The Nepali Mathematical Sciences Report
Regd No.: 53/033-034, ISSN: 2392–411X
http://www.journal.cdmathtu.edu.np

Year: 2025, Volume: 42, No. 2
Pages: 72–90
https://doi.org/10.3126/nmsr.v42i2.88518

**Research Article**

# Computational Problem and Its Time Complexity

Iswar Mani Adhikari[1], Shiva Prakash Gupta[2], and Ram Chandra Dhungana[*3]

[1]Tribhuvan University, Prithvi Narayan Campus, Pokhara, Nepal
adhikariim35@gmail.com
[2]Tribhuvan University, Tri-Chandra Campus, Kathmandu, Nepal
shivaprasadgupta99@gmail.com
[*3]Kathmandu University, Department of Mathematics, Dhulikhel, Nepal
dhungana.ramchandra@gmail.com

Received: 23 August, 2025 Accepted: 17 December, 2025 Published Online: 30 December, 2025

**Abstract**

Computational problems and their complexity analysis play a crucial role in computer science, mathematics, and real-world applications. Such tasks can be solved using computational processes known as an algorithms. Such a process consists of a set of inputs and corresponding outputs that satisfy certain conditions. Search, decision, and optimization problems are different variants of the computational problems. Complexity theory studies the measure of efficiency in solving such problems. Such a measure is expressed in terms of space and time. Regarding time complexity, its major variants include: (i) P: polynomial time, (ii) NP: nondeterministic polynomial time, (iii) NP-complete, and (iv) NP-hard.

Understanding computational complexity is essential for designing efficient algorithms, optimizing resources, and making informed decisions about problem-solving approaches. This work offers a comprehensive overview of computational problems and their associated complexities and opens a wide horizon for in-depth and broader research in this area.

**Keywords:** Computational problem, algorithm, complexity, approximation, deterministic, nondeterministic.
**AMS(MOS) Subject Classification:** 68Q25, 68Q17, 03D15.

## 1 Introduction

Computational problems can be solved using a computational process, typically through an algorithm. The computational model underlies many research areas in natural as well as physical sciences, and even in mathematical biology and neuroscience. The running time might be different

*Corresponding author

on different computer devices for the same algorithm. For this, we need to establish an agreement that the run algorithm is modeled theoretically. Mainly, the computational problems are classified as decision, optimization,and search problems. Decision problems are those with a yes/no answer like checking a number for prime. The optimization problems are to get the best one from a set of feasible solutions like to determine the shortest path between two cities. The search problems require finding a solution that satisfies given constraints, like the solution of a Sudoku puzzle.

The field of computational complexity rigorously examines the efficiency of problem-solving, focusing primarily on time and space resources. Understanding these complexities empowers us to tackle some of the most daunting challenges in computer science. Depending on the time complexity, computational problems are classified: (i) P: Deterministic polynomial Time, (ii) NP: Nondeterministic Polynomial Time, (iii) EXP: Exponential Time, (iv) NP-Complete, and (v) NP-Hard.

Section 2 presents the fundamentals of the computational problems. Different variants of the computational complexities and their characteristics are in Section 3. The most challenging and the interesting issues as the P verses NP is in Section 4 and is followed by the computational problems and their complexities in Section 5. Approximations schemes on such problems in Section 6. Finally, we conclude it in Section 7.

## 2 Fundamentals

Figure 1: Sorting a card using insertion sort

**Computation and algorithm:** In different computational problems, an algorithm produces certain output in a finite number of steps. Consider a sorting problem-

**Input**: A sequence, $< k_1, k_2, ..., k_n >$ having n numbers..
**Output:** A reordering of $< k_1, k_2, ..., k_n >$ in the form $k_1^{'} \leq k_2^{'} \leq k_3^{'} ... \leq k_n^{'}$.
For the input sequence, $< 21, 43, 39, 79, 63 >$ a sorting algorithm returns to $< 21, 39, 43, 63, 79 >$.

**Running time of an algorithm:** Consider an algorithm as an INSERTION-SORT for sorting a small number of elements like sorting of playing cards properly as in Figure 1.

The time required for the INSERTION-SORT procedure is strongly influenced by the characteristics

**Algorithm 1:** An INSERTION-SORT (X).

```
1: for $j = 2$ to $X$ do
2:     key = $X[j]$
3:     /*Insert X[j] into X[1, 2, 3, ..., j-1]*/
4:     $i = j - 1$
5:     while $i > 0$ and $X[i] > key$ do
6:         $X[i + 1] = X[i]$
7:         $i = i - 1$
8:     end while
9:     $X[i + 1] = key$
10: end for
11: Output: The final sorted array X.
```

of the input. For example, sorting of hundred numbers will take considerably longer than sorting of four. Furthermore, even two sequences of the same size can have markedly different sorting times, depending on how close they are to being sorted. As the input size increases, the running time will certainly increase. It might be machine-dependent, but it is better to be as machine-independent as possible. Comments are written in /*...*/ are not executable statements, and, therefore, do not consume any time. In above example of INSERTION-SORT, we have,

Table 1: Running time of an algorithm

| Line No. | cost | time |
|---|---|---|
| 1 | $k_1$ | n |
| 2 | $k_2$ | n-1 |
| 3 | 0 | |
| 4 | $k_4$ | n-1 |
| 5 | $k_5$ | $\sum_{j=2}^{n} t_j$ |
| 6 | $k_6$ | $\sum_{j=2}^{n}(t_j - 1)$ |
| 7 | $k_7$ | $\sum_{j=2}^{n}(t_j - 1)$ |
| 8 | $k_8$ | n-1 |

Here, the total running time becomes,

$$T(n) = k_1 n + k_2(n-1) + k_4(n-1) + k_5 \sum_{j=2}^{n} t_j + k_6 \sum_{j=2}^{n}(t_j - 1) + k_7 \sum_{j=2}^{n}(t_j - 1) + k_8(n-1).$$

If the array is already sorted, it gives the best case with,

$$T(n) = k_1 n + k_2(n-1) + k_4(n-1) + k_5(n-1) + k_8(n-1) = (k_1 + k_2 + k_4 + k_5 + k_8)n - (k_2 + k_4 + k_5 + k_8) = an + b,$$

a linear function in $n$.

The worst case is, at its reverse order sorting, i.e.,

$$T(n) = k_1n+k_2(n-1)+k_4(n-1)+k_5\left(\frac{n(n+1)}{2}-1\right)+k_6\left(\frac{n(n-1)}{2}\right)+k_7\left(\frac{n(n-1)}{2}\right)+k_8(n-1)$$

$$= \frac{1}{2}\left(2k_5+2k_6+2k_7\right)n^2+\frac{1}{2}\left(2k_1+2k_2+2k_4+k_5-k_6-k_7+2k_8\right)n-(k_2+k_4+k_5+k_8) = an^2+bn+c$$

a quadratic function in $n$.

**Example 2.1.** A simple illustrative example for insertion sort related to above case of insertion on the sequence $5, 2, 4, 6, 1, 3$ in ascending order.

Table 2: An illustrative example of insertion sort

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 5 | ② | 4 | 6 | 1 | 3 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 2 | 5 | 4 | 6 | 1 | 3 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 2 | 5 | ④ | 6 | 1 | 3 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 2 | 4 | 5 | 6 | 1 | 3 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 2 | 4 | 5 | 6 | ① | 3 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 2 | 4 | 5 | ① | 6 | 3 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 2 | 4 | ① | 5 | 6 | 3 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 2 | ① | 4 | 5 | 6 | 3 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 2 | 4 | 5 | 6 | 3 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 2 | 4 | 5 | 6 | ③ |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 2 | 4 | 5 | ③ | 6 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 2 | 4 | ③ | 5 | 6 |

| 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 |

Let $X =< 5, 2, 4, 6, 1, 3 >$. The variety in the first row represents their order. The values are sorted and appear within the circles in second row of the Table. Here, the respective iterations of the Algorithms are illustrated serially, in Table 2. The circled number is the key in each array, that is compared iteratively with the entries to its left up to final sorting.

**Growth rate of an algorithm:** The growth rate of an algorithm's execution time offers a straightforward way to assess the algorithm's efficiency, helping us evaluate the relative performance of different algorithms. This refers to the asymptotic efficiency of an algorithm. Typically, an algorithm that shows greater asymptotic efficiency will be the preferred option for nearly all input sizes except for very small ones. For details we refers to [1], [10].

**Big Θ-notation:** It gives the worst-case running time. Then, for sufficiently large n, for above *insertion sort*, it becomes, $T(n) = \Theta(n^2)$. For a given $g(n)$, the set of functions is $\Theta(g(n))$, where,

$$\Theta(g(n)) = \left\{ f(n) \ : \ \exists\, c_1 > 0,\ c_2 > 0,\ n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)\, \forall n \geq n_0. \right\}$$

For $f(n) = an^2+bn+c$ with $a, b, c \in \mathbb{R}$ and $a > 0$, we write, $\Theta(f(n)) = n^2$. Here, $\Theta(n^0) = \Theta(1)$, indicating either a continuous or a continual function.

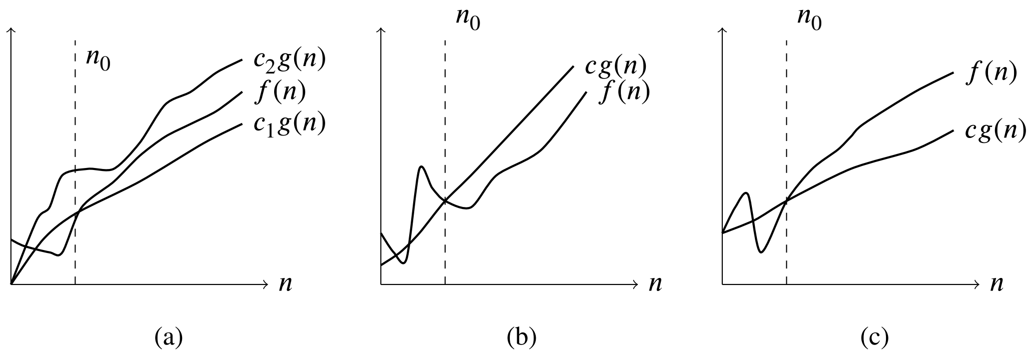

Figure 2: Asymptotic notation visualizations: (a) $\Theta(g(n))$, (b) $O(g(n))$, (c) $\Omega(g(n))$

***Big O-notation*:** It bounds asymptotically a function from both sides. But, *O-notation* is used for asymptotic upper bound, as in Figure 2(b) for this,

$$O(g(n)) = \left\{f(n) : \exists\, c > 0, n_0 > 0, \text{ such that } 0 \leq f(n) \leq cg(n) \forall n \geq n_0.\right\}$$

Here, $f(n) = O(g(n))$. Then, $f(n) = \Theta(g(n)) \implies f(n) = O(g(n))$. Hence, $\Theta(g(n)) \subseteq O(g(n))$; $\Theta$-notation is a stronger than O-notation. For any $f(n) = an^2 + bn + c$, where $a > 0$, then $f(n) \in \Theta(n^2) \implies f(n) \in O(n^2)$. The following are standard properties of Big-O notation:

- For $c \geq 0$, $c \cdot f = O(f)$ and $c \cdot O(f) = O(f)$
- For any constant $p$, $O(f_1) + \cdots + O(f_p) = O(f_1 + \cdots + f_p) = O\left(\max\{f_1, \ldots, f_p\}\right)$
- For functions $f$ and $g$, $O(f) \cdot O(g) = O(f \cdot g)$

**Big $\Omega$-notation:** It gives an asymptotic lower bound, Figure 2 (c), where,

$$\Omega(g(n)) = \left\{f(n) : \exists\, c > 0, n_0 > 0, \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0\right\}.$$

**Theorem 2.2.** *[10] Let $f(n)$ and $g(n)$ be any two functions, then*

$$f(n) = \Theta(g(n)) \quad \textit{iff} \quad f(n) = O(g(n)) \textit{ and } f(n) = \Omega(g(n)).$$

Consider, $f : \mathbb{N} \to \mathbb{R}^+$. Then it becomes, **logarithmic** if $f = O(\log n)$; **poly-logarithmic** if $f = O(\log^k n)$ for some $k \in \mathbb{N}$; **linear**, **quadratic**, or **cubic** if $f = O(n)$, $f = O(n^2)$, or $f = O(n^3)$ respectively. Likewise, it is **quasi-linear** if $f = O(n \log^k n)$ for some $k \in \mathbb{N}$; **polynomial bounded** ( **polynomial**) if $f = O(n^k)$ for some $k \in \mathbb{N}$; **super-polynomial** if $f = \Omega(n^k) \forall k \in \mathbb{N}$; **sub-exponential** if $f = O(2^{n^\varepsilon})$ for every $\varepsilon > 0$; **exponential** if $f = \Omega(2^{n^\varepsilon})$ for some $\varepsilon > 0$; and **strictly exponential** if $f = \Omega(2^{\varepsilon n})$ for some $\varepsilon > 0$.

**Classification of computational problems:** Computational problems can be classified like decision problems, optimization problems, search problems, etc. A Combinatorial optimization problem can be reformed to its decision version but each of such an optimization problem need not be a decision problem. A Hamiltonian cycle (HC) problem is a decision problem.

**Example 2.3.** Consider a graph $G = (V, E)$. Does it contain a cycle passing through each vertex only once as an HC?

**Input:** A graph $G = (V, E)$.
**Step 1:** Assume a permutation ∀ V.
**Step 2:** Check if step 1 gives an HC
**If** Yes, **then** accept the input.

A simple decision problem on the matrix is like: *Let b be an n-dimensional integer vector and A be an $m \times n$ matrix. Does there exit an m-dimensional integer vector x such that $Ax \geq b$.*

This problem is in NP. For details, we refer standard texts like [41], [9], [48]. Let us have a classical TSP:

**Problem 1.** Consider a complete connected graph of n cities with known distances. Find the shortest Hamiltonian tour as an HC in the complete graph.

Its decision version becomes,

**Problem 2.** Given a complete connected graph of n cities with known distances. Consider $K > 0$ for $K \in \mathbb{N}$, does it have a HC tour with, total distance $\geq K$ ?

If the above Problem 1 is solvable in polynomial time, then is so for the Problem 2 and vice-versa. In 1965, Admonds presented a conjecture 'TSP does not have a polynomial time solution.' After a decade in 1971, S. Cook established the $1^{st}$ NP-complete problem like:

**Problem 3.** A problem is NP-complete if it is both NP-hard and belongs to the NP class.

For detail, we refer the text like [23] and [48].

## 3 Taxonomy of algorithms

An algorithm is performed with $(i)$ assignment steps, $(ii)$ operation steps, and $(iii)$ logical steps. The time required by it is largely dependent on the problem instance, which differs from one instance to another. We can measure its performance with $(i)$ empirical $(ii)$ average-case, and $(iii)$ worst-case analysis. The empirical one is to estimate how an algorithm behaves. The average-case forecasts for expected number of steps. In contrast, the worst-case analysis provides the upper bounds. The performance of an algorithm depends on the compiler, programmer and their programming language, and the machine used which are the major drawback to empirical analysis. Moreover, it may be time-consuming and expensive to perform, and different algorithms perform differently on various classes of problems. Average-case analysis depends on the probability distribution and is challenging to determine the appropriate choice, and is extremely difficult for more complex algorithms. The majority of the drawbacks encountered by empirical and average-case analysis are improved upon in worst-case analysis.

Let $\mathscr{I} = \{\mathcal{I} : \mathcal{I} \in \mathscr{I}\}$ denote the set of all problem instances $\mathcal{I}$ for the computational problem $\mathscr{P}_c$. For an algorithm $\mathscr{A}$, the runtime function $f(n) : \mathbb{N} \to \mathbb{N}$ is determined by the input size $n$ where $|\mathcal{I}| = n$. The time complexity reflects the smallest function that achieves a running time of $O(f(n))$, indicating its worst-case performance. For a set of valid instances $\mathscr{I}_c$, the worst-case running time is given by, $\max\{\min\{f(\mathcal{I})\} : \mathcal{I} \in \mathscr{I}_c\}$. This captures the maximum execution time across all instances, highlighting the algorithm's performance in the most challenging scenarios.

Algorithmic complexity measures expected execution time and required storage space. Symbolically, $A \leq_T B$ tells that A is no more difficult than B. Thus, A and B are algorithmically similar if $A \leq_T B$ and $B \leq_T T$, i.e., $A \equiv B$, turning equivalent. There are many problems which are not computable. Still there are some very difficult computable problems that do not belong to any of the complexity classes. See in [10] and [16] for details.

## 4 P versus NP in complexity

P, NP, NP-complete and NP-hard are four major categories of the computational problems. Shortly, P: decision problems in polynomial time by a deterministic Turning machine; NP: decision problems whose "yes" instances can be verified in polynomial time by a deterministic Turning machine; NP-complete: problems in NP to which every problem in NP can be reduced in polynomial time; and NP-hard: at least as hard as NP-complete, not necessarily in NP. Figure 3 provides a simple comparison of such problems, illustrating how their complexity increases from P to NP-hard.

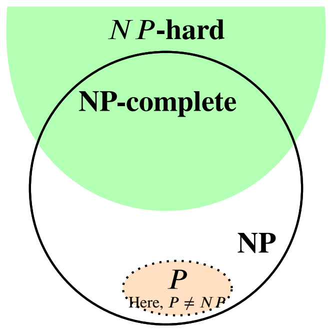

Figure 3: Comparison between $P$, $NP$, $NP-$complete, & $NP-$hard.

One of the open problem in the field of mathematics is whether P=NP. In case if P = NP, then there will be fast ways to solve problems that are currently considered very difficult. It will be the huge implications for computing, cryptography, and our daily lives. There will be no privacy in the digital domain with the complete breakdown of all cryptography, which could lead to a loss of internet security and the world being mostly a *Utopia*. By this, there would be no fundamental gap between solving a problem and recognizing the solution.

SAT, the Boolean satisfiability problem is the first problem proven to be NP-complete as in [7]. As mentioned earlier, $P \subseteq NP$, but it is an open problem whether $NP \subseteq P$. Here, we use their 3-SAT problem to demonstrate NP-completeness. Intuitively, we show that any SAT instance can be converted into an equivalent 3-SAT instance without changing whether it is satisfiable.

**Theorem 4.1.** *[7] 3-SAT is NP-complete.*

*Proof.* 3-SAT is a special case of SAT problem. In this, each clause contains at most 3 variables. So, it is NP. Now, it is sufficient to show that there is a polynomial reduction from SAT to 3-SAT.

Consider the case with m clauses. Consider a literal as a variable or its negation. A clause is the logical OR of one or more literals. Now, consider a clause with more than 3 variables with the tree diagram demonstration corresponding to $u_1 \vee \bar{u_2} \vee \bar{u_3} \vee u_4 \vee \bar{u_5}$. Then, the leaves for the new 3-SAT problem contains $u_1 \vee \bar{u_2} \vee y$, $\bar{y} \vee \bar{u_3} \vee w$, and $\bar{w} \vee u_4 \vee \bar{u_5}$.

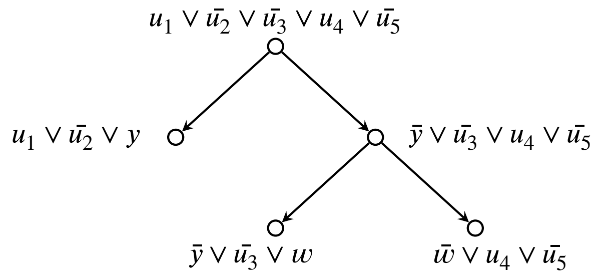
$u_1 \vee \bar{u_2} \vee \bar{u_3} \vee u_4 \vee \bar{u_5}$
$u_1 \vee \bar{u_2} \vee y$
$\bar{y} \vee \bar{u_3} \vee u_4 \vee \bar{u_5}$
$\bar{y} \vee \bar{u_3} \vee w$
$\bar{w} \vee u_4 \vee \bar{u_5}$

Figure 4: A tree diagram of SAT decomposition

In each level of the tree, the clause corresponding to the rightest node has one fewer variable than the previous one, simplifying the structure while preserving satisfiability. In a $k-3$ layers tree, we can construct $k-2$ new clauses on 3 variables. For m clauses, with $k_1, k_2, \ldots, k_m$ variables, such a problem can be constructed with $\sum_{i=1}^{i=m}(k_i-2)$ clauses. This process takes $O(2\sum_{i=1}^{i=m}(k_i-2))$ steps, polynomial in the size of input.

Clauses at level $i$ can be satisfied if and only if the clauses at level $i+1$ can be satisfied. As in Figure 4, let $u_1 \vee \bar{u_2} \vee \bar{u_3} \vee u_4 \vee \bar{u_5}$ is *True*, then either $u_1 \vee \bar{u_2}$ is *True* or $\bar{u_3} \vee u_4 \vee \bar{u_5}$ is *True*. Set $y = False$ in previous case and $y = True$, in the later one, then by such assignment, both $u_1 \vee \bar{u_2} \vee y$ and $\bar{y} \vee \bar{u_3} \vee u_4 \vee \bar{u_5}$ are satisfied. Conversely, if both $u_1 \vee \bar{u_2} \vee y$ and $\bar{y} \vee \bar{u_3} \vee u_4 \vee \bar{u_5}$ are satisfied, then if $y = True$, we know that $\bar{u_3} \vee u_4 \vee \bar{u_5}$ is true; and if $y = False$, we know that $u_1 \vee \bar{u_2}$ is *True*. Hence, the original clause is *True.*

Such satisfiability property can be satissfied for forward and backward cases. For forward, begins from the root of the tree and applies the property to the leaves. For backward, start from the leaves and move similarly to the root clause. Hence, 3-SAT problem is NP-complete. □

**NP-hard problems:** A problem $\mathcal{P}_1$ becomes NP-hard, if every problem in NP can be polynomially reduced to $\mathcal{P}_1$, i.e., $\mathcal{P}_2 \leq_p \mathcal{P}_1$ for all $\mathcal{P}_2 \in$ NP. In addition, if $\mathcal{P}_1 \in$ NP, then the problem $\mathcal{P}_1$ becomes NP-complete. Hence, the class NP-hard includes the problems not in NP.

## 5 Time complexity of some common algorithms

Having established the theoretical framework of complexity classes and NP-completeness, we now turn to concrete algorithmic examples and summarize their complexities classes.

Table 3: Time complexity of some common algorithms

| Name | Algorithm for | Complexity | Further reading |
|---|---|---|---|
| Sorting | Sorting a sequene $a_n$ for each $a_i > 0$ into non-decreasing order | $O(n^2)$ | [30] |
| Merge sort | Sorting $a_1, a_2, \dots, a_n$ in an array to $a_{i1} \leq a_{i2} \leq \cdots \leq a_{in}$ | $O(n \ln n)$ | [37], [30] |
| Quick sort | Sorting $a_1, a_2, \dots, a_n$ in an array to $a_{i1} \leq a_{i2} \leq \cdots \leq a_{in}$ | $O(n \ln n)$ | [31] |
| Heap sort | Sorting $a_1, a_2, \dots, a_n$ in an array to $a_{i1} \leq a_{i2} \leq \cdots \leq a_{in}$ | $O(n \ln n)$ | [30] |
| Counting sort | Sorting $a_1, a_2, \dots, a_n$ in an array to $a_{i1} \leq a_{i2} \leq \cdots \leq a_{in}$ with k the largest integer input | $O(n + k)$ | [10] |
| Topological sort | Sorting of nodes from $G = (V, E)$ with $m = \lvert E \rvert$, $n = \lvert V \rvert$ | $O(m)$ | [30] |
| Kruskal | A minimum spanning tree on the connected graph with no non-negative edge weight | $O(m \ln m)$ | [8], [45] |
| Ford-Fulkerson | A max-flow from $G = (V, E)$ | $O(nm^2)$ | [20], [34] |
| Edmond-Karp | A max-flow in $G = (V, E)$ with $m = \lvert E \rvert$ and $n = \lvert V \rvert$ | $O(nm^2)$ | [38], [45] |

Table 4: Time complexity of some common algorithms (Continued...)

| | | | |
|---|---|---|---|
| Dijkstra | The shortest path in no negative weight directed graph $G = (V, E)$ with $m = \|E\|, n = \|V\|$ | $O((m+n)\ln n)$ | [1], [11] |
| Breadth-First Search | The shortest path detection in a unweighted graph with $m = \|E\|, n = \|V\|$, source s | $O(m+n)$ | [10] |
| Depth-First Search | Graph traversal cycle detection with topological sort to get strongly connected components, $m = \|E\|$, $n = \|V\|$ | $O(m+n)$ | [10] |
| Bellman Ford | The min-weight of the path from the source to sink in a directed graph, $m = \|E\|$ and $n = \|V\|$ with no negative weights | $O(mn)$ | [3], [14] |
| Johnson | All pairs shortest paths in a weighted network with $m = \|E\|$ and $n = \|V\|$ with no negative cycle | $O(n^2 \ln n + mn)$ | [24], [33] |
| Greedy for Selection | Selecting a maximum subset of non-overlapping activities from a sequence of n activities | $O(n)$ | [45] |
| Greedy for Huffman Tree | Selecting a binary tree from a sequence of leaf weights | $O(n)$ | [8], [45] |
| Hopcroft-Karp | A maximum bipartite matching in a bipartite graph, $m = \|E\|$ and $n = \|V\|$ | $O\|mn\|$ | [8], [43] |

Table 5: Continued...

| | | | |
|---|---|---|---|
| Labeled-tree | Canonical guilotine partitions to get the minimum cost of labeled tree | $O(n^4)$ | [15], [39] |
| Gabow"s scaling | A single source shortest path in a directed no negative weights network | $O(mn)$ | [14], [22] |
| Floyd-Warshall | Source to sink shortest path for all pairs in a directed graph with no negative cycle | $O\|n\|^3$ | [19], [47] |
| Prim | A spanning tree from a non-negative weights graph | $O(m \ln n)$ | [27], [44] |
| Diniz | A max-flow from a network with $m = \|E\|, n = \|V\|$ | $O\|mn^2\|$ | [12], [13] |
| Max-flow min-cost | A max-flow in min-cost in a no negative cost cycle network with $m = \|E\|, n = \|V\|$ | $O\|nm^2\|$ | [2], [20] |
| Maximum graph matching | The maximum matching on the graph with $m = \|E\|$, $n = \|V\|$ | $O(mn)$ | [1], [20] |
| Goldberg-Tarjan push relabel | A max-flow in a network with $m = \|E\|, n = \|V\|$ | $O\|mn^2\|$ | [26] |
| Goldberg-Rao | A max-flow in min-cost from a sparse network with $m = \|E\|, n = \|V\|$, no negative cost cycle | $O(min(\sqrt{n}, \sqrt[3]{m}).m \ln(\frac{U}{m}))$ | [25] |
| Orlin | The max-flow on the graph, $m = \|E\|$, $n = \|V\|$ | $O(mn)$ | [1], [40] |

Table 6: Continued...

| | | | |
|---|---|---|---|
| Kelner *et al.* | An approximate max-flow on the graph with $m = \|E\|$, $n = \|V\|$, $\epsilon$ as approximation error | $O(\frac{m}{\epsilon^2})$ | [36] |
| Out-of-Kilter | Min-cost-flow in the network with $m = \|E\|$, $n = \|V\|$, U=maximum edge capacity | $O(nmU)$ | [21], [36] |
| Cheapest augmenting path | Min-cost-flow along the cheapest augmenting path with $m = \|E\|$, $n = \|V\|$, U=maximum flow capacity | $O(mnU)$ | [1], [34] |
| Cycle cancellation | A max-flow in min-cost from a network with $m = \|E\|$,$n = \|V\|$, U=maximum capacity, no negative cost cycle | $O\|mn^2U\|$ | [4] |
| Successive shortest path | The max-flow in min-cost on graph $m = \|E\|$,$n = \|V\|$, U=maximum capacity, no negative cost cycle | $O(mn \ln n + mnU)$ | [18], [21] |
| About integer vector | Given an n-dimensional integer vector b and an $m \times n$ integer matrix A, whether there exist an m-dimensional integer vector x s.t. $Ax \geq b$ | NP | [7], [17] |
| Desion version of TSP | Given n connected cities with known distances, and an integer $K > 0$, is there a Hamiltonian tour with total distance at most K? | NP-complete | [7] |

Table 7: Continued...

| | | | |
|---|---|---|---|
| SAT | In conjunctive normal form, is there a variable assignment making the formula true? | NP-complete | [7], [8] |
| 3 SAT Problem | Let a 3 SAT, each clause having three literals. Is it satisfiable? | NP-complete | [7] , [8] |
| HC | Given a completely connected network $G = (V, E)$, does G contain a HC tour. | NP-complete | [32], [35] |
| Vertex cover | Given $G = (V, E)$ & $K \in \mathbb{Z}^+$, does there exist a vertex cover of size at most $K$? | NP-complete | [6], [7] |
| Minimum vertex cover | For $G = (V, E)$, compute a minimum cardinality vertex cover | NP-complete | [5], [29] |
| Graph coloring | Can we color the vertices of $G = (V, E)$ with three colors so that no two adjacent vertices are with same color? | NP-complete | [23], [35] |
| Partition Problem | Can a given multiset of $\mathbb{N}^+$ be partitioned into two subsets having equal sums? | NP-Complete | [23] , [35] |
| Job Scheduling Problem | Can we schedule n jobs on m machines so that all jobs complete within a given time bound? | NP-complete | [23] , [35] |
| Knapsack Problem | Choose a subset of items with a total weight $\leq$ W and a total value $\geq$V | NP-complete | [23] , [35] |
| Maximum clique | Given $G = (V, E)$, find a clique with maximum size. | NP-hard | [42], [50] |

Table 8: Continued...

| | | | |
|---|---|---|---|
| Set cover | The minimum size subsets of sets that covers a given finite set | NP-hard | [51] |
| TSP | Decision version is NP-complete. Whats about the metric to optimize. | NP-hard. | [23] |
| Integer Programming | Determine whether there exits an integer solution to a system of linear equation. | NP-hard | [35] |
| Job Scheduling | Given jobs and machines with processing orders, determine the minimum time-span to complete all jobs. | NP-hard | [35] |

Many problems listed in our tables are NP-hard, so we do not expect polynomial-tme exact algorithms for them. This motivates the study of approximation algorithms, which we discuss next.

## 6 Approximations

An approximation algorithm is a polynomial-time. For an optimization problem, it provides a solution close to the optimal solution. Specifically, a c-approximation algorithm guarantees the solution's value as a factor of 'c' of the optimal value. Let OPT denote the optimal solution value, then for minimization problems, the solution is at most c.OPT, while for maximization problems, it is at least OPT/c. We refer to [28], [46] and [49] for their extensive coverage.

**Flow-Approximations:** For flows over time problems, approximation algorithms can focus on either the flow value or the time horizon. A c-flow-approximation algorithm ensures:

For maximization problems, the computed flow value is at least 1/c of the optimal flow value, where c is a constant. For minimization problems, the solution satisfies only 1/c of the demands or uses only 1/c of the supplies while being as effective as the optimal solution.

This distinction arises because finding feasible solutions can sometimes be $\mathcal{NP}$-hard. For example, in the minimum bounded-degree spanning tree problem, finding a feasible solution under strict degree constraints is intractable, but approximation algorithms can allow slight violations of these constraints.

**Time-Approximations:** Flows over time introduce additional complexity due to the need to specify flow rates at different time points for each arc. This leads to the concept of time-approximations, which ensures:

For minimizing the time horizon, the computed time horizon is at most c times the optimal time horizon. For maximizing flow value or minimizing costs, the solution is as effective as the optimal solution but can use a time horizon extended by a factor of c.

**Bicriteria Approximations:** Some algorithms approximate both time and value simultaneously. These bicriteria approximation algorithms provide guarantees such as:

To minimize the time horizon, the solution's time horizon is within a factor of c of the optimal time horizon, while satisfying only 1/c' of the demands or using only 1/c' of the supplies.

**PTAS for the Knapsack Problem:** A polynomial-time approximation scheme (PTAS) provides a $(1-\epsilon)$-approximation for any $\epsilon > 0$, the running time may depend exponentially on $1/\epsilon$.

**Example:**

1. Suppose $\epsilon = 0.1$, meaning we want a solution with at least 90% of the optimal value.
2. A PTAS might use a dynamic programming approach that scales with n (the number of items) but grows exponentially with $1/\epsilon$. For instance, the running time could be $\mathcal{O}(n^{1/\epsilon})$.
3. If n=100 and $\epsilon$=0.1, the running time could be $\mathcal{O}(100^{10})$, which is polynomial in $n$ but impractical for small $\epsilon$.

Thus, the running time is polynomial in $n$ but exponential in $1/\epsilon$. As $\epsilon$ gets smaller, the running time grows rapidly.

**FPTAS for the Knapsack Problem:** A fully polynomial-time approximation scheme (FPTAS) also provides a $(1-\epsilon)$-approximation, but the running time is polynomial in both $n$ and $1/\epsilon$.

**Example:**

1. Using $\epsilon = 0.1$, an FPTAS would provide a solution with at least 90% of the optimal value.
2. The running time might be $\mathcal{O}(n^2/\epsilon)$ or $\mathcal{O}(n^3/\epsilon)$, which is polynomial in both $n$ and $1/\epsilon$.
3. For $n = 100$ and $\epsilon = 0.1$, the running time could be $\mathcal{O}(100^2/0.1) = \mathcal{O}(100,000)$, which is much more efficient than the PTAS.

Thus, the running time is polynomial in both $n$ and $1/\epsilon$. Even for very small $\epsilon$, the algorithm remains efficient.

## 7 Conclusions

Computational problems and their complexity analysis play a crucial role in computer science, mathematics, and real-world applications. Understanding computational complexity helps in designing efficient algorithms, optimizing resources, and making informed decisions about problem-solving approaches. Determining whether a problem can be solved within a reasonable timeframe is vital for achieving sucessful outcomes. Complexity analysis distinguishes between tractable (solvable in polynomial time, P-class) and intractable ones (suspected to require exponential time, such as NP-complete problems). Complexity theory helps define fundamental classes like P, NP, EXP, NP-complete, and NP-hard, influencing fields such as cryptography and artificial intelligence, shaping algorithm design, and opening a wider horizon for further research.

**Acknowledgement:** Authors are grateful to the University Grants Commission, Nepal for their SRDIG award, SRDIG-81/82-S &T-03, SRDIG-80/81-S&T-08, and 80/81-S & T-07, respectively. Also thankful to the anonymous reviewers for their valuable comments and suggestions.

## References

[1] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. Network flows. 1988.

[2] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved time bounds for the maximum flow problem. *SIAM Journal on Computing*, 18(5):939–954, 1989.

[3] R. Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[4] R. Busacker and P. Gowen. A procedure for determining a family of minimal-cost network flow patterns, oro techn. report 15. *Operations Res. Office, Johns Hopkins Univ., Baltimore*, 1961.

[5] J. Chen, L. Kou, and X. Cui. An approximation algorithm for the minimum vertex cover problem. *Procedia engineering*, 137:180–185, 2016.

[6] S. A. Cook. A hierarchy for nondeterministic time complexity. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 187–192, 1972.

[7] S. A. Cook. The complexity of theorem-proving procedures. In *Logic, automata, and computational complexity: The works of Stephen A. Cook*, pages 143–152. 2023.

[8] W. Cook, L. Lovász, P. D. Seymour, et al. *Combinatorial optimization: papers from the DIMACS Special Year*, volume 20. American Mathematical Soc., USA, 1995.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, USA, 3 edition, 2009.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, MA, 2022.

[11] E. W. Dijkstra. A note on two problems in connexion with graphs. In *Edsger Wybe Dijkstra: his life, work, and legacy*, pages 287–290. 2022.

[12] E. A. Dinic. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pages 1277–1280, 1970.

[13] Y. Dinitz. Dinitz'algorithm: The original version and even's version. In *Theoretical Computer Science: Essays in Memory of Shimon Even*, pages 218–240. Springer, Berlin, 2006.

[14] S. Dreyfus. Richard bellman on the birth of dynamic programming. *Operations Research*, 50(1):48–51, 2002.

[15] D.-Z. Du, P. M. Pardalos, X. Hu, and W. Wu. *Introduction to Combinatorial Optimization*, volume 196 of *Texts in Computer Science*. Springer Nature, Switzerland, 2022.

[16] D.-Z. Du, P. M. Pardalos, X. Hu, and W. Wu. Np-hard problems and approximation algorithms. In *Introduction to Combinatorial Optimization*, pages 199–257. Springer International Publishing, Cham, 2022.

[17] J. Edmonds and R. M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *Journal of the ACM*, 19(2):248–264, 1972.

[18] M. Engquist. A successive shortest path algorithm for the assignment problem. *INFOR: Information Systems and Operational Research*, 20(4):370–384, 1982.

[19] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.

[20] L. R. Ford and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, Princeton, NJ, 1962.

[21] D. R. Fulkerson. An out-of-kilter method for minimal-cost flow problems. *Journal of the Society for Industrial and Applied Mathematics*, 9(1):18–27, 1961.

[22] H. N. Garbow. Scaling algorithms for network problems. *Journal of Computer and System Sciences*, 31(2):148–168, 1985.

[23] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, USA, 1979.

[24] M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some simplified np-complete problems. In *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, pages 47–63, 1974.

[25] A. V. Goldberg and S. Rao. Beyond the flow decomposition barrier. *Journal of the ACM*, 45(5):783–797, 1998.

[26] A. V. Goldberg and R. E. Tarjan. A new approach to the maximum-flow problem. *Journal of the ACM*, 35(4):921–940, 1988.

[27] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[28] M. Gross and M. Skutella. Generalized maximum flows over time. In *International Workshop on Approximation and Online Algorithms*, pages 247–260. Springer, 2011.

[29] R. Hassin and A. Levin. The minimum generalized vertex cover problem. *ACM Transactions on Algorithms (TALG)*, 2(1):66–78, 2006.

[30] M. T. Heideman, D. H. Johnson, and C. S. Burrus. Gauss and the history of the fast fourier transform. *IEEE ASSP Magazine*, 1(4):14–21, 1984.

[31] C. A. R. Hoare. Algorithm 64: Quicksort. *Communications of the ACM*, 4(7):321, 1961.

[32] K. L. Hoffman, M. Padberg, and G. Rinaldi. Traveling salesman problem. In *Encyclopedia of Operations Research and Management Science*, pages 1573–1578. Springer, Berlin, 2013.

[33] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[34] L. R. F. Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

[35] R. M. Karp. Reducibility among combinatorial problems. In *50 Years of Integer Programming 1958–2008: From the Early Years to the State-of-the-Art*, pages 219–241. Springer, Berlin, Heidelberg, 2009.

[36] J. A. Kelner, Y. T. Lee, L. Orecchia, and A. Sidford. An almost-linear-time algorithm for approximate max flow in undirected graphs, and its multicommodity generalizations. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 217–226, USA, 2014. SIAM.

[37] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*, volume 3. Addison-Wesley, USA, 2 edition, 1999.

[38] B. Korte and J. Vygen. *Combinatorial Optimization: Theory and Algorithms*. Springer, Berlin, 5 edition, 2011.

[39] J. Lee. *A First Course in Combinatorial Optimization*, volume 36. Cambridge University Press, USA, 2004.

[40] J. B. Orlin. Max flows in o(nm) time, or better. In *Proceedings of the Forty-Fifth Annual ACM Symposium on Theory of Computing*, pages 765–774, 2013.

[41] C. H. Papadimitriou and K. Steiglitz. Some complexity results for the traveling salesman problem. In *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, pages 1–9, 1976.

[42] P. M. Pardalos and J. Xue. The maximum clique problem. *Journal of Global Optimization*, 4:301–328, 1994.

[43] V. T. Paschos, editor. *Applications of Combinatorial Optimization*, volume 3. John Wiley & Sons, NY, 2014.

[44] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM (JACM)*, 49(1):16–34, 2002.

[45] A. Schrijver. A course in combinatorial optimization. Technical Report Kruislaan 413, 1098, CWI, 2003.

[46] V. V. Vazirani. *Approximation Algorithms*. Springer Science & Business Media, Berlin, 2013.

[47] S. Warshall. A theorem on boolean matrices. *Journal of the ACM (JACM)*, 9(1):11–12, 1962.

[48] I. Wegener. *Complexity Theory: Exploring the Limits of Efficient Algorithms*. Springer, Berlin, Heidelberg, 2005.

[49] D. P. Williamson and D. B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, NY, 2011.

[50] Q. Wu and J.-K. Hao. A review on algorithms for maximum clique problems. *European Journal of Operational Research*, 242(3):693–709, 2015.

[51] W. Zhang, W. Wu, W. Lee, and D.-Z. Du. Complexity and approximation of the connected set-cover problem. *Journal of Global Optimization*, 53(3):563–572, 2012.