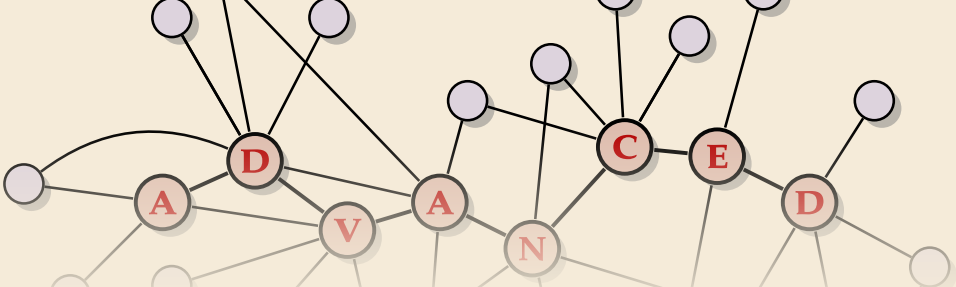


3 Pseudopolynomial Algorithms

13 May 2025

Prof. Dr. Sebastian Wild

CS627 (Summer 2025)
Philipps-Universität Marburg
version 2025-05-13 09:51

Outline

3.1 Integer Problems

Integer Problems

Definition 3.1 (Integer-Input Problem)

A problem U for which (part of the) input is a *sequence of integers* is called an *integer-input problem*.

For any instance x of an integer-input problem, we write $MaxInt(x)$ for the largest integer occurring in the input encoding. ◀

(As before, integers are encoded in binary.)

Integer Problems

Definition 3.1 (Integer-Input Problem)

A problem U for which (part of the) input is a *sequence of integers* is called an *integer-input problem*.

For any instance x of an integer-input problem, we write $MaxInt(x)$ for the largest integer occurring in the input encoding.

(As before, integers are encoded in binary.)

Examples

- TravelingSalesman
- SubsetSum
- BinPacking
- ILP
- Knapsack

MaxInt = Largest distance

Clicker Question

How large can $MaxInt(x)$ be as a function of $n = |x|$?

A	$O(1)$	**D**	$O(n^c)$	**G**	$2^{O(n)}$
B	$O(\log n)$	**E**	$O(n^{\log n})$	**H**	$O(2^{2^n})$
C	$O(n)$	**F**	$O(2^n)$	**I**	unbounded

→ *sli.do/cs627*

Clicker Question

How large can $MaxInt(x)$ be as a function of $n = |x|$?

A $O(1)$ ✓

B $O(\log n)$ ✓

C $O(n)$ ✓

D $O(n^c)$ ✓

E $O(n^{\log n})$ ✓

F $O(2^n)$ ✓

G $2^{O(n)}$ ✓ (c^n for any c)

H ~~$O(2^{2^n})$~~

I ~~unbounded~~

→ *sli.do/cs627*

Pseudopolynomial

Definition 3.2 (Pseudopolynomial algorithm)

$\neq$ quasi-polynomial ($= 2^{O(\log^c n)}$)

Let U be an integer-input problem and A an algorithm that solves U.
A has *pseudopolynomial time for* U, if there is a polynomial p in two variables with

$$Time_A(x) \quad = \quad O\Big(p\big(|x|, MaxInt(x)\big)\Big),$$

could be $2^{|x|}$

for every instance x to U.

Pseudopolynomial

Definition 3.2 (Pseudopolynomial algorithm)

≠ quasi-polynomial ($= 2^{O(\log^c n)}$)

Let U be an integer-input problem and A an algorithm that solves U.
A has *pseudopolynomial time for* U, if there is a polynomial p in two variables with

$$Time_A(x) \quad = \quad O\Big(p\big(|x|, MaxInt(x)\big)\Big),$$

for every instance x to U. ◀

Note: If $MaxInt(x) \leq h(|x|)$ for a polynomial h, then $p\big(|x|, MaxInt(x)\big) \leq g(|x|)$ for a polynomial g.

$n = |x|$

$MaxInt(x) \leq n^{10}$

$p(n, m) \leq (n \cdot m)^5$

$p(n, MaxInt(n)) \leq (n\, n^{10})^5 = n^{55}$

Pseudopolynomial Languages

Definition 3.3 (Value-Bounded Subproblem)

Let U be an integer-input problem and let $h : \mathbb{N} \to \mathbb{N}$ be weakly increasing.
The *h-bounded subproblem of U* (notation $\overbrace{Value(h)_U}$) is the problem which results from U by allowing only inputs x with $MaxInt(x) \leq h(|x|)$. ◀

Pseudopolynomial Languages

Definition 3.3 (Value-Bounded Subproblem)

Let U be an integer-input problem and let $h : \mathbb{N} \to \mathbb{N}$ be weakly increasing.
The *h-bounded subproblem of U* (notation $Value(h)_U$) is the problem which results from U by allowing only inputs x with $MaxInt(x) \leq h(|x|)$.

Theorem 3.4 (Pseudopolynomial is polynomial for small h)

Let U be an integer-input problem and A a pseudopolynomial algorithm for U.
Then for every polynomial h there is a polytime algorithm for $Value(h)_U$.

Proof:

A has running time $O(p(|x|, MaxInt(x)))$ $\quad n = |x|$

for $x \in Value(h)_U$ then $MaxInt(x) \leq h(|x|) = O(n^c)$ for constant c

$\Rightarrow$ A needs $O(p(n, O(n^c))) = O(n^d)$ for constant d

Pseudopolynomial Languages

Definition 3.3 (Value-Bounded Subproblem)

Let U be an integer-input problem and let $h : \mathbb{N} \to \mathbb{N}$ be weakly increasing.
The *h-bounded subproblem of U* (notation $Value(h)_U$) is the problem which results from U by allowing only inputs x with $MaxInt(x) \leq h(|x|)$. ◀

Theorem 3.4 (Pseudopolynomial is polynomial for small h)

Let U be an integer-input problem and A a pseudopolynomial algorithm for U.
Then for every polynomial h there is a polytime algorithm for $Value(h)_U$. ◀

Proof:

Hence if U is a decision problem then $Value(h)_U \in \mathsf{P}$,
if U is an optimization problem then $Value(h)_U \in \mathsf{PO}$.

3.2 Knapsack

Knapsack (Optimization Version)

Definition 3.5 (Knapsack (Optimization Version))

items $1, \dots, n$

Given: tuple $(w_1, \dots, w_n; v_1, \dots, v_n; b)$ of $2n + 1$ positive integers, $n \in \mathbb{N}$.
We call b the *capacity* of the knapsack, w_i the *weight* and v_i the *value* (profit) of the i-th object, $1 \leq i \leq n$.

Goal: The *optimization problem* KNAPSACK asks to find a subset $T \subseteq \{1, 2, \dots, n\}$ of items with maximal total value $cost(T) = \sum_{i \in T} v_i$ such that T fits into the knapsack, i. e., $\sum_{i \in T} w_i \leq b$.

◀

Recap: The 6 Steps of Dynamic Programming

⇝ see *Efficient Algorithms*

1. Define **subproblems** (and relate to original problem)
2. **Guess** (part of solution) ⇝ local brute force
3. Set up **DP recurrence** (for quality of solution)
4. Recursive implementation with **Memoization**
5. Bottom-up **table filling** (topological sort of subproblem dependency graph)
6. **Backtracing** to reconstruct optimal solution

- Steps 1–3 require insight / creativity / intuition;
  Steps 4–6 are mostly automatic / same each time

⇝ Correctness proof usually at level of DP recurrence

running time too! worst case time = #subproblems · time to find single best guess

Dynamic Programming Solution

- **Subproblems:** (n', b'): only items $1 \le i \le n'$ and total weight b'
- **Guess:** whether to include item n'
- **Recurrence:** $V[n', b']$ = max value in subproblem (n', b')

$$V[n', b'] = \begin{cases} 0 & n' = 0 \\ V[n'-1, b'] & n' \ge 1 \ \wedge \ w_{n'} > b' \\ \max\{ V[n'-1, b'], & \\ \quad v_{n'} + V[n'-1, b' - w_{n'}] & n' \ge 1 \ \wedge \ w_{n'} \le b' \end{cases}$$

answer: $V[n, b]$

running time: compute max, '+' $O\left(\log(\text{MaxInt}(x))\right)$

entries: $n \cdot b = O(|x| \cdot \text{MaxInt}(x))$

Pseudopolynomial Knapsack

Theorem 3.6 (DP for Knapsack is pseudopolynomial)

For every instance I to KNAPSACK we have

$$Time_{DPKP}(I) = O\big(|I| \cdot MaxInt(I) \log(MaxInt(I))\big),$$

i. e., DPKP has pseudopolynomial time for KNAPSACK. ◀

(really, b needs to be small; v_i can be large)

Beyond Knapsack

- Similar trick works for some other NP-complete problems, e. g., Partition, MakingChange
- for yet other NP-complete problems, e. g., TravelingSalesman, no such algorithms seems to exist . . .

Beyond Knapsack

- Similar trick works for some other NP-complete problems, e. g., Partition, MakingChange

- for yet other NP-complete problems, e. g., TravelingSalesman, no such algorithms seems to exist . . .

  . . . can we give evidence that likely ***no*** *pseudopolynomial algorithm is* ***possible****?*

3.3 Strong NP-hardness

Hardness

Definition 3.7 (strongly NP-hard)

An integer-input problem is called *strongly* NP*-hard*, if there exists a polynomial p such that $Value(p)_U$ is NP-hard. ◀

So: strongly NP-hard $\rightsquigarrow$ hard even for instances with "small" numbers.

Hardness

Definition 3.7 (strongly NP-hard)

An integer-input problem is called *strongly* NP-*hard*, if there exists a polynomial p such that $Value(p)_U$ is NP-hard.

So: strongly NP-hard $\rightsquigarrow$ hard even for instances with "small" numbers.

Theorem 3.8 (strongly NP-hard → no pseudopoly. algorithm)

Let $P \neq NP$ and U a strongly NP-hard (integer-input) problem.
Then there exists no algorithm with pseudopolynomial time for U.

Proof: U strongly NP-hard $\Rightarrow \exists p$ $Value(p)_U$ NP-hard

Assume A pseudopoly. alg. for U

$\Rightarrow A$ runs in time $O(h(n, \underbrace{Max|x(x)|}_{\leq p(n)}))$ $\quad x \in Value(p)_U$

$= O(n^c)$ const. c

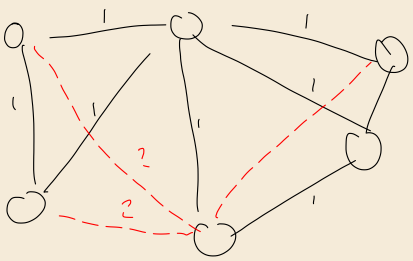
$\Rightarrow Value(p)_U \in P \Rightarrow P = NP$ ↯ ass. $\Rightarrow A$ cannot exist. ■

Example

Theorem 3.9
TravelingSalesman is strongly NP-hard.

Proof:

Show: HamiltonianCycle $\leq_p$ Value(2)$_{TSP}$



round trip of length $\leq n$

$\Leftrightarrow$ Hamiltonian cycle

It's all about the encoding

Theorem 3.10 (strongly hard iff unary hard)

An integer-input problem is strongly NP-hard if, and only if, representing its instances with unary encoding for integers remains NP-hard. ◀

Proof:

A strongly NP-hard $\rightsquigarrow$ $\exists$ polynomial p s.t. $Value(p)_A$ NP-hard

It's all about the encoding

Theorem 3.10 (strongly hard iff unary hard)

An integer-input problem is strongly NP-hard if, and only if, representing its instances with unary encoding for integers remains NP-hard. ◀

Proof:

A strongly NP-hard $\rightsquigarrow$ $\exists$ polynomial p s.t. $Value(p)_A$ NP-hard

For $x \in Value(p)_A$, unary encoding blows up $|x|$ only by polynomial factor.

$\rightsquigarrow$ A encoded with unary numbers NP-hard.

It's all about the encoding

Theorem 3.10 (strongly hard iff unary hard)

An integer-input problem is strongly NP-hard if, and only if, representing its instances with unary encoding for integers remains NP-hard. ◀

Proof:

A strongly NP-hard $\rightsquigarrow$ $\exists$ polynomial p s.t. $Value(p)_A$ NP-hard

For $x \in Value(p)_A$, unary encoding blows up $|x|$ only by polynomial factor.

$\rightsquigarrow$ A encoded with unary numbers NP-hard.

Conversely, let A with unary numbers be NP-hard.

With unary encoding, $MaxInt(x) \leq |x|$, so $Value(n \mapsto n)_A = A$ is NP-hard. ■

Summary

Pseudopolynomial algorithms can be practically efficient if numbers are (really) small

Only applicable to few problems