Complexity: Problems and Classes

Fall 2020

Algorithmics: Basic references

- Kleinberg, Tardos. Algorithm Design, Pearson Education, 2006.
- Cormen, Leisserson, Rivest and Stein. Introduction to algorithms. Second edition, MIT Press and McGraw Hill 2001.
- Easley, Kleinberg. Networks, Crowds, and Markets: Reasoning About a Highly Connected World, Cambridge University Press, 2010

Computational Complexity: Basic references

- Sipser Introduction to the Theory of Computation 2013.
- Papadimitriou Computational Complexity 1994.
- Garey and Johnson Computers and Intractability: A Guide to the Theory of NP-Completeness 1979

Growth of functions: Asymptotic notations

We consider only functions defined on the natural numbers.

$$f, g : \mathbb{N} \to \mathbb{N}$$

O-notation

For a given function $g(n)$

$$O(g(n)) = \{f(n) \mid \text{there exist a positive constant } c \text{ and } n_0 \geq 0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0\}$$

$$\begin{aligned} 5n^3 + 2n^2 &= O(2^n) \\ 5n^3 + 2n^2 &= O(n^4) \\ 5n^3 + 2n^2 &= O(n^3) \\ 5n^3 + 2n^2 &= \Theta(n^3) \\ 2^n &= O(2^{2n}) \\ 2^n &= O(2^{n \log n}) \end{aligned}$$

It is used for asymptotic upper bound.

Although $O(g(n))$ is a set we write $f(n) = O(g(n))$ to indicate that $f(n)$ is a member of $O(g(n))$

Θ-notation

For a given function $g(n)$

$$\Theta(g(n)) = \{f(n) \mid \text{there are positive constants } c_1,\ c_2, \text{ and } n_0 \geq 0 \ \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \text{ for all } n \geq n_0\}$$

$$5n^3 + 2n^2 = \Theta(n^3)$$

$$5n^3 + 2n^2 \notin \Theta(n^2)$$

It is used for asymptotic equivalence

Ω-notation

For a given function $g(n)$

$$\Omega(g(n)) = \{f(n) \mid \text{there exist positive constants } c \text{ and } n_0 \text{ such that} \; 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\}$$

$$\begin{aligned} 5n^3 + 2n^2 &= \Theta(n^3) \\ 5n^3 + 2n^2 &= \Omega(n^3) \\ 5n^3 + 2n^2 &= \Omega(n^2) \\ 2^n &= \Omega(2^{n/2}) \end{aligned}$$

It is used for asymptotic lower bound.

o-notation

For a given function $g(n)$

$$
\begin{aligned}
o(g(n)) = \{f(n) \mid & \text{for any positive constant } c \\
& \text{there is a positive constant } n_0 \text{ such that} \\
& 0 \leq f(n) < cg(n), \text{ for all } n \geq n_0\}
\end{aligned}
$$

Note that $f(n) = O(n)$ implies $f(n) \leq cg(n)$ asymptotically for some c

but $f(n) = o(n)$ implies $f(n) < cg(n)$ asymptotically for any c

and when $f(n) = o(g(n))$ it holds that $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$

It is used for asymptotic upper bounds that are not asymptotically tight.

ω-notation

$f(n) \in \omega(g(n))$ iff $g(n) \in o(f(n))$

Algorithm's analysis

- Time
- Space

Algorithm $\mathcal{A}$ on input x takes time $t(x)$.
$|x|$ denotes the size of input x.

Definition

The cost function of algorithm $\mathcal{A}$ is a function from $\mathbb{N}$ to $\mathbb{N}$ defined as

$$\mathcal{C}_{\mathcal{A}}(n) = \max_{|x|=n} t(x)$$

Fundamental growth functions

- Polynomial time
- Exponential time

Fundamental growth functions

- Polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O(n^c)$, for some constant c.
- Exponential time

Fundamental growth functions

- Polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O(n^c)$, for some constant c.
- Exponential time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(n^c)}$, for some constant c.

Fundamental growth functions

- Polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O(n^c)$, for some constant c.
- Exponential time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(n^c)}$, for some constant c.
- Quasi-polynomial time
- Pseudo polynomial time

Fundamental growth functions

- Polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O(n^c)$, for some constant c.
- Exponential time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(n^c)}$, for some constant c.

- Quasi-polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(\log^c n)}$, for some constant c.
- Pseudo-polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O((mW)^c)$, for some constant c, but input size is $O(m + \log W)$

Fundamental growth functions

- Polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O(n^c)$, for some constant c.
- Exponential time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(n^c)}$, for some constant c.
- Quasi-polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = 2^{O(\log^c n)}$, for some constant c.
- Pseudo-polynomial time
  $\mathcal{C}_{\mathcal{A}}(n) = O((mW)^c)$, for some constant c, but input size is $O(m + \log W)$
- Similar definitions replacing time by space
  Most used PSPACE polynomial space

Problem types

- Decision

  Input x

  Property $P(x)$

  Example: Given a graph and two vertices, is there a path joining them?

- Function

  Input x

  Compute y *such that* $Q(x, y)$

  Example: Given a graph and two vertices, compute the minimum distance between them.

Problem types

- Decision

  Input x

  Property $P(x)$

Problem types

- Decision

  Input x

  Property $P(x)$

Coding inputs on alphabet Σ a problem is a set

Problem types

- Decision

  Input x

  Property $P(x)$

Coding inputs on alphabet Σ a problem is a set $\{x \mid P(x)\} \in \mathcal{P}(\Sigma^*)$

Problem types

- Decision

  Input x

  Property $P(x)$

  Coding inputs on alphabet Σ a problem is a set $\{x \mid P(x)\} \in \mathcal{P}(\Sigma^*)$

- Function

  Input x

  Compute y *such that* $Q(x, y)$

Problem types

- Decision

  Input x

  Property $P(x)$

  Coding inputs on alphabet Σ a problem is a set $\{x \mid P(x)\} \in \mathcal{P}(\Sigma^*)$

- Function

  Input x

  Compute y *such that* $Q(x, y)$

  Coding inputs/outputs on alphabet Σ a deterministic algorithm solving a problem determines a function

Problem types

- Decision

  Input x

  Property $P(x)$

  Coding inputs on alphabet Σ a problem is a set $\{x \mid P(x)\} \in \mathcal{P}(\Sigma^*)$

- Function

  Input x

  Compute y *such that* $Q(x, y)$

  Coding inputs/outputs on alphabet Σ a deterministic algorithm solving a problem determines a function $f : \Sigma^* \to \Sigma^*$ s.t., for any x, $Q(x, f(x))$ is true.

Decision problem classes

- Undecidable
  No algorithm can solve the problem.
- Decidable
  There is an algorithm solving them.
  - P:
    There is an algorithm solving it with polynomial cost.
  - EXP
    There is an algorithm solving it with exponential cost.
  - PSPACE
    There is an algorithm solving it within polynomial space.

NP: non-deterministic polynomial time

It is possible to define a certificate y and a property $P(x, y)$ such that

- If x is an input with answer yes, there is y such that $P(x, y)$ is true,
- $P(x, y)$ can be decided in polynomial time, given x and y.
- y has polynomial size with respect to $|x|$.

Problems with a polynomial time verifier

NP: non-deterministic polynomial time

It is possible to define a certificate y and a property $P(x, y)$ such that

- If x is an input with answer yes, there is y such that $P(x, y)$ is true,
- $P(x, y)$ can be decided in polynomial time, given x and y.
- y has polynomial size with respect to $|x|$.

Problems with a polynomial time verifier
$\{x \mid \exists y P(x, y)\}$

Some decision problems

Bipartiteness (BIP)
Given a graph determine whether it is bipartite.

Perfect matching (PMATCH)
Given a graph determine whether it has a perfect matching.

Hamiltonian Circuit (HC)
Given a graph determine whether it has a Hamiltonian circuit.

In which classes?

NP-hardness

- It is an open question whether P $=$ NP or NP $=$ EXP. Most believed is that P $\neq$ NP

- Π is NP-hard means that a polynomial time algorithm for Π can be reused to solve in polynomial time any problem in P.
- Decision problem A is NP-complete iff $A \in$ NP and A is NP-hard.
  Look at Garey and Johnson's book for a big list of NP-hard/complete problems.

NP-hardness

- It is an open question whether P $=$ NP or NP $=$ EXP. Most believed is that P $\neq$ NP
- Π is NP-hard means that a polynomial time algorithm for Π can be reused to solve in polynomial time any problem in P.
- Decision problem A is NP-complete iff $A \in$ NP and A is NP-hard.
  Look at Garey and Johnson's book for a big list of NP-hard/complete problems.
- The NP-hardness of a problem is assessed through reductions.

Reductions

- Let A and B be decision problems
- A function $f : \mathbb{N} \to \mathbb{N}$ is a reduction from A to B if $x \in A$ iff $f(x) \in B$

Reductions

- Let A and B be decision problems
- A function $f : \mathbb{N} \rightarrow \mathbb{N}$ is a reduction from A to B if $x \in A$ iff $f(x) \in B$
- f is a polynomial time reduction if in addition f can be computed in polynomial time.

Reductions

- Let A and B be decision problems
- A function $f : \mathbb{N} \to \mathbb{N}$ is a reduction from A to B if $x \in A$ iff $f(x) \in B$
- f is a polynomial time reduction if in addition f can be computed in polynomial time.
- A is reducible to B ($A \leq B$) if there is a polynomial time reduction from A to B.

Reductions

- Let A and B be decision problems
- A function $f : \mathbb{N} \to \mathbb{N}$ is a reduction from A to B if $x \in A$ iff $f(x) \in B$
- f is a polynomial time reduction if in addition f can be computed in polynomial time.
- A is reducible to B $(A \leq B)$ if there is a polynomial time reduction from A to B.

Theorem

If $A \leq B$ and $B \in P$ then $A \in P$