Subset Sum Problem Analysis

Baranga Roxana, Pasăre Daria, and Simion Marina

Group 321CA
University POLITEHNICA of Bucharest

1 Introduction

1.1 Description

The Subset Sum Problem is a well-known member of the NP-complete class of computational problems, having no known polynomial-time algorithm. At its core, the problem involves determining whether a subset of a given set of integers can sum to a specified target value.

Definition 1. *The Subset Sum problem is defined as follows: given a set of positive integers S and a target, an integer t, determine whether there is a set S' such that $S' \subseteq S$ and $\sum(S') = t$.*

This paper aims to analyze the Subset Problem from both a theoretical and a practical perspective. We begin by discussing the problem's computational complexity, specifically focusing on the NP-hardness of the problem. Next, we review three methods for solving this problem: dynamic programming, backtracking, and a greedy approach. Dynamic programming provides an efficient way to solve the problem by breaking it down into smaller subproblems and solving them optimally. Backtracking explores potential solutions by systematically building candidates and pruning infeasible paths, while the greedy approach makes locally optimal choices in an attempt to find a globally optimal solution.

1.2 Practical use cases

1. Cryptography and Security: Some cryptographic protocols, such as the RSA algorithm, can be related to the Subset Sum Problem. Certain forms of cryptography exploit the difficulty of such problems to ensure security. The problem can be used in the construction of "trapdoor" functions, where solving the subset sum problem within a specific set of constraints is computationally infeasible without a secret key.
2. Resource Allocation and Budgeting: Can be applied when planning how to allocate a set of financial resources to various projects: given a set of expenses and a target budget, determine if there is a subset of expenses that fits within the budget.
3. Decision Making in Business and Finance: Investors may need to choose a set of assets that sum up to a specific total value, subject to certain constraints (e.g., risk levels or investment goals). This can be framed as a Subset Sum Problem where the "target" is the desired return or risk-adjusted return.

2 NP-Hard Demonstration

As stated above, the Subset Sum problem is classified as NP-Complete. In this paper, we will only focus on the NP-hardness of the problem; to demonstrate this property, we perform a reduction from another very well-known NP-Complete problem, the 3-SAT problem, more specifically a 3CNF formula.

Definition 2. *The SAT problem states as follows: Given a boolean formula in conjunctive normal form (disjunction of conjunctions), is the formula satisfiable?*

Definition 3. *The 3-SAT problem is an SAT problem in which each clause is limited to at most three literals. CNF refers to the Conjunctive Normal Form of the formula, representing a conjunction of one or more clauses, where a clause is a disjunction of literals.*

The demonstration We start with the following formula, which is obviously in the 3-CNF form. The point is to find a set S of integers and a target T, corresponding to a YES-clause of the SAT problem, so that both the 3-SAT and Subset Sum problems are satisfied.

$$\varphi = (x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor x_3) \land (x_1 \lor x_2 \lor x_3)$$

We assign 2 values to each literal, so x_i has 2 values, v_i and v_i', corresponding to the true and false values of the literal. The same is done with the clauses: clause C_i has 2 values, S_i and S_i', representing true and false. The following table shows how we assigned each of these numbers their value.

How we made this table? Let's start with the upper left corner. We only write 1 in the cells representing the values for x_i, the rest remain 0. So for x_1, we have 1 in the v_1 and v_1' cells.

In the lower left corner it's simple, all cells are 0.

Now for the upper right corner of the table, we verify the satisfiability of the clauses. For each v_i, we put 1 in the clauses that are satisfied by x_i being true. For each v_i', we put 1 in the clauses that are satisfied by x_i being false. For example, x_1 being true means the clauses C_1 and C_4 are true, so we put 1 in those cells, the rest remaining 0.

Lastly, in the lower right corner, we look at the values for C_i. In the cell representing the truth value, we put 1, and in the cell representing false, we put 2, while the rest of the cells remain 0. For example, C_1 has 1 in the S_1 cell and 2 in the S_1' cell; the rest are 0.

The last row is filled as follows: the first 3 cells are 1 (the number of literals), while the last 4 are 4. (the number of clauses)

$-$	x_1	x_2	x_3	C_1	C_2	C_3	C_4
v_1	1	0	0	1	0	0	1
v_1'	1	0	0	0	1	1	0
v_2	0	1	0	0	0	0	1
v_2'	0	1	0	1	1	1	0
v_3	0	0	1	0	0	1	1
v_3'	0	0	1	1	1	0	0
S_1	0	0	0	1	0	0	0
S_1'	0	0	0	2	0	0	0
S_2	0	0	0	0	1	0	0
S_2'	0	0	0	0	2	0	0
S_3	0	0	0	0	0	1	0
S_3'	0	0	0	0	0	2	0
S_4	0	0	0	0	0	0	1
S_4'	0	0	0	0	0	0	2
t	1	1	1	4	4	4	4

Table 1. Values assigning table

How does it correlate to Subset Sum? We claim that the target number is the bottom one, 1114444, and the set of numbers are all the rows in the table: 1001001, 1000110, 100001, 101110, 10011, 11100, 1000, 2000, 100, 200, 10, 20, 1, 2. We will call the S_i values "helper values".

Now we have to verify the reduction by using the following proposition:

Proposition 1. *3 CNF SAT has a solution if only and only if the equivalent Subset Sum has a solution.*

The left-right implication: We assume the 3 CNF SAT has a solution, called an "yes-instance"; in our case we choose $x_1 = 1$, $x_2 = 0$, $x_3 = 1$. Therefore, we select the equivalent rows: v_1, v_2' and v_3. So far, our subset is: $1001001, 101110, 10011$. Any given x_i will always have 1 of 2 values: true or false, so we either choose v_i or v_i', never both. This way, we always reach the first part of the target: the 111.

Now, for each column, we choose the appropriate helper values to reach our target digit of 4. In our case, that would be 2000, 100, 200, 20, 2. Notice that if the clause was not satisfied (there were no 1 values for the C's), the helper values would only add up to 3 per column. This assures us that the clauses are satisfied. The final subset is 1001001, 101110, 10011, 2000, 100, 200, 20, 2, which adds up to 1114444.

The right-left implication: We assume that the Subset Sum has a valid solution. Concentrating on the first part (the 111), it's implied that we either select v_i or v_i', so 1 value of 1 per column. For the second part, the 4444, we established that the helper values only add up to 3 per column. No matter which we used, we would have relied on at least 1 value of 1 from each of the columns from the upper part of the table, equivalent to the clauses being satisfied.

After proving both implications, we have proved that the Subset Sum problem can be reduced to a NP-Complete problem, therefore it is part of the NP-hard class.

3 Algorithms Analysis

3.1 The backtracking solution

Backtracking is a recursive technique used to solve the subset sum problem by exploring all possible combinations of subsets, using an approach similar to DFS. At each step, we decide whether to include or exclude an element in the current subset by checking that its sum equals the target value. If a solution is invalid, the algorithm reverts to previous decisions, removing unnecessary branches from the search space.

```
def backtracking(S, index, target, rest_sum):
if index >= len(S) or target < 0 or target > rest_sum:
    return False
if target == 0 or target == S[index] or target == rest_sum:
    return True
if backtracking(S, index + 1, target, rest_sum - S[index]):
    return True
if backtracking(S, index + 1, target - S[index], rest_sum - S[index]):
    return True
return False

def subset_sum_backtracking(nums, target):
set_sum = sum(nums)
nums.sort()
return backtracking(nums, 0, target, set_sum)
```

Complexity: The backtracking algorithm can be visualized as a binary tree, where each node represents a recursive function call. During each call, one element of the set S is processed, and the function either continues by making recursive calls to explore further or backtracks. Since each node in this tree corresponds to a potential decision point, there can be up to 2^n nodes in a complete binary tree of depth n. As a result, the worst-case time complexity of the algorithm is $O(2^n)$. The space complexity is O(n) since the depth of the recursion stack is at most n, corresponding to the maximum depth of the tree.

Advantages: The Backtracking algorithm guarantees finding a solution by exhaustively exploring all possible combinations, being an easy-to-understand and implement solution due to its simple recursive formulation. Moreover, the algorithm benefits from efficient boundary conditions, that halt the search when the sum of the remaining elements cannot reach the target, thus reducing the number of unnecessary recursive calls.

Disadvantages: Nevertheless, the algorithm becomes inefficient for large sets due to its exponential O(2^n) complexity, which grows rapidly with the size of the set. In addition, it does not use advanced optimization techniques, such as memorization or dynamic programming, which leads to redundant recalculations. Also, for dense sets, the algorithm can become expensive in terms of time and resources required to check all possible subsets. Thus, the algorithm is only recommended for small or medium sets where full searches are feasible.

3.2 The Dynamic Programming solution

Dynamic programming solves problems by breaking them into overlapping sub-problems and solving them efficiently, unlike divide-and-conquer approaches, where sub-problems are disjoint. This dynamic programming approach addresses the Subset Sum problem by determining if a subset of a given list sums to the target value. It uses a 1D array to track achievable sums, starting with summap[0] = 1, as a sum of zero is always possible. For each number in the list, the algorithm updates the array in reverse order to ensure that each number is used only once to form subsets. By the end, the value of summap[target] indicates whether the target sum can be achieved using any subset of the given numbers.

Complexity: The algorithm loops through each element in the set nums (of length n) and updates the summap vector of size target + 1 for each element. In the worst case, this involves an iteration for each number in the set and an iteration for each possible sum value, from target to 0. So the time complexity is O(n * target). Still, it is not polynomial, because the target sum is typically represented in binary, and thus the number of bits needed to represent the target grows logarithmically. As a result, the algorithm's time complexity is pseudo-polynomial, meaning it depends on the numeric values in the input, but not in a polynomial way when considering the input size in terms of bits. The algorithm uses a summap vector of size target + 1 to represent the possible sums, so the space complexity is O(target).

Advantages: The algorithm can handle large sets of numbers without requiring much memory compared to classical approaches. Using a sum vector allows efficient processing of a large set of target sum values, saving resources. Additionally, the algorithm has O(1) time complexity for each operation, making it very efficient concerning individual checks or updates in the process. Also, space complexity is reduced, so the required space is only O(target), which is much more efficient when the target is much smaller than the total number of elements in the set of numbers.

Disadvantages: One of the main disadvantages of the dynamic programming algorithm for the Subset Sum problem is its pseudo-polynomial complexity, which depends on the numerical value of the target sum and the elements in the set, rather than just the number of elements. Therefore, if the values of the numbers or the target sum are very large, the algorithm becomes time-inefficient, even for a small number of elements. Additionally, the algorithm can require a significant amount of memory because it uses a vector of size target + 1, which can lead to high resource consumption when target is a large number.

3.3 The Greedy solution

This algorithm aims to solve the subset sum problem in a greedy manner by making a locally optimal choice at each step. Descending sorting of elements is used to start with large numbers and quickly reduce the remaining amount. At each step, the algorithm decides to add a number to the current amount only if it does not exceed the target amount. This strategy attempts to build the target amount as efficiently as possible by using the largest numbers available.

Complexity: The algorithm sorts the list of numbers in descending order, and the sorting operation has complexity O(n * log n), where n represents the number of elements in the list. After sorting, the algorithm loops through the list only once to calculate the sum, and the complexity of this step is O(n). Thus, the total complexity of the algorithm is O(n * log n), since sorting is the most expensive step. In terms of space complexity, the algorithm uses a constant amount of memory for the current_sum variables and for iterative control, so the space complexity is O(1), e.g constant memory.

Advantages: The algorithm is efficient in simple cases, working quickly when the solutions involve large numbers or a small subset because it starts with the large numbers and can quickly find a solution in such situations. With a complexity of O(n log n), the algorithm is an excellent choice regarding the time, especially for medium-sized datasets, providing good performance and reduced execution time in these case.

Disadvantages: The greedy algorithm does not always guarantee the optimal solution because it does not explore all possible combinations of subsets and sometimes it may not find a valid solution even if there is one. In more complex cases, where valid subsets consist of combinations of small and large numbers, the greedy approach can lead to wrong or inefficient results.

4 Evaluation

4.1 Testing criteria

We conducted two categories of tests: one category containing only tests where a subset exists with a sum equal to the target, and another category without such subsets.
To evaluate the backtracking algorithm, we created a set of smaller tests since its time requirements were very large. Each category contains 14 tests, making a total of 28 tests. Each category of tests includes:

- 5 tests with small values
- tests with an increasing number of elements:
  - 10, 15, 17, 20, 23, 25, 27, 30, 32 - for smaller tests
  - 50, 100, 150, 200, 250, 300, 350, 400, 450, 500 - for larger tests (only for DP and greedy)

The tests include both sparse cases and cases with closely packed elements, ensuring exhaustive coverage of possible scenarios.

Input data format

The input data adheres to the following specifications:

1. The first line contains two integers, N and T, where:
   - $1 \leq N \leq 10^4$, N - the number of elements in the set
   - $1 \leq T \leq 10^7$, T - the target sum for the subset
2. The next N lines each contain one positive integer $\leq 10^7$

Output data format

The program's result will be of boolean type:

$$output = \begin{cases} \text{true}, & \text{if a subset with a sum equal to T exists} \\ \text{false}, & \text{otherwise} \end{cases}$$

4.2 System Specifications

- Operating System: Ubuntu 20.04
- RAM: 16 GB
- Processor: Intel® Core™ i7-1355U
- Python Version: 3.12.0

4.3 Results

For the smaller tests we were able to evaluate all 3 approaches. We noticed an obvious increase in time for the backtracking algorithm, as expected since the other algorithms have pseudo-polynomial complexity and this one has exponential complexity. We used polynomial interpolation to graph the execution time of each algorithm.

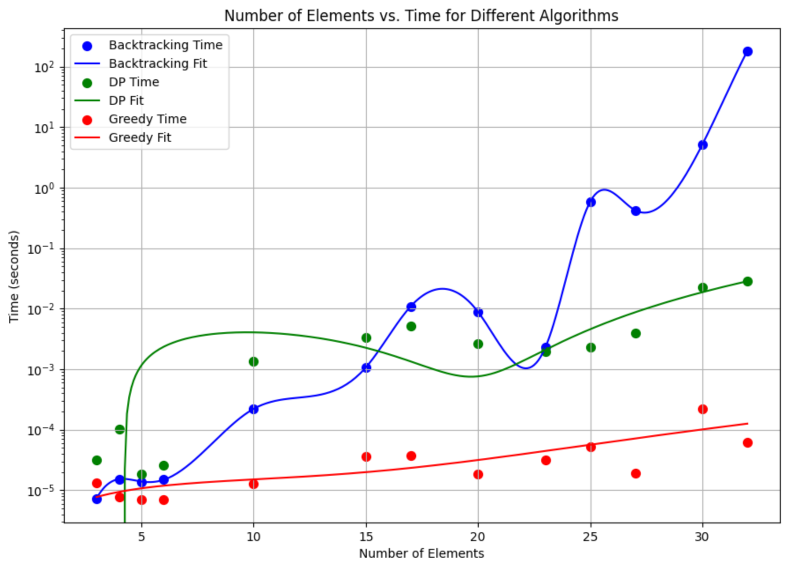

Fig. 1. Number of elements vs execution time

We also evaluated memory consumption. Here we can see how the dynamic programming approach exhibits noticeably more memory consumption compared to the other 2 approaches due to its reliance on storing intermediate results in an array.

Since our input tests were classified into tests that had the output True and tests that had the output False, we were able to compare the accuracy of the algorithms as well. The greedy algorithm seemed to start out fine, but occasionally slipped and did not notice a correct subset.
As for the larger tests, we can see on the graphic that the dynamic programming approach is significantly less efficient both in terms of memory and time consumption, but is always correct as opposed to the greedy approach.

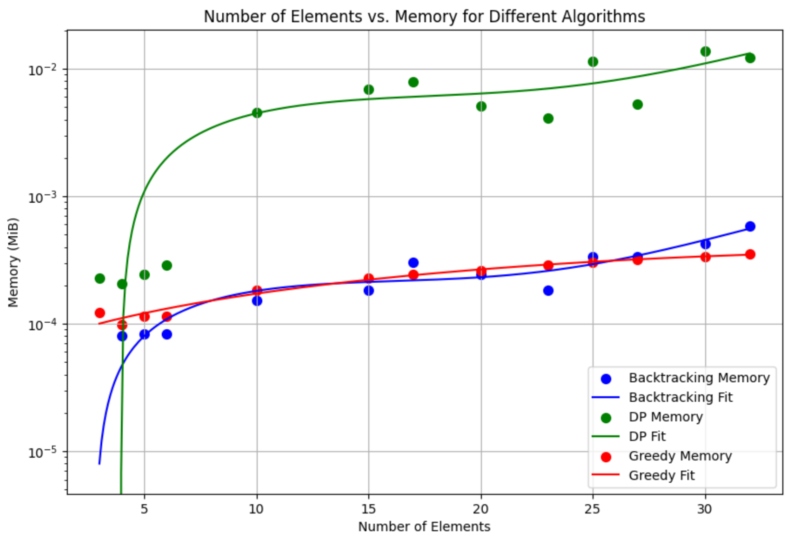

Fig. 2. Enter Caption

num elements	target	backtracking output	dp output	greedy output
3	6	TRUE	TRUE	TRUE
4	10	TRUE	TRUE	TRUE
4	10	TRUE	TRUE	TRUE
5	15	TRUE	TRUE	TRUE
6	21	TRUE	TRUE	TRUE
10	566	TRUE	TRUE	FALSE
15	875	TRUE	TRUE	TRUE
17	1002	TRUE	TRUE	TRUE
20	641	TRUE	TRUE	TRUE
23	518	TRUE	TRUE	TRUE
25	1475	TRUE	TRUE	TRUE
27	663	TRUE	TRUE	FALSE
30	1773	TRUE	TRUE	FALSE
32	1575	TRUE	TRUE	TRUE

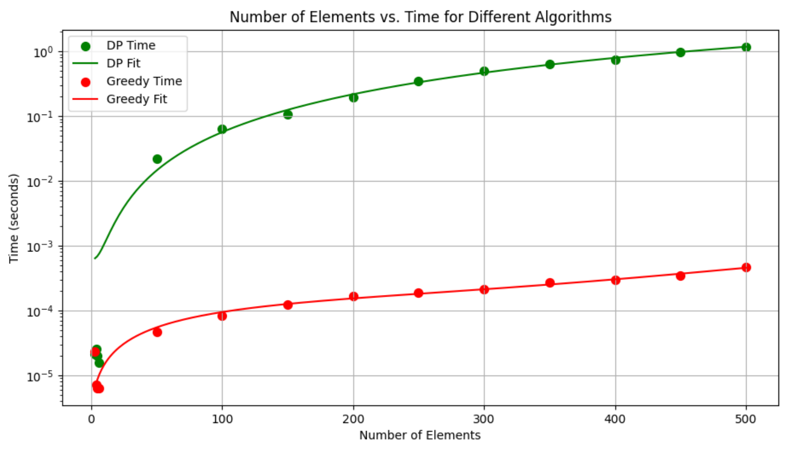

Fig. 3. Enter Caption

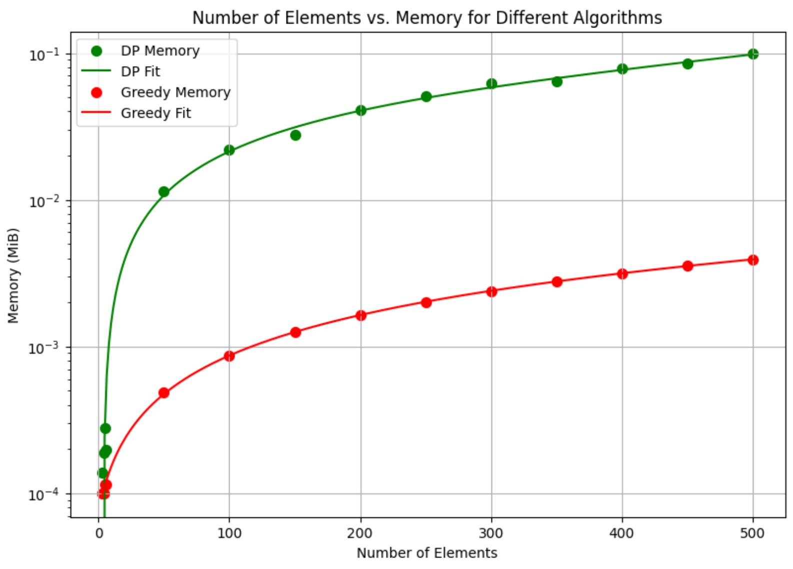

Fig. 4. Enter Caption

num elements	target	dp output	greedy output
3	6	TRUE	TRUE
4	10	TRUE	TRUE
4	10	TRUE	TRUE
5	15	TRUE	TRUE
6	21	TRUE	TRUE
50	2943	TRUE	FALSE
100	5679	TRUE	FALSE
150	7258	TRUE	FALSE
200	10630	TRUE	TRUE
250	13234	TRUE	TRUE
300	16290	TRUE	TRUE
350	16674	TRUE	TRUE
400	20449	TRUE	TRUE
450	22058	TRUE	TRUE
500	26017	TRUE	TRUE

4.4 Results interpretation

The results for the three algorithms varied, but we can notice a clear trade-off:

1. **Dynamic Programming**

   This is the most reliable algorithm. It provides a solution if one exists and in scenarios where T and N are moderate has a good enough execution time. It is however memory-intensive as it constructs an array up to the size of T.

2. **Backtracking**

   This algorithm is memory efficient while still providing accurate results. It is effective for large values of T, but not for N. It is the least time efficient since its time complexity is $O(2^n)$.

3. **Greedy**

   The Greedy algorithm is extremely fast, with low memory usage. It is effective for large values of N and T and provides a quick approximate solution. Its accuracy is very poor and it fails when subsets require combining smaller elements to reach T.

Trade Offs	Backtracking	DP	Greedy
Time	very high	moderate	low
Memory	low	high	low
Accuracy	good	good	poor

5 Conclusion

5.1 Managing the problem in practice

Depending on the context of the problem and its constraints, we can choose between the three analyzed algorithms: backtracking, dynamic programming or greedy approach.

1. Dynamic programming is particularly effective for medium-sized problems with well-defined subproblems, especially when time and space complexity are a priority. It is highly suitable for scenarios where resource allocation must fit within a fixed budget, or for portfolio optimization, where the decision process needs to take into account multiple options, while aiming for a specific return. Furthermore, in IT infrastructure management, dynamic scheduling can optimize server allocation based on performance requirements and cost constraints.

2. Backtracking is a great choice when the problem size is small and it's mandatory to explore all possible combinations to find an exact solution. For example, when organizing an event with a limited budget, this method can identify which combination of expenses (such as space rental, catering and materials) perfectly fits the amount available. Backtracking can also be used in solving logic games or math puzzles, such as choosing a subset of numbers that obey a certain rule. It is ideal in situations where accuracy is more important than execution time.

3. The greedy method excels in scenarios where the set is sorted and the exact solution is not critical, time being the priority. This approach is particularly advantageous when time constraints demand a rapid yet reasonably accurate outcome. For instance, it is highly effective in tasks such as providing a preliminary cost estimate for a project, where an approximate allocation of resources is sufficient to make an informed decision. Additionally, this algorithm is well-suited for situations like logistics, where initial approximations for space or weight allocation in transport can guide further planning.

Sample Heading (Fourth Level) The contribution should contain no more than four levels of headings. Table 2 gives a summary of all heading levels.
Displayed equations are centered and set on a separate line.

$$x + y = z \tag{1}$$

Please try to avoid rasterized images for line-art diagrams and schemas. Whenever possible, use vector graphics instead (see Fig. 5).

Table 2. Table captions should be placed above the tables.

Heading level	Example	Font size and style
Title (centered)	**Lecture Notes**	14 point, bold
1st-level heading	**1 Introduction**	12 point, bold
2nd-level heading	**2.1 Printing Area**	10 point, bold
3rd-level heading	**Run-in Heading in Bold.** Text follows	10 point, bold
4th-level heading	*Lowest Level Heading.* Text follows	10 point, italic

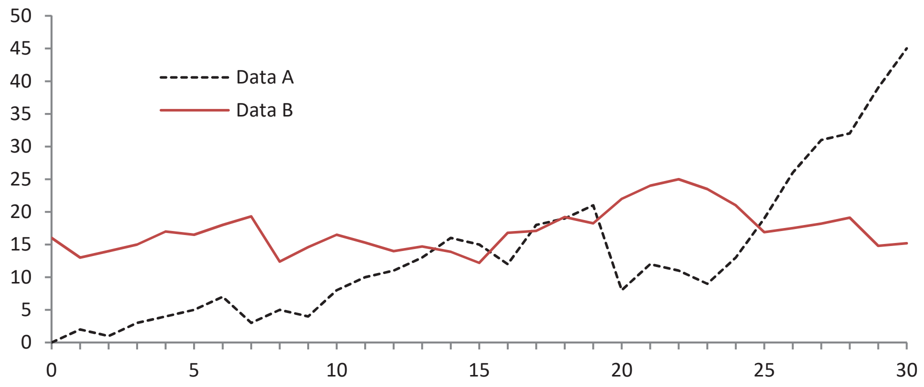

Fig. 5. A figure caption is always placed below the illustration. Please note that short captions are centered, while long ones are justified by the macro package automatically.

Theorem 1. *This is a sample theorem. The run-in heading is set in bold, while the following text appears in italics. Definitions, lemmas, propositions, and corollaries are styled the same way.*

Proof. Proofs, examples, and remarks have the initial word in italics, while the following text appears in normal font.

For citations of references, we prefer the use of square brackets and consecutive numbers. Citations using labels or the author/year convention are also acceptable. The following bibliography provides a sample reference list with entries for journal articles [**?**], an LNCS chapter [2], a book [3], proceedings without editors [4], and a homepage [5]. Multiple citations are grouped [**?**,2,3], [**?**,3,4,5].

Acknowledgments. A bold run-in heading in small font size at the end of the paper is used for general acknowledgments, for example: This study was funded by X (grant number Y).

Disclosure of Interests. It is now necessary to declare any competing interests or to specifically state that the authors have no competing interests. Please place the statement with a bold run-in heading in small font size beneath the (optional) acknowledgments[1], for example: The authors have no competing interests to declare that are relevant to the content of this article. Or: Author A has received research grants from Company W. Author B has received a speaker honorarium from Company X and owns stock in Company Y. Author C is a member of committee Z.

References

1. Github link for scripts and tests
2. Author, F., Author, S.: Title of a proceedings paper. In: Editor, F., Editor, S. (eds.) CONFERENCE 2016, LNCS, vol. 9999, pp. 1–13. Springer, Heidelberg (2016). `https://doi.org/10.10007/1234567890`
3. Author, F., Author, S., Author, T.: Book title. 2nd edn. Publisher, Location (1999)
4. Author, A.-B.: Contribution title. In: 9th International Proceedings on Proceedings, pp. 1–2. Publisher, Location (2010)
5. LNCS Homepage, `http://www.springer.com/lncs`, last accessed 2023/10/25

[1] If EquinOCS, our proceedings submission system, is used, then the disclaimer can be provided directly in the system.