lab_report_knapsack

```
library(AdvRknapsack)
```

Introduction

This vignette provides a comprehensive overview of the `AdvRknapsack` package, which offers solutions to the classic knapsack problem using three different algorithms: Brute-Force, Greedy, and Dynamic Programming. Each approach has its own strengths and trade-offs, depending on the size and nature of the input data.

Brute-Force Approach

The brute-force approach attempts every possible combination of items to find the optimal solution. While this guarantees the best solution, it is computationally expensive, especially for larger datasets.

Usage

```
# Prepare Data
set.seed(42)
knapsack_objects <- data.frame(w = sample(1:4000, size = 1000000, replace = TRUE), v = runif(n = 100000

# Run Brute-Force Knapsack
result <- brute_force_knapsack(x = knapsack_objects[1:8, ], W = 3500)
print(result)
#> $value
#> [1] 21754.76
#>
#> $elements
#> [1] 3 4 7
result <- brute_force_knapsack(x = knapsack_objects[1:12, ], W = 3500)
print(result)
#> $value
#> [1] 22689.93
#>
#> $elements
#> [1] 3 7 9
result <- brute_force_knapsack(x = knapsack_objects[1:8, ], W = 2000)
print(result)
#> $value
#> [1] 15327.2
#>
#> $elements
#> [1] 3 7
```

```r
result <- brute_force_knapsack(x = knapsack_objects[1:12, ], W = 2000)
print(result)
#> $value
#> [1] 15327.2
#>
#> $elements
#> [1] 3 7
```

```r
# Measure execution time for brute-force knapsack
time_taken <- system.time({
  result <- brute_force_knapsack(x = knapsack_objects[1:16, ], W = 2000)
})

# Display the results
print(result)
#> $value
#> [1] 23829.08
#>
#> $elements
#> [1]  7 13 15
print(time_taken)
#>    user  system elapsed
#>    0.38    0.00    0.38
```

How much time does it takes to run the algorithm for n = 16 objects?
Time taken: 0.38 seconds

Dynamic Programming Approach

Dynamic programming breaks the problem into smaller sub-problems and builds up the solution iteratively. It guarantees an optimal solution with improved efficiency over brute-force.

Usage

```r
# Run Dynamic Knapsack
result <- knapsack_dynamic(x = knapsack_objects[1:500, ], W = 2000)
print(result)
#> $value
#> [1] 124958.5
#>
#> $elements
#>  [1]  15  77  89 110 114 152 203 242 244 256 283 295 315 322 341 455 459
```

```r
# Measure execution time for Greedy Approach
time_taken <- system.time({
  result <- knapsack_dynamic(x = knapsack_objects[1:500, ], W = 2000)
})

print(result)
#> $value
```

```
#> [1] 124958.5
#>
#> $elements
#>  [1]  15  77  89 110 114 152 203 242 244 256 283 295 315 322 341 455 459
print(time_taken)
#>    user  system elapsed
#>    1.86    0.03    1.95
```

How much time does it takes to run the algorithm for n = 500 objects?
Time taken: 1.95 seconds

Greedy Approach

The greedy algorithm sorts items based on their value-to-weight ratio and picks the best option. While fast, it does not always yield the optimal solution.

Usage

```
# Run Greedy Knapsack
result <- greedy_knapsack(x = knapsack_objects[1:800, ], W = 3500)
print(result)
#> $value
#> [1] 222828.6
#>
#> $elements
#>  [1] 256 530 701  89 152 559 759 244 455  77 110 704 114 691 553 708 511 341 203
#> [20] 283 626 763  15 322 315 620 800 295 509 672 705

result <- greedy_knapsack(x = knapsack_objects[1:1200, ], W = 2000)
print(result)
#> $value
#> [1] 230018.5
#>
#> $elements
#>  [1]  256  530 1186  701 1070   89  973  951  962  152  559  759 1157  244 1106
#> [16]  897  959 1019  949 1080  455   77  110  704  114  845  691  553  708 1153
#> [31] 1068  511   75  826
```

```
# Measure execution time for Greedy Approach
time_taken <- system.time({
  result <- greedy_knapsack(x = knapsack_objects[1:1000000, ], W = 2000)
})

print(result)
#> $value
#> [1] 5153451
#>
#> $elements
#>   [1] 454690 291045 483218 908053 268238 922512 669156 784767 561647 357325
#>  [11] 148391 517539 348422 965553 453703 708876 186243 174352 423296 514669
#>  [21] 401725  36908 305455 455764  68497 227414 394109 465881 114001 868449
```

```
#>  [31] 945304 259259 616425 505318 533376 977611 609213 165736 778510 953873
#>  [41] 341275 185835 261156 984882 441787 791909 632509 246747 860950 259068
#>  [51] 724970 677100 337460 125049 552578 823841 789458 568121 241642 521138
#>  [61] 656939 674393 880792 871807 754781 886209 193214 563635 147963 928408
#>  [71]  47441  29469 686875 809935 878878 284784 476100 706615 809281 863091
#>  [81] 904139 112754 818533 314324 309349 470696 463093 543148 189928  50535
#>  [91] 584027 394170 113726 585895 100550 339121 811180 276820 713029 516242
#> [101] 695030 114916 673088 138151 558904 953020 584505 357095 919997 660272
#> [111]  20281 382135 140530 141638   6805 960752 810973 933856 798864 149282
#> [121] 193200 397922 221487 540528 698709 613397 331387 934501 111412 289712
#> [131] 997957 409019 264492 968630 424841 467321 414349 228853 137753 182345
#> [141] 111778 835229 573168 861476 667152 504629 227987 281464 866853  57109
#> [151] 950075 975273 582302 943935 962922  71017 201842 788582 943191 613727
#> [161] 825151 303068 526964   2640 156423 699861 593349 732876 775975  87830
#> [171] 624218 815482 877512 745909  93995 328347 969235 465414 156329 944744
#> [181] 521668 427402 375189  62079 191165 813849 382818 297284 427828 712688
#> [191] 830752 898707 881634 946699 826565   4640 679637 477438 980842 553563
#> [201] 776699 743431  39742 812184 979388 394013 993484 608866 166553 199594
#> [211]  21947 508299 467625 243758 668885  24006  38855 800964 487352 332341
#> [221] 965557 853891 125504 331047  63719 907792 803493 903185 933943 620885
#> [231] 581950 608233    256 221930 837801 308082 694402 675454 289552 159322
#> [241] 473238 925918  44690 757260 347386 955195 642976 448897 966530 172740
#> [251] 614344 148125 658251 604410 686452 283666 573257 934951 820681 410905
#> [261] 166125 261259 323866 747080 208201 750589 991266 643498 259766 899375
#> [271] 988923 119426  15205 812110 996698 796320 310537 500111 572251 739752
#> [281] 132876 437288 607225 257517 518661 210565 710140 865952 656064 653173
#> [291] 163020 226088 994568 531929 138925 625037 167226 856214 249748 453048
#> [301] 553505 125095  74953 724297 698176 858645 364799 627315 774133 748301
#> [311] 470519 967400 467277 853871 853218  29190 944497 851139 813774 826863
#> [321]  31592 412173 719467 235482 331577 116971 927406 365201 916468 193615
#> [331] 667505 487093 873908  66452  73768 528633 605151 107262 101942 517413
#> [341]  41116 559649 703795 357997 701665 487183  47478  77633 103300  77260
#> [351]  48130 812612 524237 397331 146068 614406 448190 619421 666841 198216
#> [361] 702974 136850 287635 815927 872488  45345 243973 387722 979601 693659
#> [371]  11468 824278 991322 634390 526328 192645 620138 740960 189897 619118
#> [381] 497260 497198 208828 677366 929590 959278 999865 356427 690234 784650
#> [391] 250323 403177 298194 598673 520339  68860  19680 253393 165941 414222
#> [401] 715042 227744 324147 530575 837465 636411 614503 247266 566766  76467
#> [411] 510661 623109 244587 858144 902472 127464 391821 303692 836141 156095
#> [421] 541202 850068 982432 295276 911305 300938  25372 818703 899408 570489
#> [431]   2828 623379 943951 690010 349667  20091 501742 628959 817970 880388
#> [441] 240667 609210 445039 798531  83526  55816 242081 912696  66481 302779
#> [451] 375476 491053 957984  76340 774835 374620 954760 628279  43054  74681
#> [461]   6966 233653 688148 408072 789954 745509 562112 327516 992802 449458
#> [471] 896788 558155 350017 684631 278070 523272 589204 889681 897293 929084
#> [481] 502392 238324 148222 558120  53053  11575 450582 969158   1751 714491
#> [491] 374076 815804  21088 105889 584427 517166 747240 937284 191970 931003
#> [501] 533897 497746 807907 937417 104519 973167 619200 444617 895146 331631
#> [511] 590476 727461 623094 576651 293205 964429 247806 827374 654478   3517
#> [521] 941165 790914 979390 626355 311115 444757 626466 179689  34442 336916
#> [531] 564905 508265 703718 546841 958266 342551 240569 911247 307432 739052
#> [541] 410769 448742 183656 475733 182785 411976 905073 793367 687263 440572
#> [551]  84421 870296 945933  26479 803067 381739  79780 424015 504535 617875
```

```
#> [561] 818272 824443 371729 768480 619120 505832 220561 239565 910268 284313
#> [571]  66099 780174 432983 448214 890998 606112 208014 464570 731700 570505
#> [581]  32870 934343 746995 414237 749574 951627 610979 340378 920647 692033
#> [591] 879241 802546 852226 621728 856169 779347 442057 740103 292467 942657
#> [601] 904406 423623 598328 318306 578319 395246  14063  74984 113522 533375
#> [611]  46276 384223 211481 891456 466548 321557 323867 965466 582576 496425
#> [621] 567553 824911 294464 850922 504431 643118 413496 606066 417678  70433
#> [631] 560546 788217 169114 833562 845084 426430 973474 458860 267657 704898
#> [641]  24435 313429 658403 381833 332706  64451 419980 916274 715933 848698
#> [651] 900984 573531 775033 812356   8597 269201 187134 141726 947710 693243
#> [661] 489291  12238 306447 991815  49221 817336 537368 426751 419932 269542
#> [671] 201766 684689 346090 101052 110036 113943 768494 328165 318488  75266
#> [681]  46852 983789 984564 446193 312510 101669 228576 677536  12530 132714
#> [691] 224475 801746 628408 237645 517558 119336  71074 813246 940057 195027
#> [701] 894203 146252 138712 799007 813412 642308 406533 965372 458892  64432
#> [711] 892719 910400 960722 911762 498952 395950 718200 529933 987438 209758
#> [721] 778468 168583 852903 504623 994131 843857 967914 466654 294230   9050
#> [731]  13575 548865 349127 998646 829240  27717 886537  37691 106971 498369
#> [741] 760868 161122 939149 567431 267576 639164
print(time_taken)
#>    user  system elapsed
#>    8.86    2.86   11.97
```

How much time does it takes to run the algorithm for n = 1000000 objects?
Time taken: 11.97 seconds

Comparing the Methods

The choice of algorithm depends on the problem size and requirements:

- **Brute-Force**: Suitable for small datasets where exact results are required.
- **Dynamic Programming**: Ideal for medium-sized datasets where an optimal solution is desired with reasonable computational effort.
- **Greedy**: Good for larger datasets when a near-optimal solution suffices.

Conclusion

The `AdvRknapsack` package provides flexible options for solving the knapsack problem using different algorithms, catering to various performance needs and dataset sizes.