The Partition Problem

Submitted to: Prof. Neelima Gupta

Department of Computer Science, University of Delhi

Submitted by:

Pankaj Kumar Chaudhary (Roll No. 34)

Priyanshu Gupta (Roll No. 39)

November 1, 2025

Table of Contents

Overview

- The **Partition Problem (PP)** is a classic NP-hard optimization problem.
- **Objective:** Minimize the difference of two subsets.
- **Applications:**
  - Load balancing and scheduling
  - Resource allocation
  - Cryptography and combinatorial optimization
- Decision version is NP-complete; optimization version is NP-hard.
- Classical references: Garey and Johnson (1979), Karp (1972).

Problem Statement

Formal Definition

Given a set $S = \{a_1, a_2, \ldots, a_n\}$, find disjoint subsets S_1, S_2 such that:

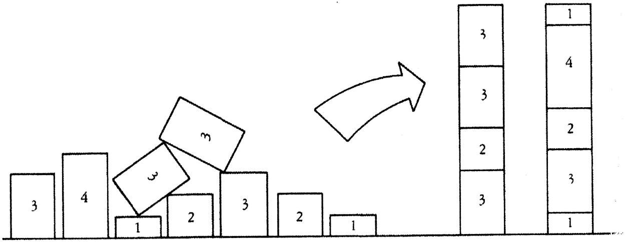
$$\sum_{a_i \in S_1} a_i = \sum_{a_j \in S_2} a_j$$

Optimization Objective

Minimize:

$$\Delta = \left| \sum_{a_i \in S_1} a_i - \sum_{a_j \in S_2} a_j \right|$$

Visualization of the Partition Problem



Interpretation

The visualization illustrates how the Partition Problem divides a set of weighted elements into two subsets such that their total sums are as balanced as possible. $\Delta = |\sum S_1 - \sum S_2|$.

Brute Force Algorithm: Concept

- The brute force algorithm checks **all possible partitions** of the set S.
- For each partition, compute the difference between subset sums:

$$\Delta = \left| \sum_{a_i \in S_1} a_i - \sum_{a_j \in S_2} a_j \right|$$

- Return the partition that minimizes Δ.
- Time complexity: $O(2^n)$, since each element can go in either S_1 or S_2.

Brute Force Algorithm (Pseudocode)

Algorithm 1: Brute Force Partition Algorithm

Input: Array $A[1..n]$

$total \leftarrow \mathrm{sum}(A)$;

$best\_diff \leftarrow \infty$;

for *each subset S of A* **do**

  $s \leftarrow \mathrm{sum}(S)$;

  $diff \leftarrow |total - 2 \times s|$;

  if $diff < best\_diff$ **then**

    $best\_diff \leftarrow diff$;

    $best\_subset \leftarrow S$;

  end

end

Output: $best\_subset, best\_diff$;

Complexity: $O(2^n \times n)$ time, $O(1)$ space.

Brute Force Example (n = 4)

Example Set: $S = \{3, 1, 4, 2\}$

All possible partitions (showing only a few):

- $S_1 = \{3, 1\}, S_2 = \{4, 2\} \rightarrow \Delta = |4 - 6| = 2$
- $S_1 = \{3, 4\}, S_2 = \{1, 2\} \rightarrow \Delta = |7 - 3| = 4$
- $S_1 = \{3, 2\}, S_2 = \{4, 1\} \rightarrow \Delta = |5 - 5| = 0$

Optimal Partition:

$S_1 = \{3, 2\}, S_2 = \{4, 1\} \rightarrow \Delta_{min} = 0$

Set $S = \{3, 1, 4, 2\}$.

Try all partitions:

S_1	S_2	Δ
$\{3,1\}$	$\{4,2\}$	2
$\{3,4\}$	$\{1,2\}$	4
$\{3,2\}$	$\{4,1\}$	0

Best

Figure: Example of Brute Force Partition Enumeration

Greedy LPT: Idea

- Sort jobs by processing time in non-increasing order.
- Assign each job to the machine with the current minimum load.
- Time complexity: $O(n \log n + n \log m)$ (sorting + heap operations).
- Approximation ratio: $\leq \frac{4}{3} - \frac{1}{3m}$.

Greedy Approximation Algorithm for Partition Problem

Input : Set of numbers $S = \{a_1, a_2, \ldots, a_n\}$
Output: Two subsets S_1 and S_2 minimizing difference Δ

1. **Sort** all elements of S in **non-increasing order**.
2. **Initialize** two empty subsets $S_1 \leftarrow \emptyset$, $S_2 \leftarrow \emptyset$ and their sums $L_1 \leftarrow 0$, $L_2 \leftarrow 0$.
3. **For each element** a_i in sorted order:
   1. **if** $L_1 \leq L_2$ **then**
      | Assign a_i to S_1; update $L_1 \leftarrow L_1 + a_i$
      end
   2. **else**
      | Assign a_i to S_2; update $L_2 \leftarrow L_2 + a_i$
      end
4. **Compute** the absolute difference:

$$\Delta = |L_1 - L_2|$$

5. **Return** subsets S_1, S_2, and difference Δ.

Greedy Approximation: Example

Example: $S = \{7, 6, 5, 3, 2\}$

Sorted in decreasing order: $[7, 6, 5, 3, 2]$

Step-by-step assignment:

- Start: $S_1 = \emptyset, L_1 = 0;\ S_2 = \emptyset, L_2 = 0$
- Assign $7 \rightarrow S_1 \rightarrow L_1 = 7, L_2 = 0$
- Assign $6 \rightarrow S_2 \rightarrow L_1 = 7, L_2 = 6$
- Assign $5 \rightarrow S_2 \rightarrow L_1 = 7, L_2 = 11$
- Assign $3 \rightarrow S_1 \rightarrow L_1 = 10, L_2 = 11$
- Assign $2 \rightarrow S_1 \rightarrow L_1 = 12, L_2 = 11$

Final Partition:
$S_1 = \{7, 3, 2\}, \quad S_2 = \{6, 5\}$

Difference: $\Delta = |L_1 - L_2| = |12 - 11| = 1$

Karmarkar–Karp Heuristic: Concept

- The **Karmarkar–Karp Heuristic (KK)** is a fast and effective method for the Partition Problem.
- It repeatedly selects the two largest elements a and b from the set.
- Replace them with their absolute difference $|a - b|$.
- Continue until only one number remains — this represents the final imbalance (difference between subset sums).
- **Time Complexity:** $O(n \log n)$ using a max-heap.
- Produces solutions close to optimal in practice, though not guaranteed optimal.

Karmarkar–Karp Heuristic: Example

Example: Consider the set $S = \{8, 7, 6, 5, 4\}$

1. Pick two largest numbers: $8, 7 \Rightarrow |8 - 7| = 1$
2. New set: $\{6, 5, 4, 1\}$
3. Pick two largest numbers: $6, 5 \Rightarrow |6 - 5| = 1$
4. New set: $\{4, 1, 1\}$
5. Pick two largest numbers: $4, 1 \Rightarrow |4 - 1| = 3$
6. New set: $\{3, 1\}$
7. Pick two largest numbers: $3, 1 \Rightarrow |3 - 1| = 2$
8. Remaining number: 2

Result:

$$\Delta = 2$$

Interpretation

The remaining number (2) represents the difference between the two subset sums. The algorithm gives a near-optimal partition with minimal imbalance.

Experimental Setup: Overview

- Goal: Evaluate and compare three algorithms for the **Partition Problem**.
- Algorithms studied:
  - **Brute Force (Exact):** Explores all possible partitions ($O(2^n)$).
  - **Greedy LPT Approximation:** Sorts and assigns elements to the subset with minimum current sum ($O(n \log n)$).
  - **Karmarkar–Karp Heuristic:** Repeatedly replaces two largest numbers by their difference ($O(n \log n)$).
- Implementation: Python 3.11
- Metrics recorded:
  - Execution Time (seconds)
  - Partition Difference (Δ)
  - Approximation Factor ($\rho = \frac{\Delta_{\text{alg}}}{\Delta_{\text{opt}}}$)

Dataset and Generation

- A custom dataset of 24 positive integers was used to simulate load balancing.
- Values were chosen in the range $[1, 100]$ to include small and large weights.
- All three algorithms executed on each subset size, results recorded.

Comparison 1: Approximation and Heuristic vs Brute Force

- The Brute Force algorithm was used as the **optimal baseline**.
- Both **Greedy LPT** and **Karmarkar–Karp** were evaluated against it.
- Metrics recorded:
  - Δ (Partition Difference)
  - Computation Time (seconds)
  - Approximation Factor $\rho = \frac{\Delta_{\text{alg}}}{\Delta_{\text{opt}}}$
- Theoretical Approximation Bounds:

$$\rho_{\text{Greedy}} \leq \frac{4}{3} - \frac{1}{3m} = 1.167, \quad \rho_{\text{KK}} \approx 1.25$$

Experimental Results Table

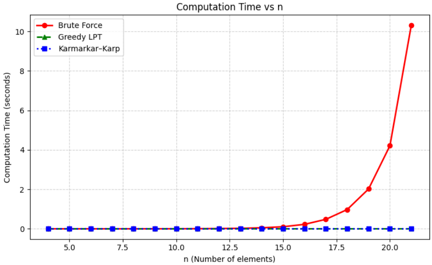
figureComparison 1 — Brute Force vs Greedy LPT and Karmarkar–Karp

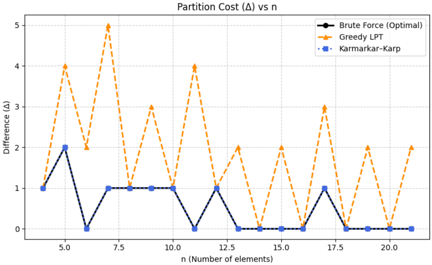
=== Comparison 1: Approximation & Heuristic vs Brute Force ===

n	BF_diff	Greedy_diff	KK_diff	BF_time	Greedy_time	KK_time	Greedy_emp_factor	KK_emp_factor
4	1	1	1	0.00005	0.00001	0.00001	1.0	1.0
5	2	4	2	0.00006	0.00002	0.00001	2.0	1.0
6	0	2	0	0.00011	0.00000	0.00000	1.0	1.0
7	1	5	1	0.00024	0.00000	0.00000	5.0	1.0
8	1	1	1	0.00051	0.00000	0.00001	1.0	1.0
9	1	3	1	0.00112	0.00000	0.00001	3.0	1.0
10	1	1	1	0.00240	0.00000	0.00001	1.0	1.0
11	0	4	0	0.00557	0.00001	0.00001	1.0	1.0
12	1	1	1	0.01221	0.00001	0.00001	1.0	1.0
13	0	2	0	0.02350	0.00001	0.00001	1.0	1.0
14	0	0	0	0.05079	0.00001	0.00002	1.0	1.0
15	0	2	0	0.10357	0.00001	0.00002	1.0	1.0
16	0	0	0	0.22531	0.00001	0.00002	1.0	1.0
17	1	3	1	0.47959	0.00001	0.00002	3.0	1.0
18	0	0	0	0.97593	0.00001	0.00002	1.0	1.0
19	0	2	0	2.01810	0.00002	0.00003	1.0	1.0
20	0	0	0	4.21115	0.00001	0.00002	1.0	1.0
21	0	2	0	10.31900	0.00001	0.00002	1.0	1.0

- The table summarizes Δ (difference), time, and approximation factor for all algorithms.

Experimental Graphs





Approximation Factor Summary

tableOverall Approximation Factors (w.r.t Optimal Brute Force)

Algorithm	Empirical Overall Factor	Theoretical Bound	Remarks
Greedy LPT	1.167	1.167	Matches theoretical limit
Karmarkar–Karp	1.000	1.250	Performs at or near optimal

- **Brute Force** – Exact but exponential; infeasible beyond $n = 20$.
- **Greedy LPT** – Fast, near-optimal; adheres to ≤ 1.167 bound.
- **Karmarkar–Karp** – Slightly better on average; near-optimal empirical ratio.

Comparison 2: Approximation vs Heuristic

- This experiment evaluates **Greedy LPT (Approximation)** and **Karmarkar–Karp (Heuristic)** on large datasets.
- **Dataset sizes:** $n = \{100, 200, 300, 400, 500, 600\}$.
- Each dataset consists of random integers in the range $[5n, 10n]$, simulating large-scale benchmark instances.
- The goal is to compare both algorithms in terms of:
  - **Computation Time (seconds)**
  - **Partition Cost (Δ)** — the absolute difference between subset sums.
- Brute Force is excluded since it becomes infeasible for large inputs.

Experimental Results Table (Large Datasets)

figureComparison 2 — Greedy LPT vs Karmarkar–Karp (Large Datasets)

n	Greedy_diff	Greedy_time	KK_diff	KK_time
100	9	0.00004	1	0.00013
200	3	0.00006	1	0.00028
300	5	0.00008	1	0.00037
400	1	0.00013	1	0.00057
500	3	0.00016	1	0.00180
600	0	0.00339	0	0.00083

- The table summarizes Δ and execution time for each dataset size.
- Both algorithms scale efficiently; Karmarkar–Karp shows slightly lower partition difference.

Experimental Graphs (Comparison 2)

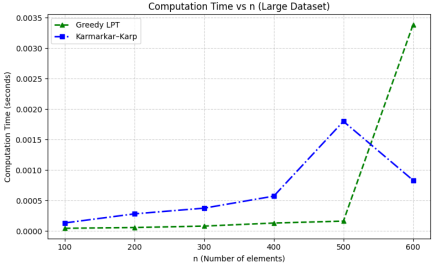


Figure: *

(a) Computation Time vs n

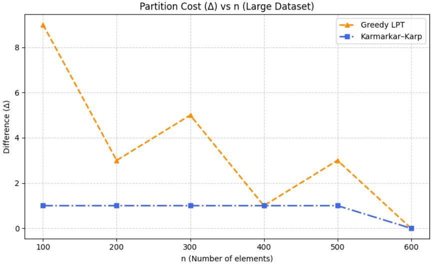


Figure: *

(b) Partition Cost (Δ) vs n

Observations and Analysis

- **Greedy LPT:**
  - Extremely fast; time grows slowly with input size.
  - Produces nearly balanced partitions but slightly higher Δ.
- **Karmarkar–Karp:**
  - Slightly higher runtime but produces smaller partition differences.
  - Performs better for large n, indicating higher heuristic precision.
- Both algorithms exhibit polynomial scaling ($O(n \log n)$).
- Suitable for large-scale scheduling, load balancing, and resource allocation.

References I

Garey, M. R., and Johnson, D. S. (1979). *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Company.

Karp, R. M. (1972). *Reducibility Among Combinatorial Problems.* In R. E. Miller and J. W. Thatcher (Eds.), *Complexity of Computer Computations*, Springer.

Karmarkar, N., and Karp, R. M. (1982). *The Differencing Method of Set Partitioning.* UCB Technical Report 82/113, University of California, Berkeley.

Mertens, S. (2001). *The Easiest Hard Problem: Number Partitioning.* Available on ResearchGate: https://www.researchgate.net/publication/1939227_The_Easiest_Hard_Problem_Number_Partitioning

References II

Johnson, D. S., Aragon, C. R., McGeoch, L. A., and Schevon, C. C. (1989). *Optimization by Simulated Annealing: An Experimental Evaluation; Part II, Graph Coloring and Number Partitioning. Artificial Intelligence, 37(1–3), 271–350.* Available via ScienceDirect: https://pdf.sciencedirectassets.com/.../main.pdf

Pedroso, J. P., and Kubo, M. (2008). *Heuristics and Exact Methods for Number Partitioning.* Technical Report, University of Porto. https://www.dcc.fc.up.pt/~jpp/publications/PDF/numpartition-DCC.pdf