CPS 3440 Project Report

Bounded Knapsack Implementations and Differential Testing and Empirical Complexity Verification

Team: Kehan Zhu, Ruiqi Tian
Course: CPS 3440 (FA25)
Instructor: Dr. Omar
Date: Dec, 2025
Repository: https://github.com/remember-4/Bounded-knapsack

1 Scope

- `knapsack1.cpp`: 2D DP for bounded knapsack (explicitly enumerates the chosen count per item type).
- `knapsack2.cpp`: 1D DP for bounded knapsack by repeating a 0/1 update exactly c_i times.
- `knapsack3.cpp`: 1D DP with **binary splitting** $(1, 2, 4, \ldots)$ to convert bounded items into multiple 0/1 items.
- `knapsack4.cpp`: 1D DP with **monotonic-queue (sliding window)** optimization grouped by capacity modulo w_i.
- `test.cpp`: differential testing harness (random input generation + compares `knapsack1.exe` vs `knapsack4.exe`).
- `verify.py`: empirical runtime scaling benchmark (auto-generates a C++ benchmark for the monotonic-queue solver, records runtimes, and plots runtime vs $V \times N$ with R^2).

No algorithms or conclusions are claimed beyond what is supported by these files and their outputs.

2 Problem Definition

Input format used by all four solvers:

- Integers N (number of item types) and V (knapsack capacity).
- For each item type i: integers (w_i, v_i, c_i) where: w_i = weight (capacity cost), v_i = value, c_i = count limit.

Output: a single integer, the maximum total value achievable with capacity V under the per-item count limits.
Formally:

$$\max \sum_{i=1}^{N} x_i v_i \quad \text{s.t.} \quad \sum_{i=1}^{N} x_i w_i \le V,\ 0 \le x_i \le c_i,\ x_i \in \mathbb{Z}.$$

3 Shared Optimization Present in All Four Solvers

All four solver files apply the same preprocessing step:

$$c_i \leftarrow \min\left(c_i, \left\lfloor \frac{V}{w_i} \right\rfloor\right).$$

Reason: even if the input allows many copies, the knapsack cannot physically contain more than $\lfloor V/w_i \rfloor$ copies of item i.

4 `knapsack1.cpp`: 2D DP (Baseline Implementation)

4.1 State

$$\texttt{dp}[i][j] = \text{maximum value using the first } i \text{ item types with capacity } j.$$

4.2 Transition

For each item type i and capacity j:

- Initialize with "take 0 copies of item i": $\texttt{dp}[i][j] \leftarrow \texttt{dp}[i-1][j]$.
- Enumerate number of copies $k = 1..c_i$ while $kw_i \le j$:

$$\texttt{dp}[i][j] \leftarrow \max\big(\texttt{dp}[i][j],\ \texttt{dp}[i-1][j-kw_i] + kv_i\big).$$

Output printed by this file is $\texttt{dp}[N][V]$.

4.3 Complexity (Directly from Loop Structure)

Time is proportional to iterating over i, j, and k: $O(\sum_i V \cdot c_i)$.
Space uses a 2D table: $O(NV)$.

Implementation limits in this file: `MAXN` $= 10^3 + 5$, `MAXV` $= 10^5 + 5$, and `dp[MAXN][MAXV]`.

5 `knapsack2.cpp`: 1D DP by Repeating a 0/1 Update c_i Times

5.1 State

$$\texttt{dp}[j] = \text{maximum value achievable with capacity } j.$$

5.2 Transition

For each item type i, the code repeats a standard 0/1 update exactly c_i times:

$$\texttt{dp}[j] \leftarrow \max(\texttt{dp}[j], \texttt{dp}[j-w_i] + v_i), \quad \text{for } j = V, V-1, \dots, w_i.$$

The loop over j is descending, which is the standard 0/1 pattern. Output printed by this file is $\texttt{dp}[V]$.

5.3 Complexity (Directly from Loop Structure)

Time is proportional to iterating over i, repeating c_i times, and scanning capacities: $O(\sum_i V \cdot c_i)$.
Space uses one array: $O(V)$.

Implementation limits in this file: `MAXN` $= 10^4 + 5$, `MAXV` $= 10^6 + 5$, and `dp[MAXV]`.

6 `knapsack3.cpp`: 1D DP with Binary Splitting

6.1 Idea as Implemented

For each item type i, let $c = c_i$. The code splits c into powers of two:

$$1, 2, 4, 8, \dots$$

Each split creates a derived 0/1 item with:

$$w = k \cdot w_i, \quad v = k \cdot v_i,$$

and then performs a standard descending 0/1 update:

$$\texttt{dp}[j] \leftarrow \max(\texttt{dp}[j], \texttt{dp}[j-w] + v), \quad j = V, V-1, \dots, w.$$

A final remainder item is created if `c>0`. Output printed by this file is $\texttt{dp}[V]$.

6.2 Complexity (From the Split Count)

Binary splitting reduces per-type updates from c_i to $O(\log c_i)$ derived items:

$$O\left(V \sum_{i=1}^{N} \log c_i\right)$$

with space $O(V)$.

7 `knapsack4.cpp`: 1D DP with Monotonic Queue Optimization

7.1 Grouping by Residue Class

For each item type i, the code iterates over $\texttt{mod} = 0, 1, \ldots, w_i - 1$ and visits:

$$j = \texttt{mod} + k \cdot w_i \quad (k = 0, 1, 2, \ldots)$$

implemented as `for (j=mod, k=0; j<=V; j+=w, k++)`.

7.2 Queue-Driven Update Used in the Code

For each such j, it computes:

$$\texttt{curVal} = \texttt{dp}[j] - k \cdot v_i.$$

It maintains arrays `q[]` and `val[]` as a deque with `head/tail`, enforcing:

- **Window size:** pop head while $k - \texttt{val[head]} > c_i$.
- **Monotonicity:** pop tail while $\texttt{q[tail-1]} \leq \texttt{curVal}$.

Then it updates:

$$\texttt{dp}[j] = \texttt{q[head]} + k \cdot v_i.$$

Output printed is $\texttt{dp}[V]$.

7.3 Complexity (From Loop Structure)

Across all residue classes, each capacity state is processed once per item with amortized $O(1)$ deque work, so the code structure is $O(NV)$ time with $O(V)$ space.

8 `test.cpp`: Differential Testing Harness

8.1 Procedure

The test runs 10,000 rounds. Each round:

1. Writes a random instance to `input.txt`.
2. Executes `knapsack1.exe < input.txt > out1.txt`.
3. Executes `knapsack4.exe < input.txt > out2.txt`.
4. Reads one integer from each output file and compares them.
5. On the first mismatch, prints the test id, prints the input via `system("type input.txt")`, and prints both answers.

8.2 Random Input Distribution

$$N \in [1, 7], \quad V \in [20, 49], \quad w \in [1, 7], \quad v \in [1, 10], \quad c \in [1, 5].$$

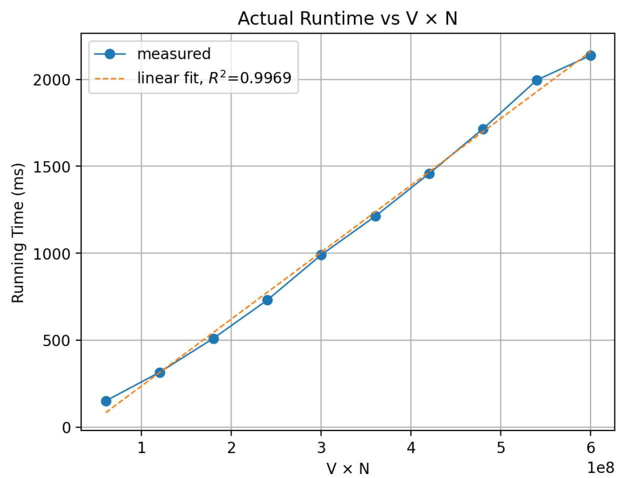

Figure 1:

9 Empirical Time-Complexity Verification

9.1 Goal

From the loop structure, `knapsack4.cpp` is expected to scale as $O(NV)$. The script `verify_knapsack_complexity.py` verifies this empirically by measuring runtimes at multiple (N, V) sizes and checking whether runtime is approximately **linear** in $V \times N$.

9.2 Benchmark-Friendly Build

The original `knapsack4.cpp` uses arrays sized by `MAXV`. If V is much smaller than `MAXV`, constant-size initialization can distort timing. The benchmark script generates a functionally equivalent monotonic-queue solver that:

- allocates `dp`, `q`, and index arrays as vectors of size $V + 1$,
- measures only the monotonic-queue DP computation for the chosen V.

This preserves the same recurrence and deque logic while making runtime scale with the selected V.

9.3 Measurement Method

For each capacity value V in a sweep, the script chooses N by:

$$N = \max(N_{\min}, \text{round}(N_{\text{scale}} \cdot V)).$$

It generates one random instance per (N, V) point (using a fixed seed), runs the solver `REPEATS` times, and records the **median** runtime in milliseconds.

9.4 Linear Fit and R^2

To check $O(NV)$ scaling, the script fits a linear model:

$$T \approx a\,(V \cdot N) + b$$

using least squares on measured pairs $\big(x = V \cdot N,\ y = T\big)$. It reports the coefficient of determination R^2 and overlays the fitted line on the plot. A value of R^2 close to 1 indicates that the measured runtime is well explained by a linear function of $V \cdot N$.

10 Conclusion

- All four solvers implement the same bounded-knapsack input format and apply the same count capping $c_i \leftarrow \min(c_i, \lfloor V/w_i \rfloor)$.
- `knapsack1.cpp` is a 2D DP baseline enumerating copy counts k per item type.
- `knapsack2.cpp` enforces boundedness by repeating a 0/1 update c_i times.
- `knapsack3.cpp` uses binary splitting to reduce updates per type from c_i to $O(\log c_i)$.
- `knapsack4.cpp` uses a monotonic queue grouped by $j \bmod w_i$ to implement bounded transitions with amortized constant work per capacity.
- `test.cpp` differentially tests `knapsack4` against `knapsack1` on randomized instances.
- `verify.py` outputs `time.out` and a plot `runtime_vs_VN.png` summarizing empirical scaling with a linear fit and R^2.