CS330 Introduction to Analysis of Algorithms

May 24, 2023
by Tiago Januario

• course information
• why algorithms?
• pseudocode
Course Staff and sections

• Instructor:
  • Prof. Tiago Januario

• Teaching Fellow:
  • Islam Faisal
Algorithm design and analysis

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.
Etymology of the word “algorithm”

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.
Course goals

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

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

**Textbook**: Algorithm Design by Kleinberg and Tardos

Additional resources:
Cormen, Leiserson, Rivest, Stein: Algorithms (CLRS)

**Course webpage** [https://cs-people.bu.edu/januario/teaching/cs330/summer2023/index.html](https://cs-people.bu.edu/januario/teaching/cs330/summer2023/index.html)
- Syllabus, schedule, slides

**Piazza** [https://piazza.com/bu/summer2023/cascs330](https://piazza.com/bu/summer2023/cascs330)
- all course communication — questions, announcements
- lab problems, homework assignments, other material

**Gradescope** [https://www.gradescope.com/courses/541943](https://www.gradescope.com/courses/541943) Entry code: ZZ XK2E
- homework submission

**Google Drive**
- slides, lab problems, homework assignments, other material

**TopHat** [https://app.tophat.com/e/677208](https://app.tophat.com/e/677208) join code 677208
- in-class poll questions
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.
Homework Assignments

• workload in this course is heavy
• Weekly assignments to be submitted via Gradescope
• posted Thursday, always due Thursday 11:59 pm
• late assignments are NOT accepted (solutions will be posted the morning after each deadline)
• regrade requests through Gradescope, not email.

Problems are mostly theory. Often involves clever application or modification of some algorithm that we study in class. Solving a problem that requires creativity takes time, make sure that you start early.
Exams and Grading

**Midterm:** June 8th - in class

**Final:** Last day of class - June 29th
- material is cumulative
- it’s your responsibility to attend the final exam, don’t make travel plans before the set date!

**Grading:**
- 5% Class participation, Top Hat, lectures, discussions, Piazza
- 30% Homework
- 30% Midterm
- 35% Final Exam

To pass the course you need to achieve at least 40% on both exams
Collaboration Policy and Academic Conduct

Exam:
• absolutely NO collaboration is allowed.
• closed book, some limited notes (crib sheet)

Homework:
• ask questions and start discussions on Piazza!
• you can collaborate with up to 1 person on each assignment
• you have to write down the solutions by yourself using your own words, including code
• list your collaborators on your assignment (or explicitly write collaborators: none)

Academic Conduct:
• participants must follow the CAS Academic Code of Conduct https://www.bu.edu/academics/policies/academic-conduct-code/
• the work you submit has to be your own (except for your collaborators or input from course staff)
• if you use some resource from outside of class (e.g. web, book) use proper citation
• you are forbidden to explicitly search for the problem solutions online or get them from a person not currently enrolled in this course
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.
Algorithm design and analysis

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

computational problem: precisely defined set of inputs and for each input the set of acceptable outputs.

examples (what are the inputs and outputs?):
- 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
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.

In order to access whether an algorithm is correct we need to
- define precisely what the inputs are
- define what it means that the output is correct (examples?)
- prove that the algorithm returns the correct output
Why study algorithms?
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?
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?
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?

• 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.

\[
\begin{array}{cccccccc}
 & a_1 & a_2 & a_3 & \ldots & \ldots & \ldots & a_{m-1} & a_m \\
+ & b_1 & b_2 & \ldots & \ldots & \ldots & b_{n-1} & b_n \\
\hline
 & a & + & b
\end{array}
\]

Running time?
- suppose a math operation on digits takes one unit of time
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></td>
<td></td>
</tr>
<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>
<td></td>
</tr>
</tbody>
</table>

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 $\left(10^{\log_{10} b}\right)$
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

FingerCounting(a,b):

\( i \leftarrow 0 \)
\( \text{while}(i \leq b): \)

\( a \leftarrow a + 1 \)
\( i \leftarrow i + 1 \)

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]
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
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 $A$
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_v5(A):

For{i = 0 to n - 1}{
    For{j = i down to 0}{
        If{A[j] > A[j + 1]}{
        }
    }
}
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*/
\( A_{-}\text{sort} \leftarrow \) BubbleSort(\( A \))

FOR \( \{i = 0 \) to \( n - 1 \}\}{

  \( B \leftarrow \) 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

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($A$):

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

Return $A$
BubbleSort

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

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.
Solving computational problems

**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></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
Big-O

Big-O Complexity Chart

Operations

Elements

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

**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?