The analysis of Algorithms and Data Structures

Data Structures and Algorithms (094224)

Yuval Emek

Winter 2022/23

Staff

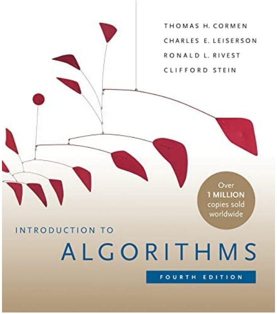
שם	תפקיד	דוא"ל	שעת קבלה	משרד
פרופ'ח יובל עמק	מרצה	yemek@technion.ac.il	יום ד' 08:30-09:30 (בתיאום מראש)	בלומפילד 309
מר יובל גיל	מתרגל אחראי	yuval.gil@campus.technion.ac.il	יום ה' 10:30-11:30	יעודכן בהמשך
גברת נגה הרלב	מתרגלת	snogazur@campus.technion.ac.il	יום ב' 10:30-11:30	קופר 424
מר אורי פרקש	מתרגל	orifa@campus.technion.ac.il	יום ג' 12:30-13:30	ייקבע בזמן אמת
מר אדם גולדבראיך	אחראי תרגילים עיוניים	sgoadam@campus.technion.ac.il	בתיאום מראש	בתיאום מראש
מר ויסאם היגא	אחראי תרגיל תכנותי	hija.wesam@campus.technion.ac.il	בתיאום מראש	בתיאום מראש

Grades

- Final exam: 75%
- Home assignments (5): 15%
- Programming exercise: 10%

Textbook

Introduction to Algorithms, 4th Edition
Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein
MIT Press, 2022



- Older editions
- Hebrew translation (Open U)

Miscellaneous

- Home assignments and programming exercise
  - Teams of 2–3 students
  - Automatic extension of 48 hours
- Email guidelines
- Studying expectations

What is an algorithm?

- Wikipedia.org: *a self-contained step-by-step set of operations to be performed*
- Whatis.com: *a procedure or formula for solving a problem*
- Dictionary.com: *a set of rules for solving a problem in a finite number of steps*
- CLRS: *any well-defined computational procedure that takes some value, or set of values, as input* ⟨קלט⟩ *and produces some value, or set of values, as output* ⟨פלט⟩
  - Algorithms existed long before (digital) computers
- Examples: long division, Gauss elimination (a.k.a. matrix row reduction)
- Word origin: Al-Khwarizmi
  - Persian mathematician and astronomer, 9th century AD
  - Introduced the Hindu-Arabic numeral system to the Europeans

What is a data structure?

- Whatis.com: *a specialized format for organizing and storing data*
- Wikipedia.org: *a particular way of organizing data in a computer so that it can be used efficiently* ⟨יעילות⟩
- Examples: array, linked list
- The algorithm-data structure interdependence:
  - Algorithms use data structures to store and manipulate data
  - Data structures support operations such as `insert`, `find`, and `delete` which are implemented via algorithms

The sorting problem

- The *sorting* ⟨מיון⟩ problem:
  - Input: an array A of n numbers
  - Output: reordering A so that $A[1] \leq A[2] \leq \cdots \leq A[n]$
- E.g., on input $A = [7, 2, 12, 6, 4, 9]$, a sorting algorithm should output $A = [2, 4, 6, 7, 9, 12]$
- An algorithm is *correct* ⟨נכונות⟩ if it halts with a valid output for any given input
  - An incorrect algorithm might halt with a wrong output or not halt at all for some inputs

    מספיק שלפעמים לא נכון כדי שנאמר שאלגוריתם מסוים לא נכון
- How do we measure the quality of a (correct) algorithm?
  - *Running time* ⟨זמן ריצה⟩ (sometimes run-time) — our main focus
  - *Space* ⟨מקום⟩ = memory requirements

A sorting algorithm

- Algorithms are described using *pseudocode*
  - Flexible and forgiving, yet unambiguous
  - Easily converted to C, C++, Java, Python, ...

`Insertion_Sort`(A)

for $j = 2$ to $A.length$ **do**
    $key = A[j]$
    ▷ insert *key* into the sorted sequence $A[1 \ldots j-1]$
    $i = j - 1$
    while $i > 0$ and $A[i] > key$ **do**
        $A[i+1] = A[i]$
        $i = i - 1$
    $A[i+1] = key$

Correctness

- Prove that certain properties hold at key points of execution
  - A.k.a. *invariants* ⟨שמורות⟩
- `Insertion_Sort`:
  At the start of each iteration j of the for loop, the subarray $A[1 \dots j-1]$ consists of the original elements of $A[1 \dots j-1]$, but in sorted order
  - Proof by induction on $j = 2, 3, \dots, n$
    - Requires arguing about the inner while loop

Computational model

- Motto: think like a novice C/C++/Java/Python programmer
- Random access memory (RAM)
  - Ignore memory hierarchies, HDD vs. SSD, etc.
  - Ignore dynamic memory (de)allocation (e.g., garbage collection)
- Instruction set: primitive operations on a fixed number of variables
  - arithmetic operations: $+$, $-$, $\times$, $\div$, $<$, $>$, $\leq$, $\geq$, $=$
  - logical operations: $\wedge$, $\vee$, $\neg$
  - bitwise operations: &, $|$, $\oplus$
  - program control: if, for, while, goto
  - Not supported: average, find-min, sort
    - Why?
- How much time each instruction takes?
  - Depends on the processor model, compiler, battery level
- Abstraction: each instruction takes a constant number of time units
  - Constant c_j for the jth instruction type
- Time unit = millisecond? microsecond? nanosecond?

Worst case behavior

- What exactly do we mean by running time?
- Running time of `Insertion_Sort` varies with the input instances
  - Larger arrays typically require longer time
  - Different arrays of the same length may require different time
- We are interested in the *worst case* ⟨המקרה הגרוע⟩
  - Strong guarantee for the user
  - Alternatives: average case, worst case on almost all input instances
- $T_{\mathcal{A}}(I)$ = running time of algorithm $\mathcal{A}$ on input instance I
- $T_{\mathcal{A}}(n) = \max_{I:|I|=n} T_{\mathcal{A}}(I)$
  - Running time of $\mathcal{A}$ on instances of size n in the worst case
  - Size of input instance =
    - number of items (e.g., if input is an array) or
    - number of bits
  - Referred to as the *running time* ⟨זמן ריצה⟩ of $\mathcal{A}$
    - function!

Asymptotic behavior

- $T_{\mathcal{A}}(I) = \sum_j \#\text{calls to } j\text{th instruction type in } \mathcal{A}(I) \cdot c_j$
  $T_{\mathcal{A}}(n) = \max_{I:|I|=n} T_{\mathcal{A}}(I)$
  - Cumbersome to analyze
- Abstraction: focus on the asymptotic behavior of $T_{\mathcal{A}}(n)$

How fast does $T_{\mathcal{A}}(n)$ grow as $n \to \infty$?

- Ignore low order terms
  - $n^2 + n + \sqrt{n} \approx n^2$
- Ignore constant coefficients
  - $7n \log n \approx n \log n$

Asymptotic analysis — pros and cons

Positive side:

- Low order terms vanish as $n \to \infty$
- Ignoring constant coefficients abstracts away differences between:
  - Processors, compilers, etc.
  - Instruction types
    - Assume that each instruction takes 1 time unit
  - Millisecond, microsecond, etc.
    - Measure time disregard of units
- Simpler analysis: $T_{\mathcal{A}}(I) = \#\text{instruction calls in } \mathcal{A}(I)$
- Easier to compare between algorithms

Negative side:

- In practice, constants matter
  - Run-time $T(n) = n$ is much faster than run-time $T(n) = 1000n$
  - 1 microsecond $\ll$ 1 millisecond
- Low order terms matter for finite values of n
  - $\lg^4 n > \sqrt{n}$ for n as large as $1,000,000,000$

1 Algorithms and data structures

2 A case study: sorting

- Running time
- Asymptotic notation
- Analysis of `Insertion_Sort`

Characterizing the growth rate of functions

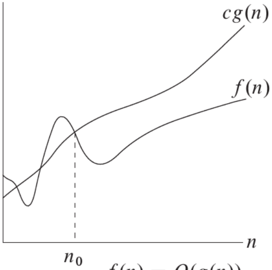
- Goal: compare quality of different algorithms in terms of running time
  - Based on the asymptotic growth rate of the running time functions
- Apply a function classification method commonly known as *asymptotic notation* ⟨סימונים אסימפטוטים⟩ (or Bachmann-Landau notation)
  - Not restricted to running time functions
- The functions we consider:
  - Domain $= \mathbb{Z}_{\geq 0}$
    - Sometimes: $\mathbb{Z}_{>0}$, $\mathbb{R}_{\geq 0}$, $\mathbb{R}_{>0}$
  - Range $= \mathbb{R}$
  - *Asymptotically positive*:
    there exists $n_0 > 0$ such that $f(n) > 0$ for all $n \geq n_0$

Big O notation

Definition (Big O)

Function $f(n)$ belongs to the class $O(g(n))$ if there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.

- Interpreted as $f \leq g$
- Often write $f(n) = O(g(n))$ rather than $f(n) \in O(g(n))$



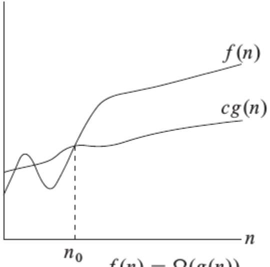
$f(n) = O(g(n))$

Big Ω notation

Definition (Big Ω)

Function $f(n)$ belongs to the class $\Omega(g(n))$ if there exist positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

- Interpreted as $f \geq g$
- Often write $f(n) = \Omega(g(n))$ rather than $f(n) \in \Omega(g(n))$



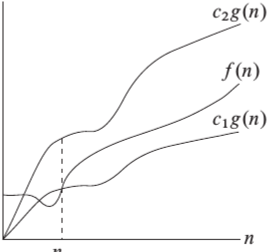
$f(n) = \Omega(g(n))$

Θ notation teta

Definition (Θ)

Function $f(n)$ belongs to the class $\Theta(g(n))$ if there exist positive constants c_1, c_2, and n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0$.

- Interpreted as $f = g$
- Often write $f(n) = \Theta(g(n))$ rather than $f(n) \in \Theta(g(n))$



$f(n) = \Theta(g(n))$

Little o notation

Definition (Little o)

Function $f(n)$ belongs to the class $o(g(n))$ if for any constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0$.

- Interpreted as $f \lneq g$ (strict inequality)
- $f(n) \in o(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
- Often write $f(n) = o(g(n))$ rather than $f(n) \in o(g(n))$

Little ω notation

Definition (Little ω)

Function $f(n)$ belongs to the class $\omega(g(n))$ if for any constant $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0$.

- Interpreted as $f \gneqq g$ (strict inequality)
- $f(n) \in \omega(g(n)) \iff \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$
- Often write $f(n) = \omega(g(n))$ rather than $f(n) \in \omega(g(n))$

Examples

- $3n + 8\sqrt{n} = O(n)$
  - $3n + 8\sqrt{n} \leq 11n$ for all $n \geq 1$
  - Demonstrated by taking $c = 11$ and $n_0 = 1$
- $n \lg(\sqrt{n}) - 5n = \Omega(n \lg n)$
  - Convention: $\lg = \log_2$, $\ln = \log_e$
  - $\lg n \geq 20$ for all $n \geq 2^{20}$
  - $\frac{1}{4} n \lg n \geq 5n$ for all $n \geq 2^{20}$
  - $n \lg(\sqrt{n}) - 5n = \frac{1}{2} n \lg n - 5n \geq \frac{1}{4} n \lg n$ for all $n \geq 2^{20}$
  - Demonstrated by taking $c = 1/4$ and $n_0 = 2^{20}$
- $1,000,000,000,000 = \Theta(1)$ (why?)
- Lesson:
  - By adjusting c (or c_1, c_2), we get rid of constant coefficients
  - By adjusting n_0, we make the low order terms vanish
    - Dominated by even a tiny fraction of the larger term
  - Exactly what we wanted!

Relations

- $f(n) = O(f(n))$, $f(n) = \Omega(f(n))$, $f(n) = \Theta(f(n))$
  $f(n) \neq o(f(n))$, $f(n) \neq \omega(f(n))$
- $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$
- $o(f(n)) \subset O(f(n))$
  $\omega(f(n)) \subset \Omega(f(n))$
- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$
  $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$
  $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$
- If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$
  - Same for Ω, Θ, o, and ω
- There exist functions $f(n)$ and $g(n)$ such that
  $f(n) \neq O(g(n)) \wedge f(n) \neq \Omega(g(n))$
  - Can $f(n) = \Theta(g(n))$? $f(n) = o(g(n))$? $f(n) = \omega(g(n))$?

    לא [illegible] לא לא

Asymptotic notation in longer expressions

- Asymptotic notation often appears as part of longer expressions in equations and inequalities
- Convention:
  - LHS of the equation/inequality: universal quantifier ($\forall$)
  - RHS of the equation/inequality: existential quantifier ($\exists$)
- Consistent with writing $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$
- $O(f(n)) + O(g(n)) = O(f(n) + g(n))$
  $O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$
  - Same for Ω, Θ, o, and ω

1. Algorithms and data structures

2. A case study: sorting
   - Running time
   - Asymptotic notation
   - Analysis of `Insertion_Sort`

The running time of Insertion_Sort

```
Insertion_Sort(A)
1: for j = 2 to A.length do
2:     key = A[j]
3:                     ▷ insert key into the sorted sequence A[1 ... j − 1]
4:     i = j − 1
5:     while i > 0 and A[i] > key do
6:         A[i + 1] = A[i]
7:         i = i − 1
8:     A[i + 1] = key
```

- $T_{\texttt{Insertion_Sort}}(A) = \#$instruction calls in `Insertion_Sort`$(A)$
- Consider arbitrary array A of length $A.length = n$
- Each line by itself contains $O(1)$ instruction calls
  - Takes $O(1)$ time
- Each iteration of the while loop of lines 5–7 takes $O(1)$ time
- In iteration j of the for loop, the while loop makes $\leq j - 1$ iterations
- Iteration j of the for loop takes $\leq (j-1) \cdot O(1) + O(1) = O(j)$ time
- $T_{\texttt{Insertion\_Sort}}(A) \leq \sum_{j=2}^{n} O(j) \leq (n-1) \cdot O(n) = O(n^2)$

The running time of `Insertion_Sort` — cont.

```
Insertion_Sort(A)
for j = 2 to A.length do
    key = A[j]
                    ▷ insert key into the sorted sequence A[1...j − 1]
    i = j − 1
    while i > 0 and A[i] > key do
        A[i + 1] = A[i]
        i = i − 1
    A[i + 1] = key
```

- Holds for any array A of length n, hence $T_{\texttt{Insertion\_Sort}}(n) = \max_{A:A.length=n} T_{\texttt{Insertion\_Sort}}(A) \leq O(n^2)$
- $\Longrightarrow T_{\texttt{Insertion\_Sort}}(n) = O(n^2)$
- Tutorial: if A is ordered in decreasing order, then $T_{\texttt{Insertion\_Sort}}(A) = \Omega(n^2)$
- $\Longrightarrow T_{\texttt{Insertion\_Sort}}(n) = \Theta(n^2)$
- Does it mean that $T_{\texttt{Insertion\_Sort}}(A) = \Theta(n^2)$ for every array A of length n?