算法分析基础
Basics of Algorithm Analysis

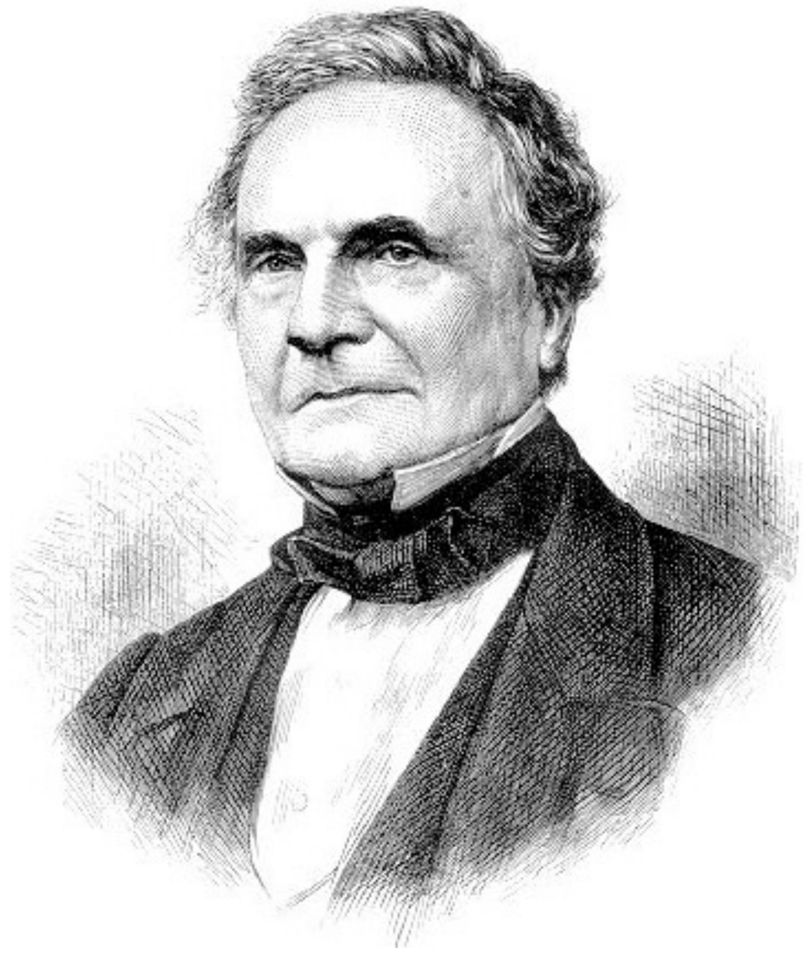
钮鑫涛 王智彬
Nanjing University
2025 Fall

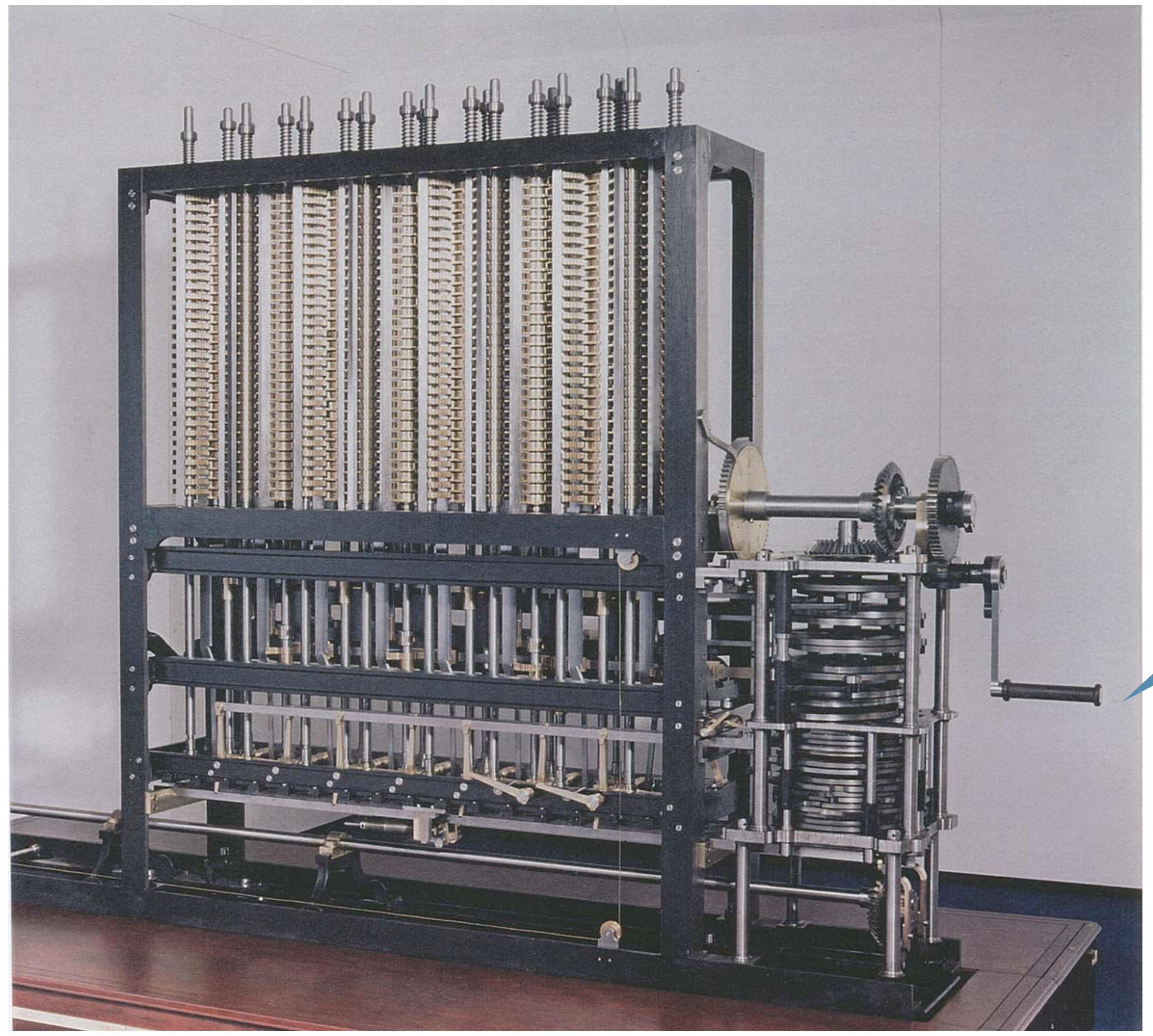
The slides are mainly adapted from the original ones shared by Chaodong Zheng, Kevin Wayne and Hengfeng Wei. Thanks for their supports!

A strikingly modern thought

“ As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise— By what course of calculation can these results be arrived at by the machine in the shortest time? ”

— Charles Babbage (1864)





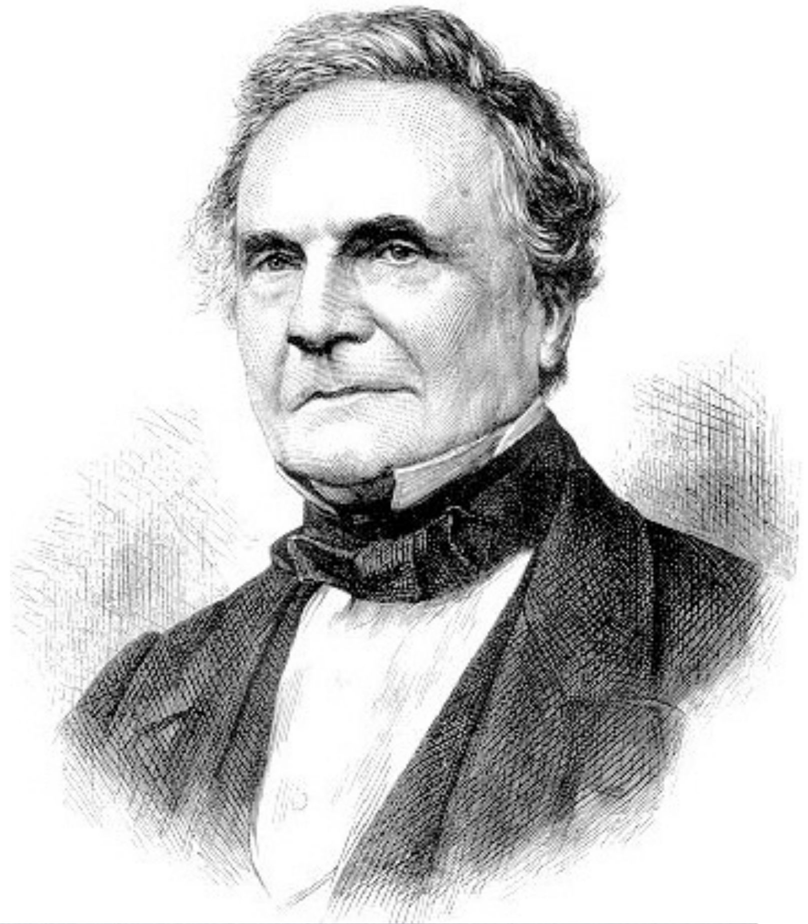
Analytic Engine

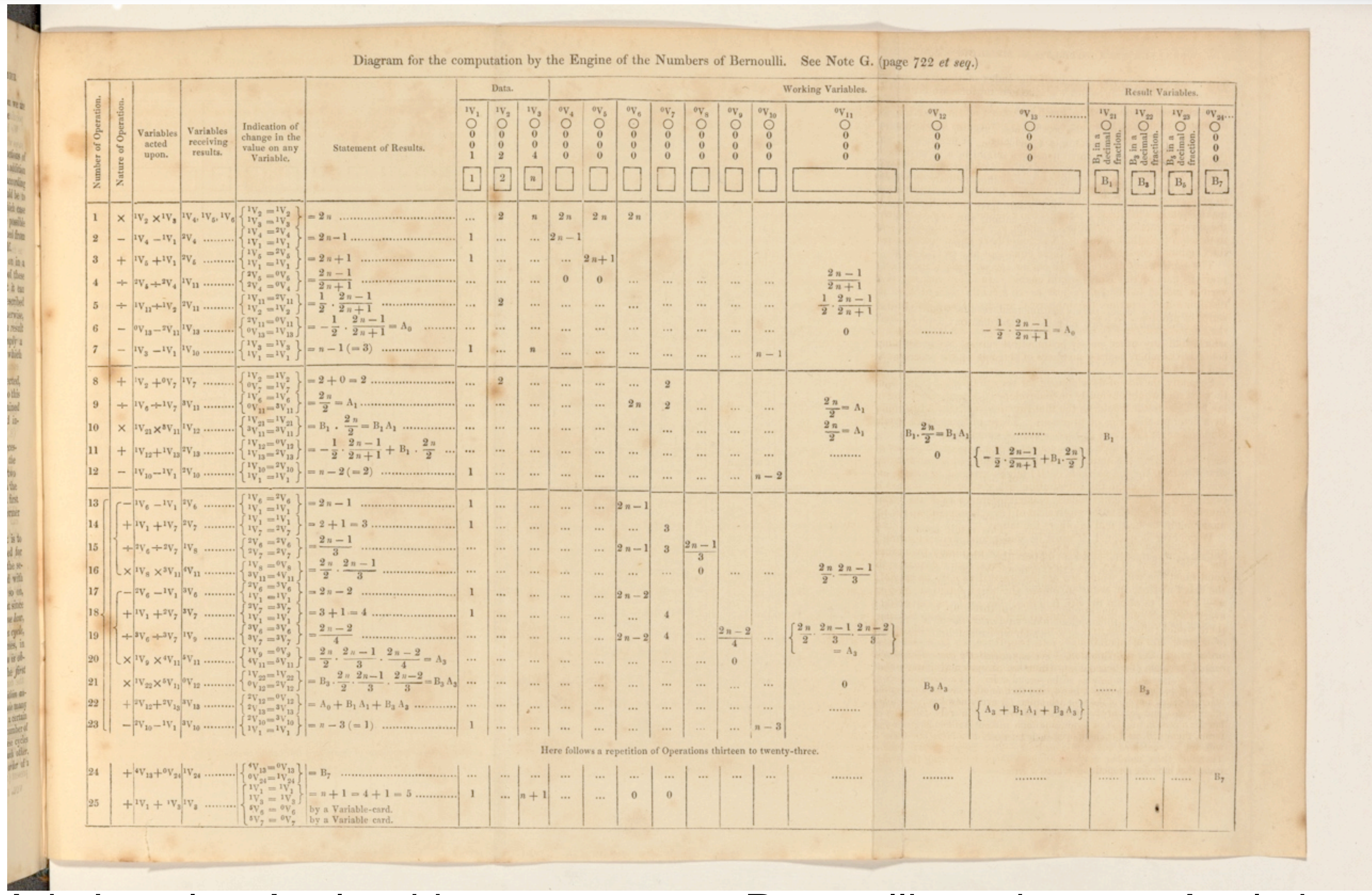
how many times do you have to turn the crank?

A strikingly modern thought

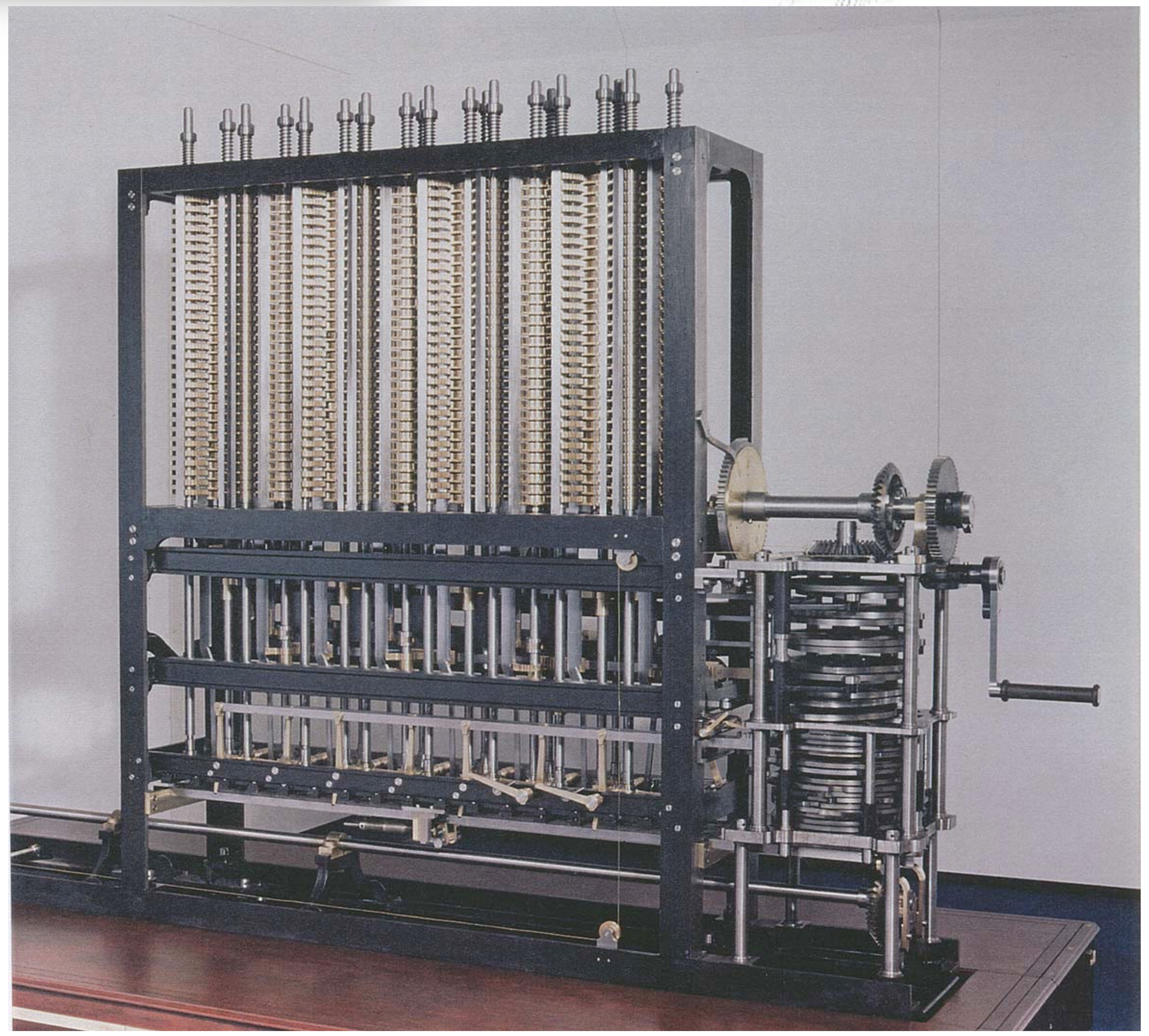
" As soon as an Analytic Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will arise—By what course of calculation can these results be arrived at by the machine in the shortest time? "

— Charles Babbage (1864)





Ada Lovelace's algorithm to compute Bernoulli numbers on Analytic Engine (1843)



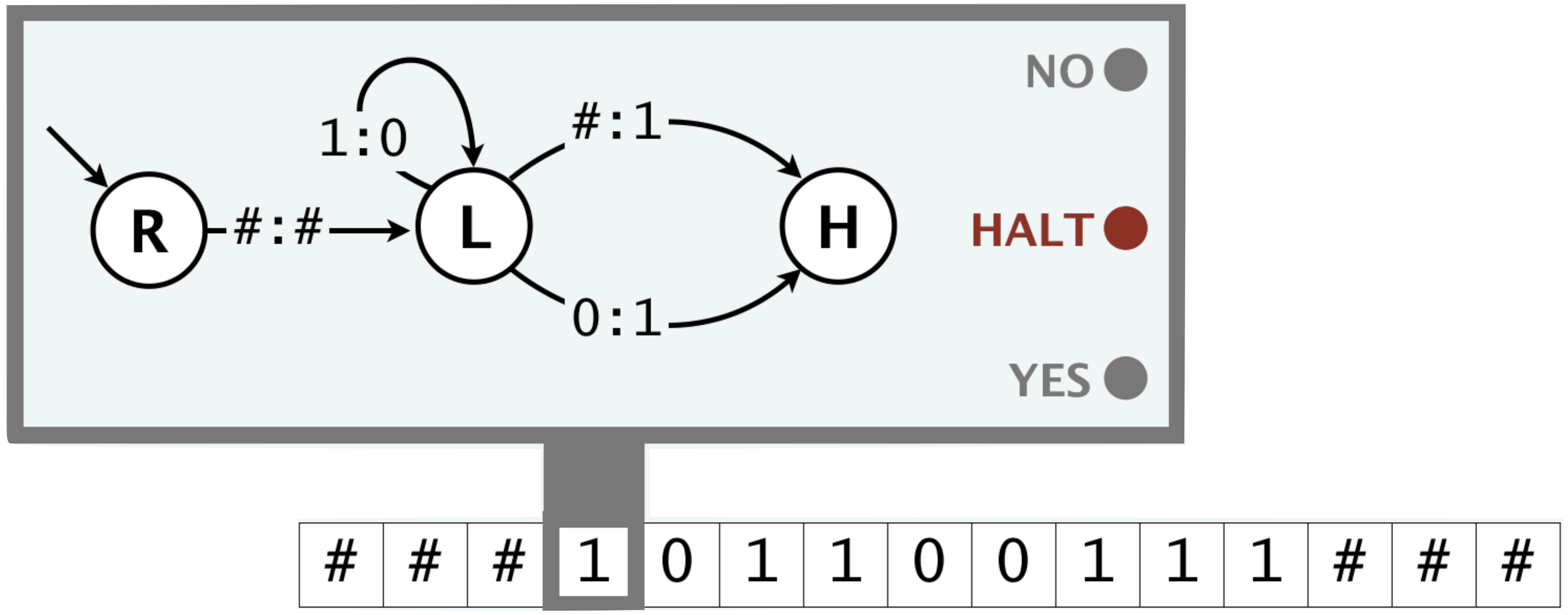
Analytic Engine

Algorithm evaluation

- There are many algorithms to solve one specific problem
  - Which are better?
- Experimental studies?
  - The implementation matters! (Language, Complier, experienced programmer)
  - Even the same implementation, different architecture of the computer makes difference (CPU, memory, operating systems)
- We need an ideal computation model
  - Which is independent of previous factors

Models of computation: Turing machines

- (Deterministic) **Turing machine** — Simple and idealistic model.



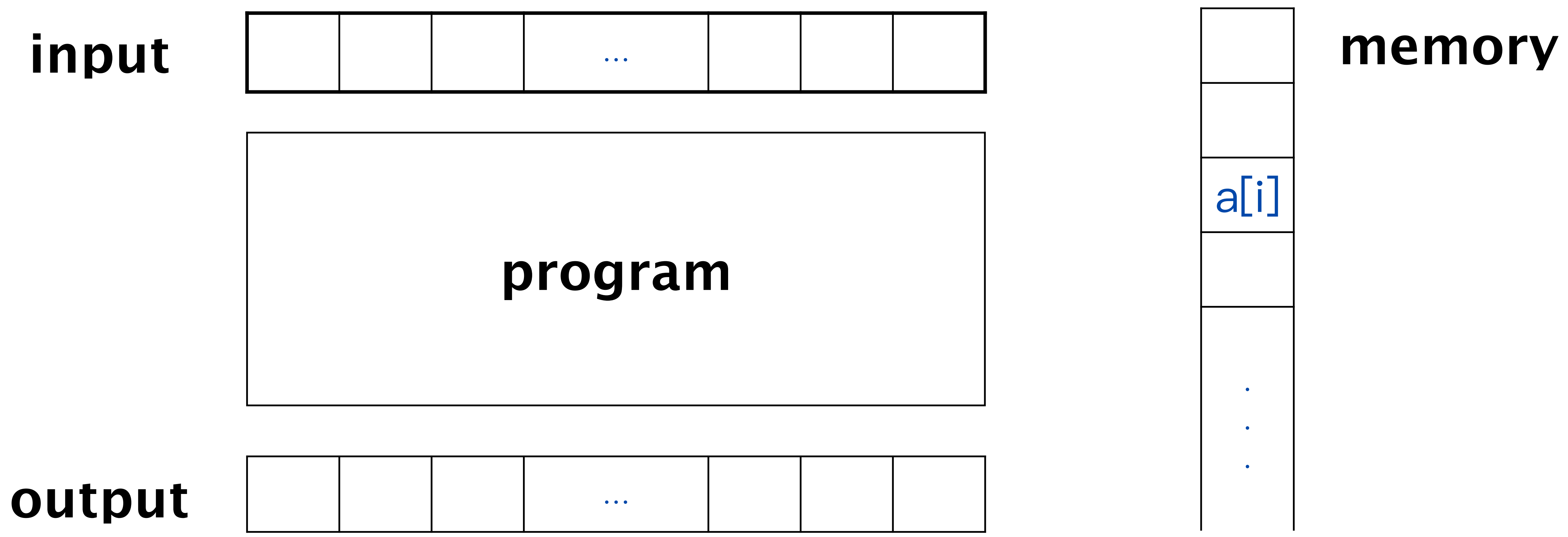
We can evaluate:

- Running time: Number of steps.
- Memory: Number of tape cells utilized.

- **Disadvantage**: No random access of memory.
  - More steps when solving problems than a normal computer

Random-Access-Machine (RAM)

- Random-Access-Machine (RAM, 随机存取机): relatively simple, yet generic and representative.
  - One processor which executes instructions one by one.
  - Memory cells supporting random access, each of limited size.
  - RAM model supports common instructions. Arithmetic, logic, data movement, control, …
  - RAM model supports common data types. Integers, floating point numbers, …
  - RAM model does not support complex instructions or data types (directly). Vector operations, graphs, …



We can evaluate:

- Running time: Number of primitive operations.
- Memory: Number of memory cells utilized.

Correctness of Algorithms

Correctness of algorithms

- When we talk about the correctness of an algorithm, we actually mean the correctness with respect to its specification.
- Specification expresses the task to be done by the algorithm, which consists of:
  - (optional) name of algorithm and list of its arguments
  - Precondition (or initial condition) — it specifies what is correct input data to the problem
  - Postcondition (or final condition) — it specifies what is the desired result of the algorithm)

Correctness of algorithms

- Specification Example:
  - **name**: $Sort(A)$
  - **input**: (pre-condition)
    - An array A of n integers
  - **output**: (post-condition)
    - A permutation of that array A that is sorted (monotonic).

A reordering, yet retaining all of the original elements

Correctness of algorithms

Definition (**Total correctness, 完全正确性**) An algorithm is called totally correct for the given specification if and only if for any **any correct input data** it:

1) **terminates**

2) **returns correct output**

- Correct input data is the data which satisfies the initial condition of the specification.
- Correct output data is the data which satisfies the final condition of the specification.

Correctness of algorithms

- Usually, while checking the correctness of an algorithm it is easier to separately:
  - Check whether the algorithm stops
  - Then checking the remaining part — This remaining part of correctness is called "Partial Correctness" of algorithm

Definition (**Partial correctness,** 部分正确性) An algorithm is partially correct if satisfies the following condition:

If the algorithm receiving correct input data stops then its result is correct

Note: Partial correctness does not make the algorithm stop.

Examples

precondition: $x = 1$
algorithm: $y := x$
postcondition: $y = 1$

Total correctness

precondition: $x = 1$
algorithm: $y := x$
postcondition: $y = 2$

Neither partial nor total correctness

precondition: $x = 1$
algorithm:
    while (true)
        $x := 0$
postcondition: $y = 1$

Partial correctness

Actually, they are Hoare triples!

More details of hoare logic: https://en.wikipedia.org/wiki/Hoare_logic

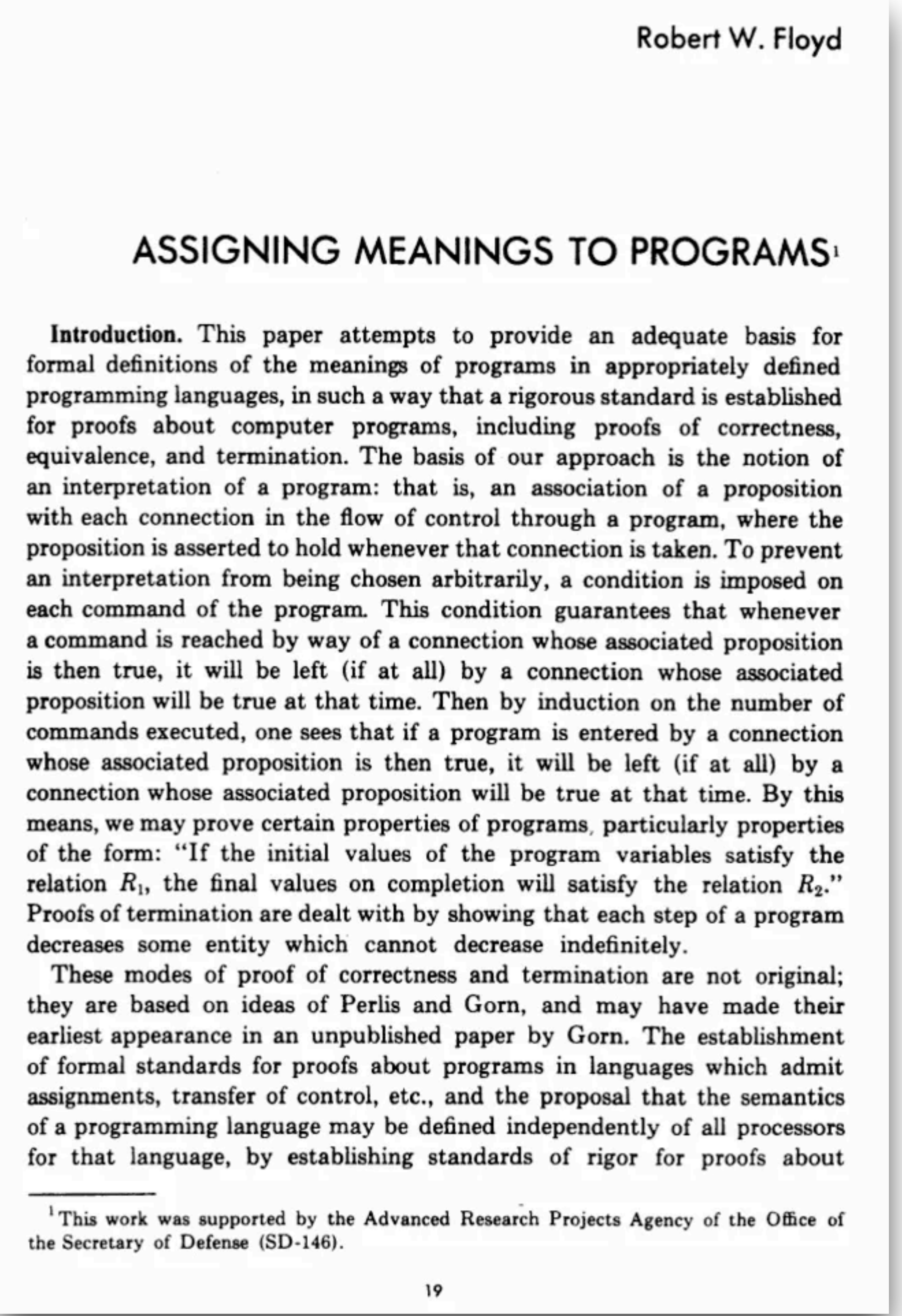
A quick quiz

- What does the implementation have to fulfill if the client violates the precondition (i.e., the algorithm receives incorrect input data)?
  - Nothing. It can do anything at all!
  - Consider the *C* language case: **return** x/0, what will output?
    - For this input, the Compilers are not required to provide any guarantees, which is the so-called "Undefined behavior"!

智能软件与工程学院
School of Intelligent Software and Engineering

Correctness of algorithms

Robert W. Floyd



Robert W. Floyd

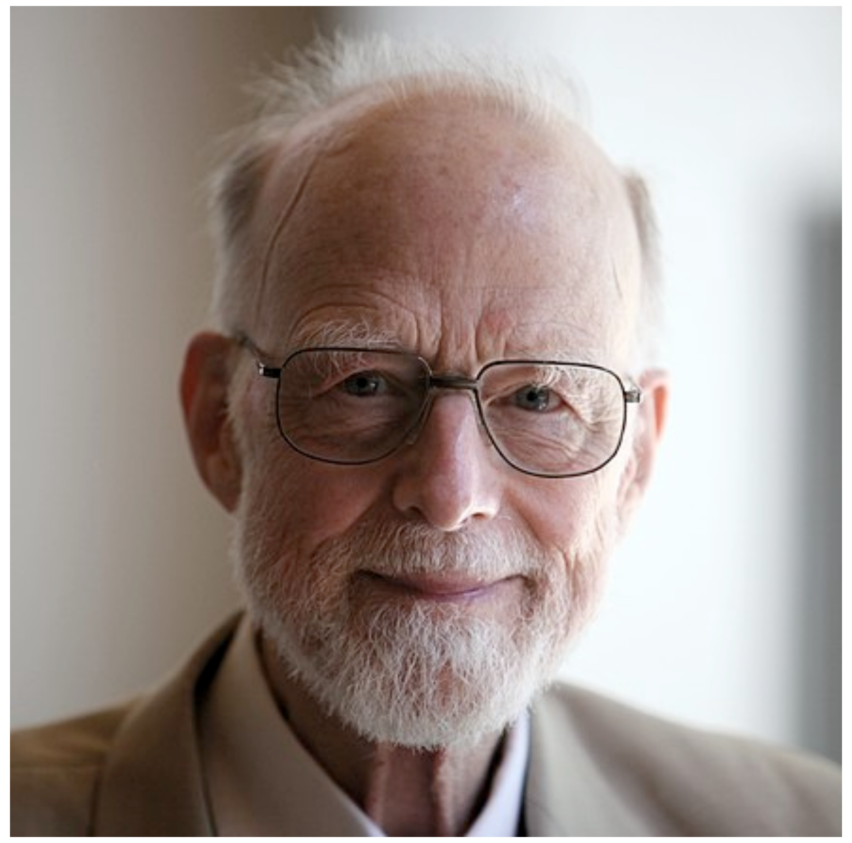
ASSIGNING MEANINGS TO PROGRAMS[1]

Introduction. This paper attempts to provide an adequate basis for formal definitions of the meanings of programs in appropriately defined programming languages, in such a way that a rigorous standard is established for proofs about computer programs, including proofs of correctness, equivalence, and termination. The basis of our approach is the notion of an interpretation of a program: that is, an association of a proposition with each connection in the flow of control through a program, where the proposition is asserted to hold whenever that connection is taken. To prevent an interpretation from being chosen arbitrarily, a condition is imposed on each command of the program. This condition guarantees that whenever a command is reached by way of a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. Then by induction on the number of commands executed, one sees that if a program is entered by a connection whose associated proposition is then true, it will be left (if at all) by a connection whose associated proposition will be true at that time. By this means, we may prove certain properties of programs, particularly properties of the form: "If the initial values of the program variables satisfy the relation R_1, the final values on completion will satisfy the relation R_2." Proofs of termination are dealt with by showing that each step of a program decreases some entity which cannot decrease indefinitely.

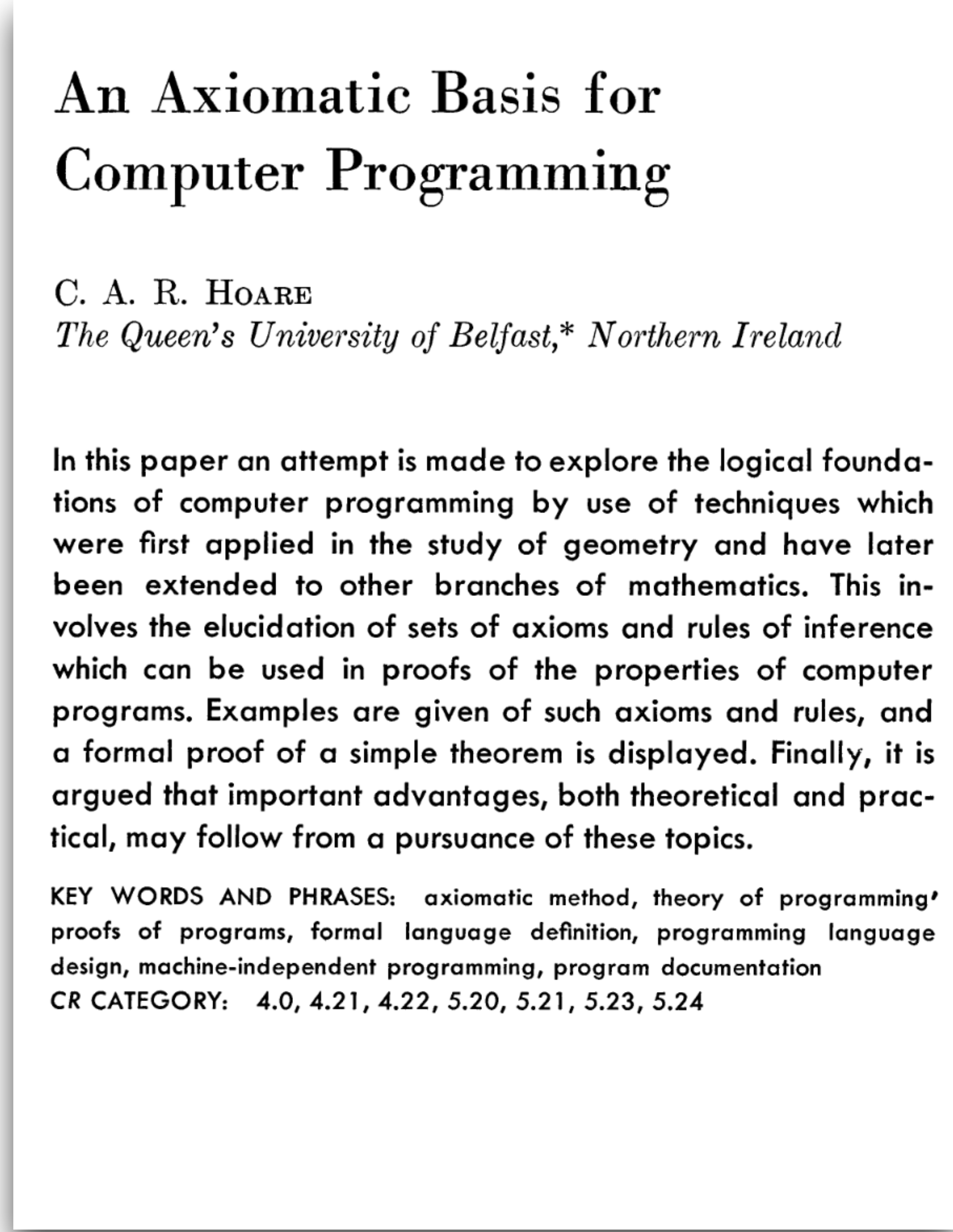
These modes of proof of correctness and termination are not original; they are based on ideas of Perlis and Gorn, and may have made their earliest appearance in an unpublished paper by Gorn. The establishment of formal standards for proofs about programs in languages which admit assignments, transfer of control, etc., and the proposal that the semantics of a programming language may be defined independently of all processors for that language, by establishing standards of rigor for proofs about

[1] This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (SD-146).

19



Tony Hoare



An Axiomatic Basis for Computer Programming

C. A. R. Hoare
The Queen's University of Belfast, Northern Ireland*

In this paper an attempt is made to explore the logical foundations of computer programming by use of techniques which were first applied in the study of geometry and have later been extended to other branches of mathematics. This involves the elucidation of sets of axioms and rules of inference which can be used in proofs of the properties of computer programs. Examples are given of such axioms and rules, and a formal proof of a simple theorem is displayed. Finally, it is argued that important advantages, both theoretical and practical, may follow from a pursuance of these topics.

KEY WORDS AND PHRASES: axiomatic method, theory of programming' proofs of programs, formal language definition, programming language design, machine-independent programming, program documentation
CR CATEGORY: 4.0, 4.21, 4.22, 5.20, 5.21, 5.23, 5.24

Invented mathematical proof systems (a.k.a. Floyd–Hoare logic) to formally prove that programs satisfy their specification

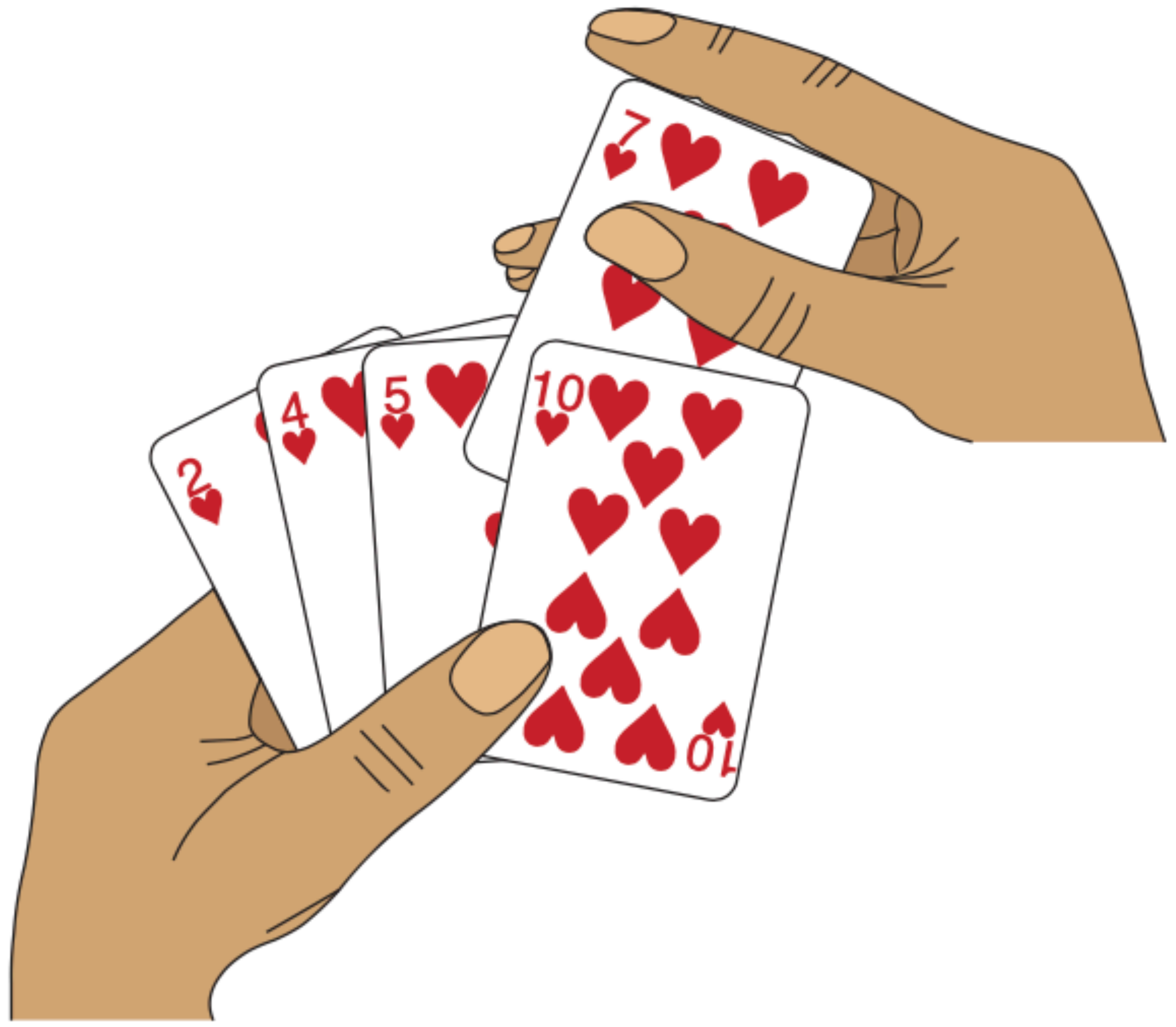
The proof of total correctness

- A proof of total correctness of an algorithm usually assumes 2 separate steps
  - 1. (to prove that) the algorithm always **terminate** for correct input data
  - 2. (to prove that) the algorithm is **partially correct.**
- Different proof methods for them, typically
  - **Variants （变式）** for "termination"
  - **Invariants（不变式）** for "partial correctness"

"Termination" is often much easier to prove

Example: Insertion Sort

Algorithm design strategy 0: wisdom from daily life



Procedure Insertion-Sort(A)

In: An array A of n integers.

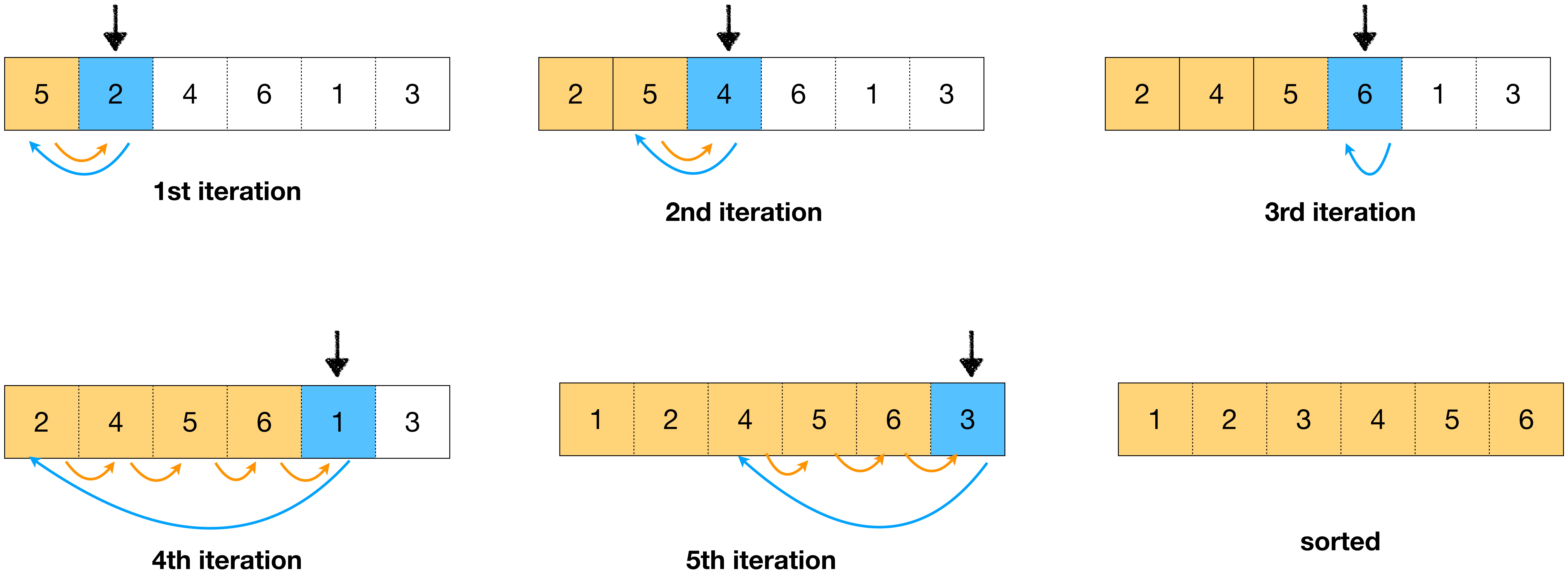
Out: A permutation of that array A that is sorted (monotonic).

for i := 2 **to** $A.length$
  key := $A[i]$
  // Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$
  j := i - 1
  while j > 0 **and** $A[j]$ > key
    $A[j + 1]$:= $A[j]$
    j := j - 1
  $A[j + 1]$:= key
return A

we omit the “end” keyword here to make it simpler

Example: Insertion Sort

- Applies algorithm Insertion-Sort to $[5, 2, 4, 6, 1, 3]$



Example: Insertion Sort

- Proof the correctness of Insertion-Sort
  - Step1: The algorithm outputs correct result on every instance (**partially correct**).
  - Step2: The algorithm terminates within finite steps on every instance (**termination**).

Step1: Using loop invariant for partial correctness

General rules for loop invariant proofs

- Initialization: It is true prior to the first iteration of the loop.
- Maintenance: If it is true before an iteration of the loop, it remains true before the next iteration.
- Termination: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct

Partial correctness of Insertion Sort

- **Loop invariant:** By the end of i^{th} iteration of outer **for** loop, the elements in subarray $A[1,\cdots,i]$ are in sorted order.
- **[Initialization]** prior the first iteration(i = 2): $A[1]$ is in sorted order.
- **[Maintenance]** Assume by the end of the i^{th} iteration, the elements in subarray $A[1,\cdots, i]$ are in sorted order; then by the end of the $(i+1)^{th}$ iteration, the elements in subarray $A[1,\cdots,i+1]$ are in sorted order.
- **[Termination]** After the iteration $i = n$, the loop invariant states that A is sorted

Procedure Insertion-Sort(A)

In: An array A of n integers.

Out: A permutation of that array A that is sorted (monotonic).

```
for i := 2 to A.length
  key := A[i]
  // Insert A[i] into the sorted subarray A[1 : i - 1]
  j := i - 1
  while  j > 0 and A[j] > key
      A[j + 1] := A[j]
      j := j - 1
  A[j + 1] := key
return A
```

Requires another loop invariant for the inner while loop

How to find the loop invariant?

- Is there only one loop invariant?
  - Another loop invariant: By the end of the i^{th} iteration of outer **for** loop, subarray $A[1,\cdots,i]$ retains all of the original elements in $A[1,\cdots,i]$ in previous iteration.
- Let this invariant be IV_2, and the previous invariant be IV_1. What is their relationship?
  - IV_2 is weaker than IV_1, since there are more possible $A[1,\cdots,i]$ that satisfy IV_2, but not satisfy IV_1.
- A good (strong) loop invariant must satisfy these three properties **[Initialization]**, **[Maintenance] and [Termination].** Note that IV_2 does not satisfy **[Termination]** property.

How to find the loop invariant?

- How to find a good loop invariant?
- Generally, the answer is:
  - We don't know
    - For simple ones, e.g., integer ranges, like $0 \leq x \leq 1024$, there exists effective techniques — e.g., abstract interpretation
    - However, for sophisticated invariants, there is no general method, and sometimes we need to provide them manually!
    - Very hot research topic!

Step2: Using loop **variant** for termination

- Wait!!! Program termination is formally undecidable!!
  - It just means that there is no general algorithm exists that solves the halting problem for **all possible** programs.
  - In fact, the partial correctness of all possible programs is also undecidable. — can you prove it?
- Using loop variant to prove the termination
  - show that some quantity **strictly** decreases.
  - it cannot decrease indefinitely (Bounded!)

Well-ordered set

- An ordered set is well-ordered if each and every nonempty subset has a smallest or least element.
    - E.g., every nonempty subset of the non-negative integers has a least element
    - Set of integers and the positive real number are not well-ordered sets
- A well-ordered set has **no infinite descending** sequences, which can be used to ensure the termination of algorithm

Termination of Insertion Sort

- **Loop Variant:** for the inner loop: j
  - For each iteration, j strictly decreases.
  - j is bounded to be larger than 0
- **Loop Variant:** for the outer loop: $A.length$ - i
  - For each iteration, $A.length$ - i strictly decreases.
  - $A.length$ - i is bounded to be larger or equal to 0

Procedure Insertion-Sort(A)

In: An array A of n integers.

Out: A permutation of that array A that is sorted (monotonic).

```
for i := 2 to A.length
  key := A[i]
  // Insert A[i] into the sorted subarray A[1 : i - 1]
  j := i - 1
  while j > 0 and A[j] > key
      A[j + 1] := A[j]
      j := j - 1
  A[j + 1] := key
return A
```

How to find the loop variant

- Again, generally, the answer is:
  - We don’t know
  - But generally speaking, it is very easy to identify!
    - For example, the induction variable of the loop (or some linear transformation of it).

Other strategies of correctness proof

- Some methods and strategies: proof by cases, proof by contraposition, proof by contradiction, etc.
- When loops and/or recursions are involved: often (if not always) use mathematical induction.
- Review your discrete math book if you feel unfamiliar with above terms…
  - [Rosen] Ch.1 (1.7, 1.8) and Ch.5 (5.1, 5.2)

Efficiency of Algorithms

Complexity

- **Time complexity:** how much time is needed before halting
- **Space complexity:** how much memory (usually excluding input) is required for successful executed
- Other performance measures, e.g., communication bandwidth, or energy consumption…
- Time complexity is typically more important than others in analysis.

Complexity

- **Observation**: larger inputs often demands more time.
  - Cost of an algorithm should be a function of ***input size***, say, $T(n)$.
- Given an algorithm and an input, when counting the cost with respect to the RAM model:
  - Each memory access takes constant time.
  - Each **"primitive"** operation takes constant time.
  - Compound operations should be decomposed.
  - At last, Counting up the number of time units.

Time complexity of Insertion Sort

	Cost	Times
Procedure Insertion-Sort(A)		
In: An array A of n integers.		
Out: A permutation of that array A that is sorted (monotonic).		
for i := 2 **to** $A.length$	c_1	n
key := $A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$		
j := i - 1	c_4	$n-1$
while $j > 0$ **and** $A[j] > key$	c_5	$\Sigma_{i=2}^{n} t_i$
$A[j+1]$:= $A[j]$	c_6	$\Sigma_{i=2}^{n}(t_i - 1)$
j := j - 1	c_7	$\Sigma_{i=2}^{n}(t_i - 1)$
$A[j+1]$:= key	c_8	$n-1$
return A		

Check one more time until false

Add them up: $T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1)$

Time complexity of Insertion Sort

- The time cost of insert sort is:

$$T(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} t_i + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7 \sum_{i=2}^{n} (t_i - 1) + c_8(n-1)$$

Depends on **which** input of size n

- The time cost of insert sort varies among inputs
  - How to fairly evaluate a algorithm — enumerate the cost of all the possible inputs? Not possible, since the input space is infinite!
  - We can check the representative inputs, but, what are they?

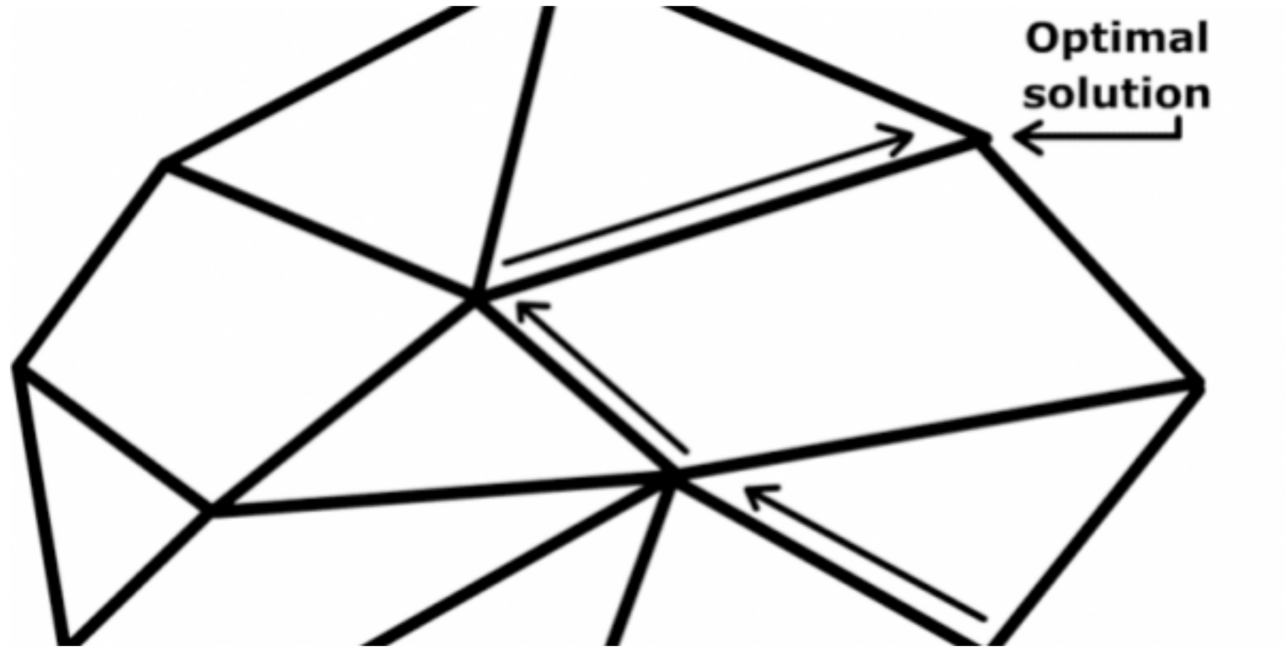
Worst, best, and average

Given one problem and an algorithm, let $\mathcal{X}_n$ be the set of all the possible inputs of size n, and $T(n)$ be the time cost of the algorithm under one input with size n.

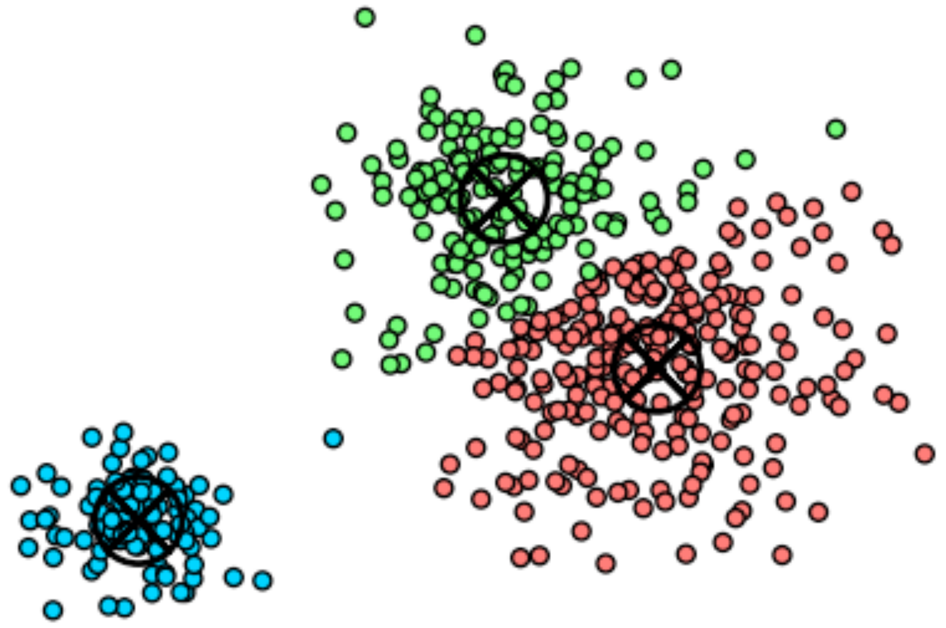
- Worst
  - *W(n)* = maximum time of algorithm on any input of size *n*, i.e., $W(n) = \max_{x \in \mathcal{X}_n} T(x)$
- Best
  - *B(n)* = minimum time of algorithm on any input of size *n*, i.e., $B(n) = \min_{x \in \mathcal{X}_n} T(x)$
- Average
  - *A(n)* = expected time of algorithm over all inputs of size *n*, i.e., $A(n) = \sum_{x \in \mathcal{X}_n} T(x) \cdot Pr(x)$
  - Note: need assumption of statistics distribution of inputs.

Mainly focus on worst-case analysis

- Worst case — Running time guarantee for **any input** of size n.
  - Generally captures efficiency in practice.
  - Draconian view, but hard to find effective alternative.
- **Exceptions.** Some exponential-time algorithms are used widely in practice because the worst-case instances don't arise.



simplex algorithm



k-means algorithm

Time complexity of Insertion Sort

Procedure Insertion-Sort(A)

In: An array A of n integers.

Out: A permutation of that array A that is sorted (monotonic).

	Cost	Times
for i := 2 **to** $A.length$	c_1	n
key := $A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$		
j := i - 1	c_4	$n-1$
while $j > 0$ **and** $A[j] > key$	c_5	$\Sigma_{i=2}^{n} t_i$
$A[j+1]$:= $A[j]$	c_6	$\Sigma_{i=2}^{n}(t_i - 1)$
j := j - 1	c_7	$\Sigma_{i=2}^{n}(t_i - 1)$
$A[j+1]$:= key	c_8	$n-1$
return A		

What is the best case?

Each time t_i is 1, which means that each time the while loop condition is false at the beginning! —> $A[j] > key$ is false every time —> the array is already sorted at the beginning!

Time complexity of Insertion Sort

Procedure Insertion-Sort(A)

In: An array A of n integers.

Out: A permutation of that array A that is sorted (monotonic).

Code	Cost	Times
for i := 2 to $A.length$	c_1	n
key := $A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$		
j := i - 1	c_4	$n-1$
while $j > 0$ and $A[j] > key$	c_5	$\Sigma_{i=2}^{n} t_i$
$A[j + 1]$:= $A[j]$	c_6	$\Sigma_{i=2}^{n}(t_i - 1)$
j := j - 1	c_7	$\Sigma_{i=2}^{n}(t_i - 1)$
$A[j + 1]$:= key	c_8	$n-1$
return A		

$$B(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} 1 \ + c_6 \sum_{i=2}^{n} (1-1) \ + c_7 \sum_{i=2}^{n} (1-1) \ + c_8(n-1) = (c_1 + c_2 + c_4 + c_5 + c_8)n - (c_2 + c_4 + c_5 + c_8)$$

Time complexity of Insertion Sort

Procedure Insertion-Sort(A)

In: An array A of n integers.

Out: A permutation of that array A that is sorted (monotonic).

Code	Cost	Times
for i := 2 **to** $A.length$	c_1	n
key := $A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$		
j := i - 1	c_4	$n-1$
while $j > 0$ **and** $A[j] > key$	c_5	$\Sigma_{i=2}^{n} t_i$
$A[j+1]$:= $A[j]$	c_6	$\Sigma_{i=2}^{n}(t_i - 1)$
j := j - 1	c_7	$\Sigma_{i=2}^{n}(t_i - 1)$
$A[j+1]$:= key	c_8	$n-1$
return A		

What is the worst case?

Each time t_i is the largest it can be, which means that each time the while loop condition is true until j is equal to 0 —> $A[j] > key$ is true every time —> the array is reversely sorted at the beginning! —> $t_i = i$

Time complexity of Insertion Sort

Procedure Insertion-Sort(A)

In: An array A of n integers.

Out: A permutation of that array A that is sorted (monotonic).

	Cost	Times
for i := 2 **to** $A.length$	c_1	n
key := $A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1 : i - 1]$		
j := i - 1	c_4	$n-1$
while j > 0 **and** $A[j]$ > key	c_5	$\Sigma_{i=2}^{n} t_i$
$A[j + 1]$:= $A[j]$	c_6	$\Sigma_{i=2}^{n}(t_i - 1)$
j := j - 1	c_7	$\Sigma_{i=2}^{n}(t_i - 1)$
$A[j + 1]$:= key	c_8	$n-1$
return A		

$$W(n) = c_1 n + c_2(n-1) + c_4(n-1) + c_5 \sum_{i=2}^{n} i \ + c_6 \sum_{i=2}^{n} (i-1) \ + c_7 \sum_{i=2}^{n} (i-1) \ + c_8(n-1)$$

$$= c_1 n + c_2(n-1) + c_4(n-1) + c_5(n+2)(n-1)/2 \ + c_6 n(n-1)/2 \ + c_7 n(n-1)/2 \ + c_8(n-1)$$

$$= \left((c_5 + c_6 + c_7)/2\right) n^2 + \left(c_1 + c_2 + c_4 + c_8 - (c_5 + c_6 + c_7)/2\right) n - (c_2 + c_4 + c_5 + c_8)$$

Time complexity of Insertion Sort

Procedure Insertion-Sort(A)

In: An array A of n integers.

Out: A permutation of that array A that is sorted (monotonic).

Code	Cost	Times
for i := 2 **to** $A.length$	c_1	n
key := $A[i]$	c_2	$n-1$
// Insert $A[i]$ into the sorted subarray $A[1 : i$ - $1]$		
j := i - 1	c_4	$n-1$
while $j > 0$ **and** $A[j] > key$	c_5	$\Sigma_{i=2}^{n} t_i$
$A[j+1] := A[j]$	c_6	$\Sigma_{i=2}^{n}(t_i - 1)$
j := j - 1	c_7	$\Sigma_{i=2}^{n}(t_i - 1)$
$A[j+1] := key$	c_8	$n-1$
return A		

What about the average case? —> the elements in the input array are randomly ordered

Hint: the number of swaps equals the number of inversions!

One more thing

- What the space complexity of insertion sort?

Procedure Insertion-Sort(A)

In: An array A of n integers.

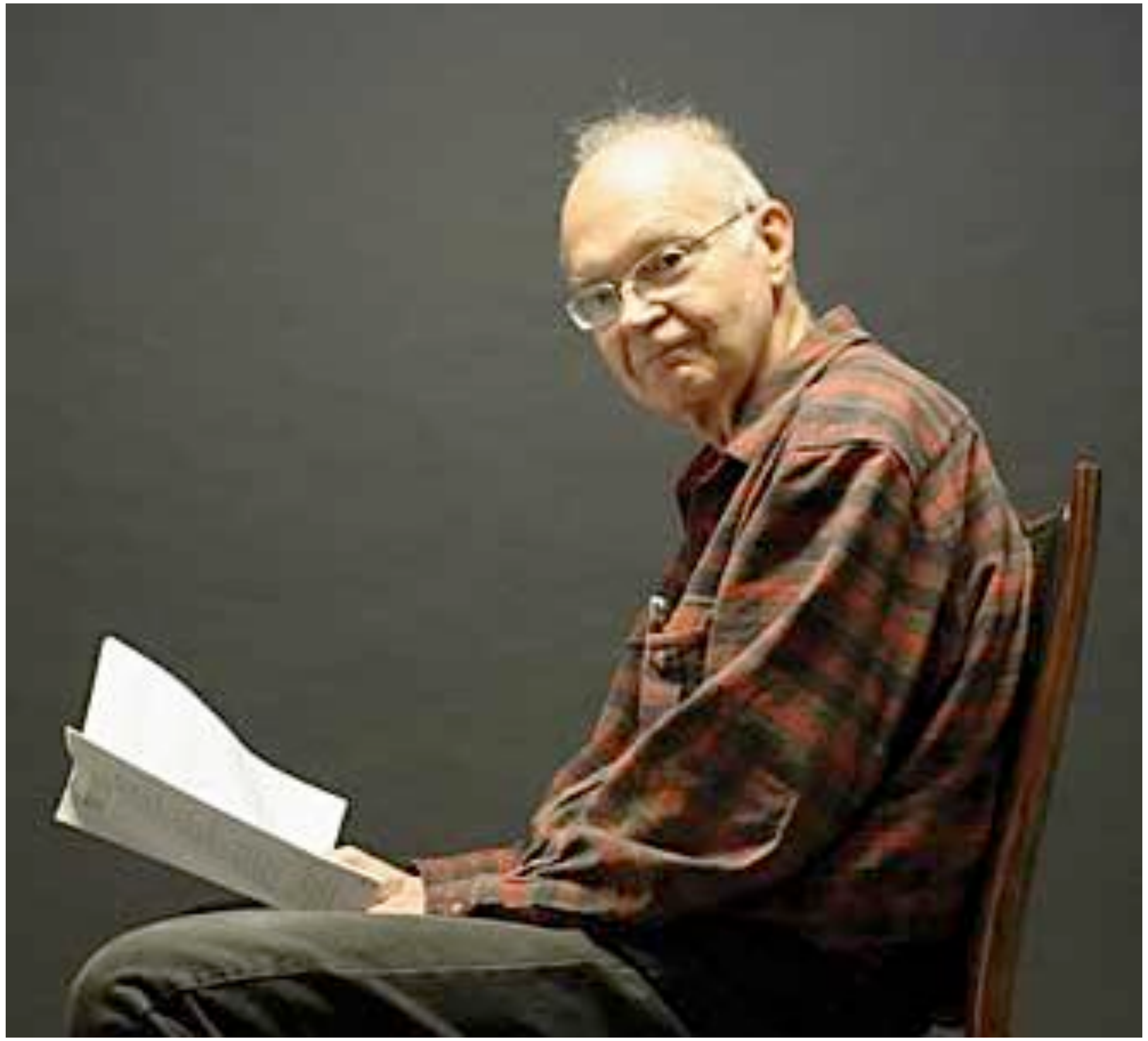
Out: A permutation of that array A that is sorted (monotonic).

```
for i := 2 to A.length
  key := A[i]
  // Insert A[i] into the sorted subarray A[1 : i - 1]
  j := i - 1
  while (j > 0 and A[j] > key)
      A[j + 1] := A[j]
      j := j - 1
  A[j + 1] := key
return A
```

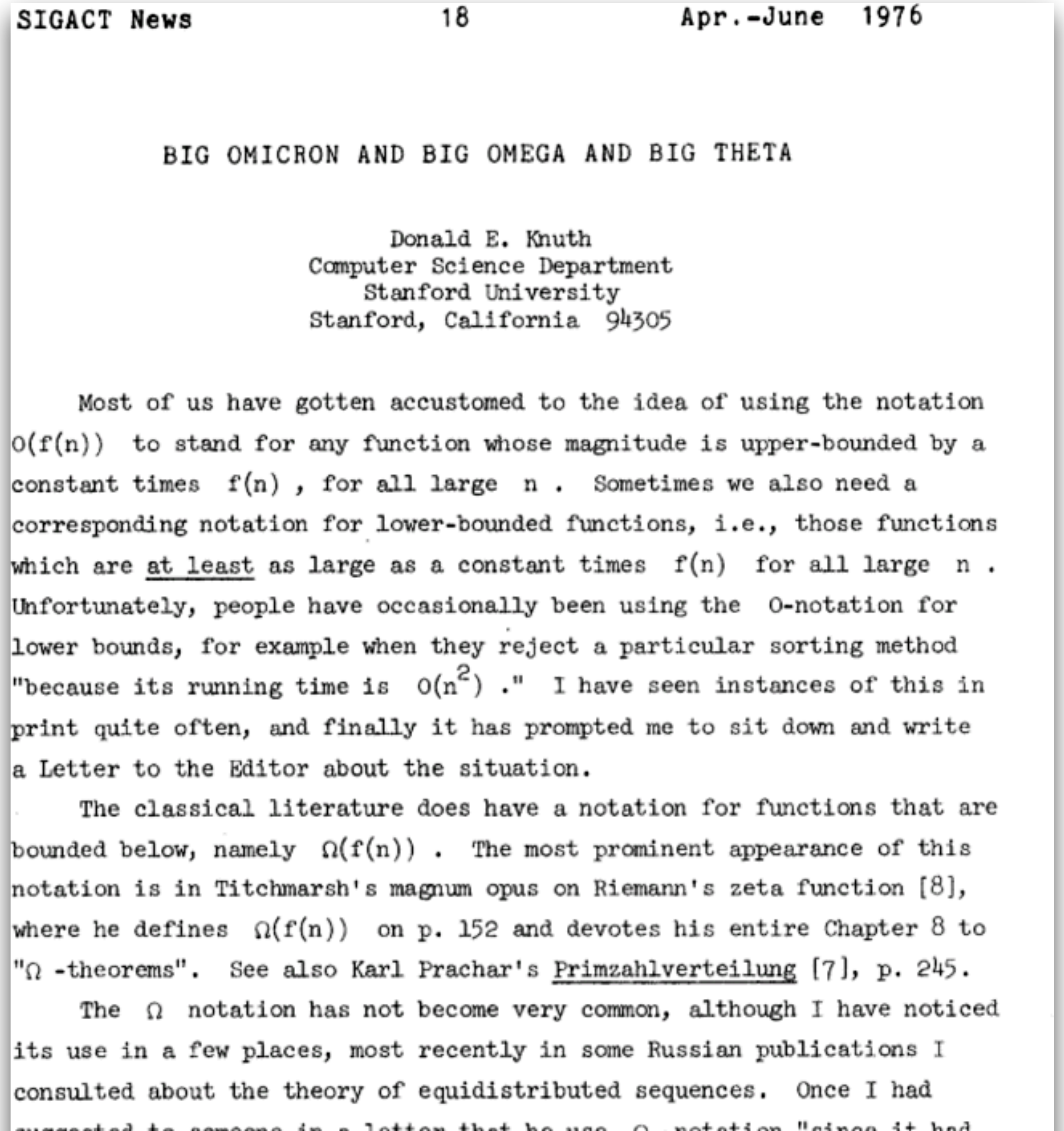
We only need three additional memory cells to store the variable key, i, and j.

Asymptotic order of growth

Asymptotic order of growth



Donald E. Knuth



SIGACT News 18 Apr.-June 1976

BIG OMICRON AND BIG OMEGA AND BIG THETA

Donald E. Knuth
Computer Science Department
Stanford University
Stanford, California 94305

Most of us have gotten accustomed to the idea of using the notation $O(f(n))$ to stand for any function whose magnitude is upper-bounded by a constant times $f(n)$, for all large n. Sometimes we also need a corresponding notation for lower-bounded functions, i.e., those functions which are at least as large as a constant times $f(n)$ for all large n. Unfortunately, people have occasionally been using the O-notation for lower bounds, for example when they reject a particular sorting method "because its running time is $O(n^2)$." I have seen instances of this in print quite often, and finally it has prompted me to sit down and write a Letter to the Editor about the situation.

The classical literature does have a notation for functions that are bounded below, namely $\Omega(f(n))$. The most prominent appearance of this notation is in Titchmarsh's magnum opus on Riemann's zeta function [8], where he defines $\Omega(f(n))$ on p. 152 and devotes his entire Chapter 8 to "Ω-theorems". See also Karl Prachar's Primzahlverteilung [7], p. 245.

The Ω notation has not become very common, although I have noticed its use in a few places, most recently in some Russian publications I consulted about the theory of equidistributed sequences. Once I had

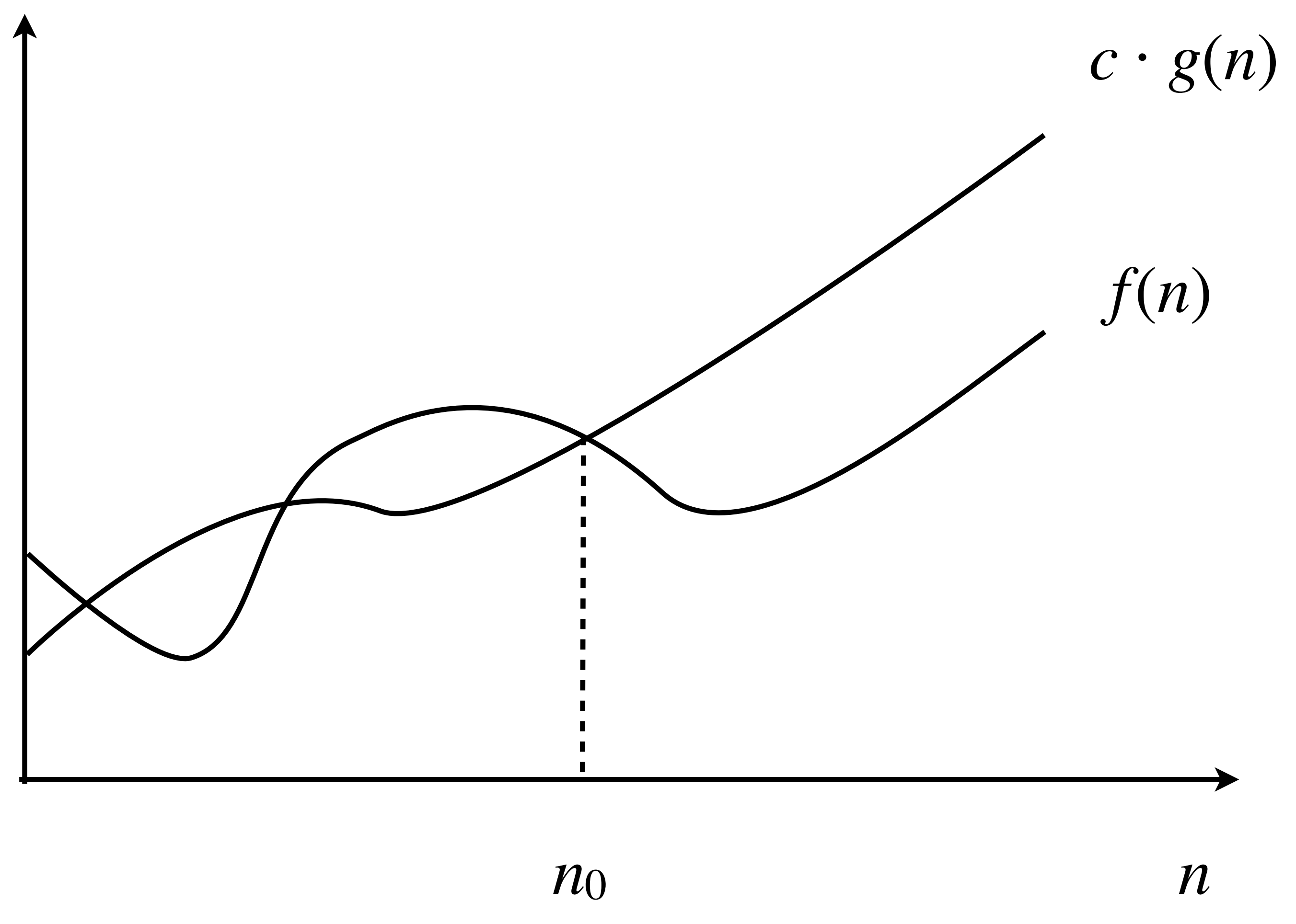
A higher-level abstraction

- In practice, we usually don't care about the unimportant details in the counted operations.
- We need one more simplifying abstraction, which can give us an intuitive feeling of the cost of an algorithm.
    - The abstractions is: the rate of growth, or order of growth, of the running time that really interests us, therefore, two factors are ignored:
        - Constant coefficients are not that important (when n is large)
        - Lower-order terms are not that important (when n is large).

Big O notation

Definition (O) Given a function $g(n)$, we denote by $O(g(n))$ the following set of functions: $O(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) \leq c \cdot g(n)\}$

- **Asymptotic upper bounds（渐近上界）** — when we say $f(n)$ is $O(g(n))$, we mean that $f(n)$ grows *no faster* than a certain rate —> is asymptotically **at most** $g(n)$.



- Ex. $f(n) = 32n^2 + 17n + 1$.
  - $f(n)$ is $O(n^2)$. ← choose $c = 50, n_0 = 1$
  - $f(n)$ is neither $O(n)$ nor $O(n \log n)$ —> why?

Big O notation abuses

- $O(g(n))$ is actually a set of functions, but computer scientists often write $f(n) = O(g(n))$ instead of $f(n) \in O(g(n))$.
- Ex. Consider $f_1(n) = 5n^3$ and $f_2(n) = 3n^2$.
  - We have $f_1(n) = O(n^3)$ and $f_2(n) = O(n^3)$.
  - But, do not conclude $f_1(n) = f_2(n)$.
- Since the worst time complexity of insertion sort is $W(n) = \left((c_5 + c_6 + c_7)/2\right) n^2 + \left(c_1 + c_2 + c_4 + c_8 - (c_5 + c_6 + c_7)/2\right) n - (c_2 + c_4 + c_5 + c_8)$

➡ Therefore, $W(n) = O(n^2)$ —> is asymptotically at most n^2.

Big O notation with multiple variables

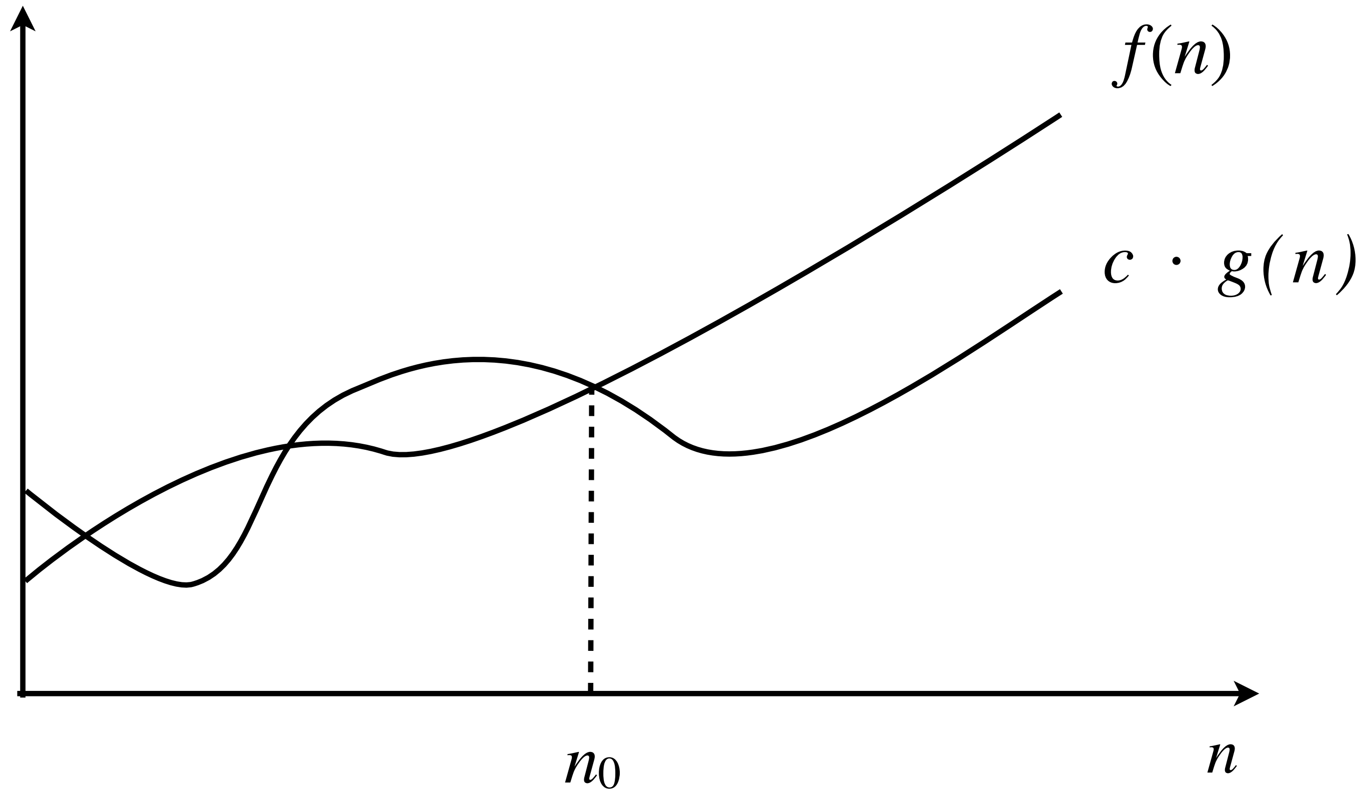
- $f(m, n)$ is $O(g(m, n))$ if there exist constants $c > 0$, $m_0 \geq 0$, and $n_0 \geq 0$ such that $0 \leq f(m, n) \leq c \cdot g(m, n)$ for all $n \geq n_0$ and $m \geq m_0$.

- Ex. $f(m, n) = 32mn^2 + 17mn + 32n^3$.
  - $f(m, n)$ is both $O(mn^2 + n^3)$ and $O(mn^3)$.
  - $f(m, n)$ is neither $O(n^3)$ nor $O(mn^2)$.

Big Ω notation

Definition (Ω) Given a function $g(n)$, we denote by $\Omega(g(n))$ the following set of functions: $\Omega(g(n)) = \{f(n) \mid \exists c > 0, \exists n_0 > 0, \forall n \geq n_0 : f(n) \geq c \cdot g(n)\}$

- **Asymptotic lower bounds（渐近下界）** — when we say $f(n)$ is $\Omega(g(n))$, we mean that $f(n)$ grows at least as fast as a certain rate —> is asymptotically **at least** $g(n)$.

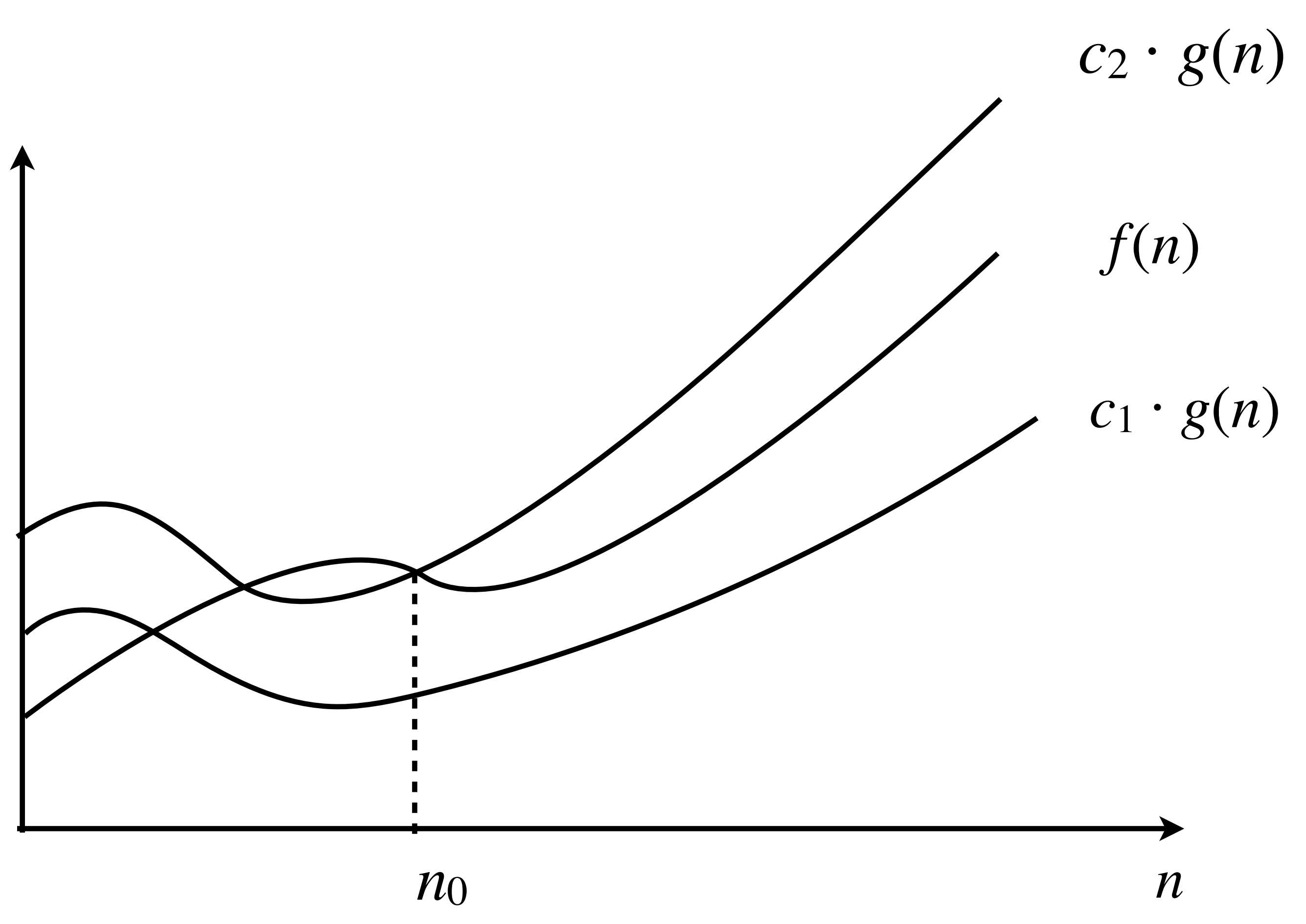


- Ex. $f(n) = 32n^2 + 17n + 1$.
  - $f(n)$ is both $\Omega(n^2)$ and $\Omega(n)$. ⟵ choose $c = 32, n_0 = 1$
  - $f(n)$ is not $\Omega(n^3)$.

Big Θ notation

Definition (Θ) Given a function $g(n)$, we denote by $\Theta(g(n))$ the following set of functions:
$$\Theta(g(n)) = \{f(n) \mid \exists c_1 > 0, \exists c_2 > 0, \exists n_0 > 0, \forall n \geq n_0 : c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

- **Asymptotic tight bounds（渐近紧确界）**
  When we say $f(n)$ is $\Theta(g(n))$, we mean that $f(n)$ grows *precisely* at a certain rate —> it is asymptotically equal to $g(n)$



- Ex. $f(n) = 32n^2 + 17n + 1$.
  - $f(n)$ is $\Theta(n^2)$. ⟵ choose $c_1 = 32, c_2 = 50, n_0 = 1$
  - $f(n)$ is neither $\Theta(n)$ nor $\Theta(n^3)$.

Q: The worst time complexity of Insertion Sort is Θ(n²)?

Small o and ω notation

- $f(n)$ is asymptotically (strictly) smaller than $g(n)$:

Definition (o) Given a function $g(n)$, we denote by $o(g(n))$ the following set of functions: $o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : 0 \leq f(n) < c \cdot g(n)\}$

- $f(n)$ is asymptotically (strictly) larger than $g(n)$:

Definition (ω) Given a function $g(n)$, we denote by $\omega(g(n))$ the following set of functions: $\omega(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 > 0, \forall n \geq n_0 : f(n) > c \cdot g(n)\}$

Q: Now that we have O, Ω, Θ and o,ω, do we have small θ?

Some properties of asymptotic notations

- Reflexivity
  - E.g., $f(n) \in O(f(n))$; but $f(n) \notin o(f(n))$.
- Transitivity
  - E.g., if $f(n) \in O(g(n))$ and $g(n) \in O(h(n))$, then $f(n) \in O(h(n))$.
- Symmetry
  - $f(n) \in \Theta(g(n))$ iff $g(n) \in \Theta(f(n))$.
- Transpose symmetry:
  - E.g., $f(n) \in O(g(n))$ iff $g(n) \in \Omega(f(n))$.

Asymptotic bounds and limits

- If cost functions are complex, it is hard to apply the definitions to get its asymptotic bounds.
- In this case, it usually easier to apply limit method.

Asymptotic bounds and limits

- Proposition. If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = c$ for some constant $0 < c < \infty$ then $f(n)$ is $\Theta(g(n))$.
- Pf.
  - By definition of the limit, for any $\epsilon > 0$, there exists n_0 such that
  - $c - \epsilon \leq \frac{f(n)}{g(n)} \leq c + \epsilon$ for all $n \geq n_0$.
  - Choose $\epsilon = \frac{c}{2} > 0$.
  - Multiplying by $g(n)$ yields $1/2\, c \cdot g(n) \leq f(n) \leq 3/2\, c \cdot g(n)$ for all $n \geq n_0$.
  - Thus, $f(n)$ is $\Theta(g(n))$ by definition, with $c_1 = 1/2\, c$ and $c_2 = 3/2\, c$. ▪

Asymptotic bounds for some common functions

a.k.a. $o(g(n))$

- Proposition. If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$, then $f(n)$ is $O(g(n))$ but not $\Omega(g(n))$.

- Proposition. If $\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$, then $f(n)$ is $\Omega(g(n))$ but not $O(g(n))$.

a.k.a. $\omega(g(n))$

Asymptotic bounds for some common functions

- Polynomials. Let $f(n) = a_0 + a_1 n + \ldots + a_d n^d$ with $a_d > 0$. Then, $f(n)$ is $\Theta(n^d)$.
  - Pf. $\lim_{n\to\infty} \dfrac{a_0 + a_1 n + \ldots + a_d n^d}{n^d} = a_d > 0$
- Logarithms. $\log_a n$ is $\Theta(\log_b n)$ for every $a > 1$ and every $b > 1$.
  - Pf. $\dfrac{\log_a n}{\log_b n} = \dfrac{1}{\log_b a}$
- Logarithms and polynomials. $\log_a n$ is $O(n^d)$ for every $a > 1$ and every $d > 0$.
  - Pf. $\lim_{n\to\infty} \dfrac{\log_a n}{n^d} = 0$

L'Hopital's Rule

Asymptotic bounds for some common functions

- Exponentials and polynomials. n^d is $O(r^n)$ for every $r > 1$ and every $d > 0$.
  - Pf. $\lim_{n \to \infty} \frac{n^d}{r^n} = 0$
- Factorials.

  $n!$ is $o(n^n)$

  $\log(n!) = \Theta(n \log n)$

  - Pf. Stirling's formula: $n! \sim \sqrt{2\pi n} \cdot (\frac{n}{e})^n$

Comparing some common functions

- $\Theta(1)$, constant
- $\Theta(\log n)$, logarithm
- $\Theta(n)$, linear
- $\Theta(n \log n)$, linearithmic
- $\Theta(n^c)$, polynomial

tractable

- $\Theta(2^n)$, exponential
- $\Theta(n!)$, factorials

intractable

Polynomial running time

- When considering brute force algorithm to solve one problem, it is usually asymptotically equal to exponential functions.
- When an algorithm has a polynomial running time, we say it is **efficient,** and the corresponding problem is so-called **easy** or **tractable.**
    - The algorithm has typically exposes some **crucial structure** of the problem.

Although, there are exceptions

- Some poly-time algorithms in the wild have galactic constants and/or huge exponents.

- Q. Which would you prefer: $20\,n^{120}$ or $n^{1\,+\,0.02\,\ln n}$?

n^{120}

Map graphs in polynomial time

Mikkel Thorup*
Department of Computer Science, University of Copenhagen
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
mthorup@diku.dk

Abstract

Chen, Grigni, and Papadimitriou (WADS'97 and STOC'98) have introduced a modified notion of planarity, where two faces are considered adjacent if they share at least one point. The corresponding abstract graphs are called map graphs. Chen et.al. raised the question of whether map graphs can be recognized in polynomial time. They showed that the decision problem is in NP and presented a polynomial time algorithm for the special case where we allow at most 4 faces to intersect in any point — if only 3 are allowed to intersect in a point, we get the usual planar graphs.

Chen et.al. conjectured that map graphs can be recognized in polynomial time, and in this paper, their conjecture is settled affirmatively.

Further reading

- [CLRS] Ch.2 (2.1, 2.2), Ch.3
- [Rosen] Ch.1 (1.7, 1.8) and Ch.5 (5.1, 5.2)