2 Brief Introduction/Recap to Computational Complexity

The running time of an algorithm is the number of elementary operations that it performs.

- $+, -, \cdot, /$
- comparisons : $\leq, <, =, \ldots$
- assignments : $a = b$ with $b \in \mathbb{Q}$

Example

Scalar product between two vectors $x, y \in \mathbb{Z}^n$.

$$\langle x, y \rangle = \sum_{i=1}^{n} x_i y_i$$

$$\left.\begin{array}{lll} \#\text{ multiplications} & : & n \\ \#\text{ additions} & : & n-1 \end{array}\right\} \text{total \# of elem operations: } 2n-1$$

Example

Let $x, y, z \in \mathbb{Z}^n$. How many elementary operations does it take to compute $x \cdot y^T \cdot z$?

$$\begin{pmatrix} \\ x \\ \\ \end{pmatrix} \cdot \begin{pmatrix} & y^T & \end{pmatrix} \cdot \begin{pmatrix} \\ z \\ \\ \end{pmatrix}$$

Option 1

$$x \cdot y^T \cdot z = (x \cdot y^T) \cdot z \text{, i.e.,}$$

first compute $x \cdot y^T$ and then $(x \cdot y^T) \cdot z$

$$x \cdot y^T = \begin{pmatrix} x_1 y_1 & \cdots & x_1 y_n \\ \vdots & & \vdots \\ x_n y_1 & \cdots & x_n y_n \end{pmatrix} \quad : \; n^2 \text{ multiplications}$$

$(x \cdot y^T) \cdot z$; n scalar products

$$\rightarrow n \cdot \underbrace{(2n-1)}_{\substack{\#\text{ elem operations} \\ \text{of scalar product in } \mathbb{Z}^n}} = 2n^2 - n$$

$\Longrightarrow$ Total # of elem. operations : $\underline{3n^2 - n}$

Option 2

$$x \cdot y^T \cdot z = x \cdot (y^T \cdot z) \text{ , i.e.,}$$

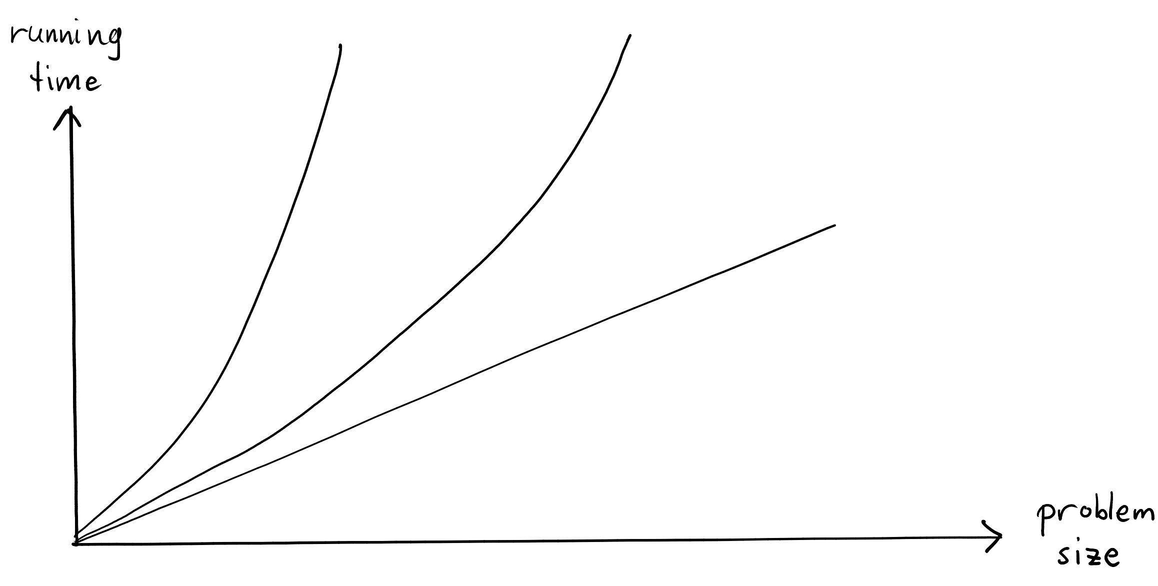
first compute $y^T z$ and then $x \cdot (y^T z)$.

$$\left.\begin{array}{lll} y^T z & : & 2n-1 \text{ elem. operations} \\ x \cdot (y^T z) & : & n \text{ elem. operations} \end{array}\right\} \; 3n-1 \text{ elem. operations}$$

Landau notation

Above way to measure running times is too fine-grained.
We want to know how an algorithm scales when problem size increases.

$\hookrightarrow$ For this, it suffices to determine the running time up to constant factors.



Definition 2.2: Landau notation

Let f and g be two functions.

(i) We write $f = O(g)$ if and only if

$$\exists M > 0, c > 0 \quad \text{such that} \quad |f(s)| \leq c \cdot |g(s)| \quad \forall s \geq M \ .$$

(ii) We write $f = \Omega(g)$ if and only if $g = O(f)$.

(iii) We write $f = \Theta(g)$ if and only if $f = O(g)$ and $f = \Omega(g)$.

$3n-1 = O(n)$ $\qquad 3n-1 \leq 3n$

$n + \log n = O(n)$ $\qquad n + \log n \leq 2n$

$n + 10^6 = O(n)$ $\qquad n + 10^6 \leq 2n$ for $n \geq 10^6$

$$\sqrt{n} + (\log n) \cdot n^{2/3} = O(n^{3/4}) = O(n)$$

$n^2 \log n = O(n^3) \iff n^3 = \Omega(n^2 \log n)$

$$n^2 + \log n = \Theta(n^2)$$

Worst-case assumption

When analyzing the running time of an algorithm for a certain problem class, we want to get a running time upper bound that holds for any problem instance of a certain size.

Example

Given : $a_1, a_2, a_3, \dots, a_n \in \mathbb{Z}$

Task : Decide whether $a_1 = a_2 = \dots = a_n$.

One possible algorithm:

```
for i = 2, 3, ..., n
    if a_i ≠ a_1
        return false
return true
```

What's the running time of this algorithm?

If $a_1 \neq a_2$ $\longrightarrow$ running time : $\Theta(1)$

If $a_1 = a_2 = \dots = a_n$ $\longrightarrow$ " " : $\Theta(n)$

$\rightarrow$ Clearly, the running time depends on the input, even when fixing n.

The running time is measured with respect to worst-case instance.

$\Longrightarrow$ running time : $O(n)$

$\boxed{\text{Input size}}$

# of bits needed to save problem instance.

<u>Saving $a \in \mathbb{Z}_{\geq 0}$ in binary</u>:

$a = 9$ 1001

8 4 2 1

We need $\lceil \log_2(a+1) \rceil$ bits.

$a=1$	1
$a=2$	10
	11
⋮	100
	101
	110
	111
$a=8$	1000

To also save negative numbers, we can use extra bit for sign.

$\rightarrow$ $\underline{\lceil \log_2(a+1) \rceil + 1}$

Example

Task : Compute scalar product $\langle x, y \rangle$ with $x, y \in \mathbb{Z}^n$.

What is the input size of this problem?

$$\langle \text{input} \rangle = \Theta\left(\sum_{i=1}^{n} \underbrace{\left(\lceil \log(|x_i|+1) \rceil + 1 \right)}_{\#\text{ bits to save } x_i} + \sum_{i=1}^{n} \underbrace{\left(\lceil \log(|y_i|+1) \rceil + 1 \right)}_{\#\text{ bits to save } y_i} \right)$$

$$= \Omega(n)$$

Definition 2.3: Polynomial algorithms and problems

An algorithm is *polynomial* or *efficient* if its running time $f(\langle \text{input} \rangle)$ is bounded by a polynomial in the size of the input, i.e., there is a polynomial g such that

$$f = O(g) \ .$$

A problem is *solvable in polynomial time* if it can be solved by a polynomial algorithm.

$\boxed{\text{Complexity classes}}$ ← When talking about complexity classes, we restrict ourselves to decision problems.

yes/no - problems

$\mathcal{P}$: Class of all decision problems that can be solved in polynomial time.

$\mathcal{NP}$: Class of all decision problems for which a yes-instance can be certified efficiently.

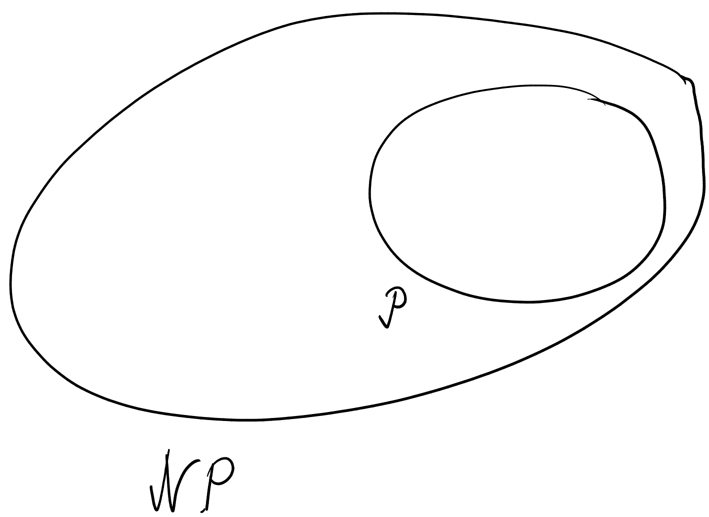
This means that there is a polynomial-size certificate and an efficient algorithm such that, given the instance and the certificate, the algorithm can verify that it is a yes-instance.

Example of a problem in $\mathcal{NP}$

Input : $x \in \mathbb{Z}_{\geq 2}$

Task : Decide whether x is not prime.

$$x = a \cdot b \qquad a, b \in \mathbb{Z}_{\geq 2}$$


P
NP