Lecture 6

Algorithms: Introduction

So, what is an algorithm?

Different definitions of an algorithm:

- *an unambiguous specification of how to solve a class of problems. Algorithms can perform calculation, data processing and automated reasoning* **Wikipedia**
- *a set of rules for solving a problem in a finite number of steps, as for finding the greatest common divisor* **Random House**
- *a procedure for solving a mathematical problem in a finite number of steps that frequently involves repetition of an operation* **Merriam-Webster**
- *a process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer* **Oxford**

A stamp showing Persian mathematician Muhammad ibn Musa **Al-Khwarizmi** whose last name was transliterated to ***Algorithmi***

Algorithms?

Algorithm of happiness

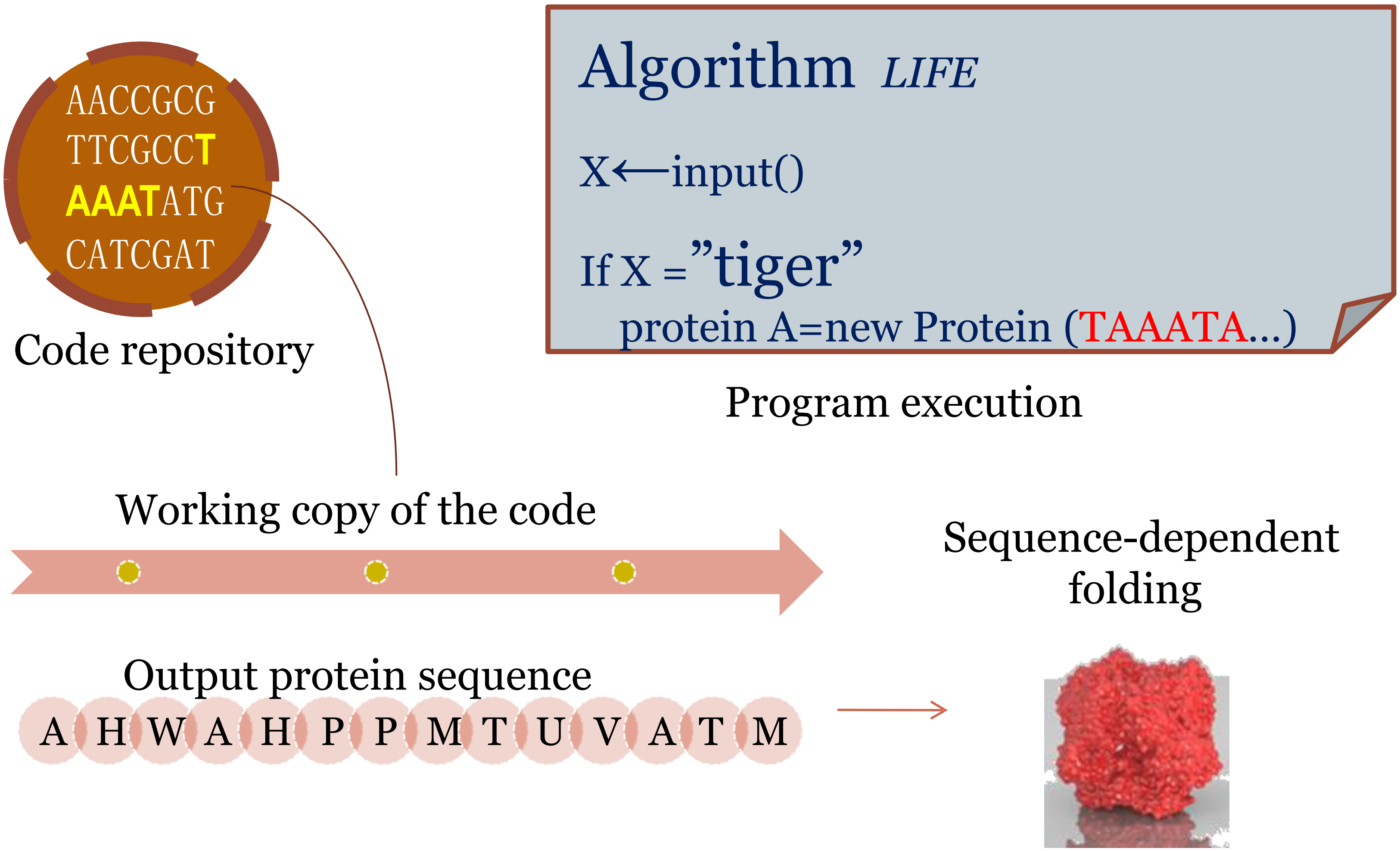
1. Focus upon problem-solving, not just venting
2. Build quality relationships with supportive people
3. Practice gratitude
4. Be kind to yourself, rather than overly self-critical
5. Set meaningful goals
6. Build intrinsic motivation

Algorithm of success in STEM classes

1. Break information into small chunks
2. Intensively concentrate on each chunk for at least 20 minutes
3. Take a break to let new material settle
4. Create a visual metaphor/story about each new concept
5. Solve problems several times until stable connection in your brain is formed

Reference

Algorithms in living systems (on biochemical hardware)



In this course:

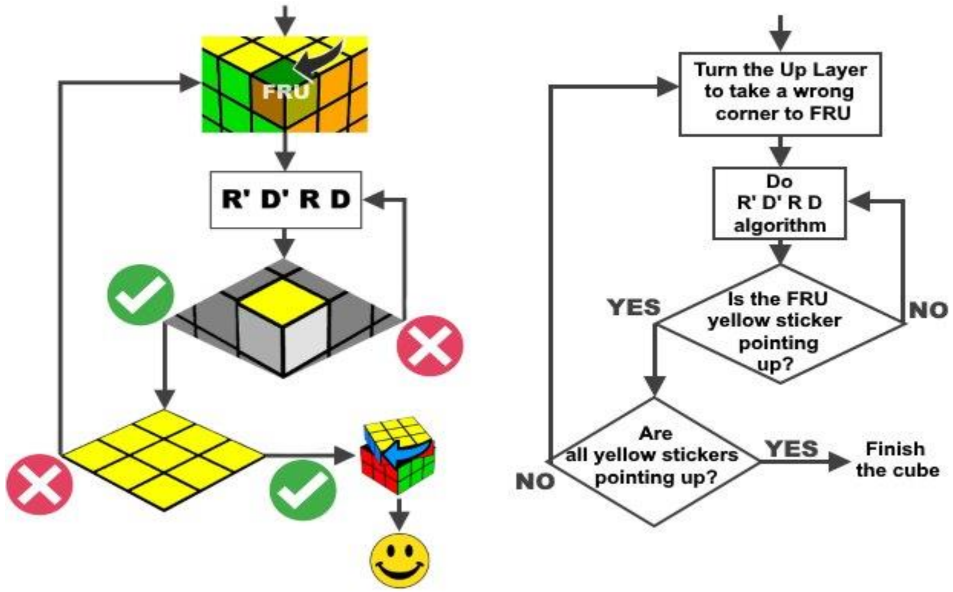
- We limit ourselves to algorithms for performing computational tasks
- The algorithms should be very precise and unambiguous, so they can be communicated to a machine
- **Every computational problem is an algorithmic problem**

Sample problem: compute min value in the array

```
public static <T extends Comparable<T>>
                                T min (T [] input) {
      T result = null;
      for (T entry: input) {
            if (result == null ||
                                entry.compareTo(result) < 0)
                  result = entry;
      }
      return result;
}
```

Why study algorithms?

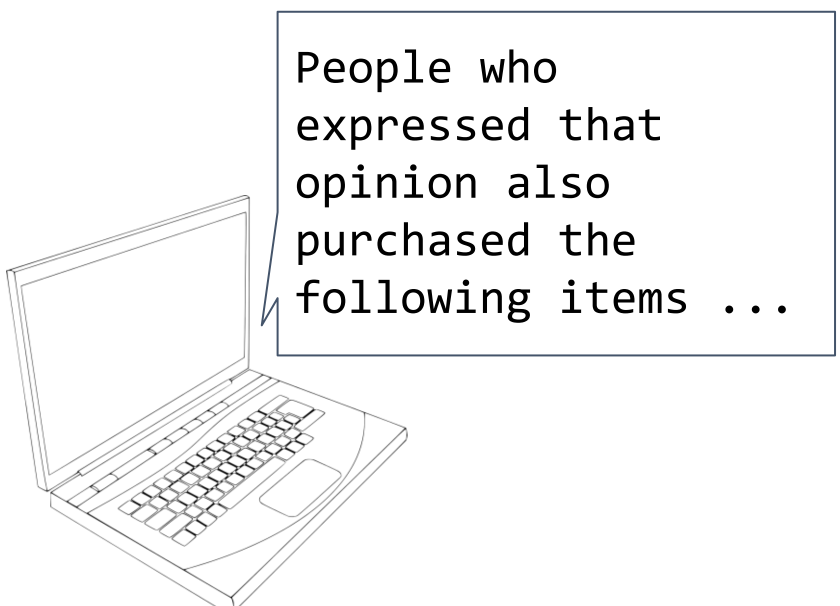
- Mastery of algorithms is required for all branches of Computer Science: Cryptography? Networks? Graphics? Bioinformatics? AI?
- Algorithms play a key role in innovations of modern life
- Challenging yourself is good for brain development
- Fun, addictive activity which can make you a better problem-solver in general

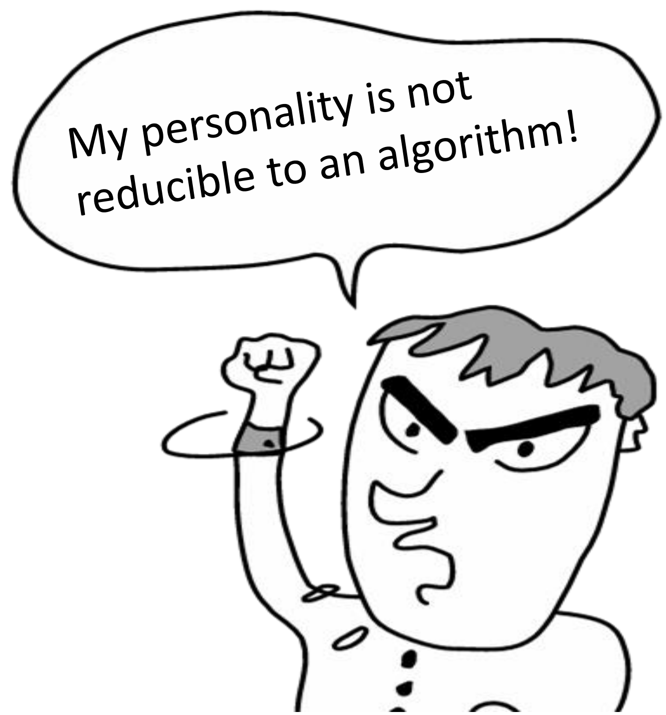


Rubik cube solving algorithm

Algorithms that changed modern world

- Google search: *page-rank*
- Online banking: *concurrent transactions*
- Online payments: *public-key cryptography*
- Reliable communication: *error-correcting codes*
- GPS systems: *shortest paths*





We still need efficient algorithms

Memory and processing power constraints

- Ancient consoles
- Mobile devices
- Browsers

Ever more ambitious tasks

- Big data
- Machine learning/AI
- DNA analysis

What kind of algorithms: Problem vs. problem instance

- Problem instances:
  1. ***What is the position of element 11 in array A={2,10,4,1,3,11,33}?***
  2. ***What is the median of A?***

Median is the middle value in the sorted array

We are interested in solving these

⇩

- **General** algorithmic problems:
  1. **Given an array A of integers, and the target integer *t* find the position of the first occurrence of *t* in *A***
  2. **Given an array of integers *A* find its median**

Developing Algorithms: steps

1. Formalize the problem: input and output
2. Brainstorm solution
3. Express solution: pseudocode
4. Prove correctness (outside the scope of this course)
5. Estimate running time
6. Estimate space requirements

1. Formalizing problem

- Sample problem instance: what is the Greatest Common Divisor (GCD) of 12 and 99?
- Formalized general problem: input and output

Problem: Compute GCD

Input: 2 integers a, b. $b > 1$, $a > 1$, $a > b$
Output: gcd(a, b).

We want it to work on large numbers:
gcd(3918848, 1653264)

Problem instance

2. Brainstorming

GCD: Formal Definition

For integers, *a* and *b*, their ***greatest common divisor*** or ***gcd(a, b)*** is the largest integer *d* s.t. *d* divides both *a* and *b* *(without remainder)*.

Why would we want to compute it:

Put fraction a/b into simplest form.

Need to check remainders of (a/d) (b/d)

d should divide both *a* and *b*.

Want *d* to be as large as possible.

a=45, b=15

both 45 and 15 are divisible by 3, 5,15

we want to find 15

Go over an example

Solution

Problem: Compute GCD

Input: 2 integers a, b. $b > 1$, $a > 1$, $a > b$

Output: $\gcd(a, b)$.

According to the problem and the definition of gcd:

We need to go over integers 1, 2, …

Check if each such integer *d* divides both *a* and *b* without remainder

Keep the largest such number

Stop when $d = min(a,b) = b$

This is algorithm in plain English

a=45, b=15

both 45 and 15 are divisible by 3, 5,15

we want to find 15

Three ways of expressing algorithmic solutions

- English
- Pseudocode
- Program

Increasing precision

Pseudocode: example

```
FOR i from 1 TO 100 DO
    IF i is divisible by 3 AND i is divisible by 5 THEN
        OUTPUT "Both"
    ELSE IF i is divisible by 3 THEN
        OUTPUT "By 3"
    ELSE IF i is divisible by 5 THEN
        OUTPUT "By 5"
    ELSE
        OUTPUT i
```

Pseudocode does not have specific syntax requirements: it just has to be **clear and unambiguous**

Some specifics

- Assignment operator:
  X **:=** 5
  X ← 5 (you can use x=5, but then use == for equality)
- Comparing for equality:
  if x **=** y (you can use x==y)
- *FOR* loops:
  for each element x in sequence:
  for i from 1 to n:
  for i from 1 to n step 2:
  for i from n down to 1:
- *WHILE* loop:
  same as if

Pseudocode does not have specific syntax

But keep in mind the goal:
pseudocode **must be easily translatable into a working program** (in **any** language).

⇩

Avoid language-specific instructions

Pseudocode for GCD

English:

Try every integer from 1 to *b* (*b* < *a* without lost of generality).
If the integer divides both *a* and *b*, remember the best *gcd* so far.
Since the integers we test are increasing,
the algorithm will remember the last – the greatest common divisor for *a* and *b.*

Pseudocode:

```
Algorithm GCD(a, b)

best = 1
for d from 2 to b:
    if (d divides a) and (d divides b):
        best = d
return best
```

Exercise: Develop algorithm for searching in the array

- Formalize the problem: input, output
- Brainstorming?
- Now write the pseudocode

Solution: Pseudocode for search in Array

```
Algorithm find (array A, target)
n: = length of A
for i from 0 to n-1:
     if A[i] = target:
          return i
return -1
```

Developing Algorithms: steps

1. Formalize the problem: input and output
2. Brainstorm solution
3. Express solution: pseudocode
4. Prove correctness (outside the scope of this course)

➢ Estimate running time

1. Estimate space usage

Sample problem

Find the maximum product of two distinct numbers drawn from a sequence of non-negative integers.

My understanding:

Given: A sequence of non-negative integers (each number is either 0 or positive).

Need to find: The maximum value that can be obtained by multiplying two different elements from the sequence (*which by themselves are not necessarily distinct?*).

Ask and clarify!

Go over an example

Given: A sequence of non-negative integers (each number is either 0 or positive).

Need to find: The maximum value that can be obtained by multiplying two different elements from the sequence.

Sample input:								
7	5	14	2	8	8	10	1	2
Sample output: 140								

Sample input:					
7	5	8	8	1	3
Sample output: 64 and not 56					

Formalize the problem

Maximum pairwise product problem

Input: a sequence of n integers $a_0, . . . , a_{n-1}$ | $a_i \geq 0$, $\forall i$ in $[0 \ldots n-1]$

Output: max $(a_i * a_j)$, $0 \leq i \neq j < n$

Brainstorm solution

Maximum pairwise product problem

Input: a sequence of n integers $a_1, \ldots, a_n \mid a_i \geq 0$, $\forall i$ in $[0 \ldots n\text{-}1]$

Output: max $(a_i * a_j)$, $0 \leq i \neq j < n$

The first solution follows directly from the problem definition:

we need to check all pairs of integers in a sequence and find which pair produces the largest product

Solution: pseudocode

Algorithm maxPairwiseProduct1(*A*[0 . . . *n*-1]):

```
product ← 0
for i from 0 to n-1:
  for j from i + 1 to n-1:
    product ← max(product, A[i] · A[j])
return product
```

Step … Think!

Sample input:				
5	6	2	7	4
Sample output: ?				

Maybe there is a better solution?

Another solution

Algorithm maxPairwiseProduct2(A[0 . . . n-1]):

```
index ← 0
for i from 1 to n - 1:
  if A[i] > A[index]:
    index ← i
swap A[index] and A[n - 1]

index ← 0
for i from 1 to n - 2:
  if A[i] > A[index]:
    index ← i
swap A[index] and A[n − 2]

return A[n − 2] · A[n − 1]
```

Which solution is better?

- Should we implement both, run and measure how long does it take for n=100,000 and n = 1,000,000?
- Can we compare them without implementing?
- By analyzing and comparing our algorithms BEFORE implementing them, we can thus avoid implementing algorithms that will require too much time to run
  - A little analysis could save us a lot of programming effort!

Counting instructions

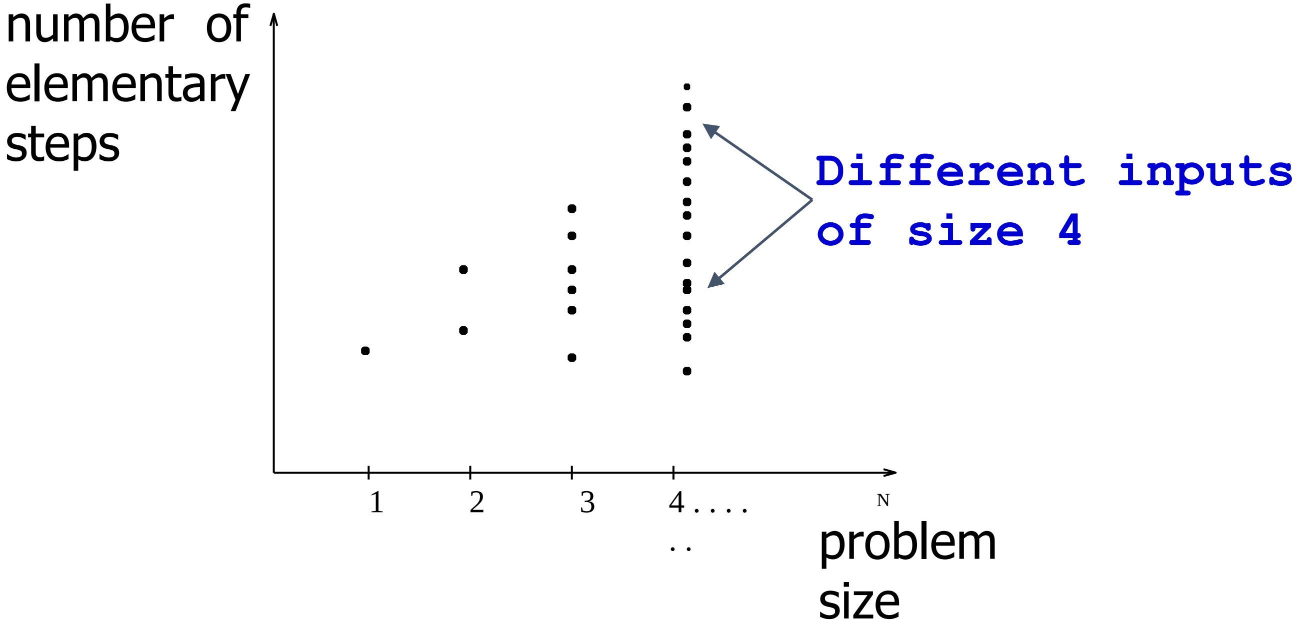
The pseudocode makes it easy to count the total number of steps as it relates to the input size *n* and the nature of the input

```
Algorithm find (array A, target)
n = length of A
for i from 0 to n-1:
      if A[i] == target:
            return i
return -1
```

- It may happen that algorithm finds `target` already on the first iteration: 1 comparison and we are done
- However, it may take *n* comparisons in case that `target` is not in `A`: *n* operations in total

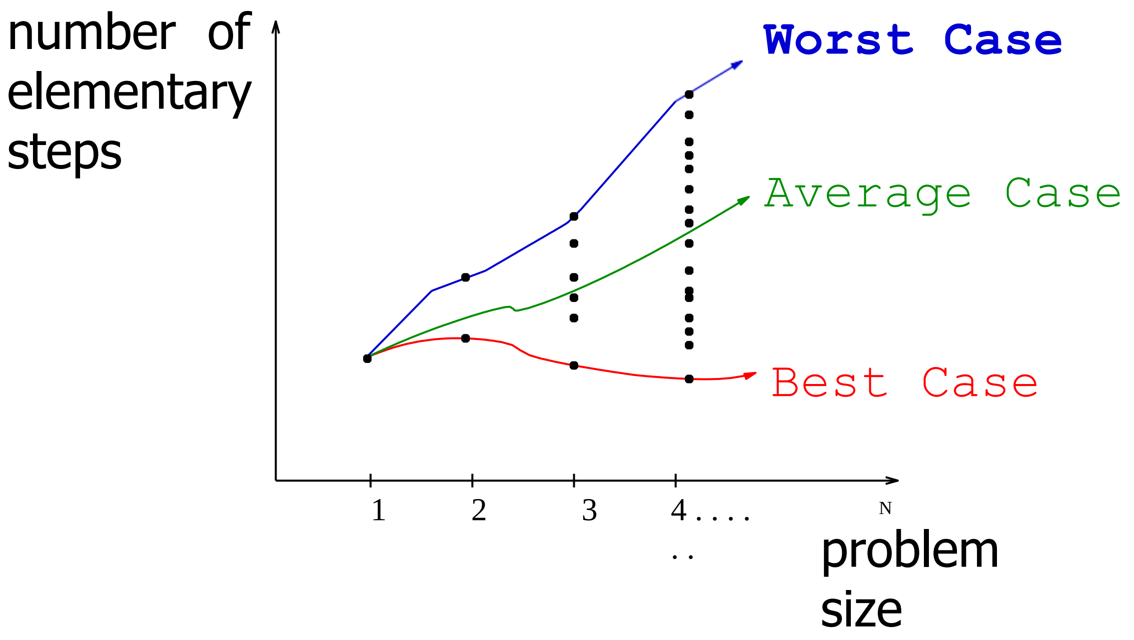
Number of operations vs. input size

- We can count number of steps for a variety of inputs and for different values of *n* and plot the results



Number of steps as function of *n*

- We want to discover function f(n) from the input size *n* to the total number of steps
- We also see that there is the best case and the worst case for each *n*



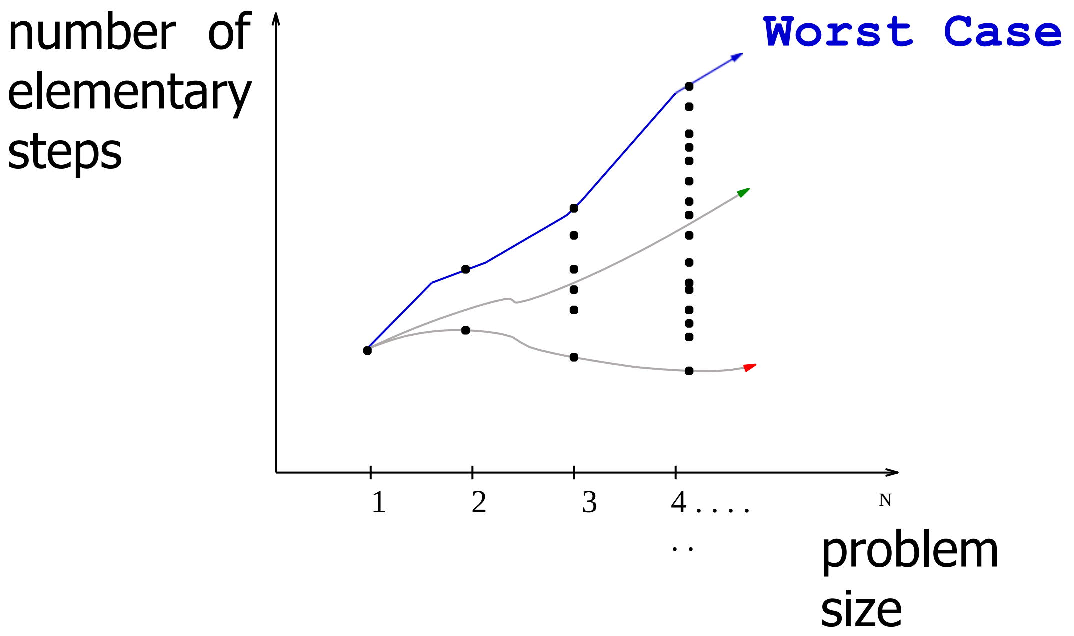
Time complexity

- The best case time complexity of an algorithm is the function defined by the minimum number of steps taken on any instance of size *n*.
- The average-case complexity of the algorithm is the function defined by an average number of steps taken on any instance of size *n*.
- The worst case complexity of an algorithm is the function defined by the maximum number of steps taken on any instance of size *n*.

- Each of these complexities defines a **numerical function**: number of operations vs. size of the input

We are more interested in the worst case

- The nature of the input is generally not known in advance
- We concentrate on the worst-case: we want to know if it is practical to run this algorithm on large inputs of unknown nature



Counting steps: RAM model

The process of counting computer operations is greatly simplified if we accept **the RAM model of computation**:

- Access to each memory element takes a constant time (1 step)
- Each "simple" operation (+, -, =, /, if, call) takes 1 step.
- Loops and function/method calls are *not* simple operations: they depend upon the size of the data and the contents of a subroutine:
    - "sort()" is not a single-step operation
    - "max(list)" is not a single-step operation
    - " if x in list" is not a single-step operation

The RAM model is useful and accurate in the same sense as the **flat-earth model** (which ***is*** useful)!

Loops

The running time of a loop is, at most, the running time of the statements inside the loop (including if tests) multiplied by the total number of iterations.

```
m = 0
for i from 0 to n-1:  # repeat n times:
                      # 2 operations –
                      # increment i, test condition
    m = m + 2    #one assignment
```

Total steps = 1 + 2n + n = 3n +1

Nested loops

Analyze from the inside out.

Total number of operations is the product of the sizes of all the nested loops.

```
for i from 0 to n-1:         # outer loop - 2n times
    for j from 0 to n-1:     # inner loop - 2n times
         k = k+1                        # 1 time
```

Total time = $3\ n \times 2\ n = 6n^2$

Consecutive statements

Add the time complexity of each statement.

```
x = x + 1                    # 1
for i from 0 to n-1:         # 2n times
     m = m+2                 # 1 time

for i from 0 to n-1:         # 2n times
     for j from 0 to n-1:    # 2n times
          k = k+1            # 1 time
```

Total time = $1 + 3n + 2n \times 3n = 6n^2 + 3n + 1$

If-then-else statements

Operations: the test, plus either the then part or the else part: whichever is the largest.

```
if len(t) == 0:                         # test: 1
     return false                       # then part: 1
else:                                   # else part:
     for n from 0 to len(t)-1:          # loop: 2n
          if t[n] == p[n]:              # if: 1 (no else)
               return false
     return true
```

Total time = 1 + (3 n + 1)= 3n + 2

Counting instructions: 1

Algorithm max_pairwise_product1(A**[0 . . .** n**-1]):**

$product \leftarrow$ **0**
for i **from 0 to** $n - 2$**:**
  for j **from** $i + 1$ **to** $n - 1$**:**
    $product \leftarrow$ **max(**$product$, A**[**i**]** ▪ A**[**j**])**
return $product$

Counting instructions: 2

Algorithm max_pairwise_product2(*A*[0 . . . *n*-1]):

```
index ← 0
for i from 1 to n - 1:
  if A[i] > A[index]:
    index ← i
swap A[index] and A[n - 1]

index ← 0
for i from 1 to n - 2:
  if A[i] > A[index]:
    index ← i
swap A[index] and A[n − 2]

return A[n − 2] · A[n − 1]
```