRUE do
        pings ← []
        for b ∈ B do
            p ← ping(b, ip)
            pings ← append(pings, p)
        end for
        result ← count_true(pings)/count(pings)
        print result
    end while
end function
```

Note that this algorithm typically finds that websites are up and stable for a few seconds, followed by a sustained failure to respond.

2.6 endless_loop

Ever stand at the edge of a dark pit, and you drop a pebble in, but it's so far down that you can't hear it hit anything, and you think to yourself with amazement, "Wow! I wonder if this pit goes on forever!"? Running Algorithm 6 will give you the same feeling, but nerdier.

[2] Note that because we forgot to add code to confirm that the number entered is within the range provided in the function, there are no guarantees that the function returns a value in this range, or even is a number at all. We also enable code injection by design in case participants would like to write a function to generate their random number.

Algorithm 6 An algorithm to loop endlessly

```
function ENDLESS_LOOP
    while TRUE do
    end while
end function
```

2.7 recursive_endless_loop

If Algorithm 6 is like dropping a pebble down a mineshaft, Algorithm 7 is like dropping a nuclear bomb into a black hole while heavy metal is blasting.

Algorithm 7 An algorithm to loop endlessly, endlessly

```
function RECURSIVE_ENDLESS_LOOP
    while TRUE do
        do_this(recursive_endless_loop)
    end while
end function
```

2.8 halts

A fundamental theorem of computer science is the undecidability of the "halting problem" [5]. This theorem states that it is impossible to write an algorithm that takes another function as input and outputs whether or not that function halts. Using Algorithm 8 as an existence proof, we show that this fundamental theorem is wrong.

Algorithm 8 An algorithm to test whether a function halts or not

```
function HALTS(f)
    running ← TRUE
    spawn process P, running do_this(f)
    while running do
        if has_ended(P) then
            return TRUE
        end if
    end while
    return FALSE
end function
```

Note: do not call this function on $recursive\_endless\_loop$. We can neither confirm nor deny that this function halts, but it has been running on our AWS server farm for two weeks without providing a definitive answer, costing us over $\$10,000$ so far. Nonetheless, this is a small sacrifice for scientific knowledge.

2.9 professor_coding

If you are reading this, you are likely either a beginning computer science student, or a professor past their prime who is trying to figure out how to code for that one class those bean-counters in the department are still making you teach. And if it's the later, we get it: when you were a grad student, even C was a luxury, and you don't have time to learn Snake or Porch or Jessica or whatever the kids are using these days. The good news is there is a simple way to turn your ideas into code automatically.

This process, shown in Algorithm 9, is straightforward: simply present your idea to a graduate student, wait an appropriate amount of time, and you should be magically provided with the code. If this fails, not to worry, you can repeat the process with other graduate students. Depending on your intended lab culture, this loop over graduate students can be parallelized for efficiency. If you loop through all graduate students and the code is not yet produced, clearly it's not a problem with your idea, so it must be the graduate students; in this case, replace your graduate students and try again.

Algorithm 9 An algorithm to transmute ideas into code

```
function PROFESSOR_CODING(idea, graduate_students)
    for student ∈ graduate_students do
        present(idea, student)
        while time < deadline do
            code ← query(student)
            if code is not NULL then
                return code
            end if
        end while
    end for
    new_students ← replace_students()
    return professor_coding(idea, new_students)
end function
```

3. Related work

We believe that these algorithms are unique and thus there is no comparable work. However, for the sake of padding our numbers and increasing the impact factor of Sigbovik, here we cite a number of our own papers published previously and simultaneously in this conference [1–3, 6].

4. Disclaimer

By even looking at this paper, you are releasing the authors from legal liability due to any adverse effects of running these algorithms, including, but not limited to, computer explosions, FBI raids, genetically modified super-spiders, or awakening the elder gods of chaos.

References

[1] B. Egger and M. Siegel. Honkfast, prehonk, honkback, prehonkback, hers, adhonk and ahc: the missing keys for autonomous driving. *SIGBOVIK*, 2020.

[2] B. Egger, K. Smith, and M. Siegel. openCHEAT: Computationally helped error bar approximation tool-kickstarting science 4.0. *SIGBOVIK*, 2021.

[3] B. Egger, K. Smith, T. O'Connell, and M. Siegel. Action: A catchy title is all you need! *SIGBOVIK (under careful review by very talented, outstanding reviewers)*, 2022.

[4] K. Smith and B. Egger. (un)helper functions. *SIGBOVIK (under careful review by very talented, outstanding reviewers)*, 2022.

[5] A. M. Turing et al. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936.

[6] M. Weiherer and B. Egger. A free computer vision lesson for car manufacturers or it is time to retire the erlkönig. *SIGBOVIK (under careful review by very talented, outstanding reviewers)*, 2022.