UCL COMP0078: Supervised Learning
Time Complexity & P versus NP

Notes by Stephen Pasteris & Antonin Schrab

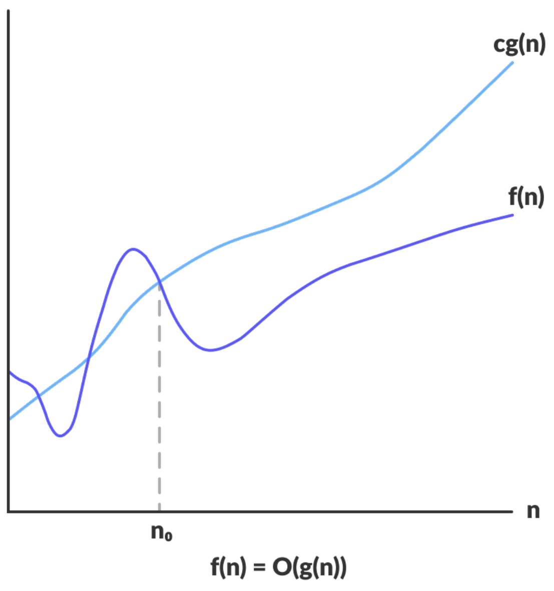
If you find any mistakes/typos, please contact sl-support@cs.ucl.ac.uk.

1 'Big O' $\mathcal{O}$, 'Big Omega' Ω and 'Big Theta' Θ notations

For functions $f, g : \mathbb{N} \to (0, \infty)$, we define the notations $\mathcal{O}$, Ω and Θ.[1]

'Big O' notation $\mathcal{O}$:

- **notation**: $f(n) = \mathcal{O}(g(n))$
- **intuition**: f is bounded *above* by g asymptotically (up to a constant)
- **informal definition**: $f(n) \leq Cg(n)$ for some constant $C > 0$ and for all n large enough
- **formal definition**: $\exists C > 0\ \exists n_0 \in \mathbb{N} : f(n) \leq Cg(n)\ \forall n > n_0$
- **examples:** $n^2 + 3n + 10 = \mathcal{O}(n^2)$, $\log(n) = \mathcal{O}(n)$, $n^{10} = \mathcal{O}(2^n)$, $2^n = \mathcal{O}(n!)$, $n! = \mathcal{O}(n^n)$

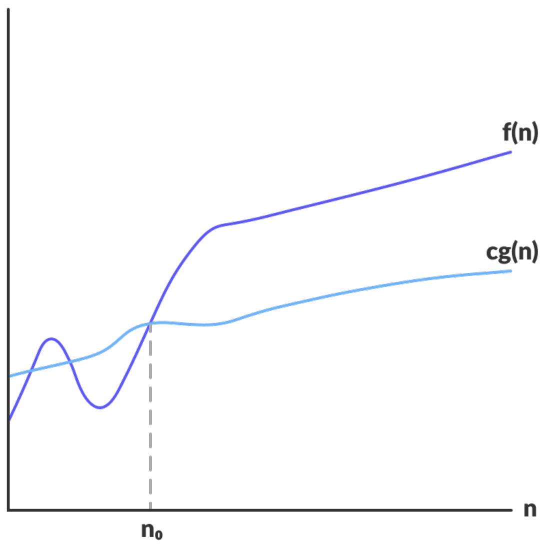


'Big Omega' notation Ω:

- **notation**: $f(n) = \Omega(g(n))$
- **intuition**: f is bounded *below* by g asymptotically (up to a constant)
- **informal definition**: $f(n) \geq Cg(n)$ for some constant $C > 0$ and for all n large enough
- **formal definition**: $\exists C > 0\ \exists n_0 \in \mathbb{N} : f(n) \geq Cg(n)\ \forall n > n_0$

[1] Source for figures: 'Asymptotic Analysis: Big-O Notation and More' `https://www.programiz.com/dsa/asymptotic-notations`.

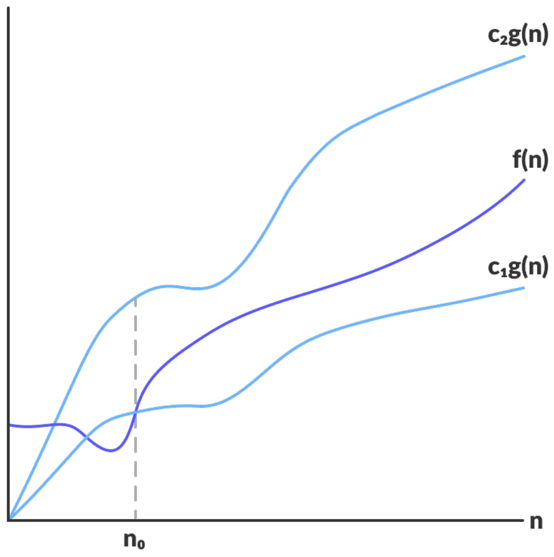
- **examples:** $\sqrt{n} = \Omega(\log n)$, $n^4 + 5n^2 = \Omega(n^2)$, $n^2 = \Omega(n \log(n))$, $\log(n) = \Omega(\log(\log(n))$



f(n) = Ω(g(n))

'Big Theta' notation Θ:

- **notation**: $f(n) = \Theta(g(n))$
- **'Big' definition**: $f(n) = \Omega(g(n))$ and $f(n) = \mathcal{O}(g(n))$
- **intuition**: f is bounded *below* and *above* by g asymptotically (up to a constant)
- **informal definition**: $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for some $C_1, C_2 > 0$ and for all n large enough
- **formal definition**: $\exists C_1 > 0\ \exists C_2 > 0\ \exists n_0 \in \mathbb{N} : C_1 g(n) \leq f(n) \leq C_2 g(n)\ \forall n > n_0$
- **examples:** $2n^3 + n^2 + 2n + 3 = \Theta(n^3)$ since $2n^3 + n^2 + 2n + 3 \leq 8n^3$ and $2n^3 + n^2 + 2n + 3 \geq 2n^3$, observe that $\Theta(2^n) \neq \Theta(e^n) \neq \Theta(3^n)$ which we encourage you to check.



f(n) = Θ(g(n))

The exact same definitions hold for functions $f, g : \mathbb{R}^D \to (0, \infty)$ by replacing $n > n_0$ by $\boldsymbol{x} > \boldsymbol{x}_0$, that is, $\boldsymbol{x}_i > (\boldsymbol{x}_0)_i$ for $i = 1, \ldots, D$.

Note that the 'Big O' $\mathcal{O}$ notation is often abused in the literature, and used as a substitute for the 'Big Theta' Θ notation.

2 Time complexity of algorithms

Many algorithms take as inputs either vectors, matrices or tensors, and, often, the time to run the algorithm increases with the dimensions of the inputs. Intuitively, the *time complexity* of an algorithm is the amount of the time it takes for the algorithm to run, expressed as a function of the dimensions of the inputs (using the notations $\mathcal{O}$, Ω and Θ).

Aside: Instead of the amount of time, one can consider the amount of memory required, this defines the *space complexity*. For parallel algorithms, by considering the number of processors required, we obtain the *size complexity*.

- Time complexity $\mathcal{O}(1)$ is called "constant time".[2]
- Time complexity $\mathcal{O}(\ln(x))$ is called "logarithmic time".
- Time complexity $\mathcal{O}(x^n)$ for some $n \in \mathbb{N}$ is called "polynomial time". Specifically:
  - $n = 1$ is called "linear time".
  - $n = 2$ is called "quadratic time".
  - $n = 3$ is called "cubic time".
  - $n = 4$ is called "quartic time".
- Time complexity $\mathcal{O}(c^x)$ for some $c > 0$ is called "exponential time". Note, however, that $\Theta(2^n) \neq \Theta(e^n) \neq \Theta(3^n)$, which we encourage you to check.

We now provide some examples of algorithms and their time complexities.

- Addition/subtraction/multiplication/division of scalars: constant time $\mathcal{O}(1)$
- Finding maximum/mean/median of a list of N entries: linear time $\mathcal{O}(N)$
- Nested for loops:

```
$x = 0$
for $i = 1, \dots, I$ :
    for $j = 1, \dots, J$ :
        for $k = 1, \dots, K$ :
            $x = x + ijk$
```

  Time complexity $\mathcal{O}(IJK)$ since step '$x = x + ijk$' can be done in constant time $\mathcal{O}(1)$

- Matrix addition $C = A + B$ for matrices A and B of size (M, N)

```
for $i = 1, \dots, M$ :
    for $j = 1, \dots, N$ :
        $c_{ij} = a_{ij} + b_{ij}$
```

  Time complexity $\mathcal{O}(MN)$

[2]This is true under the Random Access Machine (RAM) model of computation (infinite memory and equal access cost), which is the standard model of computation used in the Machine Learning community.

- Matrix multiplication by a scalar $C = \lambda A$ for matrix A of size (M, N) and $\lambda > 0$

  ```
  for $i = 1, \ldots, M$ :
      for $j = 1, \ldots, N$ :
          $c_{ij} = \lambda a_{ij}$
  ```

  Time complexity $\mathcal{O}(MN)$

- Matrix multiplication $C = AB$ for matrix A of size (M, L) and matrix B of size (L, N)

  ```
  for $i = 1, \ldots, M$ :
      for $j = 1, \ldots, N$ :
          $c_{ij} = \sum_{\ell=1}^{L} a_{i\ell} b_{\ell j}$
  ```

  Time complexity $\mathcal{O}(MNL)$

- Matrix multiplication $C = AB$ for matrix A of size (M, L) and matrix B of size (L, N)

  ```
  for $i = 1, \ldots, M$ :
      for $j = 1, \ldots, N$ :
          $c_{ij} = \sum_{\ell=1}^{L} a_{i\ell} b_{\ell j}$
  ```

  Time complexity $\mathcal{O}(MNL)$

- Matrix inversion $C = A^{-1}$ for matrix A of size (N, N)

  Gauss-Jordan elimination: cubic time $\mathcal{O}(N^3)$

  Advanced methods: between quadratic and cubic time $\mathcal{O}(N^{2.373})$

- Sorting a list of N elements:

  Insertion sort: quadratic time $\mathcal{O}(N^2)$

  Merge sort: linearithmic time $\mathcal{O}(N \log N)$

3 P versus NP

A *decision problem* for a given dataset is a question about the dataset which can be answered by True or False. A typical example is a problem asking whether the dataset satisfy some property (i.e. 'Does there exist X such that Y holds?').

A *solving algorithm* for the decision problem outputs the correct answer (either True or False) for every dataset.

P (polynomial time): The set of decision problems for which there exists a solving algorithm which produces the correct answer in polynomial time. Those problems are considered *'easy'* as they can be solved by algorithms in polynomial time.

We now define the concept of *verification algorithm* for a decision problem, which is an algorithm which takes as input a dataset and a *proof* (see below for intuition), and satisfies the following two conditions.

1. Given a dataset for which the decision problem is TRUE, there exists a proof such that the verification algorithm terminates and outputs TRUE.

2. Given a dataset for which the decision problem is FALSE, there does not exist a proof such that the verification algorithm outputs TRUE.

Intuitively, for a given dataset for which decision problem is TRUE, the *proof* explains why it is TRUE.

NP (non-deterministic polynomial time): The set of decision problems for which there exists a verification algorithm which outputs TRUE in condition 1 in polynomial time.

$\mathbf{P} \subseteq \mathbf{NP}$: Every decision problem that can be solved in polynomial time can be verified in polynomial time (for example by solving it).

$\mathbf{P} \overset{?}{=} \mathbf{NP}$: Open problem which can be summarised as: *If the solution to a problem is easy to check for correctness, must the problem be easy to solve?* One of the Millennium Prize Problems from the Clay Mathematics Institute: if you solved it, you get one million dollars!

Examples of decision problems:

- **Set cover problem** $\in$ **NP.** Given a collection $\mathcal{S}$ of sets and some $K \in \mathbb{N}$ does there exist a subset $\mathcal{T} \subseteq \mathcal{S}$ such that $|\mathcal{T}| \leq K$ and $\bigcup \mathcal{T} = \bigcup \mathcal{S}$? Suppose there does exist such a subset $\mathcal{T}$, then our proof is simply the set $\mathcal{T}$ as all a verification algorithm needs to do is check that $|\mathcal{T}| \leq K$ and $\bigcup \mathcal{T} = \bigcup \mathcal{S}$ which is easy to do in polynomial time. Hence, the set cover problem is in **NP**. It can be solved in exponential time using a brute-force search algorithm, but it is not known whether it can be solved in polynomial time (*i.e.* whether or not it belongs to **P**).

- **Hamiltonian path problem** $\in$ **NP.** Given a graph does it have a Hamiltonian path (a path that visits each vertex exactly once)? Suppose there does exist a Hamiltonian path, then our proof is the Hamiltonian path itself since checking that it is Hamiltonian takes polynomial time. Hence, the Hamiltonian path problem is in **NP**. It can be solved in exponential time using a brute-force search algorithm, but it is not known whether it can be solved in polynomial time (*i.e.* whether or not it belongs to **P**).

- **Shortest path problem** $\in$ **P.** Given a graph, two vertices u, v and some $K \in \mathbb{N}$, does there exist a path from u to v of length less that K? Finding a shortest path can be done in polynomial time, so a verification algorithm needs no proof: it just computes the length of the shortest path. Hence, the shortest path problem is in $\mathbf{P} \subseteq \mathbf{NP}$.

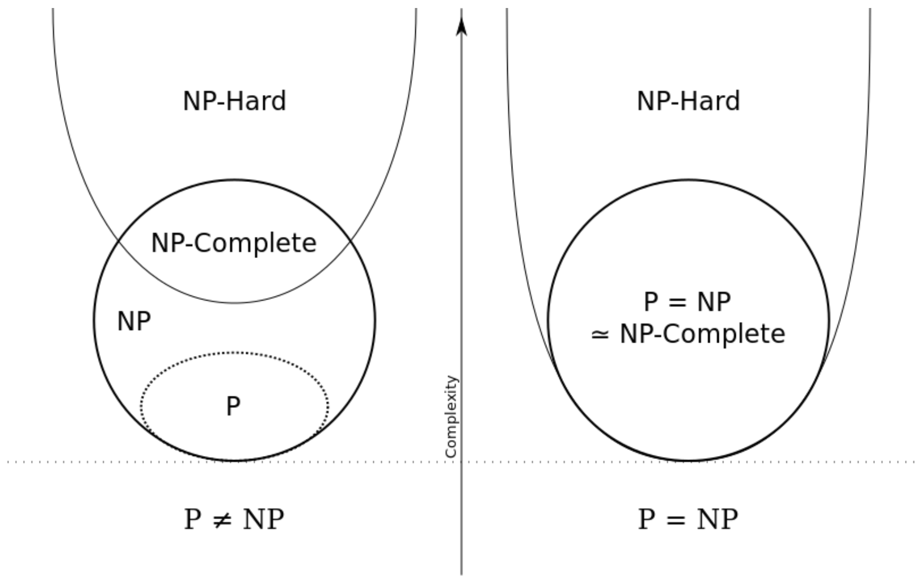
NP-complete: An **NP** problem is **NP-complete** if it can be used to simulate every other **NP** problem. Intuitively, **NP-complete** problems are the hardest **NP** problems. If we could solve one **NP-complete** problem in polynomial time, we would be able to all **NP** problems in polynomial time, which would prove that $\mathbf{P} = \mathbf{NP}$.

Example: The *Hamiltonian path problem* presented above has been shown to be **NP-complete**.

NP-hard: A problem (not necessarily a decision problem) is **NP-hard** if an algorithm for solving it can be translated in polynomial time into one for solving any **NP** problem. Equivalently, a problem is **NP-hard** if any **NP** problem can be reduced to it in polynomial time. Informally, **NP-hard** problems are 'at least as hard as the hardest problems in **NP**'.

Examples: For separable datasets, *finding a linear classifier which perfectly classifies the data* is a problem which can be solved in polynomial time. However, for non-separable datasets, *finding a linear classifier with minimum zero-one classification error* is an **NP-hard** problem.

Aside: Note that not every decision problem belongs to **P** or **NP**, some decision problems can neither be solved or verified in polynomial time, but they can in exponential time.



Source for figure: 'NP (complexity)' `https://en.wikipedia.org/wiki/NP_(complexity)`.