Algorithm design and analysis: theoretical study of how to solve computational problems.

Examples:
- sorting a list of numbers
- finding the shortest route on a map
- scheduling when to work on your assignments for different courses
- answering web search queries

Computational problem: precisely defined set of inputs and for each input the set of acceptable outputs.
Muhammad ibn Musa al-Khwarizmi (c. 780 - 850 AD)
  • Persian astronomer, geographer, mathematician

“On Calculating with Hindu Numerals”
  • a treatise in Arabic in 825
  • translated into Latin in the 12th century: “Agoritmi de Numero Indorum”
  • the word agoritmi comes from the Latinized version of al-Khwarizmi
  • starts with “Dixit Algorizmi” = “Thus spoke al-Khwarizmi”
  • Algorizmi meaning: “calculation methods”
What are algorithms?

Algorithm: a finite set of *unambiguous instructions* for solving a problem.

When is it correct? If on *all* legitimate input it outputs the correct answer in *finite* amount of time.

* A recipe that consists only of a list of ingredients is *not* an algorithm.

* Procedures and algorithms are two different things:

  *Procedure*: set of instructions that help accomplish a task

  *Algorithm*: a specific set of precisely defined instructions to complete a specific task
Course goals

Material:
- classical algorithms
- analysis of algorithms
- design techniques

Skills:
- algorithmic thinking
- problem solving & mathematical skills
- technical writing

- Pseudocode in English
Prerequisites

**CS112**
- time and space efficiency
- data structures
- modularity, reusability
- understanding of how algorithms translate to computer code

**CS131 (MA293)**
- precise language
- proofs

one of CS132 (MA242, EK103), CS235 (MA294), CS237 (MA581, EK381)
- mathematical language, comfort
Course Resources II.

• **Discussions** (labs)
  • problems related to the algorithms we study
  • review of course material
  • hints for the hw

• **Office hours**
  • clarification on lecture, lab problems
  • help with homework

• **Piazza**
  • course announcements, share material
  • ask any major or minor question related to the material
  • logistics questions
  • (check out pinned post for links, oh info, etc.)

• **Online**
  • Algorithms is part of the CS curriculum in most CS programs, there is lots of material available
  • tutorials, animations, YouTube lectures, practice exercises, etc.
How to do well?

• Attend lectures and discussions!
  • go over the material after each class
  • trace/run each algorithm
  • review the TopHat questions

• ask!
  • Piazza: no question is too small or too trivial.
  • go to office hours - multiple if needed

• find a study buddy
  • person with the same learning style as you

• start early with assignments
• solve many problems

• if you are having difficulties, either academic or personal, reach out to us so that we can help.
Why study algorithms?

- Improve problem-solving skills
  By breaking down a complex problem into more manageable steps
- Enhance analytical skills
  Analyze and evaluate the efficiency of your solution
- Enhance programming skills
  Helping you writing more efficient and optimized code.
- Prepare for technical interviews
  Many companies often include algorithmic questions in their hiring process
- It is interesting!
Why study algorithms?

- A *language* to explain a sequence of computational steps, talk about program behavior
- Standard set of algorithms and design techniques
- Feasibility (what can and cannot be done)
- Analyzing correctness and resource usage
- Computation is fundamental to understanding the world
- Gives you a certain way of thinking
- (get a job)
- it’s fun!
What is a good algorithm?

- Efficient in both time and space
- Correct ALL-THE-TIME
- Scalable
- Simple
  - Comprehensive
  - Doesn't need a bunch of resources
  - Reusable
What is a good algorithm?

• Returns correct answer

• Fast
  • Running time: expressed as the number of ‘computational steps’ the algorithm takes to return the correct solution

• simplicity
• modularity
• maintainable
• robust
• user-friendliness
• extensibility
• optimized (e.g. for space)
• compatible with legacy code
• etc.
Example: integer addition — pre-school

Input: integers $a$ and $b$
Output: $a+b$

FingerCounting($a,b$):

\[
i \leftarrow 0
\]
while($i \leq b$):

\[
a \leftarrow a + 1
\]
\[
i \leftarrow i + 1
\]
return $a$

Running time?

i.e. how many computational steps?

ex: what if $b$ is a 3-digit number?

\[
a = 237
\]
\[
b = 999
\]

number of iterations is $b$

if $b = \text{100 digits}$ number then

\[
\text{# of iterations is } \left[ 50^{100} - 1 \right]
\]
Example: integer addition — pre-school

Input: integers $a$ and $b$
Output: $a+b$

FingerCounting($a$, $b$):

\[
i \leftarrow 0 \\
\text{while}(i \leq b): \\
\quad a \leftarrow a + 1 \\
\quad i \leftarrow i + 1 \\
\text{return } a
\]

Running time?

- running time scales with $b$
- if $b$ has 100 digits, it could take as many as $10^{100} - 1$ executions of the While loop!
- (this number is called googol)
- (The Sun will die out in about $10^{27}$ cycles of a typical PCs processor)
Example: integer addition — grade school

Grade school algorithm:
1. write $a$ and $b$ in decimal format.
2. Write one over the other, adding columns right to left, taking the carry with you as you go.

<table>
<thead>
<tr>
<th></th>
<th>$a_1$</th>
<th>$a_2$</th>
<th>$a_3$</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>$a_{m-1}$</th>
<th>$a_m$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$+$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>$b_1$</td>
<td>$b_2$</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Running time?
• suppose a math operation on digits takes one unit of time

$\# \text{ of steps} = \max(m, n)$

The larger number determines the $\# \text{ of steps}$

$\max(\log_{10} a, \log_{10} b) = \# \text{ of digits}$
Example: integer addition — grade school

Grade school algorithm:
1. write $a$ and $b$ in decimal format.
2. Write one over the other, adding columns right to left, taking the carry with you as you go.

<table>
<thead>
<tr>
<th>a_1</th>
<th>a_2</th>
<th>a_3</th>
<th>...</th>
<th>...</th>
<th>...</th>
<th>a_{m-1}</th>
<th>a_m</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>b_1</td>
<td>b_2</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>b_{n-1}</td>
<td>b_n</td>
</tr>
</tbody>
</table>

|       | a | + | b |

Running time?
• suppose a math operation on digits takes one unit of time
• algorithm takes $\sim 3(m+n)$ operations
• total time scales with $\log_{10} a + \log_{10} b$
• exponentially faster than the FingerCounting algorithm $(10^{\log_{10} b})$
Language to describe an algorithm

Algorithm: a finite set of *unambiguous instructions* for solving a problem.

How to *express* an algorithm?
- text in natural language (e.g. English)
- pseudocode
- flow charts
- computer code
Language to describe an algorithm

// Nice comments are part of the description
// Input: two positive integer numbers 'a' and 'b'
// Output: the sum 'a + b'

FingerCounting(a,b):

i ← 0
while(i ≤ b):
    a ← a + 1
    i ← i + 1
return a
BubbleSort

Input: an array A of n integers
Output: array A with the integers in ascending order

Description: While we find any two numbers next to each other that are out of order, we swap them.

example: A = [1,5,3,7,4] → 5 and 3 are out of order
[1,3,5,7,4] → 7 and 4 are out of order
[1,3,5,4,7] → 5 and 4 are out of order
[1,3,4,5,7] → sorted 😊
BubbleSort — developing pseudocode

Input: an array $A$ of $n$ integers
Output: array $A$ with the integers in ascending order

BubbleSort_v1($A$):

While (there are two neighbors out of order)
{
    swap them
}

Return $A$
BubbleSort — developing pseudocode

Input: an array $A$ of $n$ integers
Output: array $A$ with the integers in ascending order

BubbleSort_v1($A$):

While there are two neighbors out of order{
    swap them
}
Return $A$

BubbleSort_v2($A$):

WHILE{there are two neighbors out of order}{
    swap them
}
Return $A$
BubbleSort — developing pseudocode

Input: an array $A$ of $n$ integers
Output: array $A$ with the integers in ascending order

BubbleSort_v1($A$):

While there are two neighbors out of order{
    swap them
}
Return $A$

BubbleSort_v2($A$):

WHILE{there are two neighbors out of order}{
    swap them
}
Return $A$

BubbleSort_v3($A$):

WHILE{there is $i$: $A[i] > A[i + 1]$}{
    swap $A[i]$ and $A[i + 1]$
}
Return $A$

How $i$ changes?
BubbleSort — developing pseudocode

Input: an array A of n integers
Output: array A with the integers in ascending order

BubbleSort_v1(A):
While there are two neighbors out of order{
    swap them
}
Return A

BubbleSort_v2(A):
WHILE{there are two neighbors out of order}{
    swap them
}
Return A

BubbleSort_v3(A):
WHILE{there is i: A[i] > A[i + 1]}{
    swap A[i] and A[i + 1]
}
Return A

BubbleSort_v4(A):
For{i = 0 to n – 1}{
    For{j = i down to 0}{
        If{A[j] > A[j + 1]}{
            swap A[j] and A[j + 1]
        }
    }
}
Return A
BubbleSort — developing pseudocode

Input: an array \( A \) of \( n \) integers
Output: array \( A \) with the integers in ascending order

BubbleSort\(_v1\)(\( A \)):

While there are two neighbors out of order{
    swap them
}
Return \( A \)

BubbleSort\(_v2\)(\( A \)):

WHILE{there are two neighbors out of order}{
    swap them
}
Return \( A \)

BubbleSort\(_v3\)(\( A \)):

WHILE{there is \( i \): \( A[i] > A[i + 1] \)}{
    swap \( A[i] \) and \( A[i + 1] \)
}
Return \( A \)

BubbleSort\(_v4\)(\( A \)):

For\( i = 0 \) to \( n - 1 \)\{
    For\( j = i \) down to \( 0 \)\{
            swap \( A[j] \) and \( A[j + 1] \)
        }  
    }  
}
Return \( A \)

BubbleSort\(_v5\)(\( A \)):

For\( i = 0 \) to \( n - 1 \)\{
    For\( j = i \) down to \( 0 \)\{
        }  
    }  
}
Return \( A \)
Standard items to use in pseudocode

• Explicit inputs and outputs — input $A =$ unsorted array of ints, output: sorted $A$

• Sometimes English is good enough: swap $A[i]$ and $A[i+1]$

• Use control structure commands — WHILE, IF-ELSE, FOR, OR, AND
  • capitalize, use indents, brackets

• use comments

• Precise function calls — specify input, store output in variable
  • used for subroutine or recursive function calls

• be concise
  • make use of known subroutines, ds

• don’t spell out trivial steps

MyAlgo($A$):
/* $A$ is an unsorted array of ints */

$A_{\text{sort}} \leftarrow \text{BubbleSort}(A)$

FOR{$i = 0$ to $n - 1$}

$B \leftarrow \text{SomeOtherAlgo}(A_{\text{sort}})$
/* $B$ is a data structure of type xxx containing $y$ */

$B \leftarrow$ some operation on $B$

some other instruction 1

some other instruction 2

...

}$

Return some variable(s)
**BubbleSort**

**Input:** an array $A$ of $n$ integers

**Output:** array $A$ with the integers in ascending order

BubbleSort$_v4(A)$:

For $i = 0$ to $n - 1$ {
    For $j = i$ down to 0 {
            swap $A[j]$ and $A[j + 1]$
        }
    }
}

Return $A$

Solving the sorting problem is not done yet!

- We need to prove it is correct
  - State precisely when the algorithm is correct!
  - What do we need to prove to verify this statement?
- We need to compute its (asymptotic) running time to gauge efficiency

In your homework assignments a full credit solution will discuss all of these.
BubbleSort

**Input:** an array $A$ of $n$ integers

**Output:** array $A$ with the integers in ascending order

**English description:**

Iterate over the indices of an unsorted array $A$ from lowest to highest index. For every index $i$ compare its value with its right neighbor’s and swap if it is larger. Repeat this step on index $i-1$ down to 0.

BubbleSort_v4($A$):

$$\text{For} \{ i = 0 \text{ to } n - 1 \}$$

$$\text{For} \{ j = i \text{ down to } 0 \}$$


- swap $A[j]$ and $A[j+1]$

Return $A$

\[1 + 2 + 3 + \ldots + n - 1 = \frac{n(n+1)}{2} \text{ steps} \leq O(n^2)\]
Brute force algorithm: check the outcome for every possible combination and pick the best one.

- This works for almost all problems
- Typically takes $2^n$ computational steps or worse for inputs of size $n$.

We need better algorithms!
When is an algorithm efficient?

Good algorithm: works in practice! - what does that mean?

- algorithm is correct
- efficient: runs reasonable fast

What is ‘reasonable fast’?

- note: the running time in seconds depends on the size of the input, the processor, worst and best case data, etc.
  - what is a good notion of ‘running time’?
Running time in terms of the input size $n$

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds $10^{25}$ years, we simply record the algorithm as taking a very long time.

<table>
<thead>
<tr>
<th>$n$</th>
<th>$n$</th>
<th>$n \log_2 n$</th>
<th>$n^2$</th>
<th>$n^3$</th>
<th>$1.5^n$</th>
<th>$2^n$</th>
<th>$n!$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n = 10$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>4 sec</td>
</tr>
<tr>
<td>$n = 30$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>18 min</td>
<td>10$^{25}$ years</td>
</tr>
<tr>
<td>$n = 50$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>11 min</td>
<td>36 years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>12,892 years</td>
<td>10$^{17}$ years</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>1 sec</td>
<td>18 min</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 10,000$</td>
<td>&lt; 1 sec</td>
<td>&lt; 1 sec</td>
<td>2 min</td>
<td>12 days</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 100,000$</td>
<td>&lt; 1 sec</td>
<td>2 sec</td>
<td>3 hours</td>
<td>32 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
<tr>
<td>$n = 1,000,000$</td>
<td>1 sec</td>
<td>20 sec</td>
<td>12 days</td>
<td>31,710 years</td>
<td>very long</td>
<td>very long</td>
<td>very long</td>
</tr>
</tbody>
</table>

image source: Kleinberg, Tardos, ch. 2.1 page 34.
Asymptotic

Goal: analysis that is

• machine independent
• allows comparing how different algorithms scale
source: https://towardsdatascience.com/understanding-time-complexity-with-python-examples-2bda6e8158a7
Running time of an algorithm

Running time: the number of computational steps an algorithm takes on an input of size $n$

- $T(n) =$ number of steps as a function of $n$
- we have to agree on what “one computational step” is - depends on the application
  - one read / write operation
  - one math operation
  - function calls ⇒ consider case by case

Goal: describe the value of $T(n)$ by a mathematical formula
Asymptotic running time

Worst case. Running time guarantee for any input of size $n$.
Ex. BubbleSort requires at most $n^2$ comparisons to sort $n$ elements.

Probabilistic. Expected running time of a randomized algorithm.
Ex. The expected number of compares to quicksort $n$ elements is $\sim 2n \ln n$.

Average-case. Expected running time for a random input of size $n$.
Ex. The expected number of character compares performed by 3-way radix quicksort on $n$ uniformly random strings is $\sim 2n \ln n$.

(Best-case.) fewest possible computational steps on the most favorable input of size $n$.
  • when should we think about this case?
    * We expect most real-life data to be pretty favorable