Solving the Knapsack 0/1 Problem: A Real-World Application

Presented by:

Ananda Irwin
Adam Light
Brent Maxwell

Why Do We Care About the Knapsack 0/1 Problem?

A real-life parallel to this project is a common problem encountered with devices known as microcontrollers (MCUs): A loss of power event. In many situations, an embedded system may receive more inputs than it has storage for.

When a device determines that it is losing power, it needs to determine:

- Which inputs are worth recording
- How many inputs can be recorded given available storage
- The time needed to write those inputs into storage.

Pictured: Arduino (left), Raspberry Pi (Right)

Which Algorithm Fits Best?

We will attempt to solve the Knapsack problem using the following algorithms:

- Brute force
- Greedy heuristic
- Dynamic Programming (Bottom up)
- Memoization (Top down with recursion)
- Fully-Polynomial-Time-Approximation-Schema (FPTAS)

There is a window of time from when the MCU detects that it is losing power, and when the device will no longer function. (Explained by the graphic on the right.)

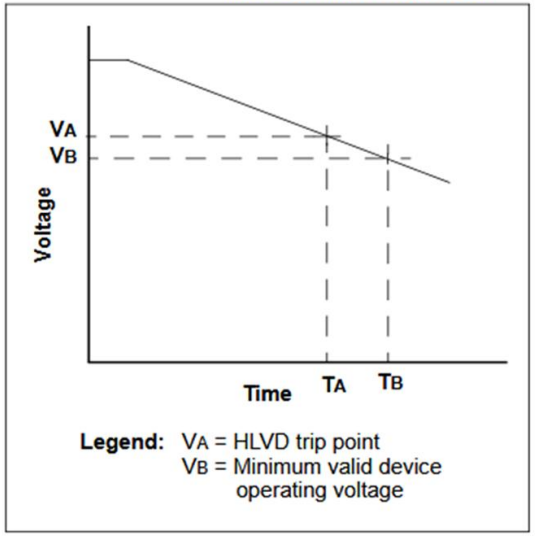
We want the most optimal solution in the least amount of time. The goal is to find an algorithm that best fits these goals.

39.5 Applications

In many applications, it is desirable to detect a drop below, or rise above, a particular voltage threshold. For example, the HLVD module could be periodically enabled to detect Universal Serial Bus (USB) attach or detach. This assumes the device is powered by a lower voltage source than the USB when detached. An attach would indicate a High-Voltage Detect from, for example, 3.3V to 5V (the voltage on USB) and vice versa for a detach. This feature could save a design a few extra components and an attach signal (input pin).

For general battery applications, Figure 39-4 shows a possible voltage curve. Over time, the device voltage decreases. When the device voltage reaches voltage, VA, the HLVD logic generates an interrupt at time, TA. The interrupt could cause the execution of an Interrupt Service Routine (ISR), which would allow the application to perform "housekeeping tasks" and a controlled shutdown before the device voltage exits the valid operating range at TB. This would give the application a time window, represented by the difference between TA and TB, to safely exit.

FIGURE 39-4: TYPICAL LOW-VOLTAGE DETECT APPLICATION



Legend: VA = HLVD trip point
VB = Minimum valid device operating voltage

Excerpt from the PIC18F27K42 datasheet, a production-level MCU

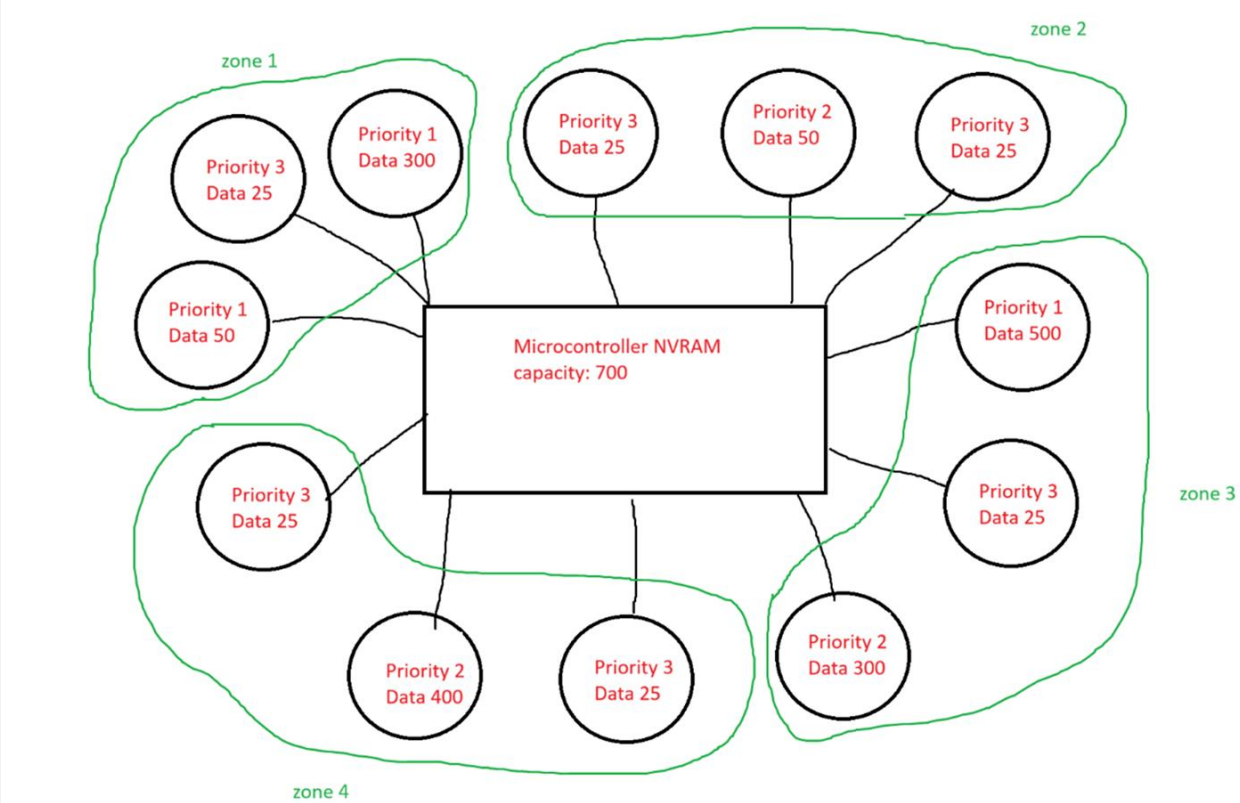
Scenario:

The diagram to the right shows several moisture sensors covering various zones are connected to a MCU. Each sensor has a priority level[1] and an amount of accumulated data.

Our theoretical MCU has a limited amount of non-volatile storage for readings.

The goal is to determine the maximum amount of high priority data we can log, while covering all 4 zones.

[1] Lower is higher priority



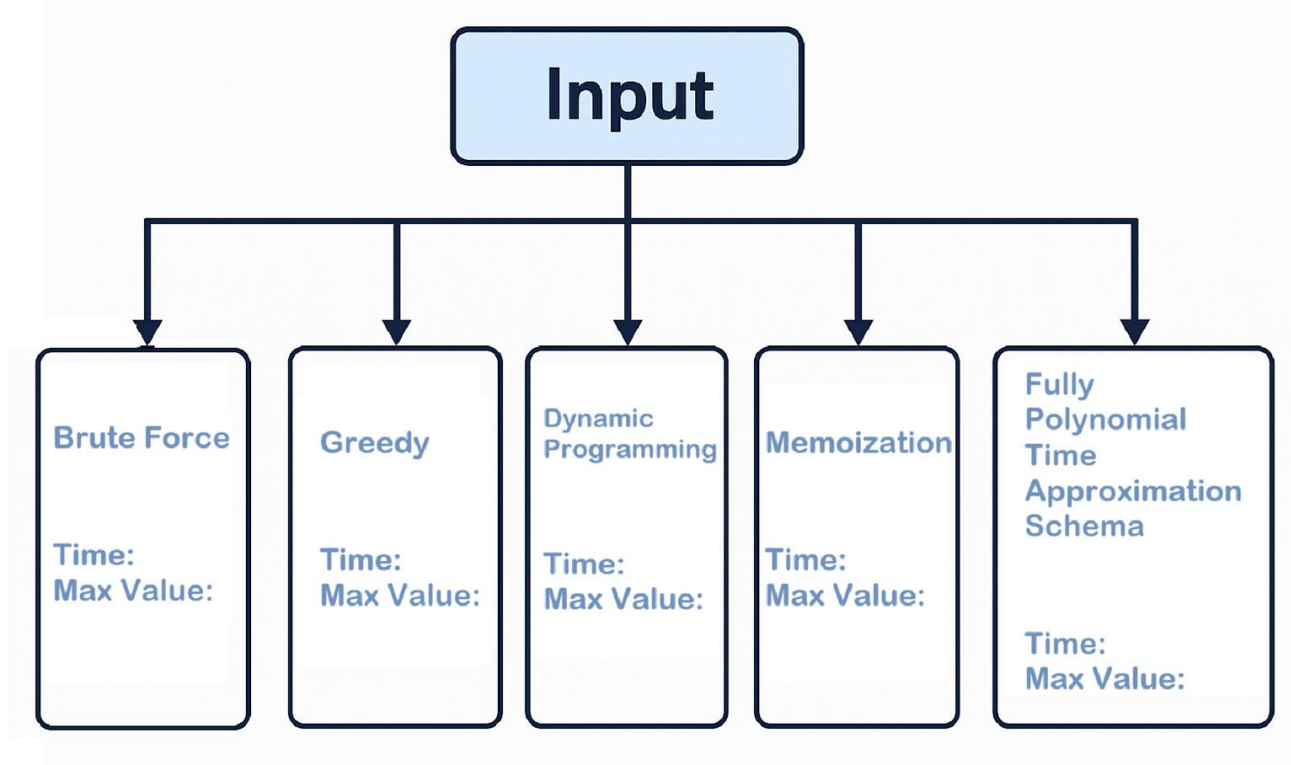
Setting up the input:

Because of the limited system resources of an MCU, the input data size had to be limited:

- Items will be a struct with attributes for their value (uint16) and weight (uint8)
- The items will be contained in a vector of size **65536** (occupying a total of **192**KB of system resources)

This input data will then be fed into each algorithm to determine two factors:

- Time to finish
- Maximum value able to fit into memory. (Value from our knapsack problem.)



BRUTE FORCE

Time Complexity: O(n^2)

Explores every combination of items and compares their weights and values

Greatest total value under the weight limit is returned

ALWAYS returns Optimal Solution

Assumptions:

- We expect for this not to be a viable algorithm after relatively small input sizes.

Questions:

- Will this be viable for very small input sizes because of little overhead?

GREEDY

Time Complexity: O($n \log n$)

Computes each items value to weight ratio

Greedily chooses items with the highest value to weight ratio until the knapsack capacity has been met

May not return the Optimal Solution, but rather a local optimal solution.

Assumptions:

- This algorithm should provide a very quick alternative to all other algorithms, even at large input sizes

Questions:

- Will the solution returned be within an acceptable range from the optimal solution?

DYNAMIC PROGRAMMING

Time Complexity: O(nW)
(W is the capacity of the knapsack)
Or more precisely: O($n * 2^b$)
(b is bits required to represent W)

Creates a 2D array that stores the solution to subproblems, storing item and weight values.

Then backtracks to find the items used to yield the solution.

ALWAYS returns Optimal Solution.

Assumptions:

- We expect this to be faster than brute force, but slower than greedy.

Questions:

- Will this be viable for larger input sizes?
- Is the guaranteed optimal solution worth the extra computation power vs the greedy solution?

MEMOIZATION

Time Complexity: O(nW)
(W is the capacity of the knapsack)
Or more precisely: O($n * 2^b$)
(b is bits required to represent W)

Creates a 2D array that stores the solution to subproblems, storing item and weight values, but does not recalculate any subproblems.

Then backtracks to find the items used to yield the solution.

ALWAYS returns Optimal Solution.

Assumptions:

- We expect this to be faster than brute force, but slower than greedy.

Questions:

- Will this be viable for larger input sizes?
- Is the guaranteed optimal solution worth the extra computation power vs the greedy solution?
- How much faster is it than normal dynamic programming?

FULLY POLYNOMIAL TIME APPROX. SCHEME

Time Complexity: $O(n^2 \left\lfloor \frac{n}{\varepsilon} \right\rfloor)$

(Epsilon is the difference from the Optimal Solution)

Scales the magnitude of the values to be bounded by *n.*

Uses the same algorithm as the dynamic programming solution otherwise.

ALWAYS returns solution within $(1 - \varepsilon)$ * Optimal Solution

Assumptions:

- We expect this to be faster than brute force, but slower than greedy.

Questions:

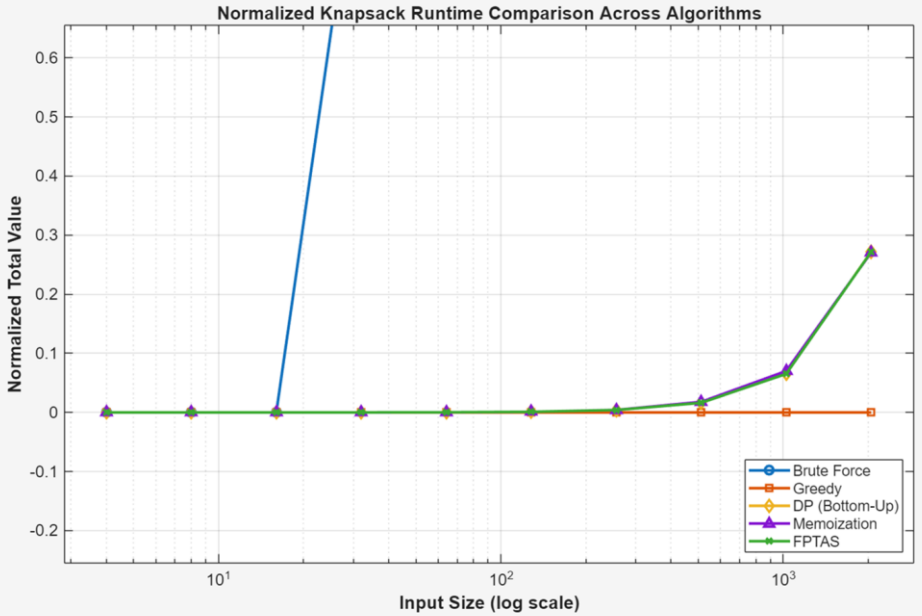
- Will this be viable for larger input sizes?
- Is the guaranteed optimal solution worth the extra computation power vs the greedy solution?
- Will this be faster than dynamic programming?
- How practical are the applications of this algorithm?
- Is the polynomial time actually an improvement with the data we are working with?

Which Algorithm is Suitable?

Based on a 20ms **best case** window for a microcontroller to write to memory, the greedy algorithm is the clear winner.

- Greedy is the *only* algorithm capable of a 20ms write window at higher input sizes.
- Brute force is viable and on par with greedy at *small* input sizes while yielding the optimal solution.
- The recursive algorithms perform better than brute, but don't provide significant accuracy over greedy.

All algorithms but greedy surpassed the 20ms window of write time at large input sizes.



Excel spreadsheet for runtime (in microseconds):

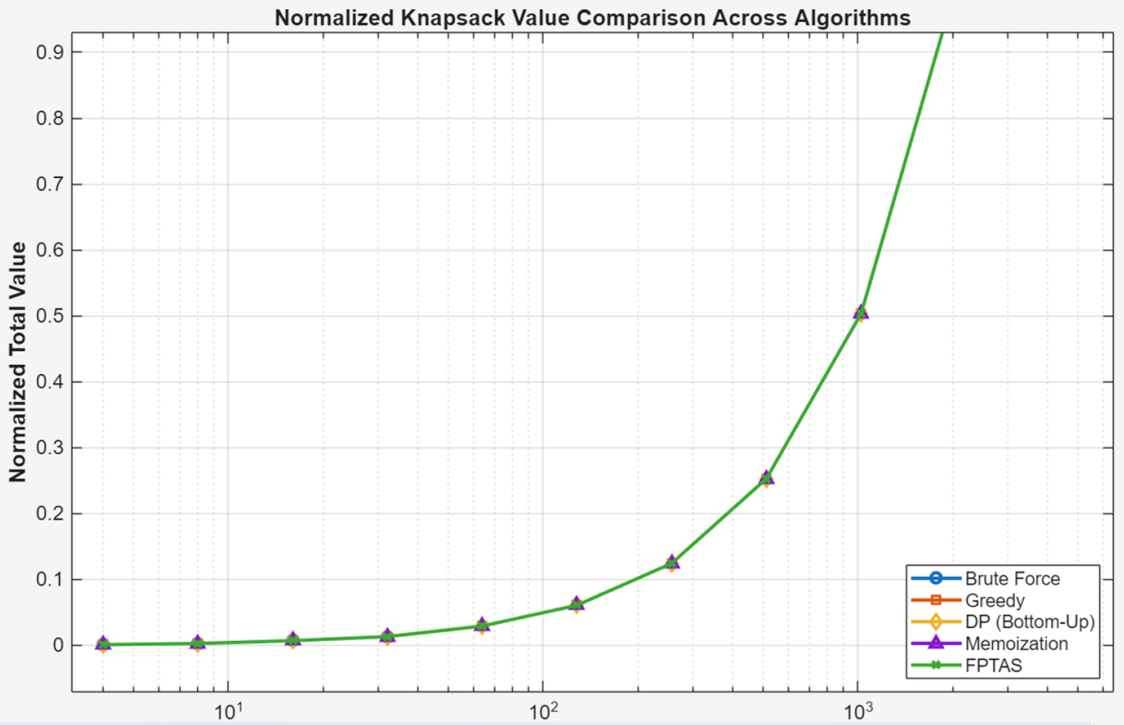
A	B	C	D	E	F
	Brute	Greedy	Dynamic P	Memo	FPTAS
4	0	0	17	0	17
8	1	0	65	2	69
16	271	1	275	93	279
32	17570605	3	1107	762	1102
64		6	4483	3722	4497
128		13	17842	17496	17865
256		32	72486	73503	71998
512		71	287488	313156	286231
1024		164	1148618	1230553	1145722
2048		361	4778543	4754414	4772794

Maximum Value Results:

There is no noticeable difference for the maximum value each algorithm provided from the chart. Viewing the raw data reveals the minor differences from the optimal solution.

Values diverge between greedy, FPTAS, and the remaining algorithms.

This proves the greedy algorithm does not always provide the overall maximum value, but it *does* produce a value that is within 1% of the optimal solution.



A	B	C	D	E	F
	Brute	Greedy	Dynamic P	Memo	FPTAS
4	58981	58981	58981	58981	57882
8	180420	180420	180420	180420	179337
16	466267	466267	466267	466267	464500
32	884758	884758	884758	884758	882994
64		1961825	1961825	1961825	1960304
128		4092156	4094428	4094428	4092855
256		8380759	8380904	8380904	8379403
512		17052703	17052703	17052703	17051147
1024		33957707	33958370	33958370	33956795
2048		67461780	67462366	67462366	67460782

Overall Conclusion

Greedy is the best algorithm for a hypothetical MCU writing sensor data to memory:

- Gives us an approximation close to the optimal solution.
- Very fast with linear time complexity.
- With variable power outage times, a greedy approach could be used to write data until the device powers off.

Answers to our questions:

- FPTAS with a 5% error runs practically identically to dynamic prog. It is not very practical.
- Memoization does generally run better than dynamic and FPTAS but with diminishing returns on the speed.
- The additional time of the recursive methods are not significant for the sample data.

Sources

[1] Microchip, "PIC18F27K42-I/SO Datasheet (PDF)," [Online]. Available: https://www.mouser.com/datasheet/2/268/PIC18_L_F26_27_45_46_47_55_56_57K4 2_Data_Sheet_400-3442767.pdf. [Accessed 28 7 2025].

[2] K. Lai, "The Knapsack Problem and Fully Polynomial Time," 10 3 2006. [Online]. [Accessed 28 7 2025].

[3] A. Levitin, Introduction to The Design and Analysis of Algorithms (3rd Edition), New Jersey: Pearson, 2012.

[4] S. Fleeman, "The Trick to Large Value, Low-Tolerance Capacitors," 21 12 2022. [Online]. Available: https://www.engineering.com/the-trick-to-large-value-low-tolerance-capacitors/.

The only use of generative AI for this project was in generating a MATLAB script to graph the spreadsheet data gathered for runtime, total value and number of basic operations. This was the prompt used to generate the script which was then modified:
“In MATLAB, is there a way I could import a CSV spreadsheet and graph the performance of different algorithms?”