2 Complexity Theory Recap

22 April 2025

Prof. Dr. Sebastian Wild

CS627 (Summer 2025)
Philipps-Universität Marburg
version `2025-04-22 08:17`

Outline

2 Complexity Theory Recap

2.1 P and NP Informally

Hard problems

- Some algorithmic problems are **"hard nuts" to crack.**
  - e. g., the *Traveling Salesperson Problem (TSP)*:

    Given: n cities $S_1, \ldots, S_n$,
    all $n(n-1)$ pairwise
    distances $d(S_i, S_j) \in \mathbb{N}$ $\quad (i \neq j)$

    d

    Goal: Shortest round trip through all cities

    permutation

    $S_{i_1}, S_{i_2}, \ldots, S_{i_n}$ $\qquad \{i_1, \ldots, i_n\} = [n]$

    $[n] = \{1, \ldots, n\}$

    $\mathrm{cost}(\ ) = \sum_{j=1}^{n-1} d(S_{i_j}, S_{i_{j+1}}) + d(S_{i_n}, S_{i_1})$

    always exact, always correct $\qquad$ polytime

  - no general, efficient algorithm known!
    (despite decades of intensive research . . .)

Hard problems

- Some algorithmic problems are **"hard nuts" to crack.**
  - e. g., the *Traveling Salesperson Problem (TSP):*

    Given: n cities $S_1, \ldots, S_n$,
    all $n(n-1)$ pairwise
    distances $d(S_i, S_j) \in \mathbb{N}$ $(i \neq j)$

    Goal: Shortest round trip through all cities

    always exact, always correct   polytime

  - no general, efficient algorithm known!
    (despite decades of intensive research . . .)

$\rightsquigarrow$ It *seems* as if there is no efficient algorithm for TSP!

Hard problems

- Some algorithmic problems are **"hard nuts" to crack.**
  - e. g., the *Traveling Salesperson Problem (TSP):*

    Given: n cities $S_1, \ldots, S_n$,
    all $n(n-1)$ pairwise
    distances $d(S_i, S_j) \in \mathbb{N}$ $(i \neq j)$

    Goal: Shortest round trip through all cities

    always exact, always correct polytime
  - no general, efficient algorithm known!
    (despite decades of intensive research . . .)

$\rightsquigarrow$ It *seems* as if there is no efficient algorithm for TSP!

- **But:** can we *prove* that?
- Despite similarly intensive research: **No!** (not yet)

Doesn't sound like a shining example for theoretical computer science? . . . stay tuned!

United in incapacity

"I can't find an efficient algorithm, but neither can all these famous people."

Garey, Johnson 1979

Complexity Theory

- *Complexity theory* allows us to *compare* the *hardness* of algorithmic problems.

A: old problem
Consensus: hard

B: new problem
Status: unknown
(seems hard for *us* . . .)

Complexity Theory

- *Complexity theory* allows us to *compare* the *hardness* of algorithmic problems.

$\leq_p$

A: old problem
Consensus: hard

B: new problem
Status: unknown
(seems hard for *us* . . .)

Intuitive idea:

1. If A is a known hard nut, and
2. $\boxed{B \text{ is at least as hard as } A,}$

then B is a hard nut, too!

Complexity Theory

- *Complexity theory* allows us to *compare* the *hardness* of algorithmic problems.

"reduce A to B"

$$\leq_p$$

A: old problem
Consensus: hard

B: new problem
Status: unknown
(seems hard for *us* . . .)

Intuitive idea:

1. If A is a known hard nut, and
2. $\boxed{B \text{ is at least as hard as } A,}$

then B is a hard nut, too!

Formally:

efficient = polytime

1. A is NP-hard: probably $\nexists$ eff. alg. for A
2. $\boxed{A \leq_p B}$: $\exists$ eff. alg. for $B \implies \exists$ eff. alg. for A

$\leadsto$ B is NP-hart: probably $\nexists$ eff. alg. for B!

P and NP – Intuitive Synopsis

$$P = NP?$$

P and NP – Intuitive Synopsis

- $\boxed{\text{P} = \text{class of problems for which there is an algorithm } A \text{ and a polynomial } p \text{ such that } A \textbf{ solves} \text{ every instance } I \text{ in time } O(p(|I|)).}$
  - P for "polynomial" – i. e., all problems where a solution can be *found* by a (deterministic) algorithm in polynomial time.

P = NP?

P and NP – Intuitive Synopsis

- $\boxed{\text{P = class of problems for which there is an algorithm } A \text{ and a polynomial } p \text{ such that } A \textbf{ solves} \text{ every instance } I \text{ in time } O(p(|I|)).}$
  - P for "polynomial" – i. e., all problems where a solution can be *found* by a (deterministic) algorithm in polynomial time.

- $\boxed{\text{NP = class of problems for which there is an algorithmus } A \text{ and a polynomial } p \text{ such that } A \text{ can } \textbf{verify} \text{ a given candidate solution } l(I) \text{ of a given instance } I \text{ in time } O(p(|I|)) \text{, i. e., check whether } l(I) \text{ solves } I \text{ or not.}}$
  - NP for "nondeterministically polynomial" – i. e., all problems where a solution can be *found* by a *nondeterministic* algorithm in polynomial time.
  - This is equivalent to the above characterization via verification.

P = NP?

P and NP – Intuitive Synopsis

- P = class of problems for which there is an algorithm A and a polynomial p such that A **solves** every instance I in time $O(p(|I|))$.
  - P for "polynomial" – i. e., all problems where a solution can be *found* by a (deterministic) algorithm in polynomial time.

- NP = class of problems for which there is an algorithmus A and a polynomial p such that A can **verify** a given candidate solution $l(I)$ of a given instance I in time $O(p(|I|))$, i. e., check whether $l(I)$ solves I or not.
  - NP for "nondeterministically polynomial" – i. e., all problems where a solution can be *found* by a *nondeterministic* algorithm in polynomial time.
  - This is equivalent to the above characterization via verification.

P = NP?

- We know P $\subseteq$ NP. We *think* P $\subsetneq$ NP, i. e., P $\neq$ NP.
  The question "P = NP?" is one of the famous **millenium problems** and arguably the **most important open problem of theoretical computer science**.

2.2 Models of Computation

Clicker Question

What is the cost of *adding* two (decimal) d-digit integers?
(For example, for $d = 5$, what is $45\,235 + 91\,342$?)

- **A** constant time
- **B** logarithmic in d
- **C** proportional to d
- **D** quadratic in d
- **E** no idea what you are talking about

→ `sli.do/cs627`

Clicker Question

What is the cost of *adding* two (decimal) d-digit integers?
(For example, for $d = 5$, what is $45\,235 + 91\,342$?)

- **A** constant time ✓
- **B** ~~logarithmic in d~~
- **C** proportional to d ✓
- **D** ~~quadratic in d~~
- **E** no idea what you are talking about ✓

→ sli.do/cs627

Mathematical Models of Computation

- complexity classes talk about sets of problems based upon whether they allow an algorithm of a certain cost
- in general, this depends on the allowable algorithms and their costs!

$\rightsquigarrow$ need to fix a machine model

Mathematical Models of Computation

- complexity classes talk about sets of problems based upon whether they allow an algorithm of a certain cost
- in general, this depends on the allowable algorithms and their costs!

$\rightsquigarrow$ need to fix a machine model

A *machine model* decides

- what algorithms are possible
- how they are described (= programming language)
- what an execution *costs*

Goal: Machine models should be
detailed and powerful enough to reflect actual machines,
abstract enough to unify architectures,
simple enough to analyze.

Random Access Machines

Standard model for detailed complexity analysis:

Random access machine (RAM)

more detail in §2.2 of *Sequential and Parallel Algorithms and Data Structures* by Sanders, Mehlhorn, Dietzfelbinger, Dementiev

- unlimited *memory* $\texttt{MEM}[0], \texttt{MEM}[1], \texttt{MEM}[2], \ldots$
- fixed number of *registers* $R_1, \ldots, R_r$ (say $r = 100$)

Random Access Machines

Standard model for detailed complexity analysis:

Random access machine (RAM)

more detail in §2.2 of *Sequential and Parallel Algorithms and Data Structures* by Sanders, Mehlhorn, Dietzfelbinger, Dementiev

- unlimited *memory* $\texttt{MEM}[0], \texttt{MEM}[1], \texttt{MEM}[2], \ldots$
- fixed number of *registers* $R_1, \ldots, R_r$ (say $r = 100$)
- memory cells $\texttt{MEM}[i]$ and registers R_i store w-bit integers, i. e., numbers in $[0..2^w - 1]$
  w is the word width/size; typically $\boxed{w \propto \lg n}$ $\rightsquigarrow$ $2^w \approx n$

Random Access Machines

Standard model for detailed complexity analysis:

Random access machine (RAM)

more detail in §2.2 of *Sequential and Parallel Algorithms and Data Structures* by Sanders, Mehlhorn, Dietzfelbinger, Dementiev

- unlimited *memory* $\texttt{MEM}[0], \texttt{MEM}[1], \texttt{MEM}[2], \ldots$
- fixed number of *registers* $R_1, \ldots, R_r$ (say $r = 100$)
- memory cells $\texttt{MEM}[i]$ and registers R_i store w-bit integers, i. e., numbers in $[0..2^w - 1]$
  w is the word width/size; typically $\boxed{w \propto \lg n}$ $\rightsquigarrow$ $2^w \approx n$
- Instructions:
  - load & store: $R_i := \texttt{MEM}[R_j]$ $\texttt{MEM}[R_j] := R_i$
  - operations on registers: $R_k := R_i + R_j$ (arithmetic is *modulo* 2^w!)
    also $R_i - R_j$, $R_i \cdot R_j$, $R_i \operatorname{div} R_j$, $R_i \bmod R_j$
    C-style operations (bitwise and/or/xor, left/right shift)
  - conditional and unconditional jumps
- time cost: number of executed instructions
- space cost: total number of touched memory cells

RAM-Program Example

Example RAM program

1 *// Assume:* R_1 *stores number* N
2 *// Assume:* $\texttt{MEM}[0..N)$ *contains list of* N *numbers*
3 $R_2 := R_1$;
4 $R_3 := R_1 - 2$;
5 $R_4 := \texttt{MEM}[R_3]$;
6 $R_5 := R_3 + 1$;
7 $R_6 := \texttt{MEM}[R_5]$;
8 **if** $(R_4 \le R_6)$ goto line 11;
9 $\texttt{MEM}[R_3] := R_6$;
10 $\texttt{MEM}[R_5] := R_4$;
11 $R_3 := R_3 - 1$;
12 **if** $(R_3 \ge 0)$ goto line 5;
13 $R_2 := R_2 - 1$;
14 **if** $(R_2 > 0)$ goto line 4;
15 *// Done:*

RAM-Program Example

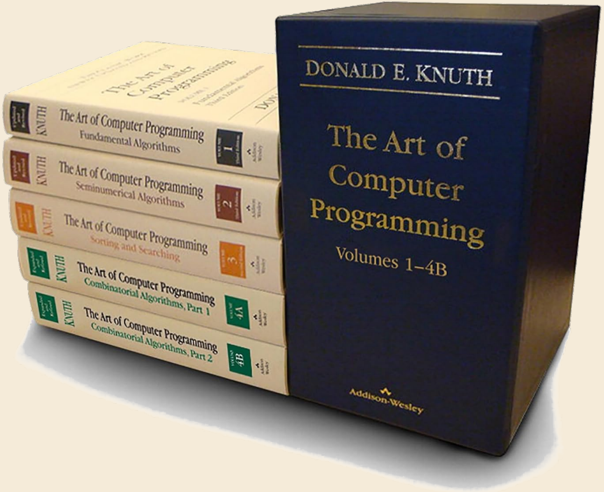
Example RAM program

```
1  // Assume: R1 stores number N
2  // Assume: MEM[0..N) contains list of N numbers
3  R2 := R1;
4  R3 := R1 − 2;
5  R4 := MEM[R3];
6  R5 := R3 + 1;
7  R6 := MEM[R5];
8  if (R4 ≤ R6) goto line 11;
9  MEM[R3] := R6;
10 MEM[R5] := R4;
11 R3 := R3 − 1;
12 if (R3 ≥ 0) goto line 5;
13 R2 := R2 − 1;
14 if (R2 > 0) goto line 4;
15 // Done: MEM[0..N) sorted
```

RAM-Program Example

Example RAM program

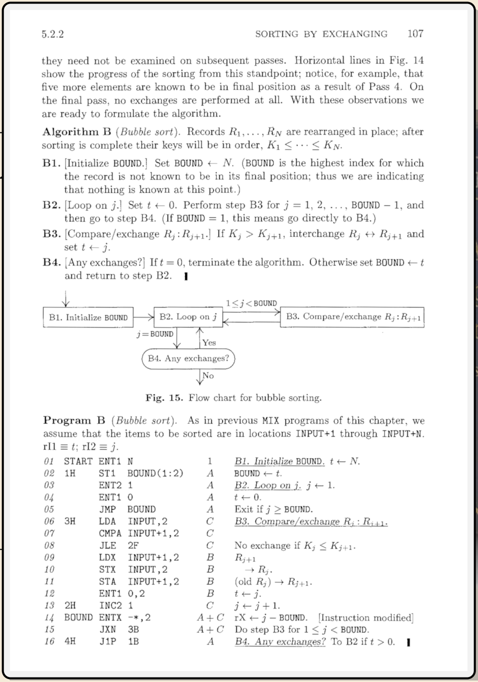
```
1  // Assume: R_1 stores number N
2  // Assume: MEM[0..N) contains list of N numbers
3  R_2 := R_1;
4  R_3 := R_1 − 2;
5  R_4 := MEM[R_3];
6  R_5 := R_3 + 1;
7  R_6 := MEM[R_5];
8  if (R_4 ≤ R_6) goto line 11;
9  MEM[R_3] := R_6;
10 MEM[R_5] := R_4;
11 R_3 := R_3 − 1;
12 if (R_3 ≥ 0) goto line 5;
13 R_2 := R_2 − 1;
14 if (R_2 > 0) goto line 4;
15 // Done: MEM[0..N) sorted
```



RAM-Program Example

Example RAM program

1 // *Assume:* R_1 *stores number* N
2 // *Assume:* $\mathtt{MEM}[0..N)$ *contains list of* N *number*
3 $R_2 := R_1;$
4 $R_3 := R_1 - 2;$
5 $R_4 := \mathtt{MEM}[R_3];$
6 $R_5 := R_3 + 1;$
7 $R_6 := \mathtt{MEM}[R_5];$
8 **if** $(R_4 \le R_6)$ goto line 11;
9 $\mathtt{MEM}[R_3] := R_6;$
10 $\mathtt{MEM}[R_5] := R_4;$
11 $R_3 := R_3 - 1;$
12 **if** $(R_3 \ge 0)$ goto line 5;
13 $R_2 := R_2 - 1;$
14 **if** $(R_2 > 0)$ goto line 4;
15 // *Done:* $\mathtt{MEM}[0..N)$ *sorted*



5.2.2 SORTING BY EXCHANGING 107

they need not be examined on subsequent passes. Horizontal lines in Fig. 14 show the progress of the sorting from this standpoint; notice, for example, that five more elements are known to be in final position as a result of Pass 4. On the final pass, no exchanges are performed at all. With these observations we are ready to formulate the algorithm.

Algorithm B (*Bubble sort*). Records $R_1, \ldots, R_N$ are rearranged in place; after sorting is complete their keys will be in order, $K_1 \le \cdots \le K_N$.

B1. [Initialize BOUND.] Set BOUND $\leftarrow N$. (BOUND is the highest index for which the record is not known to be in its final position; thus we are indicating that nothing is known at this point.)

B2. [Loop on j.] Set $t \leftarrow 0$. Perform step B3 for $j = 1, 2, \ldots,$ BOUND $- 1$, and then go to step B4. (If BOUND $= 1$, this means go directly to B4.)

B3. [Compare/exchange $R_j : R_{j+1}$.] If $K_j > K_{j+1}$, interchange $R_j \leftrightarrow R_{j+1}$ and set $t \leftarrow j$.

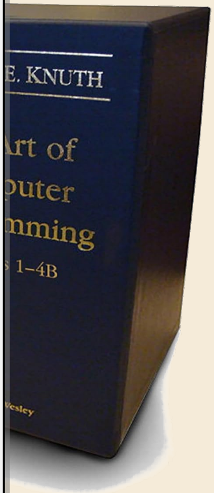
B4. [Any exchanges?] If $t = 0$, terminate the algorithm. Otherwise set BOUND $\leftarrow t$ and return to step B2. ▌

B1. Initialize BOUND
B2. Loop on j
$1 \le j <$ BOUND
B3. Compare/exchange $R_j : R_{j+1}$
$j =$ BOUND
Yes
B4. Any exchanges?
No

Fig. 15. Flow chart for bubble sorting.

Program B (*Bubble sort*). As in previous MIX programs of this chapter, we assume that the items to be sorted are in locations INPUT+1 through INPUT+N. rI1 $\equiv t$; rI2 $\equiv j$.

```
START ENT1 N            1      B1. Initialize BOUND. t ← N.
1H    ST1  BOUND(1:2)   A      BOUND ← t.
      ENT2 1            A      B2. Loop on j. j ← 1.
      ENT1 0            A      t ← 0.
      JMP  BOUND        A      Exit if j ≥ BOUND.
3H    LDA  INPUT,2      C      B3. Compare/exchange R_j : R_{j+1}.
      CMPA INPUT+1,2    C
      JLE  2F           C      No exchange if K_j ≤ K_{j+1}.
      LDX  INPUT+1,2    B      R_{j+1}
      STX  INPUT,2      B        → R_j.
      STA  INPUT+1,2    B      (old R_j) → R_{j+1}.
      ENT1 0,2          B      t ← j.
2H    INC2 1            C      j ← j + 1.
BOUND ENTX -*,2         A+C    rX ← j − BOUND. [Instruction modified]
      JXN  3B           A+C    Do step B3 for 1 ≤ j < BOUND.
4H    J1P  1B           A      B4. Any exchanges? To B2 if t > 0. ▌
```



Keep it Simple, Stupid

- word-RAM (rather) realistic, but complicated
  - note that the machine has to grow with the inputs(!)
- for a coarse distinction of running time complexity, simpler models suffice
  - useful to reason about "all algorithms"
  - machine is fixed for all inputs sizes apart from storage for input