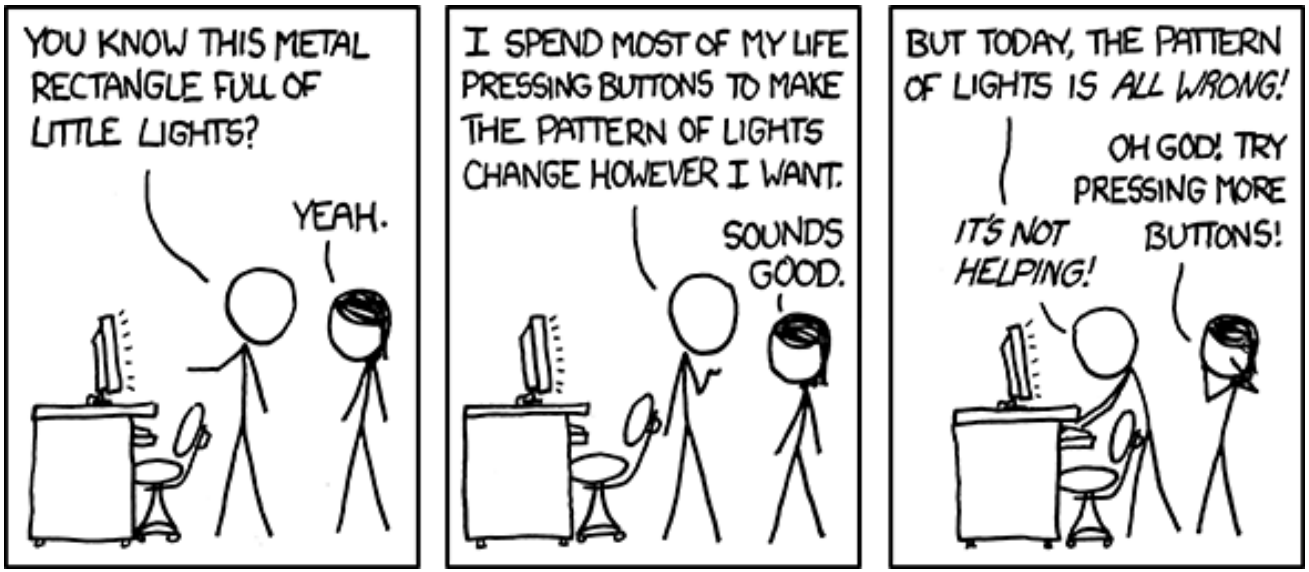
About Algorithms

Pedro Ribeiro

DCC/FCUP

2024/2025



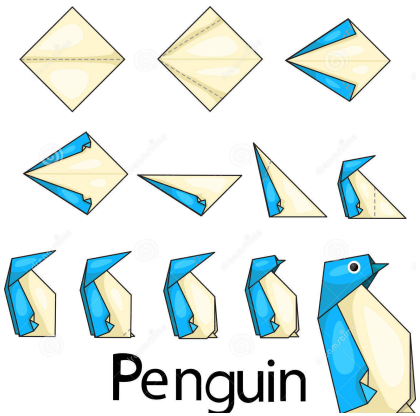
Algorithms as the core ideas

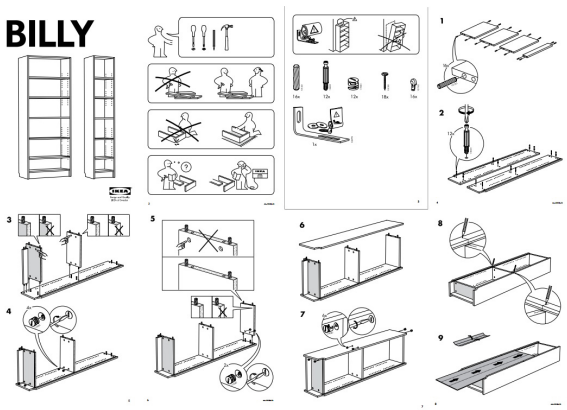
What is an algorithm?

A set of executable **instructions** to solve a **problem**

- The problem is the **motivation** for the algorithm
- The instructions need to be **executable**
- There are generally **several algorithms** for the same problem [How to choose?]

- **Representation**: description of the instructions clear enough for its "audience"






```c
int n, sum = 0;

printf("Enter values; 0 ends.\n");
scanf("%d", &n);    // first value
while (n != 0) {    // while not finished
  sum += n;         // accumulate
  scanf("%d", &n); // read next value
}
printf("The sum is: %d\n", sum);
```

What is an algorithm?

Computer Science version

- The algorithms are the **ideas** behind programs
  They are independent from programming language, machine, ...
- An algorithm solves a **problem**
- The problem is characterized by the description of its **input** and **output**

A classic example:

Sorting Problem

Input: a sequence $\langle a_1, a_2, \ldots, a_n \rangle$ of n numbers
Output: a permutation of the numbers $\langle a_1', a_2', \ldots, a_n' \rangle$ such that $a_1' \leq a_2' \leq \ldots \leq a_n'$

Example for the Sorting Problem

Input: 6 3 7 9 2 4
Output: 2 3 4 6 7 9

Properties we want on an algorithm

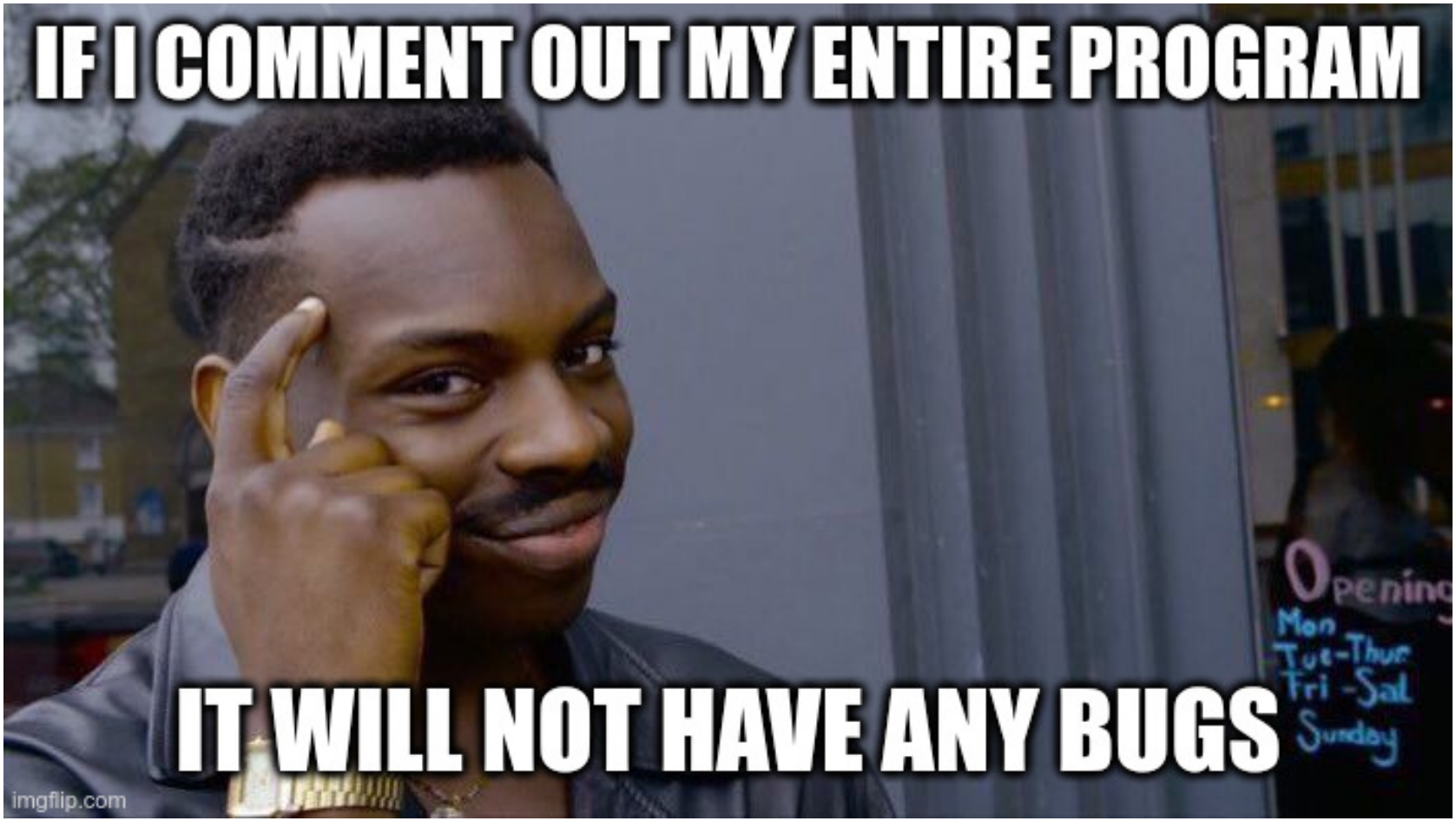
Correction

It should solve correctly **all instances** of the problem

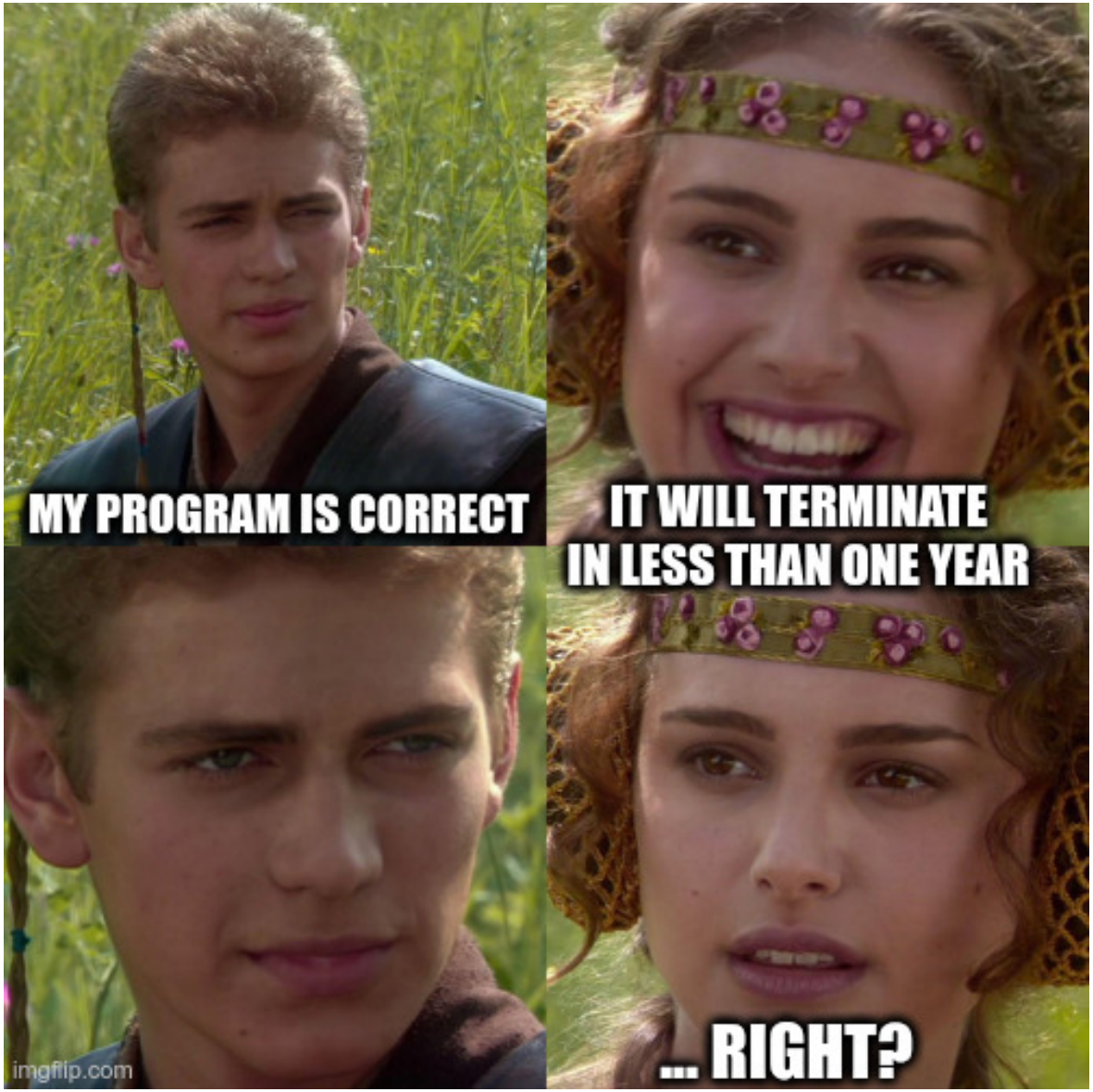
Efficiency

(**Time** and **Memory**) performance has to be adequate

About correction



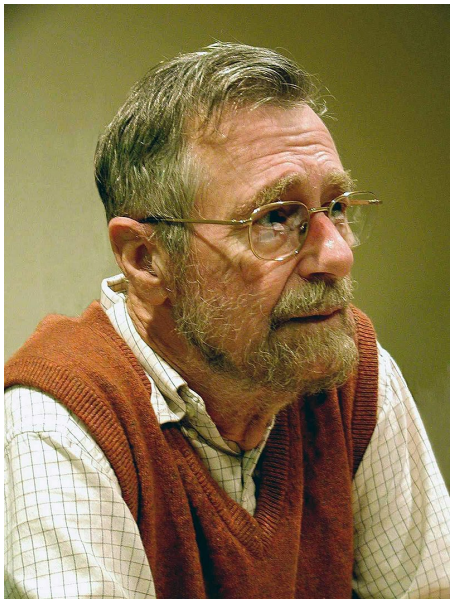
About efficiency



Correction of an algorithm

- **Instance**: A concrete example of a valid input
- A correct algorithm solves **all possible instances**
  Examples for sorting: repeated numbers, already sorted sequences, ...
- It is not always easy to **prove** the correction of an algorithm and even less it is obvious if an algorithm is correct

Dijkstra



Edsger W. Dijkstra (Wikipedia entry) (Wikiquote)
[1972 Turing Award]

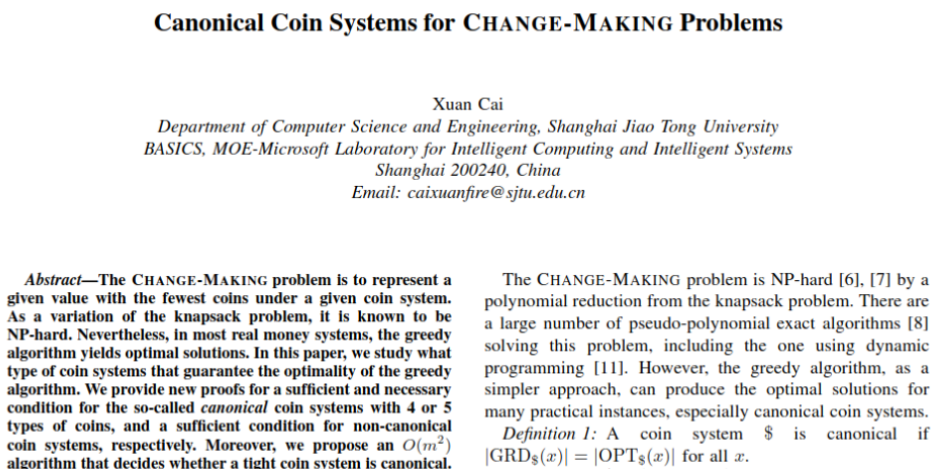
"Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence."

An Example of (in)Correction

- How to use the **smallest possible number of coins** to make a certain amount?
  (assuming we have an infinite supply of coins)
- **Greedy idea:** use the **largest possible coin** still lower or equal, repeat the process with the remaining amount
- Example: $3.45€ = 2€ + 1€ + 0.20€ + 0.20€ + 0.05€$ (5 coins)

An Example of (in)Correction

- Will this algorithm always produce the minimum number of coins?
- For common coin systems (e.g. euro, dollar): yes!
- For a general coin system... no!
- Example: coins $\{1, 5, 6\}$
  - Greedy algorithm would give: $10 = 6 + 1 + 1 + 1 + 1$ **(5 coins)**
  - Minimum would be: $10 = 5 + 5$ **(2 coins)**



Canonical Coin Systems for CHANGE-MAKING Problems

Xuan Cai
Department of Computer Science and Engineering, Shanghai Jiao Tong University
BASICS, MOE-Microsoft Laboratory for Intelligent Computing and Intelligent Systems
Shanghai 200240, China
Email: caixuanfire@sjtu.edu.cn

***Abstract*—The CHANGE-MAKING problem is to represent a given value with the fewest coins under a given coin system. As a variation of the knapsack problem, it is known to be NP-hard. Nevertheless, in most real money systems, the greedy algorithm yields optimal solutions. In this paper, we study what type of coin systems that guarantee the optimality of the greedy algorithm. We provide new proofs for a sufficient and necessary condition for the so-called *canonical* coin systems with 4 or 5 types of coins, and a sufficient condition for non-canonical coin systems, respectively. Moreover, we propose an $O(m^2)$ algorithm that decides whether a tight coin system is canonical.**

The CHANGE-MAKING problem is NP-hard [6], [7] by a polynomial reduction from the knapsack problem. There are a large number of pseudo-polynomial exact algorithms [8] solving this problem, including the one using dynamic programming [11]. However, the greedy algorithm, as a simpler approach, can produce the optimal solutions for many practical instances, especially canonical coin systems.

Definition 1: A coin system $\$$ is canonical if $|\mathrm{GRD}_{\$}(x)| = |\mathrm{OPT}_{\$}(x)|$ for all x.

(this is an actual studied problem in algorithms)

Another Example of (in)Correction

Real (Pascal) problem made by a student of R. Backhouse:
(here translated to python - we will talk about strings in C later)

```
def equal_strings(s1, s2):
    is_same = (len(s1) == len(s2))

    if is_same:
        for i in range(0, len(s1)):
            is_same = (s1[i] == s2[i])

    return is_same
```

```
>>> equal_strings("university", "university")
True
```

```
>>> equal_strings("course", "course")
True
```

```
>>> equal_strings("", "")
True
```

all valid instances of equal strings

Another Example of (in)Correction

```
def equal_strings(s1, s2):
    is_same = (len(s1) == len(s2))

    if is_same:
        for i in range(0, len(s1)):
            is_same = (s1[i] == s2[i])

        return is_same
```

```
>>> equal_strings("university", "course")
False
```

```
>>> equal_strings("tables", "course")
False
```

all valid instances of different strings

```
>>> equal_strings("pure", "true")
True
```

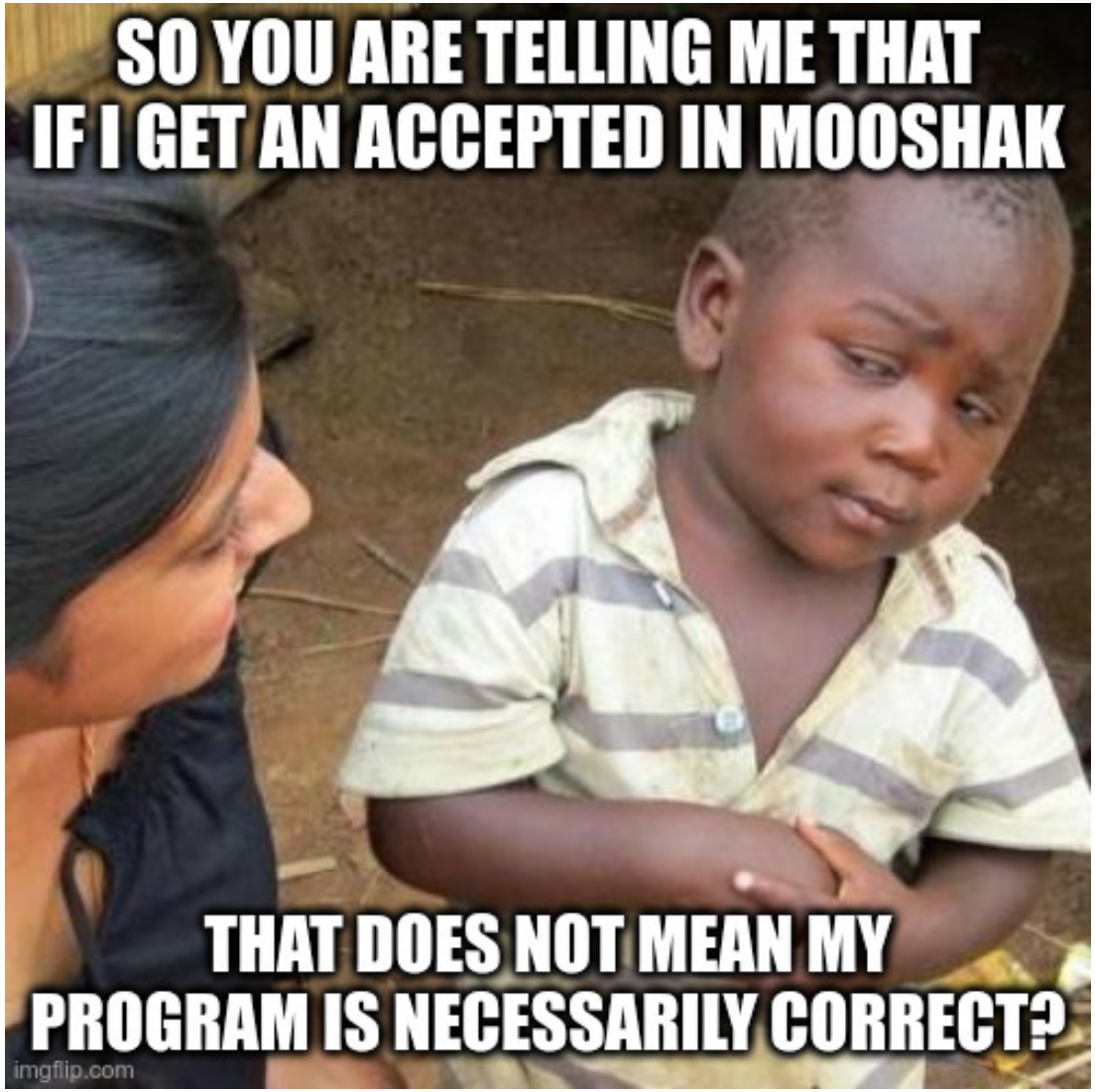
here is an incorrect instance!

(the function is just testing if the length and the last letter are the same)

About the correction of algorithms

- In a computer science degree you would study more formally about algorithm correction on other courses (here we will mainly be relying on intuition or very brief sketches of a proof)
- **Remember to always be careful about correction**
- Test your own program! (Mooshak does not exist "on the real world")
- Using Mooshak does not guarantee correction! It simply says your code "passed" the tests someone (the professor) put there

Mooshak and Correction



Correction

- Finding a program we thought was correct to be actually incorrect can happen even to computer scientists in published scientific papers!

Maximum-area triangle in a convex polygon, revisited

Ivor van der Hoog [a], Vahideh Keikha [b], Maarten Löffler [a], Ali Mohades [b], Jérôme Urhausen [a]

Show more

+ Add to Mendeley Share Cite

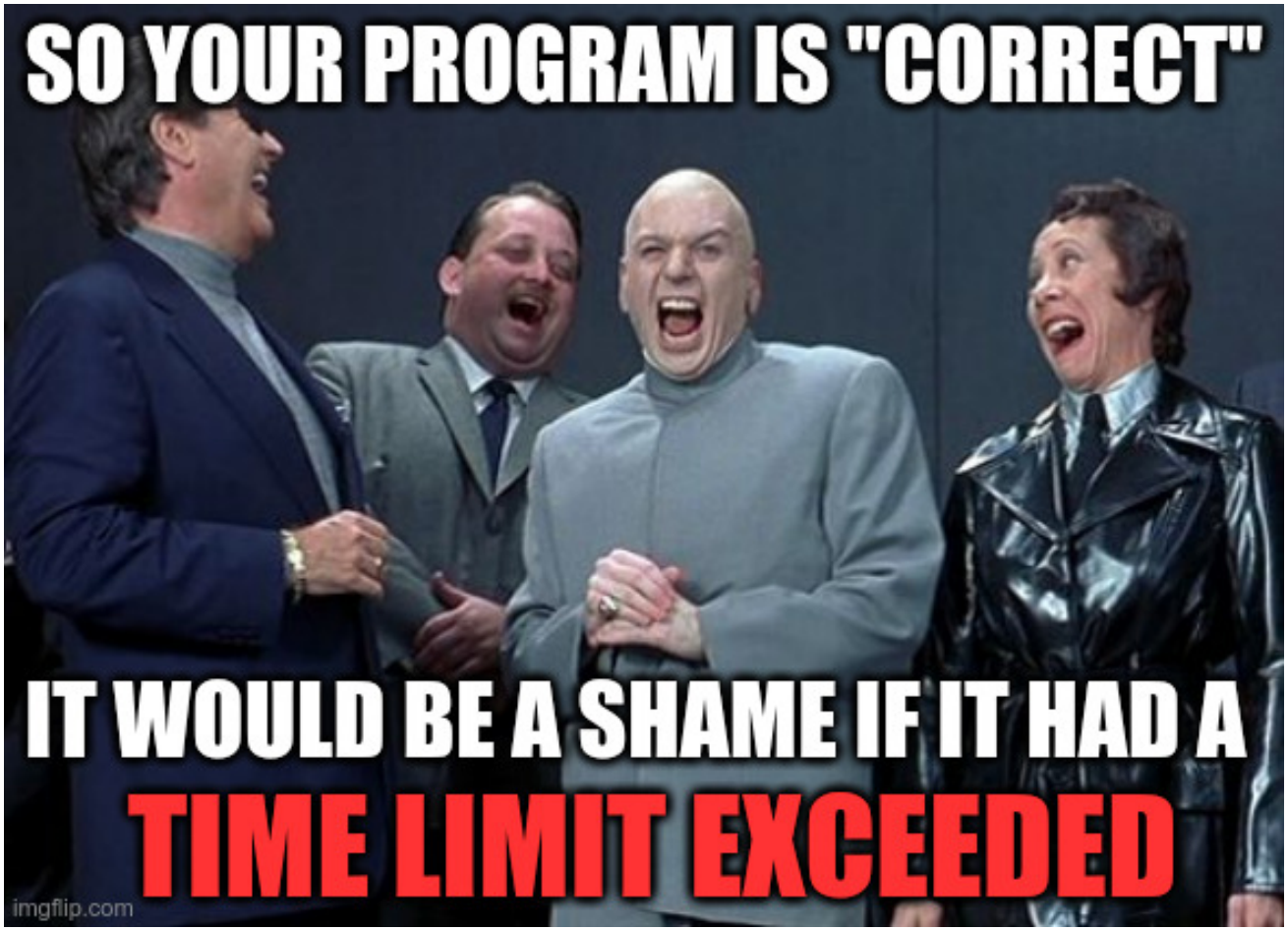
https://doi.org/10.1016/j.ipl.2020.105943 Get rights and content

Highlights

- This is a short letter, with just one message: the algorithm for computing the largest-area inscribed triangle described in 1979 by Dobkin and Snyder is incorrect.
- We present a 9-vertex polygon on which the algorithm by Dobkin and Snyder for computing the largest-area triangle fails.
- We present a 16-vertex polygon on which the algorithm by Dobkin and Snyder for computing the largest-area quadrangle fails.

Algorithmic Efficiency

- So your program is correct... but will it run on time? (and within memory limits)



Algorithmic Efficiency

- In a computer science degree you would study more formally about efficiency on other courses (asymptotical analysis and Big O notation)
- At this course we will sometimes start to give you intuitions and make you seek efficiency (yes, there will be some problems with lots of TLEs)
- On this lecture I want to provide some simple examples to give you a hands-on experience (building a program from scratch *vs* to seeing written code)

Algorithmic Efficiency

(the following slides about finding primes describe thee live coding session I did in class)

Algorithmic Efficiency - Finding Primes

- On the class I did live coding for the following problem:

Finding Primes

Print the list of all prime numbers smaller or equal than N

- The goal was to be able to find all primes smaller than 10 million in less than 1 second.

Algorithmic Efficiency - Finding Primes

- The solution I coded was to call a function `is_prime(i)` for all possible primes i; the function would simply traverse all integers smaller than i and check if they divide i: `primes_v1.c` (source)

```c
#include <stdio.h>

int is_prime(int i) {
  for (int j=2; j<i; j++)
    if (i % j == 0)
      return 0;
  return 1;
}

int main(void) {

  int n;
  scanf("%d", &n);
  for (int i=2; i<=n; i++)
    if (is_prime(i) == 1)
      printf("%d\n", i);

  return 0;
}
```

Algorithmic Efficiency - Finding Primes

- To compile this code we can use for instance:

```
gcc -Wall -o primes_v1 primes_v1.c
```

- The code asks for an input integer. One way to make this is to create a file with the desired input.

  Imagine we have a file `input.txt` with the following contents:

```
10
```

  To call the program with that input we can simply redirect the input:

```
./primes_v1 < input.txt
```

  Which will produce the following output:

```
2
3
5
7
```

Algorithmic Efficiency - Finding Primes

- We can also redirect the output:

```
./primes_v1 < input.txt > output.txt
```

  This will put the output on file `output.txt`
- In a Linux terminal we could check the number of lines on the output with the command `wc -l output.txt` which we print `4` in this case (primes less or equal than 10)
- In Linux, to measure the time, we could use the command `time`:

```
time ./primes_v1 < input.txt > output.txt
```

  Which could give as output something as:

```
real    0m0,001s
user    0m0,001s
sys     0m0,000s
```

  We are interested in the **user** time which corresponds to the CPU time used (*real* is the elapsed time and could be influenced by other processes you are running on your computer).

Algorithmic Efficiency - Finding Primes

- With this, we could start measuring the time spent as we increase N (the upper limit of the primes to find):
  - $N = 100$: 0.001s (essentially *instantaneous*)
  - $N = 1\,000$: 0.002s
  - $N = 10\,000$: 0.021s
  - $N = 100\,000$: 1.067s
  - $N = 1\,000\,000$: 90.431s
  - $N = 10\,000\,000$: ...

  We can observe that the time grows really fast as we are increasing N

  This is what we are interested on algorithmic (time) efficiency: **how does the runtime time grow as we increase the input size?**

Algorithmic Efficiency - Finding Primes

- We can make a simple optimization, by changing the `is_prime(i)` function to firt check if i is even and then only check odd divisors:

`primes_v2.c` (source)

```
int is_prime(int i) {
  if (i==2) return 1;
  if (i%2 == 0) return 0;
  for (int j=3; j<i; j+=2)
    if (i % j == 0)
      return 0;
  return 1;
}
```

- This will allows to roughly spend 2x less time, but we are not fundamentally changing the *rate of growth* of the execution time as we increase N:
  - $N = 10\,000$: 0.009s
  - $N = 100\,000$: 0.511s
  - $N = 1\,000\,000$: 37.876s

Algorithmic Efficiency - Finding Primes

- Now we could make a really big improvement by noticing `i` only needs to check the primes up to $\sqrt{i}$

  Note: this is because if $i = a \times b$ then one of a or b will necessarily be $\leq sqrt(i)$ (or else their product would be bigger than i)

  `primes_v3.c` (source)

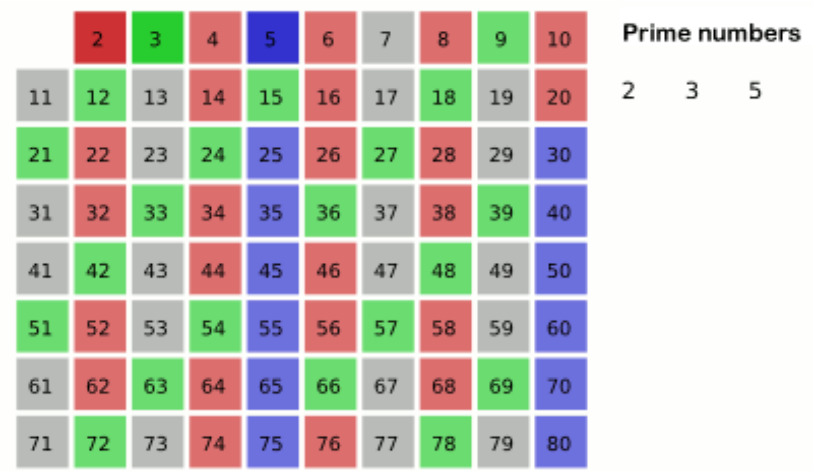
```
int is_prime(int i) {
  if (i==2) return 1;
  if (i%2 == 0) return 0;
  for (int j=3; j*j<=i; j+=2) // note the j*j<=i
    if (i % j == 0)
      return 0;
  return 1;
}
```

- This makes a real impact on the time need (which decreases by orders of magnitude) and on the growth ratio of this same time:
  - $N = 10\,000$: 0.001s
  - $N = 100\,000$: 0.009s
  - $N = 1\,000\,000$: 0.107s
  - $N = 10\,000\,000$: 2.110s *(almost on what we want...)*

Algorithmic Efficiency - Finding Primes

- This could already be acceptable, but there is still plenty of room for improvement!
- Eratosthenes (born on 276 BC) was an ancient greek mathematician that among other discoveries, introduced an efficient method for finding primes known as the **sieve of Eratosthenes**.

  Instead of doing trial division to sequentially try all possible divisors, we can **iteratively mark as non-primes the multiples of primes**
  - We first mark all multiples of 2 as non-primes
  - The next prime number is 3, we mark its multiples as non-primes
  - The next prime number is 5, we mark its multiples as non-primes
  - ...



Algorithmic Efficiency - Finding Primes

primes_v4.c (source)

```c
#include <stdio.h>

#define N 10000000

int is_prime[N+1]; // 0:  non-prime; 1:  prime

void sieve() {
  for (int i=2; i<=N; i++) // initialize all numbers as prime
    is_prime[i] = 1;

  for (int i=2; i<=N; i++)
    if (is_prime[i])
      for (int j=i+i; j<=N; j+=i)
        is_prime[j] = 0;
}

int main(void) {
  sieve();
  for (int i=2; i<=N; i++)
    if (is_prime[i])
      printf("%d\n", i);

  return 0;
}
```

Algorithmic Efficiency - Finding Primes

- With the (naive) sieve of Eratosthenes code would already reach our initial goal:
  - $N = 10\,000\,000$: 0.264s

- Note how now this code needs more memory (to have an array of size $N + 1$), and the maximum N would now be limited by the memory we have on our computer
  We are effectively trading memory for time
  (this can happen in many problems)

- We could continue improving our code, but this already exposed the most important concepts I was trying to convey:
  - There are **several possible correct algorithms** for the same task
  - Different algorithms have **different time and memory efficiency**
  - The time efficiency of an algorithm can me measured by looking at the **rate of growth of the execution time as the input size increases**

Algorithmic Efficiency - Searching

- Let's now look at another problem:

The search problem

Input:

- an array **v** storing **n** elements
- a target element **key** to search for

Output:

- **Index** *i* of **key** where **v[i]==key**
- **-1** (if **key** is not found)

Example:

v =

5	2	6	8	4	12	3	9

search(2, v, n) = 1
search(7, v, n) = -1
search(3, v, n) = 6
search(14,v, n) = -1

Algorithmic Efficiency - Searching

- variants for the case of arrays with **repeated values**:
  - indicate the position of the first occurrence
  - indicate the position of the last occurrence
  - indicate the position of any occurrence
  - indicate all the occurrences

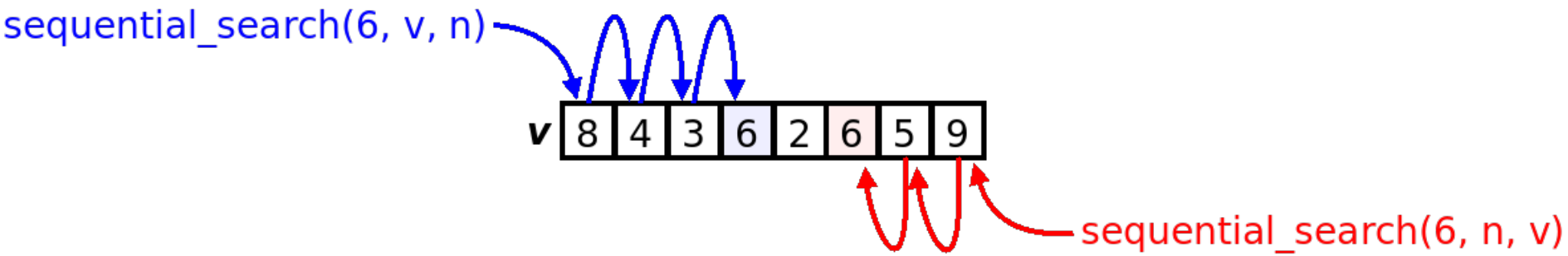
8	4	3	6	2	6	5	9

Sequential Search - Algorithm

Sequential Search Algorithm

Sequentially checks each element of the array, from the first to the last[a] or from the last to the first[b] , until a match is found or the end of the array is reached

[a] if you want to know the position of the first occurrence
[b] if you want to know the position of the last occurrence



suitable for small or unordered arrays

Sequential Search - An implementation

Search for an element *key* in a vector *v* of **n** elements. Returns the index of the first occurrence of *key*, if found, or -1, otherwise (e.g. for integers).

`sequential_search.c` (source)

```c
int sequential_search(int key, int v[], int n) {
  for (int i=0; i<n; i++)
    if (v[i] == key)
      return i; // found key
  return -1;    // not found
}
```

```c
  int n = 8;
  int v[] = {8,4,3,6,2,6,5,9};

  printf("%d\n", sequential_search(8, v, n));
  printf("%d\n", sequential_search(7, v, n));
  printf("%d\n", sequential_search(6, v, n));
```

```
0
-1
3
```

Sequential Search - Efficiency

- When analyzing the (time or memory) efficiency of a program, we can consider several cases:
  - The **best** case (in the what types of inputs it the code faster?)
  - The **worst** case (in the what types of inputs it the code faster?)
  - The **average** case (this implies we know what the average input looks like)

- In what concerns **memory**, sequential search does not use more memory as we increase n (we only need an extra variable i for the cycle)

- In what concerns **time**:
  - **Best case:** the key is on the first position of the array
  - **Worst case:** the key is not on the array
  - **Average case:** depends on the input

  Usually, when talking about algorithmic efficiency we are referring to the **worst case** (if we know input distribution, we could consider the average)
  - Example: on Mooshak I might have an input which is the worst possible case for you code (and your code need to pass it to have Accepted)

Sequential Search - Time Efficiency

- So what happens when we increase N on the worst case?

  The time needed grows linearly with N

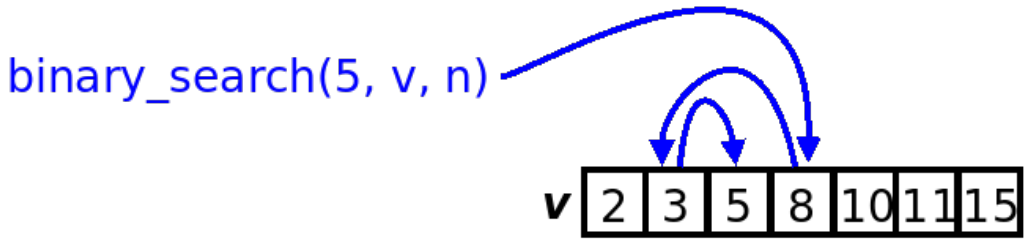
  - If N is $2\times$ times bigger, the time is also $2\times$ bigger
  - If N is $10\times$ times bigger, the time is also $10\times$ bigger
  - ...

- Using `test_sequential.c` (source) to perform 100 000 queries on an array of size N we get roughly the linear growth we expected:
  - $N = 1\,000$: 0.204s
  - $N = 10\,000$: 1.885s ($\approx$9.40$\times$ bigger)
  - $N = 100\,000$: 18.782s ($\approx$9.96$\times$ bigger)
  - $N = 1\,000\,000$: 196.132s ($\approx$10.44$\times$ bigger)
  - ...

- Can we do better for the **search problem**?

Searching in sorted arrays

- Suppose the array is **ordered**
  (arranged in increasing or non-decreasing order)
  - Sequential search on a sorted array still takes linear time
  - Can exploit sorted structure by performing **binary search**
  - Strategy: inspect middle of the array so that half of it is discarded at every step



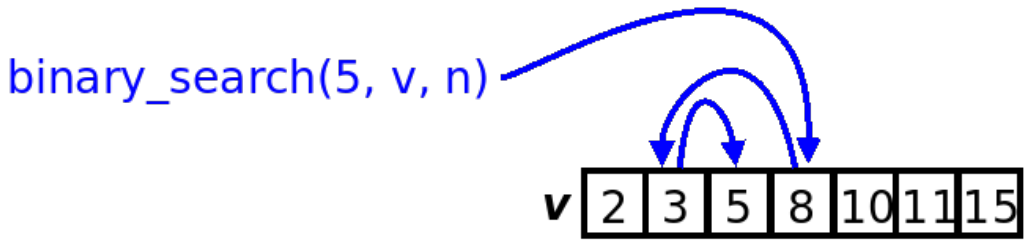
Binary Search - Algorithm

Binary Search Algorithm

compares the element in the middle of the array with the target element:

- is equal to the target element → found
- is greater than the target element → continue searching (in the same way) in the sub-array to the left of the inspected position
- is less than the target element → continue searching (in the same way) in the sub-array to the right of the inspected position

if the sub-array to be inspected reduces to an empty vector, we can conclude that the target element does not exist



Binary Search - Implementation

`binary_search.c` (source)

```
int binary_search(int key, int v[], int n) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int middle = low + (high - low) / 2;
        if (key < v[middle])       high = middle - 1;
        else if (key > v[middle])  low = middle + 1;
        else return middle; // found key
    }
    return -1; // not found
}
```

v = | 2 | 5 | 6 | 8 | 9 | 12 |

binary_search(8, v, n)

$low = 0, high = 5, middle = 2$

Since $8 > v[2]$: $low = 3, high = 5, middle = 4$

Since $8 < v[4]$: $low = 3, high = 3, middle = 3$

Since $8 == v[3]$: **return(3)**

Bugs in binary search

(if `low` and `high` are really high, `low+high` might overflow)

Binary Search - Time Efficiency

- So what happens now when we increase N on the worst case?
  - At the start we have N elements
  - With only 1 comparison, we reduce to $\approx N/2$ elements
  - With only 2 comparisons, we reduce to $\approx N/4$ elements
  - With only 3 comparisons, we reduce to $\approx N/8$ elements
  - With only 4 comparisons, we reduce to $\approx N/16$ elements
  - ...
  - With k comparisons, we reduce to $\approx N/2^k$ elements

- So how many comparisons do we need to reduce the search space to only one element?

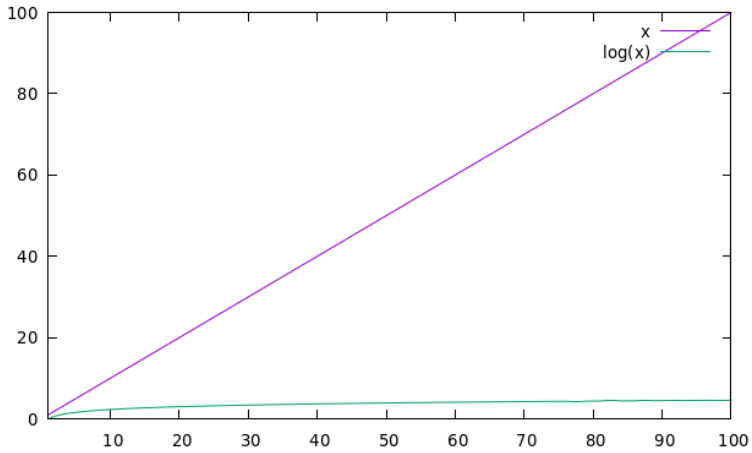
  We only need $\approx \log_2(N)$ comparisons!
  (the logarithm is the "inverse" of the exponential function)

Binary Search - Time Efficiency

- So what happens now when we increase N on the worst case?

  The time needed now grows logarithmically with N

  Note how the logarithm grows much slower than the linear function:



  - If N is $2\times$ bigger, we will only need one more comparison!
  - With 10 comparisons we can go up to $N = 2^{10} = 1\,024$
  - With 20 comparisons we can go up to $N = 2^{20} = 1\,048\,576$
  - With 30 comparisons we can go up to $N = 2^{30} = 1\,073\,741\,824$
  - ...

Binary Search - Time Efficiency

- So what happens now when we increase N on the worst case?

  The time needed now grows logarithmically with N

- Using `test_binary.c` (source) to perform 100 000 queries on an array of size N we get roughly the linear growth we expected:
  - $N = 1\,000$: 0.014s ($\approx$14$\times$ faster than sequential search)
  - $N = 10\,000$: 0.016s ($\approx$118$\times$ faster than sequential search)
  - $N = 100\,000$: 0.020s ($\approx$939$\times$ faster than sequential search)
  - $N = 1\,000\,000$: 0.026s ($\approx$7543$\times$ faster than sequential search)
  - ...

- It is orders of magnitude faster than sequential search and the difference keeps growing as the input increases because fundamentally the grows ratio of the time is much smaller.

Searching for elements - is this all?

- Is this everything that there is to know about searching?
  No! (but it is more than enough for this particular half-semester course)

- Here are some example follow up questions:
  - How much time do we need to sort? (it must be worthwhile to sort once so that afterwards we can use binary search)
  - What if need to add/remove elements from the array? Do we need to sort again? (e.g.: balanced binary search trees)
  - Are there other efficient searching strategies that do not involve some kind of "sorting"? (e.g.: Hash Tables)

There are plenty of interesting algorithms and data structures that you can learn if you choose to continue studying programming and computer science
(e.g.: here we are really just "scratching the surface)