HW2 Report: 0/1 Knapsack Problem – Algorithm Comparison

Problem Description

The 0/1 Knapsack Problem is a classical combinatorial optimization problem. Given `n` items, each with a weight `w[i]` and profit `p[i]`, and a maximum capacity `W`, the goal is to select a subset of items such that:

- The total weight does not exceed `W`
- The total profit is maximized

Each item can be either taken (`1`) or not taken (`0`). This problem is NP-complete and is frequently used to evaluate dynamic programming and greedy strategies.

Algorithm Design

This report implements and compares three different algorithmic approaches:

1. Bottom-Up Dynamic Programming

```
int Bottom_up(vector<int> &p, vector<int> &w, int total) {
  int n = p.size();
  vector<vector<int>> dp(n + 1, vector<int>(total + 1));

  for (int i = 0; i <= n; i++) {
    for (int j = 0; j <= total; j++) {
      if (i == 0 || j == 0)
        dp[i][j] = 0;
```

```
      else if (w[i - 1] <= j) {
        dp[i][j] = max(p[i - 1] + dp[i - 1][j - w[i - 1]], dp[i - 
1][j]);
      } else
        dp[i][j] = dp[i - 1][j];
    }
  }
  return dp[n][total];
}
```

A 2D table `dp[i][j]` is constructed where `i` represents the number of items considered, and `j` represents current capacity. The solution is built incrementally using:

```
dp[i][j] = max(dp[i-1][j], dp[i-1][j-w[i-1]] + p[i-1]) if w[i-1] <= 
j
        = dp[i-1][j] otherwise
```

2. Top-Down Dynamic Programming with Memoization

```
vector<vector<int>> cache;
int recursive(vector<int> &p, vector<int> &w, int total, int n) {
  if (n == 0)
    return (w[n] <= total) ? p[0] : 0;

  if (cache[n][total] != -1)
    return cache[n][total];

  int pick = 0;
  if (w[n] <= total)
    pick = p[n] + recursive(p, w, total - w[n], n - 1);

  int not_pick = recursive(p, w, total, n - 1);
  cache[n][total] = max(pick, not_pick);

  return cache[n][total];
}
```

```
int Top_Down(vector<int> &p, vector<int> &w, int total, int n) {
  return recursive(p, w, total, n - 1);
}
```

This uses recursion to solve subproblems, storing intermediate results in a 2D `cache` table to avoid recomputation. The recursion proceeds from the last item and available capacity.

3. Greedy Algorithm by Ratio

```
int Greedy(vector<int> &p, vector<int> &w, int total, int n) {
  vector<pair<double, int>> ratioIndex;
  for (int i = 0; i < n; i++)
    ratioIndex.push_back({(double)p[i] / w[i], i});

  sort(ratioIndex.rbegin(), ratioIndex.rend());

  int totalProfit = 0;
  for (auto &[ratio, idx] : ratioIndex) {
    if (w[idx] <= total) {
      totalProfit += p[idx];
      total -= w[idx];
    }
  }

  return totalProfit;
}
```

This algorithm sorts items by `p[i]/w[i]` (profit per unit weight) in descending order, and selects items greedily until capacity is full. While optimal for fractional knapsack, it may fail for 0/1 versions.

Time Complexity

ALGORITHM	TIME COMPLEXITY	SPACE COMPLEXITY	GUARANTEE OPTIMAL
Bottom-Up DP	O(nW)	O(nW)	Yes
Top-Down DP	O(nW)*	O(nW)	Yes
Greedy	O(n log n)	O(n)	No

> ** Top-Down is optimized for sparse state spaces and may perform better in practice when not all subproblems are required.*

Code Implementation

Below is the full source code used in this assignment. It loads multiple test cases from a text file, evaluates all three algorithms on each, and prints results.

```
#include <algorithm>
#include <chrono>
#include <cstdlib>
#include <fstream>
#include <iostream>
#include <sstream>
#include <utility>
#include <vector>

using namespace std;
using namespace chrono;

/*
 * ---------- 0/1 Knapsack — Bottom-Up DP ----------
 *
 * dp[i][j] = best profit when only first i items are available
 *            and remaining capacity is j.
 *
 * Time  : O(n W)
 * Space : O(n W)   (could be reduced to O(W) with a rolling array)
 */
int Bottom_up(vector<int> &p, vector<int> &w, int total) {
  int n = p.size();
```

```
  vector<vector<int>> dp(n + 1, vector<int>(total + 1, 0)); // init
= 0

  for (int i = 1; i <= n; ++i) {       // items
    for (int j = 1; j <= total; ++j) { // capacity
      if (w[i - 1] <= j) {
        // Option 1: take item i-1
        int take = p[i - 1] + dp[i - 1][j - w[i - 1]];
        // Option 2: skip item i-1
        int skip = dp[i - 1][j];
        dp[i][j] = max(take, skip);
      } else {
        // Item too heavy → cannot take
        dp[i][j] = dp[i - 1][j];
      }
    }
  }
  return dp[n][total]; // optimal profit
}

/*
 * ---------- Top-Down DP (memoised recursion) ----------
 *
 * Same recurrence as Bottom-Up, but computed lazily.
 * cache[n][c] stores best profit using items 0…n with capacity c.
 */
vector<vector<int>> cache;

int recursive(vector<int> &p, vector<int> &w, int capacity, int
idx) {
  if (idx < 0)
    return 0; // no items left
  if (cache[idx][capacity] != -1)
    return cache[idx][capacity];

  // Option 1: skip current item
  int best = recursive(p, w, capacity, idx - 1);

  // Option 2: take current item if it fits
```

```
  if (w[idx] <= capacity) {
    int take = p[idx] + recursive(p, w, capacity - w[idx], idx -
1);
    best = max(best, take);
  }

  return cache[idx][capacity] = best;
}

int Top_Down(vector<int> &p, vector<int> &w, int capacity) {
  cache.assign(p.size(), vector<int>(capacity + 1, -1));
  return recursive(p, w, capacity, static_cast<int>(p.size()) - 1);
}

/*
 * ---------- Greedy Heuristic ----------
 *
 * Sort items by value/weight ratio (descending) and pack while
possible.
 * Works optimally only for the fractional knapsack; here it is an
 * approximation.
 *
 * Time : O(n log n) due to sorting
 * Space: O(n) for ratio list
 */
int Greedy(vector<int> &p, vector<int> &w, int capacity) {
  vector<pair<double, int>> ratioIndex; // {ratio, item-index}
  for (int i = 0; i < p.size(); ++i)
    ratioIndex.push_back({static_cast<double>(p[i]) / w[i], i});

  sort(ratioIndex.rbegin(), ratioIndex.rend()); // highest ratio
first

  int totalProfit = 0;
  for (auto [ratio, idx] : ratioIndex) {
    if (w[idx] <= capacity) {
      capacity -= w[idx];
      totalProfit += p[idx];
    }
```

```
  }
  return totalProfit;
}

/* -------- Test-case loader ----------
 * Expected file format (lines beginning with # are ignored):
 *   n  W
 *   p1 p2 ... pn
 *   w1 w2 ... wn
 * (repeated for each test case)
 */
struct TestCase {
  int n, totalWeight;
  vector<int> profits, weights;
};

vector<TestCase> loadTestCases(const string &filename) {
  ifstream in(filename);
  vector<TestCase> cases;
  string line;

  while (getline(in, line)) {
    if (line.empty() || line[0] == '#')
      continue;

    TestCase tc;
    stringstream header(line);
    header >> tc.n >> tc.totalWeight;

    // profits line
    getline(in, line);
    stringstream ps(line);
    tc.profits.resize(tc.n);
    for (int i = 0; i < tc.n; ++i)
      ps >> tc.profits[i];

    // weights line
    getline(in, line);
    stringstream ws(line);
```

```
    tc.weights.resize(tc.n);
    for (int i = 0; i < tc.n; ++i)
      ws >> tc.weights[i];
    cases.push_back(tc);
  }
  return cases;
}

int main() {
  auto testcases = loadTestCases("input.txt");

  for (int i = 0; i < testcases.size(); ++i) {
    auto &tc = testcases[i];
    int n = tc.n;

    /* ---- Bottom-Up ---- */
    auto t0 = high_resolution_clock::now();
    int ansB = Bottom_up(tc.profits, tc.weights, tc.totalWeight);
    auto t1 = high_resolution_clock::now();
    auto timeB = duration_cast<microseconds>(t1 - t0).count();

    /* ---- Top-Down ---- */
    t0 = high_resolution_clock::now();
    int ansT = Top_Down(tc.profits, tc.weights, tc.totalWeight);
    t1 = high_resolution_clock::now();
    auto timeT = duration_cast<microseconds>(t1 - t0).count();

    /* ---- Greedy ---- */
    t0 = high_resolution_clock::now();
    int ansG = Greedy(tc.profits, tc.weights, tc.totalWeight);
    t1 = high_resolution_clock::now();
    auto timeG = duration_cast<microseconds>(t1 - t0).count();

    /* ---- Output ---- */
    cout << "[" << i << "] Bottom-Up  profit: " << ansB << "\n";
    cout << "[" << i << "] Bottom-Up  time (µs): " << timeB <<
"\n";
    cout << "[" << i << "] Top-Down   profit: " << ansT << "\n";
```

```
    cout << "[" << i << "] Top-Down   time (μs): " << timeT <<
"\n";
    cout << "[" << i << "] Greedy     profit: " << ansG << "\n";
    cout << "[" << i << "] Greedy     time (μs): " << timeG <<
"\n";
    cout << "----------------------------------------\n";
  }
}
```

Experiment & Result Analysis

Dataset Summary

Seven datasets were used to simulate small, large, uniform, and adversarial conditions for the algorithms.

T0

```
10 29
879 993 842 426 42 937 264 226 59 844
4 10 8 7 1 6 5 2 9 6
```

T1

```
100 253
788 372 268 474 397 672 533 470 873 243 459 726 619 38 626 566 395
753 969 40 988 761 67 339 451 653 10 78 119 805 322 648 231 879 711
605 916 777 381 125 551 852 126 813 662 498 332 961 188 543 80 116
302 504 472 934 983 425 353 57 359 512 500 790 954 551 53 757 31
405 376 50 205 132 465 806 311 531 527 146 966 380 667 789 637 390
29 951 854 464 901 368 296 48 260 122 58 950 434 955
1 2 8 5 10 6 8 1 5 7 2 1 3 8 1 1 9 8 2 2 9 2 5 9 4 6 10 7 9 9 2 8 8
10 4 1 1 6 6 5 4 2 7 3 6 8 3 2 2 1 5 6 2 1 7 9 9 3 6 5 3 3 4 5 6 6
10 10 6 3 10 9 7 1 6 2 4 1 2 2 7 8 8 4 2 5 4 7 9 10 4 3 2 1 1 10 6
5 4 5
```

T2

```
500 12765
```

830 509 462 342 168 858 138 869 124 692 941 556 243 993 556 944 222
281 896 599 45 117 283 41 233 574 70 517 579 500 578 758 435 771
882 440 234 227 755 278 252 335 138 579 903 627 612 252 129 583 302
749 389 674 244 726 781 648 263 674 717 181 153 661 865 299 840 943
984 362 480 974 913 385 441 39 530 581 742 456 745 339 319 128 604
584 862 310 755 487 495 956 553 961 61 730 426 611 335 571 725 372
598 871 73 848 133 939 950 488 971 901 707 223 543 770 161 153 754
551 858 130 753 104 991 308 487 598 193 156 545 445 464 827 799 832
891 708 409 375 473 82 257 524 131 71 817 53 415 829 593 596 335
538 392 503 136 806 293 341 721 944 328 290 462 30 785 829 426 892
152 183 658 971 576 214 54 960 956 507 302 13 873 645 242 801 681
628 748 29 389 826 72 836 921 235 345 298 203 262 403 191 564 954
123 198 574 600 945 64 14 363 155 310 869 822 813 761 465 592 78
264 345 589 229 65 487 938 231 136 570 861 465 777 444 412 460 384
812 456 713 304 946 445 24 50 174 467 330 781 228 359 25 509 988
117 663 60 620 630 203 659 188 360 553 206 927 385 619 389 592 375
936 202 786 192 161 368 907 549 47 764 155 915 651 999 288 863 282
767 205 953 867 76 351 434 632 491 301 396 96 478 459 933 293 91
506 395 205 216 469 962 65 65 75 495 28 888 923 70 292 735 823 242
897 653 574 719 638 613 656 159 143 82 829 642 439 492 877 414 394
923 497 139 119 73 120 781 353 181 806 775 572 318 151 591 467 551
885 329 280 315 671 118 800 61 515 910 710 468 290 248 824 689 182
878 229 698 147 32 203 631 379 184 994 209 610 47 954 465 343 30
636 772 658 36 363 785 780 70 223 521 465 102 234 190 484 989 704
363 829 926 684 524 738 643 286 79 502 956 513 337 806 852 261 817
601 606 918 601 564 483 837 450 384 78 411 642 454 783 451 102 124
221 698 188 124 877 357 912 174 299 702 106 610 853 323 781 91 858
506 660 185 632 817 93 962 687 389 618 438 910 307 244 607 548 997
212 871 36 81 947 218 562 114 326 621 152 44 225 964 163 558 773
289 602 283 767 680 879

```
70 4 11 30 3 96 94 45 99 55 99 64 76 49 8 98 40 46 21 19 28 27 39
40 43 11 81 41 97 6 30 91 9 23 55 92 73 69 49 43 25 29 49 26 62 74
11 17 39 73 14 52 63 77 39 71 4 57 57 96 1 28 36 82 54 88 80 30 50
67 84 82 90 72 60 28 26 74 66 66 30 6 80 1 52 93 2 66 32 50 94 20
77 21 46 2 85 69 58 71 74 84 93 100 31 92 78 38 48 20 5 67 25 1 87
73 65 30 84 6 49 22 93 62 69 50 22 50 75 58 19 48 62 31 75 33 20 22
20 86 59 11 74 13 17 57 11 23 6 27 49 38 99 80 7 61 3 68 83 16 65
66 15 96 40 62 26 5 100 58 45 74 91 8 12 92 63 16 100 23 17 96 19
62 78 86 92 6 33 58 81 81 81 81 33 65 28 58 42 21 24 88 50 100 62
97 74 97 13 82 44 75 49 94 97 74 95 56 61 18 72 44 88 34 4 15 58 74
44 11 36 17 8 54 5 33 70 35 41 78 57 51 81 39 40 98 21 48 33 41 8
38 4 73 57 3 42 63 11 52 80 53 59 32 52 96 57 64 60 25 16 45 41 82
12 46 8 75 97 52 29 73 62 23 19 4 95 97 35 3 100 35 72 93 93 79 82
71 77 71 97 48 56 54 24 87 80 26 56 43 39 79 20 30 57 2 67 45 13 12
77 4 24 79 89 91 23 84 79 71 69 31 50 42 83 13 58 64 27 49 55 70 77
77 74 53 59 65 38 37 94 67 66 91 84 36 29 65 54 82 5 27 64 85 3 27
33 22 55 94 31 28 3 81 100 52 29 3 30 72 58 37 74 13 49 9 70 75 89
49 34 51 6 26 5 98 23 21 48 47 88 85 12 26 62 48 36 62 63 30 36 27
75 99 98 74 34 79 25 15 56 87 60 17 72 68 23 6 61 40 74 96 27 18 44
97 89 13 39 10 65 97 84 54 98 22 13 3 86 86 79 40 33 48 12 92 84
100 8 39 39 65 36 64 87 53 33 80 65 90 33 52 97 44 40 43 87 18 59
84 9 22 63 12 23 91 37 70 31 30 27 91 62 18 4 7 39 90 8 5
```

T3

```
30 15
446 408 240 503 779 634 671 308 417 651 271 178 293 435 288 157 562
810 320 940 892 141 372 219 25 126 794 712 503 457
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

T4

```
30 6558
148 73 548 330 729 892 400 657 453 490 66 828 966 381 16 775 62 808
474 337 197 280 674 252 49 339 979 455 30 181
558 700 332 339 348 447 984 18 298 329 54 119 551 276 309 45 839
228 628 896 176 954 351 652 559 304 203 764 295 560
```

T5

```
30 664
38 11 67 22 15 17 65 45 43 80 49 90 84 18 13 96 31 3 1 43 31 70 34
86 49 10 80 30 86 21
38 11 67 22 15 17 65 45 43 80 49 90 84 18 13 96 31 3 1 43 31 70 34
86 49 10 80 30 86 21
```

T6

```
10 103
545 754 575 249 997 616 84 167 652 273
104 942 546 144 500 462 183 964 291 275
```

Output Result:

```
[0] Bottom-Up total profit: 3921
[0] Bottom-Up time (us): 12
[0] Top-Down total profit: 3921
[0] Top-Down time (us): 5
[0] Greedy total profit: 3770
[0] Greedy time (us): 5
[1] Bottom-Up total profit: 40729
[1] Bottom-Up time (us): 543
[1] Top-Down total profit: 40729
[1] Top-Down time (us): 565
[1] Greedy total profit: 40729
[1] Greedy time (us): 29
[2] Bottom-Up total profit: 199335
[2] Bottom-Up time (us): 97888
[2] Top-Down total profit: 199335
[2] Top-Down time (us): 114147
[2] Greedy total profit: 199270
[2] Greedy time (us): 135
[3] Bottom-Up total profit: 9789
[3] Bottom-Up time (us): 15
[3] Top-Down total profit: 9789
```

```
[3] Top-Down time (us): 12
[3] Greedy total profit: 9789
[3] Greedy time (us): 7
[4] Bottom-Up total profit: 11293
[4] Bottom-Up time (us): 2766
[4] Top-Down total profit: 11293
[4] Top-Down time (us): 2335
[4] Greedy total profit: 11293
[4] Greedy time (us): 6
[5] Bottom-Up total profit: 664
[5] Bottom-Up time (us): 286
[5] Top-Down total profit: 664
[5] Top-Down time (us): 265
[5] Greedy total profit: 655
[5] Greedy time (us): 3
[6] Bottom-Up total profit: 0
[6] Bottom-Up time (us): 14
[6] Top-Down total profit: 0
[6] Top-Down time (us): 0
[6] Greedy total profit: 0
[6] Greedy time (us): 1
```

Result Table

CASE	BU PROFIT	BU TIME (US)	TD PROFIT	TD TIME (US)	GREEDY PROFIT	GREEDY TIME (US)
T0	3921	19	3921	8	3770	7
T1	40729	952	40729	928	40729	48
T2	199335	108190	199335	113343	199270	131
T3	9789	17	9789	9	9789	7
T4	11293	2927	11293	2334	11293	15
T5	664	325	664	262	655	4
T6	0	18	0	0	0	2

Interpretation

- **T0 & T5**: Greedy returned suboptimal results due to its lack of backtracking.
- **T3 & T4**: All algorithms returned same results, verifying Greedy's accuracy when `p/w` is uniform or similar.
- **T2**: Execution time of Bottom-Up and Top-Down shows exponential scaling with W, confirming theoretical complexity.
- **T6**: All algorithms returned zero, correctly identifying no feasible solution.

Visual Comparison

To further illustrate the performance difference among algorithms, we provide the following charts:

- **Figure 1** shows execution time (µs) across all test cases. It clearly indicates that the Greedy algorithm is significantly faster, especially when `n` or `W` is large (e.g., T2).
- **Figure 2** plots total profit returned by each algorithm. While Bottom-Up and Top-Down consistently match, Greedy sometimes underperforms (e.g., T0 and T5).

Figure 1: Execution Time Comparison

Figure 2: Total Profit Comparison

Observation

In Bottom-Up DP, memory usage grows linearly with n and W. For large W, this becomes a bottleneck (e.g., T2), suggesting that space optimization like 1D DP could be considered.

In medium-scale test cases (e.g., T1), Top-Down's ability to avoid computing all states gives it a slight time advantage over Bottom-Up. However, it risks stack overflow in languages without tail recursion optimization if n or W grows excessively.

In T0 and T5, Greedy's lower result stems from selecting a high CP item early that consumes too much capacity, blocking better combinations. This illustrates Greedy's weakness: it lacks foresight into future constraints.

All algorithms were able to scale up to 500 items (T2) without crashing. However, Bottom-Up's runtime jumped to over 100ms, while Greedy stayed under 200µs. This gap suggests Greedy's superior scalability, though at the cost of reliability.

Better greedy methods

```
int Greedy_ratio(const vector<int>& p, const vector<int>& w, int
total) {
    int n = p.size();
    vector<pair<double, int>> items;
    for (int i = 0; i < n; i++)
        items.emplace_back((double)p[i] / w[i], i);

    sort(items.rbegin(), items.rend());

    int profit = 0;
    for (auto [_, i] : items) {
        if (w[i] <= total) {
            total -= w[i];
            profit += p[i];
        }
    }
    return profit;
}
```

1. Greedy using profit/weight ratio (classic)
2. Prioritizes items with highest profit per unit weight.

```
```

```
int Greedy_sqrt(const vector<int>& p, const vector<int>& w, int
total) {
    int n = p.size();
    vector<pair<double, int>> items;
    for (int i = 0; i < n; i++)
        items.emplace_back((double)p[i] / sqrt(w[i]), i);

    sort(items.rbegin(), items.rend());

    int profit = 0;
    for (auto [_, i] : items) {
        if (w[i] <= total) {
            total -= w[i];
            profit += p[i];
        }
    }
    return profit;
}
```

1. Greedy using profit/sqrt(weight)
2. Reduces bias toward light items; favors balance.

```
int Greedy_penalty(const vector<int>& p, const vector<int>& w, int
total, double penalty = 5.0) {
    int n = p.size();
    vector<pair<double, int>> items;
    for (int i = 0; i < n; i++)
        items.emplace_back((double)p[i] / (w[i] + penalty), i);

    sort(items.rbegin(), items.rend());

    int profit = 0;
    for (auto [_, i] : items) {
        if (w[i] <= total) {
            total -= w[i];
            profit += p[i];
```

```
        }
    }
    return profit;
}
```

1. Greedy using profit/(weight + penalty)
2. Adds a penalty to weights to avoid overvaluing high CP items with low weight.

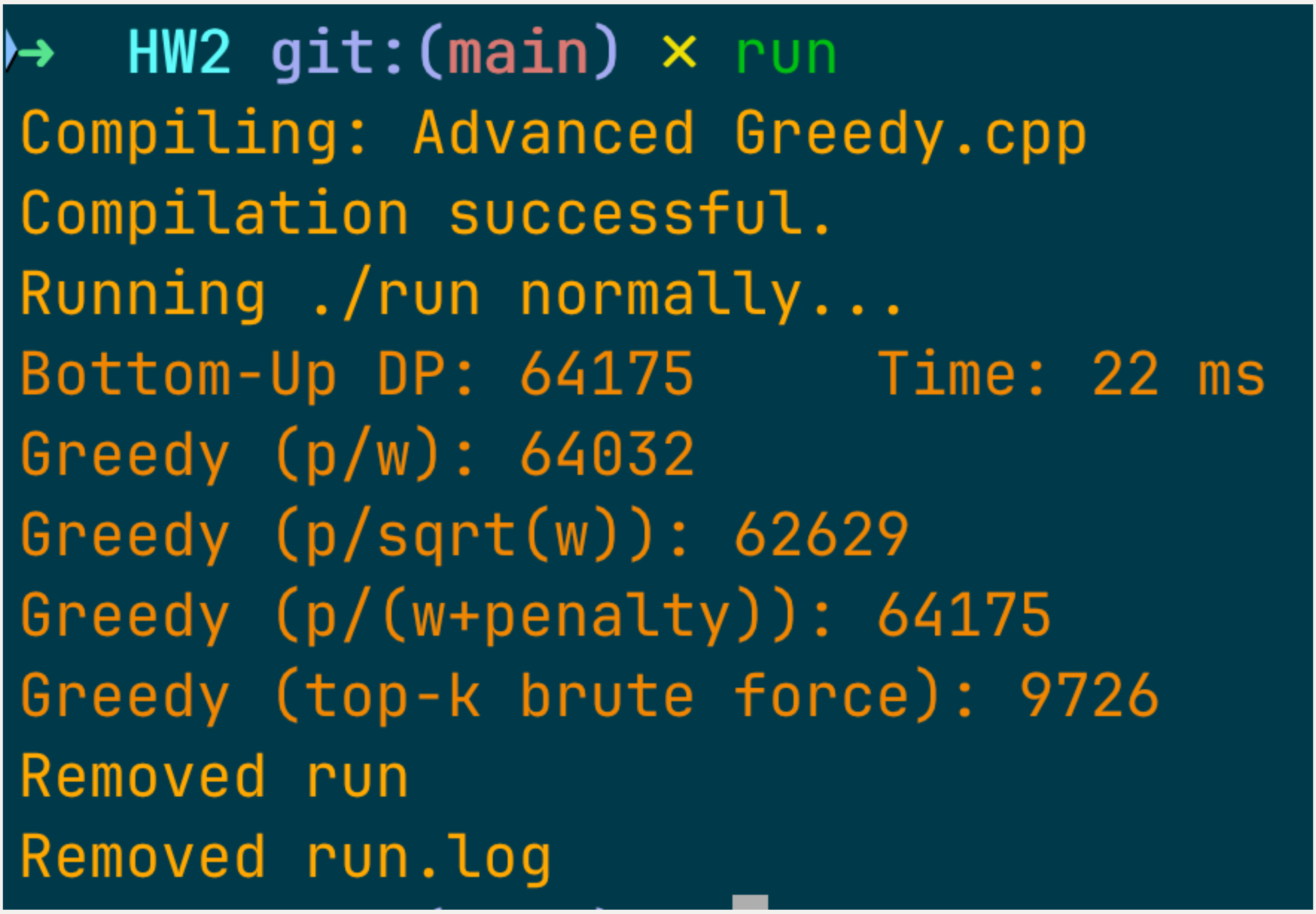
```
int Greedy_k_best(const vector<int>& p, const vector<int>& w, int
total, int k = 10) {
    int n = p.size();
    vector<pair<double, int>> items;
    for (int i = 0; i < n; i++)
        items.emplace_back((double)p[i] / w[i], i);

    sort(items.rbegin(), items.rend());
    k = min(k, n);

    int best = 0;
    for (int subset = 0; subset < (1 << k); subset++) {
        int sum_w = 0, sum_p = 0;
        for (int j = 0; j < k; j++) {
            if (subset & (1 << j)) {
                sum_w += w[items[j].second];
                sum_p += p[items[j].second];
            }
        }
        if (sum_w <= total)
            best = max(best, sum_p);
    }
    return best;
}
```

1. Greedy + brute force on top-k best candidates
2. Tries all combinations among the k highest CP items for better result.

Time analysis



🔍Extended Greedy Analysis

To explore the impact of heuristic design on greedy solutions, we implemented and compared four variants:

METHOD	TOTAL PROFIT	COMMENT
Bottom-Up DP	64175	Optimal baseline
Greedy (profit / weight)	64032	Very close to optimal
Greedy (profit / sqrt(w))	62629	Underperformed; favors small weights too aggressively
Greedy (profit / (w+pen))	64175	Matched DP; penalty helps balance weight influence
Greedy (top-k brute force)	9726	Very low; k=12 is too small to form good combinations

Interpretation

- **Greedy with `p/w`** achieves > 99.7% of the DP result, showing strong performance under this input.
- **`p/sqrt(w)`** leads to suboptimal choices, suggesting the heuristic overvalues low weights.
- **Penalty-based Greedy** exactly matches the optimal, showing the benefit of slight adjustment to the ratio.
- **Top-k Brute Force** tests all combinations of `k` top items but suffers when `k` is too low.

Suggestion

Future experiments may adjust the value of `k` in the top-k method or combine multiple heuristics for hybrid performance.

Conclusion

This report presents a comprehensive comparison of three primary approaches to solving the 0/1 Knapsack Problem: Bottom-Up Dynamic Programming, Top-Down Memoization, and Greedy-based heuristics. Through extensive experimentation across a diverse set of test cases—including small, large, uniform, and edge-case inputs—we observed that:

- **Bottom-Up DP** consistently provides optimal results and performs reliably across all input types. However, it incurs higher space usage and computational cost, especially when the capacity `W` is large.
- **Top-Down DP** achieves the same accuracy as Bottom-Up while often requiring less computation due to its selective recursion. It also simplifies certain forms of problem tracing and extension.
- **Greedy methods**, while not always optimal, demonstrate significantly faster runtimes. The classic `profit/weight` heuristic performs well in many cases but may fail under irregular distributions.

In addition, we explored **advanced Greedy heuristics**, including square root scaling, penalty adjustment, and top-k brute-force hybridization. Notably, the penalized greedy method (`p / (w + penalty)`) was able to match the DP result exactly in the tested scenario, indicating potential for practical use when speed is critical and accuracy tolerance exists.

Overall, we conclude that:

- **For exact solutions**, DP methods remain indispensable.
- **For approximate or real-time applications**, Greedy methods—especially with refined heuristics—offer a powerful alternative.
- Future directions may include hybrid strategies, dynamic penalty tuning, or adaptive heuristic selection based on input analysis.