Lecture 2 - Computational Complexity and Algorithm Design

Lan Peng, Ph.D.

School of Management, Shanghai University, Shanghai, China

``import numpy as np''

1 Preliminaries

1.1 How difficult is to solve an integer program?

- Hard!
- What is "hard"? $\Rightarrow$ measurement of difficulties.
- $\Rightarrow$ computational complexity $\Rightarrow$ the art of classifying computational problems according to their resource usage.
- What is "algorithm"? $\Rightarrow$ Given a string s, an algorithm takes the input s and within finite number of steps, returns an output string s'.
- What is "resource usage"? $\Rightarrow$ How much *time*/*space* it takes for an algorithm to process s?

1.2 Asymptotic Notation

O-Notation Let $f(n)$ and $g(n)$ be real functions. We say $f(n) = O(g(n))$ if there exist positive constants n_0 and c such that

$$0 \leq f(n) \leq cg(n), \quad \forall n \geq n_0$$

A common trick that finds $O(g(n))$ for $f(n)$ is to find function $g(n)$ such that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \quad \text{(constant)}$$

The O-Notation is referred as the asymptotic upper bound. The O-Notation is more widely used, since it can provide a clear estimate of the worst case time/space complexity of an algorithm that exclusively considers the critical grow factors.

Ω-Notation Let $f(n)$ and $g(n)$ be real functions. We say $f(n) = \Omega(g(n))$ if there exist positive constants n_0 and c such that

$$0 \leq cg(n) \leq f(n), \quad \forall n \geq n_0$$

The Ω-Notation is referred as the asymptotic lower bound. In addition:

Theorem 1.1. $f(n) = O(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$

Θ-Notation Let $f(n)$ and $g(n)$ be real functions. We say $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that

$$c_1 g(n) \leq f(n) \leq c_2 g(n), \quad \forall n \geq n_0$$

The Θ-Notation is referred as the asymptotic tight bound. In addition:

Theorem 1.2. *$f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$*

2 P v.s. NP

2.1 P and NP

Decision problem A decision problem X is the set of strings on which the output is yes, i.e., $s \in X$ iff the correct output for the input s is 1 (yes).

Example. The optimization version of the knapsack problem: Given a set of items N, each item $i \in N$ having a weight w_i and a value c_i, a capacity b and a threshold value k, find a collection of items $S \subseteq N$ of the maximum value whose total weight is less than or equal to b. The output is a set S.

The decision version of the knapsack problem: Given a set of items N, each item $i \in N$ having a weight w_i and a value c_i, a capacity b and a threshold value k, determine if there exists a collection of items $S \subseteq N$ whose total weight is less than or equal to b and its total value is at least k. The output is a boolean value, True/False.

If we have an algorithm to solve the optimization version of knapsack problem, we can immediately solve the decision version.

Class P

Definition 2.1 (polynomial running time)**.** Algorithm A has polynomial running time if there is a polynomial function $p()$ such that for every string s, A terminates on s in at most $p(|s|)$ steps.

Definition 2.2 (Class P)**.** The complexity class P is the set of decision problems X that can be solved in polynomial time.

That is, there is a known algorithm that provides the solution of any instance of size n in time $n^{O(1)}$.

Example. The following problems are known as in class P:

- Shortest path problem
- Maximum flow problem
- Spanning tree problem
- Linear programming (but not the simplex method)
- Assignment problem

Class NP

Definition 2.3 (certifier, certificate)**.** B is an efficient certifier for a problem X if

- B is a polynomial-time algorithm that takes two input strings s and t
- There is a polynomial function p such that $s \in X$ if and only if there is a string t such that $|t| \leq p(|s|)$ and $B(s,t) = 1$

The string t such that $B(s,t) = 1$ is called a certificate.

Example. Q: Given a graph $G = (V, E)$ with a Hamiltonian cycle, how can one convince another there exists a Hamiltonian cycle?

A: One can send a string (certificate) to another, the other will run an algorithm (certifier) to check if the string represents a Hamiltonian cycle.

Definition 2.4 (class NP)**.** The complexity class NP is the set of all problems for which there exists an polynomial-time certifier.

P v.s. NP

- Is $P = NP$? This is the most famous and fundamental open problem in computer science.
- Most people believe $P \neq NP$.
- If $P = NP$, that means if one can check a solution in polynomial time, one can solve it in polynomial time.
- We are sure that $P \subseteq NP$

2.2 Polynomial Time Reduction

Polynomial-time reducible Given a black box algorithm A that solves a problem X, if any instance of a problem Y can be solved using a polynomial number of standard computational steps, plus a polynomial number of calls to A, then we say Y is polynomial-time reducible to X, denoted as $Y \leq_P X$.

Suppose $Y \leq_P X$:

- If X can be solved in polynomial time, then Y can be solved in polynomial time.
- If Y cannot be solved in polynomial time, then X cannot be solved in polynomial time.

X is at least as hard as Y.

2.3 NP-Completeness

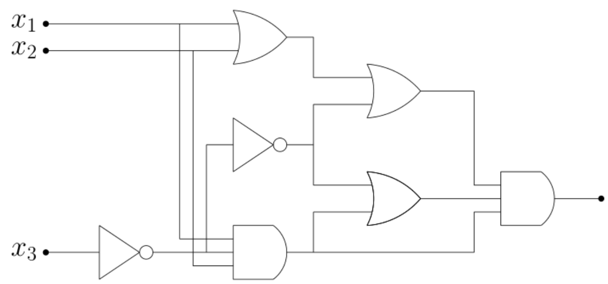
NP-Completeness A problem X is called NP-complete if

- $X \in NP$, and
- $Y \leq_P X$, for every $Y \in NP$

If any NP-complete problem can be solved in polynomial time, then $P = NP$. Unless $P = NP$, a NP-complete problem cannot be solved in polynomial time.

Circuit-Sat The Circuit Satisfiability problem is the first NP-complete problem.

Theorem 2.1 (Cook's Theorem)**.** *SAT is NP-Complete*



- key fact: any algorithm that takes n bits as input and outputs 0/1 with running time $T(n)$ can be converted into a circuit of size $p(T(n))$ for some polynomial function $p(\cdot)$.
- Then, we can show that any problem $Y \in NP$ can be reduced to Circuit-Sat

Reductions of NP-Complete problems We will show a SAT $\leq_P$ 3-SAT

Definition 2.5 (3-CNF)**.** 3-CNF is a special case of formula:

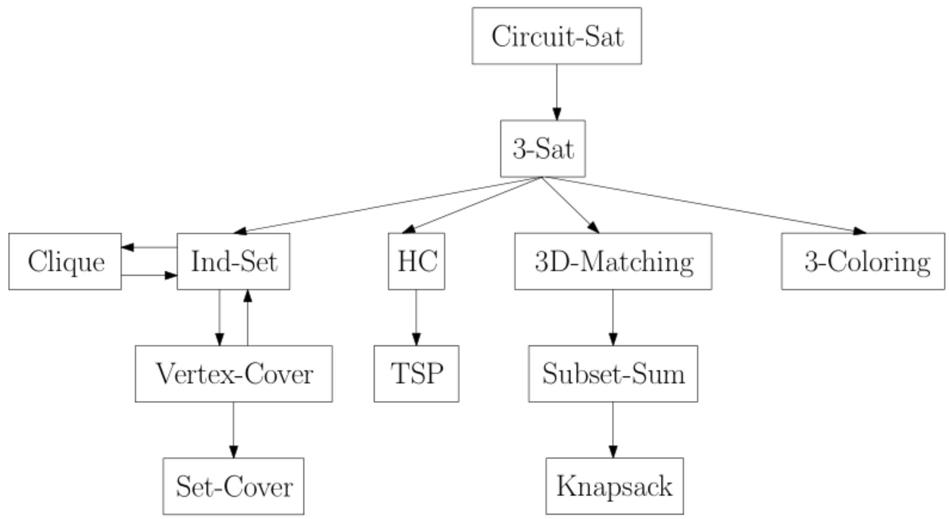
- Boolean variables: $x_1, x_2, \cdots, x_n$
- Literals: x_i or $\bar{x}_i$
- Clause: disjunction ("or") of at most 3 literals
- 3-CNF formula: conjunctino ("and") of clauses.

Example. This is a 3-CNF: $(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x_3} \vee x_4)$

To satisfy a 3-CNF, we need to satisfy all clauses. To satisfy a clause, we need to satisfy at least 1 literal. Associate every wire with a new variable, the circuit will be equivalent to a formula. Each formula can be transformed into a 3-CNF.

The SAT is satisfiable iff the 3-CNF is satifiable, and the size of the 3-CNF formula is polynomial in the size of the circuit. Thus SAT $\leq_P$ 3-SAT.

For other problems, here is a polynomial-reducible relation graph for reference.



3 Algorithm Analysis and Design

3.1 Three programming paradigms

Greedy Algorithm

- Make a greedy choice
- At each step, make an irrevocable decision using a “reasonable” strategy
- Prove that the greedy choice is safe
- Show that the remaining task after applying the strategy is to solve a/many **smaller instance**(s) of the same problem
- Usually for optimization problems.

Dynamic Programming

- Break up a problem into many **overlapping** sub-problems
- Build solutions for larger and larger sub-problems
- Use a table/dictionary to store solutions for sub-problems for reuse

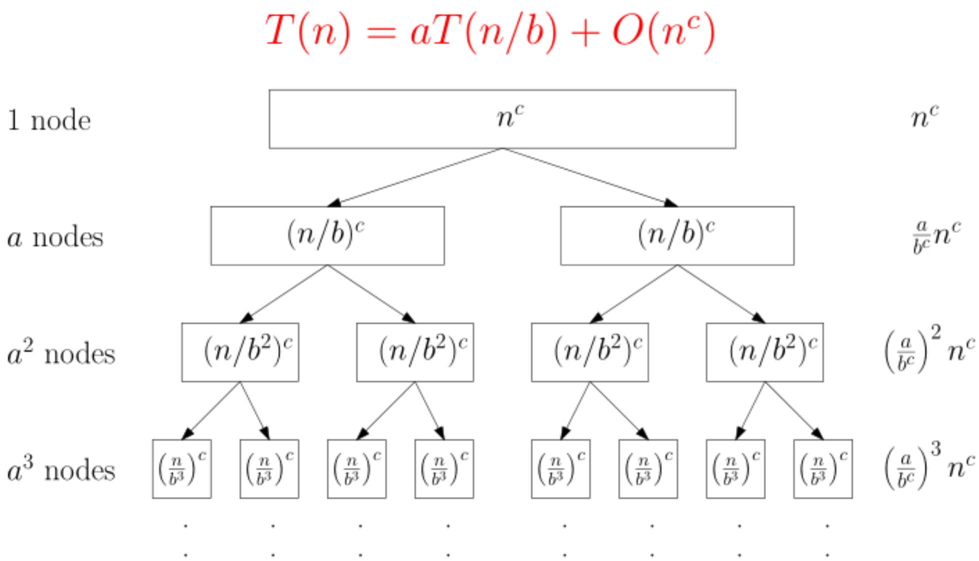
Divide and Concur

- Break a problem into many **independent** sub-problems
- Solve each sub-problem separately
- Combine solutions for sub-problems to form a solution for the original one
- Usually used to design more efficient algorithms

3.2 Master Theorem

Theorem 3.1. *$T(n) = aT(n/b) + O(n^c)$, where $a \geq 1, b > 1, c \geq 0$ are constants. Then,*

$$T(n) = \begin{cases} O(n^{\log_b a}) & \textit{if } c < \log_b a \\ O(n^c \log n) & \textit{if } c = \log_b a \\ O(n^c) & \textit{if } c > \log_b a \end{cases}$$



- $c < \log_b a$: bottom-level dominates, $(\dfrac{a}{b^c})^{\log_b n} n^c = O(n^{\log_b a})$
- $c = \log_b a$: all levels have same time, $n^c \log_b n = O(n^c \log n)$
- $c > \log_b a$: top-level dominates, $O(n^c)$

4 Some examples

4.1 Minimum Spanning Tree Problem

Basic Concepts

Example. A company wants to build a communication network for their offices. For a link between office v and office w, there is a cost c_{vw}. If an office is connected to another office, then they are connected to with all its neighbors. Company wants to minimize the cost of communication networks.

Definition 4.1 (Minimum spanning tree problem)**.** Given a connected graph graph G, and a cost $C_e, \forall e \in E$, find a minimum cost spanning tree of G

Kruskal's Algorithm Let's start with a greedy algorithm

Algorithm 1 MST-Greedy(G, w)

$F = \emptyset$
Sort edges in E in non-decreasing order of weight w
for $e = (u, v)$ **do**
  if u and v are not connected by a path of edge in F **then**
    $F = F \cup \{(u, v)\}$
return F

The Kruskal's algorithm is as follows

Algorithm 2 Kruskal's Algorithm

$F \leftarrow \emptyset$
$S \leftarrow \{\{v\} : v \in V\}$
Sort edges in E in non-decreasing order of weight w
for $e = (u, v)$ **do**
  $S_u \leftarrow$ the set in S containing u
  $S_v \leftarrow$ the set in S containing v
  if $S_u \neq S_v$ **then**
    $F \leftarrow F \cup \{(u, v)\}$
    $S \leftarrow S \setminus \{S_u\} \setminus \{S_v\} \cup \{S_u \cup S_v\}$
return F

Prim's Algorithm Let's start with another greedy algorithm

Algorithm 3 MST-Greedy(G, w)

$S \leftarrow \{s\}$, where s is arbitary vertex in V
$F \leftarrow \emptyset$
while $S \neq V$ **do**
  $(u, v) \leftarrow$ the lightest edge between S and $V \setminus S$, where $u \in S$ and $v \in V \setminus S$
  $S \leftarrow S \cup \{v\}$
  $F \leftarrow F \cup \{(u, v)\}$

The Prim's algorithm is as follows

Algorithm 4 Prim's Algorithm

$s \leftarrow$ arbitary vertex in G
$S \leftarrow \emptyset$, $d(s) \leftarrow 0$ and $d(v) \leftarrow \infty$ for every $v \in V \setminus \{s\}$
while $S \neq V$ **do**
  $u \leftarrow$ vertex in $V \setminus S$ with minimum $d(u)$
  $S \leftarrow S \cup \{u\}$
  for Each $v \in V \setminus S$ such that $(u, v) \in E$ **do**
    if $w(u, v) < d(v)$ **then**
      $d(v) \leftarrow w(u, v)$
      $\pi(v) \leftarrow u$
return $\{(u, \pi(u)) | u \in V \setminus \{s\}\}$

4.2 Knapsack Problem

Integer program model

$$\max \quad \sum_{i \in S} v_i \tag{1}$$

$$\text{s.t.} \quad \sum_{i \in S} w_i \leq W \tag{2}$$

$$w_i \geq 0, \forall i \in S \tag{3}$$

Solve knapsack problem by dynamic programming

- Let $opt[i, W']$ be the optimum value when budget is W' and the items are $\{1, 2, 3, \cdots, i\}$.
- If $i = 0, opt[i, W'] = 0$ for every $W' = 0, 1, \cdots, W$. Then
- $opt[i, W'] = \begin{cases} 0, & i = 0 \\ opt[i-1, W'], & i > 0, w_i > W' \\ \max\left\{opt[i-1, W'], opt[i-1, W' - w_i] + v_i\right\}, & i > 0, w_i \leq W' \end{cases}$

4.3 Fibonacci Numbers

Fibonacci sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, \cdots \tag{4}$$

Naive algorithm Use recursion (in a stupid way)

Algorithm 5 Fib(n)

```
if n = 0 then
    return 0
if n = 1 then
    return 1
return Fib(n − 1) + Fib(n − 2)
```

The runtime is exponential $O(2^n)$. This is stupid.

Reasonable algorithm Use dynamic programming

Algorithm 6 Fib(n)

```
F(0) ← 0
F(1) ← 1
for i ← 2 to n do
    F(i) ← F(i − 1) + F(i − 2)
return F(n)
```

The runtime is $O(n)$.

Even better algorithm Notice that

$$\begin{aligned}
\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} \\
\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^2 \begin{bmatrix} F_{n-2} \\ F_{n-3} \end{bmatrix} \\
&\cdots \\
\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} &= \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix}
\end{aligned}$$

First define power(n)

Algorithm 7 power(n)

if $n = 0$ **then**
    return $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
$R \leftarrow$ power(n/2.floor())
$R \leftarrow R \times R$
if n is odd number **then**
    $R \leftarrow R \times \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$
return R

Then the Fibonacci sequence is calculated by

Algorithm 8 Fib(n)

if $n = 0$ **then**
    return 0
$M \leftarrow$ power($n - 1$)
return M[1][1]

The time complexity is $T(n) = T(n/2) + O(1) = O(\log n)$

4.4 Multiplications

Polynomial Multiplication Given two polynomials of degree $n-1$, we need to find the product of two polynomials.

Example. Let $A = (4, -5, 2, 3)$ and $B = (-5, 6, -3, 2)$ as input. The expected output $C =$

$(-20, 49, -52, 20, 2, -5, 6)$, since

$$\begin{aligned}
&(3x^3 + 2x^2 - 5x + 4) \times (2x^3 - 3x^2 + 6x - 5) \\
=&6x^6 - 9x^5 + 18x^4 - 15x^3 + 4x^5 - 6x^4 + 12x^3 - 10x^2 \\
&- 10x^4 + 15x^3 - 30x^2 + 25x + 8x^3 - 12x^2 + 24x - 20 \\
=&6x^6 - 5x^5 + 2x^4 + 20x^3 - 52x^2 + 49x - 20
\end{aligned}$$

A naive algorithm for polynomial multiplication is as follows

Algorithm 9 naivePolyMultiply(A, B, n)

Let $C[k] = 0$ for every $k = 0, 1, \cdots, 2n-2$
for $i \leftarrow 0$ to $n-1$ **do**
    for $j \leftarrow 0$ to $n-1$ **do**
        $C[i+j] \leftarrow C[i+j] + A[i] \times B[j]$
return C

Obviously, the running time is $O(n^2)$. Now we try to use divide-and-conquer to improve. For a polynomial $p(x)$ with degree of $n-1$, denote

$$p(x) = p_H(x)x^{\frac{n}{2}} + p_L(x)$$

then, $p_H(x)$ and $p_L(x)$ are polynomials of degree $n/2 - 1$. Consider

$$pq = (p_H x^{\frac{n}{2}} + p_L)(q_H x^{\frac{n}{2}} + q_L) = p_H q_H x^n + (p_H q_L + p_L q_H) x^{\frac{n}{2}} + p_L q_L$$

The recurrence time is

$$T(n) = 4T(n/2) + O(n) = O(n^2)$$

still not good... Consider

$$\begin{aligned}
pq &= (p_H x^{\frac{n}{2}} + p_L)(q_H x^{\frac{n}{2}} + q_L) \\
&= p_H q_H x^n + ((p_H + p_L)(q_H + q_L) - p_H q_H - p_L q_L) x^{\frac{n}{2}} + p_L q_L
\end{aligned}$$

The new recurrence time is

$$T(n) = 3T(n/2) + O(n) = O(n^{\log_2 3}) = O(n^{1.585})$$

Better!!! The algorithm is given as follows

Algorithm 10 polyMultiply(A, B, n)

1: $A_L \leftarrow A[0..n/2-1]$, $A_H \leftarrow A[n/2..n-1]$
2: $B_L \leftarrow B[0..n/2-1]$, $B_H \leftarrow B[n/2..n-1]$
3: $C_L \leftarrow polyMultiply(A_L, B_L, n/2)$
4: $C_H \leftarrow polyMultiply(A_H, B_H, n/2)$
5: $C_M \leftarrow polyMultiply(A_L + A_H, B_L + B_H, n/2)$
6: Initialize $C \leftarrow [0..0]$ $(2n-1)$ 0s
7: **for** $i \leftarrow 0$ to $n-2$ **do**
8: $\quad C[i] \leftarrow C[i] + C_L[i]$
9: $\quad C[i+n] \leftarrow C[i+n] + C_H[i]$
10: $\quad C[i+n/2] \leftarrow C[i+n/2] + C_M[i] - C_L[i] - C_H[i]$
11: **return** C

Notice that the name of Algorithm 10 appears inside itself (line 3, 4, 5). This is referred as *recursion*. The key to recursion functions is to break complex problems into smaller instances of the **same** problems, so we need to "think beyond timescape".

Matrix multiplication Given two matrices $\mathbf{A} \in \mathbb{R}^{n\times}$ and $\mathbf{B} \in \mathbb{R}^{n\times}$, we need to find the production, $\mathbf{C} = \mathbf{AB} \in \mathbb{R}^{n\times}$.

A naive algorithm for matrix multiplication is as follows

Algorithm 11 naiveMatMultiply(A, B, n)

for $i \leftarrow 1$ to n **do**
$\quad$ **for** $j \leftarrow 1$ to n **do**
$\quad\quad C[i,j] \leftarrow 0$
$\quad\quad$ **for** $k \leftarrow 1$ to n **do**
$\quad\quad\quad C[i,j] \leftarrow C[i,j] + A[i,k] \times B[k,j]$
return C

The running time is $O(n^3)$. Try divided-and-conquer. Let

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{bmatrix}$$

each sub-matrix is $n/2 \times n/2$. Then

$$\mathbf{C} = \begin{bmatrix} \mathbf{A}_{11}\mathbf{B}_{11} + \mathbf{A}_{12}\mathbf{B}_{21} & \mathbf{A}_{11}\mathbf{B}_{12} + \mathbf{A}_{12}\mathbf{B}_{22} \\ \mathbf{A}_{21}\mathbf{B}_{11} + \mathbf{A}_{22}\mathbf{B}_{21} & \mathbf{A}_{21}\mathbf{B}_{12} + \mathbf{A}_{22}\mathbf{B}_{22} \end{bmatrix}$$

Still not good enough because $T(n) = 8T(n/2) + O(n)$.

Strassen's Algorithm In 1969, the first matrix multiplication algorithm with time complexity lower than $O(n^3)$ was introduced by Volker Strassen, the famous Strassen's Algorithm.

For $\mathbf{A}, \mathbf{B}, \mathbf{C}$, let

$$\begin{aligned}
\mathbf{M}_1 &= (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22}) \\
\mathbf{M}_2 &= (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11} \\
\mathbf{M}_3 &= \mathbf{A}_{11}(\mathbf{B}_{12} - \mathbf{B}_{22}) \\
\mathbf{M}_4 &= \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11}) \\
\mathbf{M}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22} \\
\mathbf{M}_6 &= (\mathbf{A}_{21} - \mathbf{A}_{11})(\mathbf{B}_{11} + \mathbf{B}_{12}) \\
\mathbf{M}_7 &= (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22})
\end{aligned}$$

Then,

$$\begin{aligned}
\mathbf{C}_{11} &= \mathbf{M}_1 + \mathbf{M}_4 - \mathbf{M}_5 + \mathbf{M}_7 \\
\mathbf{C}_{12} &= \mathbf{M}_3 + \mathbf{M}_5 \\
\mathbf{C}_{21} &= \mathbf{M}_2 + \mathbf{M}_4 \\
\mathbf{C}_{22} &= \mathbf{M}_1 - \mathbf{M}_2 + \mathbf{M}_3 + \mathbf{M}_6
\end{aligned}$$

In total, $T(n) = 7T(n/2) + O(n) = O(n^{\log_2 7})$.

Why is this important? Matrix multiplication has been widely used in machine learning such as calculating convolutions. For $n > 300$, the Strassen's algorithm will run significantly faster than the naive algorithm.

There are some algorithms even faster than Strassen, e.g., the Coppersmith-Winograd matrix multiplication algorithm.