Michael Setteducati

Dr. Simari

CS462 - Algorithm Analysis

18 April 2018

0-1 Knapsack Problem - Algorithm Analysis

1 Introduction

The 0-1 Knapsack Problem is a hypothetical situation where a thief has a limited amount of weight that he can carry in his Knapsack and he wants to maximize the value of the items that he takes. It is the 0-1 rendition of the problem because the thief must take the entire item or leave it (as opposed to taking a portion of the item).

I created this problem in Python by generating a random array of possible items with random weights and values. The maximum capacity of the thief's knapsack is 75% of the total weight of all of the random items that were created. Then, I implemented three algorithms to solve for a solution using a brute force approach, greedy approach, and dynamic programming approach.

I ran the brute force approach on 29 times for problem sizes of n = 1, 2, 3, …, 29 (29 took about 6 minutes, so I did not wait to run it for n = 30). The brute force solution uses a recursive call to calculate every possible combination of items taken and takes the items which maximize the value. The brute force algorithm yields an optimal solution.

I ran the greedy approach 30 times for problem sizes of n = 100k, 200k, 300k, …, 3M. The greedy algorithm sorts the items by value density and takes the items which are most value-dense until the knapsack has reached its weight capacity. This solution is not optimal, but is often close to it.

I ran the dynamic programming approach 30 times for problem sizes of n = 25, 50, 75, …, 750, and subsequently ran the greedy algorithm on the same problem instance to calculate the optimality of the greedy solution for that problem instance. The dynamic programming approach creates a 2-dimensional array where remaining weight is represented by the row and the item is represented by the column. It then calculates every index of the array in a bottom-up fashion, so that once it reaches the max index for available weight and item, it has calculated the optimal

solution. This dynamic algorithm yields an optimal solution (which is why the greedy solution is compared to it to compute optimality).

I created two runner files to create problem instances and run these algorithms. Both scripts can be invoked with the flag -h to get help descriptions for the required positional arguments, as well as the optional arguments. The first runner script, test-runner.py, takes the number of items in the problem instance as a positional argument, as well as the algorithm(s) that should be used to compute the solution as optional arguments. It also has an optional flag --log, which outputs more detailed information about the problem instance, algorithm and solution. This flag was used in debugging and ensuring the correctness of my algorithm.

The second runner script, runner.py, takes the minimum number of items, maximum number of items, and step size between runs as required positional arguments, as well as the algorithm(s) that should be used to compute the solution as optional arguments. This script also takes an optional --log flag to output detailed information about the problem instance, algorithm and solution. For example, to run each algorithm for n = 3, 4, 5, one would invoke:

```
Michaels-MacBook-Pro:knapsack michaelsetteducati$ python3 runner.py 3 5 1 --bruteforce --greedy --dynamic
Brute Force Algorithm Runs:
n       time (s)
3        4.00543212890625e-05
4        2.8848648071289062e-05
5        4.100799560546875e-05
Greedy Algorithm Runs:
n       time (s)
3        8.606910705566406e-05
4        4.887580871582031e-05
5        4.315376281738281e-05
Dynamic Programming Algorithm Runs:
n       time (s)              optimality
3        4.696846008300781e-05   1.0
4        8.392333984375e-05      1.0
5        9.965896606445312e-05   1.0
```

If you only wanted to run the brute force algorithm, only that argument would be included. If no algorithm arguments are included, the program does nothing.

2 Analysis

2.1 Random Problem Generation Algorithm

The random problem generation algorithm runs in *O(n)* time. This is the code for the random problem generation function:

```
4   def generate_random_knapsack_problem(n, log=False):
5       def generate_random_items():
6           to_return = []
7
8           for x in range(n):
9               random_weight = random.randint(1, n)
10              random_value = random.randint(1, n)
11              to_return.append(Item(random_weight, random_value))
12
13          return to_return
14
15      def compute_weight_capacity(items):
16          sum_weights = sum([item.weight for item in items])
17          return int(sum_weights * 0.75)
18
19      my_items = generate_random_items()
20      my_weight_capacity = compute_weight_capacity(my_items)
21
22      return KnapsackProblem(n, my_items, my_weight_capacity, log=log)
```

The random.randint() function on lines 9 and 10 runs in *O(1)* (constant) time, as does the append call on line 11. It is nested in a for loop that takes *O(n)* time to run (line 8). Computing the weight capacity also takes *O(n)* time to run because it must loop through the array of size *n* to compute the sum (line 16). The arithmetic on line 17 runs in constant time.

Since the longest operations (the loops on line 8 and line 16) both run in *O(n)* time, we have that the function runs in *O(2n)* time. After dropping the constant, the overall runtime of the algorithm is *O(n)*.

2.2 Brute Force Algorithm

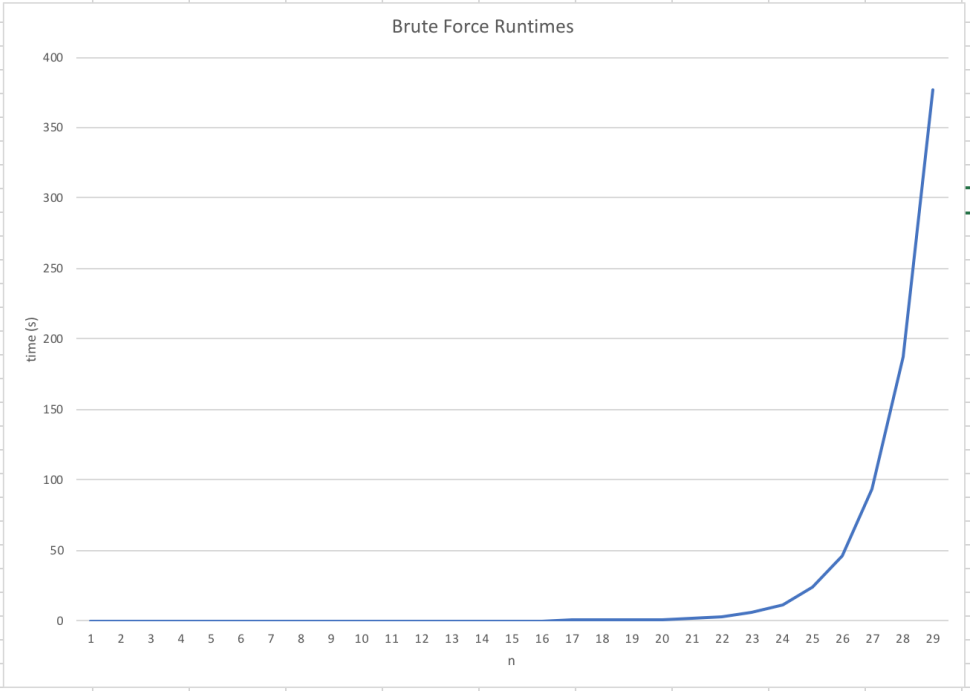
The brute force algorithm runs in $O(2^n)$ time. This is the code for the brute force algorithm:

```
51      def brute_force_solution(self):
52          """ ... """
56
57          def solve_by_brute_force(cur_i, available_weight):
58              """ ... """
64              # base case: we have looked at every element or cur_weight is at capacity
65              if cur_i == -1 or available_weight == 0:
66                  return 0
67
68              # if item doesn't fit, leave it
69              if self.items[cur_i].weight > available_weight:
70                  return solve_by_brute_force(cur_i - 1, available_weight)  # leave item
71
72              # solve in both cases: take item and leave item
73              take = self.items[cur_i].value + solve_by_brute_force(cur_i - 1, available_weight - self.items[cur_i].weight)
74              leave = solve_by_brute_force(cur_i - 1, available_weight)
75
76              # take the max of leaving or taking item
77              return max(take, leave)
78
79          optimal_value = solve_by_brute_force(self.n - 1, self.weight_capacity)
80
81          if self.log:
82              print("Brute Force Solution\n", "Optimal Value: ", optimal_value, "\n")
83
84          return optimal_value
```

The base cases on line 65 runs in constant time. There is either one or two recursive calls that are made. If the item doesn't fit, only one recursive call is made (line 70). If the item does fit, two recursive calls are made to find the value if the item is left and if the item is taken (lines 73-74). For this runtime analysis, we must consider the worst case and that two recursive calls are made. For each recursive call made, there are potentially two more recursive calls that are made, and this behavior continues until the base case. The worst-case scenario base case is cur_i == -1, because the available_weight will hit zero before this happens. This is because there is simply not enough available weight for every item. Therefore, since each recursive call might happen twice, and this might happen *n* times, the overall runtime of the brute force algorithm is $O(2^n)$.

This is a sample runtime table from one case of running the program, as well as a graph depicting this data:

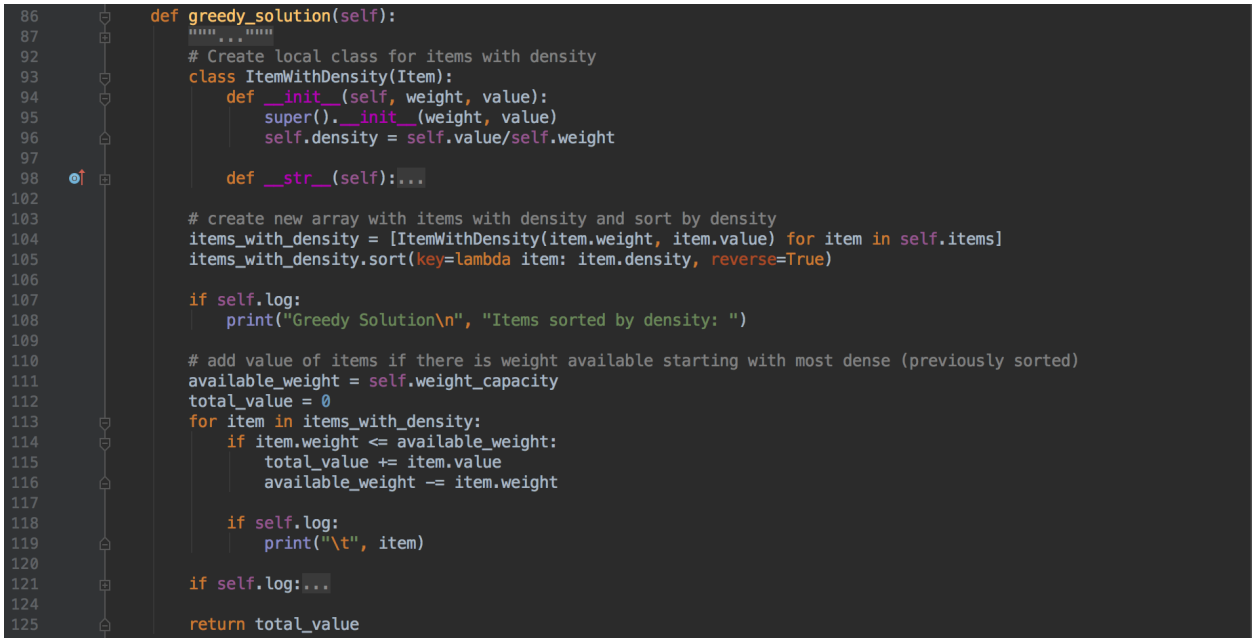
n	time (s)
1	0.000005483627319
2	0.000011444091797
3	0.000016212463379
4	0.000028371810913
5	0.000046014785767
6	0.000100612640381
7	0.000198125839233
8	0.000410318374634
9	0.000813961029053
10	0.001668214797974
11	0.003338813781738
12	0.006238698959351
13	0.012343406677246
14	0.022453308105469
15	0.038567543029785
16	0.063440799713135
17	0.119877099990844
18	0.240283012390136
19	0.445997714996337
20	0.730315208435058
21	1.466278314590450
22	2.936762094497680
23	5.814035654067990
24	11.688157320022500
25	23.413741588592500
26	46.409200191497800
27	93.565321922302200
28	187.123391628265000
29	376.975018739700000



This graph indeed appears to show that the runtime increases exponentially as a function of 2^n as n increases.

2.3 Greedy Algorithm

The greedy algorithm runs in *O(n log n)* time. This is the code for the greedy algorithm:



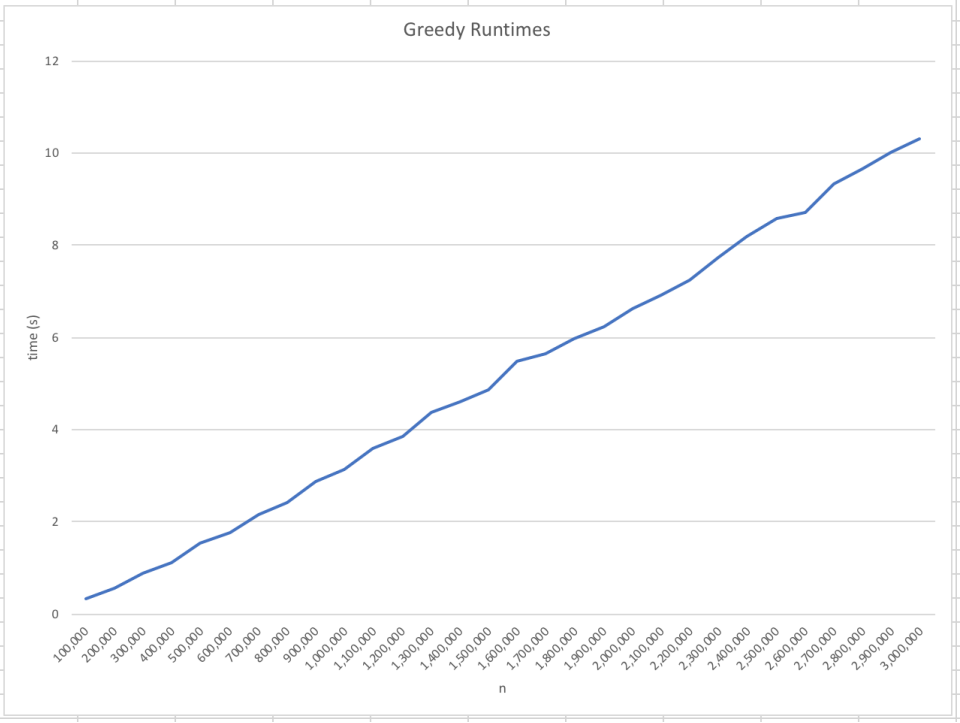
```
86      def greedy_solution(self):
87          """..."""
92          # Create local class for items with density
93          class ItemWithDensity(Item):
94              def __init__(self, weight, value):
95                  super().__init__(weight, value)
96                  self.density = self.value/self.weight
97
98              def __str__(self):...
102
103         # create new array with items with density and sort by density
104         items_with_density = [ItemWithDensity(item.weight, item.value) for item in self.items]
105         items_with_density.sort(key=lambda item: item.density, reverse=True)
106
107         if self.log:
108             print("Greedy Solution\n", "Items sorted by density: ")
109
110         # add value of items if there is weight available starting with most dense (previously sorted)
111         available_weight = self.weight_capacity
112         total_value = 0
113         for item in items_with_density:
114             if item.weight <= available_weight:
115                 total_value += item.value
116                 available_weight -= item.weight
117
118             if self.log:
119                 print("\t", item)
120
121         if self.log:...
124
125         return total_value
```

The creation of a new array for items with density runs in *O(n)* time because each item must be created and there are *n* items (line 104). The sorting of this new array in line 105 runs in *O(n log n)* time, as this is the typical runtime of a sorting algorithm (specifically, python uses an adaptive merge sort called “Timsort” that runs in *O(n log n)* time on average). Looping through the items on line 113 runs in *O(n)* time. Deciding whether or not to add the items, and doing this arithmetic on lines 114-116 runs in constant time.

Since the longest operation is the sort function on line 105, this dictates the overall runtime of the greedy algorithm. Therefore, the overall runtime of the greedy algorithm function is *O(n log n)*.

This is a sample runtime table from one case of running the program, as well as a graph depicting this data:

n	time (s)
100000	0.320485353469848
200000	0.554017066955566
300000	0.881973266601562
400000	1.125880241394040
500000	1.519998073577880
600000	1.771317720413200
700000	2.141910076141350
800000	2.427339792251580
900000	2.882771492004390
1000000	3.120078325271600
1100000	3.574099063873290
1200000	3.837089300155630
1300000	4.362854957580560
1400000	4.588498830795280
1500000	4.869853734970090
1600000	5.468279838562010
1700000	5.630125761032100
1800000	5.975736856460570
1900000	6.244175434112540
2000000	6.611451864242550
2100000	6.926522016525260
2200000	7.241564750671380
2300000	7.727230072021480
2400000	8.203099250793450
2500000	8.580536127090450
2600000	8.705816507339470
2700000	9.332063198089600
2800000	9.647487640380860
2900000	10.006412029266300
3000000	10.300064086914000



This graph might at first appear to depict a roughly linear relationship between runtime and n as n increases. However, this graph is consistent with the graph of $n \log n$, so it does indeed confirm that the runtime increases as a function of $n \log n$ as n increases.

2.4 Dynamic Programming Algorithm

The dynamic programming algorithm runs in $O(n^2)$ time. This is the code for the dynamic programming algorithm:

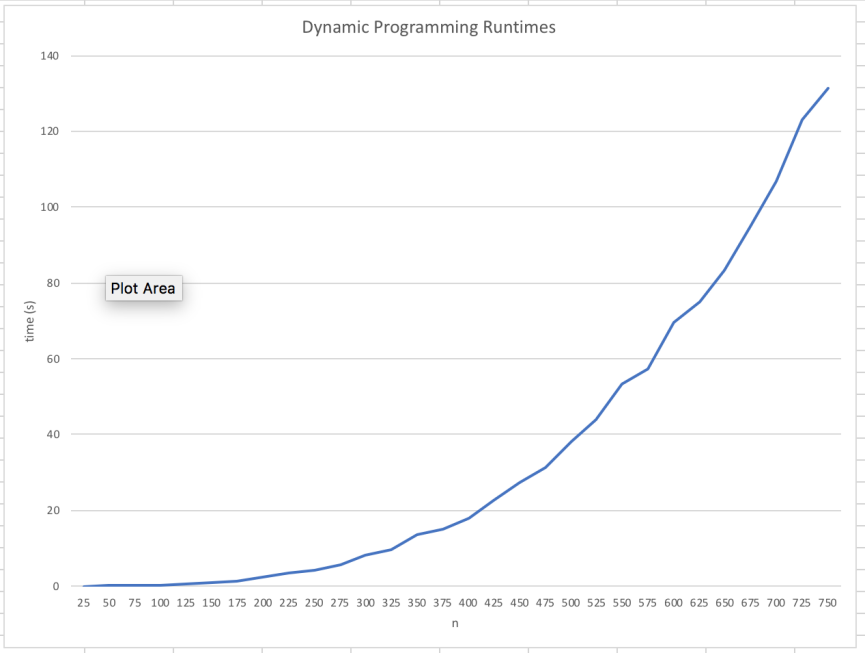
```
127     def dynamic_solution(self):
128         """..."""
134         if self.log:...
137
138         # array for storing already computer solutions; rows are weights, columns are indices
139         # ex: computed_slns[weight][i]
140         computed_slns = [[None] * (self.n + 1) for row in range(self.weight_capacity + 1)]
141
142         for row in range(self.weight_capacity + 1):
143             for col in range(self.n + 1):
144                 # if the weight available or items available is 0, the solution is 0
145                 if row == 0 or col == 0:
146                     computed_slns[row][col] = 0
147
148                 # if the weight of this item is less than or equal to the weight available
149                 elif self.items[col-1].weight <= row:
150                     # get values for this item
151                     my_weight = self.items[col-1].weight
152                     my_value = self.items[col-1].value
153
154                     # get the max of whether this item is left or taken
155                     # if item is taken, add the value and get the solution from the remaining weight
156                     # if item is left, the solution is the same as previous item
157                     computed_slns[row][col] = max(my_value + computed_slns[row-my_weight][col-1],
158                                                   computed_slns[row][col-1])
159
160                 # weight of item does not fit, so this solution is same as the previous item
161                 else:
162                     computed_slns[row][col] = computed_slns[row][col-1]
163
164         optimal_value = computed_slns[self.weight_capacity][self.n]
165
166         if self.log:...
178
179         return optimal_value
```

Creating a new array for computed solutions in line 140 runs in $O(n * w)$ time because it is essentially a doubly nested for loop with the inner loop being n items and the outer loop being the weight capacity, w. The bulk of this code is in another doubly nested for loop with the inner loop again being n items and the outer loop being the weight capacity, w. Note that although both of these aforementioned doubly nested for loops actually have a + 1 at the end, this constant can be ignored for the purpose of this big-O analysis. All of the operations inside of this for loop run in constant time: comparing row and col to 0 (line 145), looking up the items weight and comparing it to the current row (line 149), looking up the items weight and value (lines 151-152), looking up previous values in the computed_slns array and taking the max (lines 157-158), and looking up a previous value in the array (line 162). Therefore, this doubly nested for loop has the longest runtime, so it dictates the runtime of the algorithm.

The doubly nested for loop takes $O(n * w)$ time to run. However, as per the assignment document and as is coded in the problem generation function, the maximum weight capacity, w, is 75% of the total weight of all n items. So the weight capacity is actually a function of n items, times some constant number. Ignoring that constant number, we can equate w to n for the purpose of this analysis. Therefore, the overall runtime of the dynamic programming algorithm is $O(n^2)$.

This is a sample runtime table from one case of running the program, as well as a graph depicting this data:

n	time (s)	optimality
25	0.010589361190796	0.996466431
50	0.066541910171509	1
75	0.155781269073486	1
100	0.396508216857910	0.998277718
125	0.638076305389404	1
150	0.901561498641967	0.999910056
175	1.453228473663330	0.999859964
200	2.444883346557610	0.999569545
225	3.293499469757080	0.999743995
250	4.269793987274170	1
275	5.717692852020260	1
300	8.173082590103150	1
325	9.746555566787720	0.999880404
350	13.404993295669500	1
375	15.167048454284600	0.999937103
400	18.054074287414500	1
425	22.762452602386400	0.999893799
450	27.412940025329500	0.999807275
475	31.411478281021100	1
500	37.978738546371400	0.999787731
525	43.796776533126800	0.999865849
550	53.355222702026300	0.999798285
575	57.301986217498700	0.999955149
600	69.691458463668800	1
625	74.953995466232300	0.999945624
650	83.240161895751900	0.999950237
675	95.013511657714800	0.999894878
700	106.771566152572000	0.999962853
725	123.221772432327000	0.999995996
750	131.337115049362000	0.999938765



This graph indeed appears to show that the runtime increases as a function of n^2 as n increases.