Change-Making Project

Greedy, Exhaustive Enumeration, Dynamic Programming Target, and Branch&Bound

Team: Abbas Abdallah Hasan Bazzi Bilal Zahraman

October 5, 2025

Contents

1 Introduction **3**

2 Problematic **3**

3 State of the Art **3**

4 Code **3**

5 Results **6**

5.1 5.1 Input Data 6
5.2 5.2 Best Solution and Enumeration Totals 6
5.3 5.3 Top, Worst, and Ordering Effects 6
5.4 5.4 Profiler Summary (from `metrics_summary.csv`) 7

6 Conclusion **7**

1 Introduction

We study the classic *making change* problem: given an amount and a set of coin denominations, generate all ways to pay exactly and find one with the *fewest coins*. We implement and compare several algorithms (greedy, exhaustive enumeration, DP target with early stop, branch&bound), and we measure time, memory, and I/O.

2 Problematic

Let A be the amount (in cents) and $D = \langle d_1 > \cdots > d_n \rangle$ the denominations (in cents). A solution is a vector $(k_1, \ldots, k_n) \in \mathbb{N}^n$ with

$$\sum_{i=1}^{n} k_i d_i = A.$$

The objective is to minimize the cost $C = \sum_i k_i$ (fewest coins). We also need to enumerate all solutions and study performance characteristics.

3 State of the Art

Greedy is optimal for canonical coin systems (e.g., EUR/US), but not in general. Dynamic programming (unbounded coin change) yields the optimal *count* in $O(A \cdot n)$. Exhaustive enumeration is exponential; branch&bound exploits lower bounds to prune the search tree.

4 Code

We work in **integer cents** to avoid floating-point issues. Repository: github.com/abbass03/TP. Below we show only the essential functions with brief justifications.

4-1 Greedy Baseline

Listing 1: Greedy change in cents (assumes descending coin order)

```
def greedy_change(amount_cents, coin_values):
    sol, r = [], amount_cents
    for d in coin_values:                  # expect DESC order
        k = r // d                         # take as many of coin d as possible
        sol.append((d, k))
        r -= k * d
        if r == 0:
            sol += [(dd, 0) for dd in coin_values[len(sol):]]
            break
    total_coins = sum(k for _, k in sol)
    return sol, total_coins, r
```

Why/when. One pass; fast $O(n)$. Optimal for canonical coin systems (EUR/US); not guaranteed optimal for arbitrary systems.

4-2 Exhaustive Enumeration (Iterative DFS)

Listing 2: Iterative DFS enumerating all valid combinations

```
def all_solutions_iter(amount_cents, coin_values):
```

```python
    n = len(coin_values)
    stack = [(0, amount_cents, [])]  # (index, remainder, prefix)
    while stack:
        i, r, pref = stack.pop()
        if r == 0:                   # found a full solution
            yield pref + [(coin_values[j], 0) for j in range(i, n)]
            continue
        if i == n:                   # no coin types left
            continue
        d = coin_values[i]
        max_k = r // d
        for k in range(0, max_k + 1):
            stack.append((i + 1, r - k * d, pref + [(d, k)]))
```

Why/when. Completeness without recursion; exponential in #solutions. Printing each solution can dominate runtime.

4-3 Exhaustive Enumeration (Recursive DFS)

Listing 3: Recursive DFS; tries max count down to 0 (order matters)

```python
def all_solutions_recursive(amount_cents, coin_values, i=0, prefix=None, counter=None):
    if counter is not None:
        counter['calls'] = counter.get('calls', 0) + 1
    if prefix is None:
        prefix = []
    if amount_cents == 0:
        if counter is not None:
            counter['solutions'] = counter.get('solutions', 0) + 1
        yield prefix + [(d, 0) for d in coin_values[i:]]
        return
    if i == len(coin_values):
        return
    d = coin_values[i]
    for k in range(amount_cents // d, -1, -1):  # max .. 0
        if counter is not None:
            counter['nodes'] = counter.get('nodes', 0) + 1
        yield from all_solutions_recursive(
            amount_cents - k * d, coin_values, i + 1, prefix + [(d, k)], counter=counter
        )
```

Why/when. Mirrors the iterative search; easy to instrument via `counter`. "Max→0" order often helps branch&bound by finding a good incumbent early.

4-4 DP Target & Early Stop

Listing 4: Unbounded coin-change DP for the optimal coin count

```python
def optimal_coin_count(amount_cents, coin_values):
    INF = 10**9
    dp = [0] + [INF] * amount_cents
    for a in range(1, amount_cents + 1):
        best = INF
        for d in coin_values:
            if d <= a and dp[a - d] + 1 < best:
                best = dp[a - d] + 1
```

```python
        dp[a] = best
    return dp[amount_cents]
```

Listing 5: Stop at the first enumerated solution matching the DP optimum

```python
def best_solution_stop_early(amount_cents, coin_values):
    target = optimal_coin_count(amount_cents, coin_values)
    if target >= 10**9:
        return None
    for sol in all_solutions_iter(amount_cents, coin_values):
        if sum(k for _, k in sol) == target:
            return sol
    return None
```

Why/when. DP yields the global minimum coin count. Early-stop returns the first enumerated solution with that count—optimal without scanning everything (if an optimum appears early).

4-5 Branch & Bound (Cut)

Listing 6: Exact DFS with pruning via partial-cost lower bound

```python
def best_solution_branch_and_bound(amount_cents, coin_values, max_first=True):
    best = None
    best_cost = float("inf")
    nodes, calls = 0, 0

    def dfs(i, r, prefix, partial_cost):
        nonlocal best, best_cost, nodes, calls
        calls += 1
        if r == 0:
            if partial_cost < best_cost:
                best = prefix + [(coin_values[j], 0) for j in range(i, len(
                    coin_values))]
                best_cost = partial_cost
            return
        if i == len(coin_values):
            return
        d = coin_values[i]
        max_k = r // d
        k_range = range(max_k, -1, -1) if max_first else range(0, max_k + 1)
        for k in k_range:
            nodes += 1
            new_cost = partial_cost + k
            if new_cost >= best_cost:    # lower bound: prune
                continue
            dfs(i + 1, r - k * d, prefix + [(d, k)], new_cost)

    dfs(0, amount_cents, [], 0)
    return {"best": best, "best_cost": best_cost, "nodes": nodes, "calls": calls
        }
```

Why/when. Partial coins used is a valid lower bound on any completion; if it's already $\geq$ incumbent best, the branch cannot improve $\rightarrow$ safe to cut. Order `max_first` typically prunes much more on canonical coins.

4-6 Ranking Helpers (Optional)

Listing 7: Keep the top-k fewest-coin solutions using a heap

```
def k_best_solutions(amount_cents, coin_values, k=5):
    heap = []  # (-cost, idx, sol)
    idx = 0
    for sol in all_solutions_iter(amount_cents, coin_values):
        cost = sum(c for _, c in sol)
        item = (-cost, idx, sol)         # max-heap by cost via negative key
        if len(heap) < k:
            heapq.heappush(heap, item)
        else:
            if -heap[0][0] > cost:
                heapq.heapreplace(heap, item)
        idx += 1
    top = sorted([(-c, i, s) for (c, i, s) in heap])
    return [s for _, _, s in top]
```

Listing 8: Best and two least-good (largest coin counts)

```
def best_and_two_less_worst(amount_cents, coin_values):
    best = None
    best_cost = float("inf")
    all_solutions = []
    idx = 0
    for sol in all_solutions_iter(amount_cents, coin_values):
        cost = sum(k for _, k in sol)
        all_solutions.append((cost, idx, sol))
        if cost < best_cost:
            best, best_cost = sol, cost
        idx += 1
    two_worst = sorted(all_solutions, key=lambda x: (-x[0], x[1]))[:2]
    return best, [w[2] for w in two_worst]
```

Why/when. Heaped top-k keeps memory $O(k)$ (good). The "two worst" demo stores metadata for all solutions (OK for this instance; heavy at scale).

5 Results

5.1 5.1 Input Data

$$A = 1235 \text{ cents} (= 12.35), \quad D = \{500, 200, 100, 50, 20, 10, 5\} \text{ cents.}$$

5.2 5.2 Best Solution and Enumeration Totals

- Greedy solution: $\{5.00 \times 2 + 2.00 \times 1 + 0.20 \times 1 + 0.10 \times 1 + 0.05 \times 1\}$ (coins: 6).
- Total solutions (iterative and recursive): **266,724**.
- Recursive stats: calls = **12,607,231**; nodes = **12,607,230**.
- DP target confirms optimum coin count = **6**.

5.3 5.3 Top, Worst, and Ordering Effects

- Top-5 (fewest coins): first is the 6-coin solution above; then several 7–8 coin variants.
- Two "less-worst": 0.05×247 (247 coins) and $0.10 \times 1 + 0.05 \times 245$ (246 coins).
- B&B ordering: max-first explores **77,609** nodes in $\approx$ **0.0056** s (core), while min-first explores **3,776,302** nodes in **0.397** s.

5.4 5.4 Profiler Summary (from `metrics_summary.csv`)

Algorithm	Time (s)	Peak Mem (KB)	Printed Chars
Greedy	0.000616	1.4	83
Exhaustive/Iter (print all)	40.913459	23,913.5	17,859,559
Exhaustive (store all)	37.908426	61,126.5	34
Early-Stop	0.009865	19.8	75
Branch&Bound (profiled)	0.137244	2.2	102

Note: The profiled B&B time includes measurement overhead; the raw core search (Prg7) is $\approx$ 5.6 ms.

6 Conclusion

For the Euro-like denominations and $A = 12.35$, greedy, DP target, and B&B all yield a 6-coin optimal solution. Exhaustive printing is dominated by I/O. Branch&Bound with max-first ordering prunes the search from millions of nodes to tens of thousands, giving millisecond-level core runtimes.

Repository: https://github.com/abbass03/TP.