The Knapsack Problem

You find yourself in a vault chock full of valuable items. However, you only brought a knapsack of capacity S pounds, which means the knapsack will break down if you try to carry more than S pounds in it). The vault has n items, where item i weighs s_i pounds, and can be sold for v_i dollars. You must choose which items to take in your knapsack so that you'll make as much money as possible after selling the items. The weights, s_i, are all integers.

Decisions and State

A solution to an instance of the Knapsack problem will indicate which items should be added to the knapsack. The solution can be broken into n true / false decisions $d_0 \dots d_{n-1}$. For $0 \leq i \leq n-1$, d_i indicates whether item i will be taken into the knapsack.

In order to decide whether to add an item to the knapsack or not, we need to know if we have enough capacity left over. So the "current state" when making a decision must include the available capacity or, equivalently, the weight of the items that we have already added to the knapsack.

Dynamic Programming Solution

$dp[i][j]$ is the maximum value that can be obtained by using a subset of the items $i \dots n-1$ (last $n-i$ items) which weighs **at most** j pounds. When computing $dp[i][j]$, we need to consider all the possible values of d_i (the decision at step i):

1. Add item i to the knapsack. In this case, we need to choose a subset of the items $i+1 \dots n-1$ that weighs at most $j - s_i$ pounds. Assuming we do that optimally, we'll obtain $dp[i+1][j - s_i]$ value out of items $i+1 \dots n-1$, so the total value will be $v_i + dp[i+1][j-s_i]$.

2. Don't add item i to the knapsack, so we'll re-use the optimal solution for items $i+1 \dots n-1$ that weighs at most j pounds. That answer is in $dp[i+1][j]$.

We want to maximize our profits, so we'll choose the best possible outcome.

$$dp[i][j] = \max \begin{pmatrix} dp[i+1][j] \\ dp[i+1][j-s_i] + v_i \quad \text{if } j \geq s_i \end{pmatrix}$$

To wrap up the loose ends, we notice that $dp[n][j] = 0, \forall 0 \leq j \leq S$ is a good base case, as the interval $n \dots n-1$ contains no items, so there's nothing to add to the knapsack, which means the total sale value will be 0. The answer to our original problem can be found in $dp[0][S]$. The value in $dp[i][j]$ depends on values of $dp[i+1][k]$ where $k < j$, so a good topological sort would be:

$$\begin{array}{cccc} dp[n][0] & dp[n][1] & \dots & dp[n][S] \\ dp[n-1][0] & dp[n-1][1] & \dots & dp[n-1][S] \\ \vdots & \vdots & & \vdots \\ dp[0][0], & dp[0][1] & \dots & dp[0][S] \end{array}$$

This topological sort can be produced by the pseudo-code below.

KNAPSACKDPTOPSORT(n, S)

```
1  for i in {n, n − 1 ... 0}
2      for j in {0, 1 ... S}
3          print (i, j)
```

The full pseudo-code is straight-forward to write, as it closely follows the topological sort and the Dynamic Programming recurrence.

KNAPSACK(n, S, s, v)

```
 1  for i in {n, n − 1 ... 0}
 2      for j in {0, 1 ... S}
 3          if i == n
 4              dp[i][j] = 0 // initial condition
 5          else
 6              choices = []
 7              APPEND(choices, dp[i + 1][j])
 8              if j ≥ s_i
 9                  APPEND(choices, dp[i + 1][j − s_i] + v_i)
10              dp[i][j] = MAX(choices)
11  return dp[0][S]
```

DAG Shortest-Path Solution

The Knapsack problem can be reduced to the single-source shortest paths problem on a DAG (directed acyclic graph). This formulation can help build the intuition for the dynamic programming solution.

The state associated with each vertex is similar to the dynamic programming formulation: vertex (i, j) represents the state where we have considered items $0 \ldots i$, and we have selected items whose total weight is at most j pounds. We can consider that the DAG has $n + 1$ layers (one layer for each value of i, $0 \leq i \leq n$), and each layer consists of $S + 1$ vertices (the possible total weights are $0 \ldots S$).

As shown in figure 1, each vertex (i, j) has the following outgoing edges:

- Item i is not taken: an edge from (i, j) to $(i + 1, j)$ with weight 0
- Item i is taken: an edge from (i, j) to $(i + 1, j + s_i)$ with weight $-v_i$ if $j + s_i \leq S$

The source vertex is $(0, 0)$, representing an initial state where we haven't considered any item yet, and our backpack is empty. The destination is the vertex with the shortest path out of all vertices (n, j), for $0 \leq j \leq S$.

Alternatively, we can create a virtual source vertex s, and connect it to all the vertices $(0, j)$ for $0 \leq j \leq S$, meaning that we can leave j pounds of capacity unused (the knapsack will end up weighing $S - j$ pounds). In this case, the destination is the vertex (n, S). This approach is less intuitive, but matches the dynamic programming solution better.

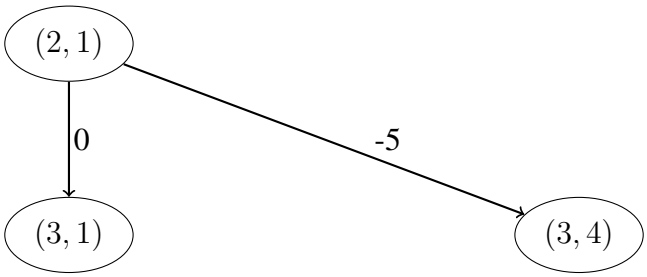

Figure 1: Edges coming out of the vertex $(2, 1)$ in a Knapsack problem instance where item 2 has weight 3 and value 5. If item 2 is selected, the new total weight will be $1 + 3 = 4$, and the total value will increase by 5. Edge weights are negative so that the shortest path will yield the maximum knapsack value.

Running Time

The dynamic programming solution to the Knapsack problem requires solving $O(nS)$ sub-problems. The solution of one sub-problem depends on two other sub-problems, so it can be computed in $O(1)$ time. Therefore, the solution's total running time is $O(nS)$.

The DAG shortest-path solution creates a graph with $O(nS)$ vertices, where each vertex has an out-degree of $O(1)$, so there are $O(nS)$ edges. The DAG shortest-path algorithm runs in $O(V+E)$, so the solution's total running time is also $O(nS)$. This is reassuring, given that we're performing the same computation.

The solution running time is **not polynomial** in the input size. The next section explains the subtle difference between **polynomial** running times and **pseudo-polynomial** running times, and why it matters.

Polynomial Time vs Pseudo-Polynomial Time

To further understand the difference between algorithms with polynomial and pseudo-polynomial running times, let's compare the performance of the Dynamic Programming solution to the Knapsack problem with the performance of Dijkstra's algorithm for solving the single-source shortest paths problem.

Dynamic Programming for Knapsack

The input for an instance of the Knapsack problem can be represented in a reasonably compact form as follows (see Figure 2):

- The number of items n, which can be represented using $O(\log n)$ bits.
- n item weights. We notice that item weights should be between $0 \ldots S$ because we can ignore any items whose weight exceeds the knapsack capacity. This means that each weight can be represented using $O(\log S)$ bits, and all the weights will take up $O(n \log S)$ bits.
- n item values. Let V be the maximum value, so we can represent each value using $O(\log V)$ bits, and all the values will take up $O(n \log V)$ bits.

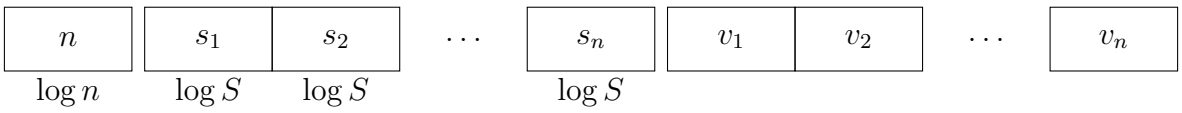

Figure 2: A compact representation of an instance of the Knapsack problem. The item weights, s_i, should be between 0 and S. An item whose weight exceeds the knapsack capacity ($s_i > S$) can be immediately discarded from the input, as it wouldn't fit the knapsack.

The total input size is $O(\log(n) + n(\log S + \log V)) = O(n(\log S + \log V))$. Let $b = \log S$, $v = \log V$, so the input size is $O(n(v + b))$. The running time for the Dynamic Programming solution is $O(nS) = O(n \cdot 2^b)$. Does this make you uneasy? It should, and the next paragraph explains why.

In this course, we use asymptotic analysis to understand the changes in an algorithm's performance as its input size increases – the corresponding buzzword is "scale", as in "does algorithm X scale?". So, how does our Knapsack solution runtime change if we double the input size?

We can double the input size by doubling the number of items, so $n' = 2n$. The running time is $O(nS)$, so we can expect that the running time will double. Linear scaling isn't too bad!

However, we can also double the input size by doubling v and b, the number of bits required to represent the item weights and values. v doesn't show up in the running time, so let's study the impact of doubling the input size by doubling b. If we set $b' = 2b$, the $O(n \cdot 2^b)$ result of our algorithm analysis suggests that the running time will increase quadratically!

To drive this point home, Table 1 compares the solution's running time for two worst-case inputs of $6,400$ bits against a baseline worst-case input of $3,200$ bits.

Metric	Baseline	Double n	Double b
n	100 items	200 items	100 items
b	32 bits	32 bits	64 bits
Input size	$3,200$ bits	$6,400$ bits	$6,400$ bits
Worst-case S	$2^{32} - 1 = 4 \cdot 10^9$	$4 \cdot 10^9$	$2^{64} - 1 = 1.6 \cdot 10^{19}$
Running time	$4 \cdot 10^{11}$ ops	$8 \cdot 10^{11}$ ops	$1.6 \cdot 10^{21}$ ops
Input size	1x	2x	2x
Time	1x	2x	$4 \cdot 10^9$x

Table 1: The amounts of time required to solve some worst-case inputs to the Knapsack problem.

The Dynamic Programming solution to the Knapsack problem is a **pseudo-polynomial** algorithm, because the running time will not always scale linearly if the input size is doubled. Let's look at Dijkstra's algorithm, for comparison.

Dijkstra for Shortest-Paths

Given a graph G with V vertices and E edges, Dijkstra's algorithm implemented with binary heaps can compute single-source shortest-paths in $O(E \log V)$[1].

The input graph for an instance of the single-source shortest-paths problem can be represented in a reasonably compact form as follows (see Figure 3):

- The number of vertices V, which can be represented using $O(\log V)$ bits.
- The number of edges E, which can be represented using $O(\log E)$ bits.
- E edges, represented as a list of tuples (s_i, t_i, w_i) (directed edge i goes from vertex s_i to vertex t_i, and has weight w_i). We notice that s_i and t_i can be represented using $O(logV)$ bits, since they must address V vertices. Let the maximum weight be W, so that each w_i takes up $O(\log W)$ bits. It follows that all edges take up $O(E(\log V + \log W))$ bits.

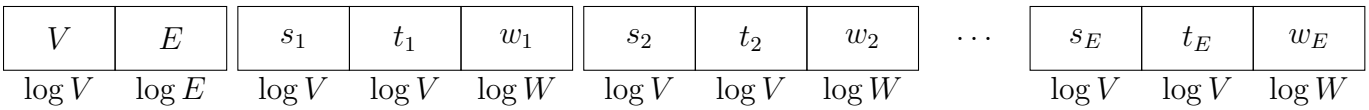

Figure 3: A compact representation of an instance of the single-source shortest-paths problem. Each edge i is represented by a tuple, (s_i, t_i, w_i), where s_i and t_i are between 0 and $V - 1$ and w_i is the edge's weight.

The total input size is $O(\log(V) + \log(E) + E(2 \log V + \log W)) = O(E(\log V + \log W))$. Let $b = \log W$, so the input size is $O(E(\log V + b))$.

[1] A Fibonacci heaps-based implementation would have a running time of $O(E + V \log V)$. Using that time would our analysis a bit more difficult. Furthermore, Fibonacci heaps are only important to theorests, as their high constant factors make them slow in practice.

Note that the graph representation above, though intuitive, is not suitable for running Dijkstra. However, the edge list representation can be used to build an adjacency list representation in $O(V + E)$, so the representation choice does not impact the total running time of Dijkstra's algorithm, which is $O(E \log V)$.

Compare this result to the Knapsack solution's running time. In this case, the running time is polynomial (actually linear) in the number of bits required to represent the input. If we double the input size by doubling E, the running time will double. If we double the input size by doubling the width of the numbers used to represent V (in effect, we'd be squaring V, but not changing E), the running time will also double. If we double the input size by doubling b, the width of the numbers used to represent edge weights, the running time doesn't change. No matter the reason why how the input size doubles, there is no explosion in the running time.

Computation Model

Did that last paragraph make you uneasy? We just said that doubling the width of the numbers used to represent the edge weights doesn't change the running time at all. How can that be?!

We're assuming the RAM computation model, where we can perform arithmetic operations on word-sized numbers in $O(1)$ time. This simplifies our analysis, but the main result above (Dijkstra is polynomial, Knapsack's solution is pseudo-polynomial) hold in any reasonable (deterministic) computational model. You can convince yourself that, even if arithmetic operations take time proportional to the inputs sizes ($O(b)$, $O(n)$, or $O(\log V)$, depending on the case) Dijkstra's running time is still polynomial in the input size, whereas the Dynamic Programming solution to the Knapsack problem takes an amount of time that is exponential in b.

Subset Sum

In a variant of the Knapsack problem, all the items in the vault are gold bars. The value of a gold bar is directly proportional to its weight. Therefore, in order to make the most amount of money, you must fill your knapsack up to its full capacity of S pounds. Can you find a subset of the gold bars whose weights add up to exactly S?

Boolean Logic for Dynamic Programming

Dynamic Programming is generally used for optimization problems that involve maximizing an objective. The Subset Sum problem asks us for a boolean (TRUE / FALSE) answer to the question "Is there a combination of bars whose weights add up to S or not?" Fortunately, we can use the boolean logic formulation below to transform this into a maximization problem.

We'll use the C language representation of booleans: TRUE is 1, FALSE is 0. This is motivated by the fact that we want to arrive to an answer of TRUE whenever that is possible, and only arrive to an answer of FALSE otherwise.

Formally, let $\mathcal{S}$ be the set of all possible subsets of gold bars, and let $good(s)$ be a function that takes a subset of gold bars and returns TRUE if it is a good solution (the total weight of the bars is S) and FALSE otherwise. If we let TRUE $= 1$ and FALSE $= 0$, the answer to our initial problem is

$$\max_{s \in \mathcal{S}} good(s)$$

We can define the familiar Boolean arithmetic operations AND, OR, and NOT as follows. Let a and b be Boolean variables.

- AND(a, b): $a \wedge b = a \cdot b$ (AND is multiplication)
- OR(a, b): $a \vee b = \max(a, b)$ (OR is $\max$)
- NOT(a): $\neg a = 1 - a$ (NOT is 1's complement)

Dynamic Programming Solution

The Subset Sum solution has identical decisions and state to the Knapsack solution, so we can go straight to writing the Dynamic Programming solution.

$dp[i][j]$ is TRUE (1) if there is a subset of the gold bars $i \ldots n-1$ (last $n-i$ bars) which weighs **exactly** j pounds. When computing $dp[i][j]$, we need to consider all the possible values of d_i (the decision at step i, which is whether or not to include bar i):

1. Add bar i to the knapsack. In this case, we need to choose a subset of the bars $i+1 \ldots n-1$ that weighs exactly $j - s_i$ pounds. $dp[i+1][j-s_i]$ indicates whether such a subset exists.
2. Don't add item i to the knapsack. In this case, the solution rests on a subset of the bars $i+1 \ldots n-1$ that weighs exactly j pounds. The answer of whether such a subset exists is in $dp[i+1][j]$.

Either of the above avenues yields a solution, so $dp[i][j]$ is TRUE if at least one of the decision possibilities results in a solution. Recall that OR is implemented using $\max$ in our Boolean logic. The recurrence (below) ends up being simpler than the one in the Knapsack solution!

$$dp[i][j] = \max \begin{pmatrix} dp[i+1][j] \\ dp[i+1][j-s_i] \quad \text{if } j \geq s_i \end{pmatrix}$$

The initial conditions for this problem are $dp[n][0] = 1$ (TRUE) and $dp[n][j] = 0$ (FALSE) $\forall 1 \leq j \leq S$. The interval $n \ldots n-1$ contains no items, the corresponding knapsack is empty, which means the only achievable weight is 0.

Just like in the Knapsack problem, the answer the original problem is in $dp[0][S]$. The topological sort is identical to Knapsack's, and the pseudo-code only has a few small differences.

```
SUBSETSUM(n, S, s)
 1  for i in {n, n − 1 ... 0}
 2      for j in {0, 1 ... S}
 3          if i == n // initial conditions
 4              if j == 0
 5                  dp[i][j] = 1
 6              else
 7                  dp[i][j] = 0
 8          else
 9              choices = []
10              APPEND(choices, dp[i + 1][j])
11              if j ≥ s_i
12                  APPEND(choices, dp[i + 1][j − s_i])
13              dp[i][j] = MAX(choices)
14  return dp[0][S]
```

DAG Shortest-Path Solution

The DAG representation of the problem is also very similar to the Knapsack representation, and slightly simpler.

A vertex (i, j) represents the possibility of finding a subset of the first i bars that weighs exactly j pounds. So, if there is a path to vertex (i, j) (the shortest path length is less than $\inf$), then the answer to sub-problem (i, j) is TRUE.

The starting vertex is $(0, 0)$ because we can only achieve a weight of 0 pounds using 0 items. The destination vertex is (n, S), because that maps to the goal of finding a subset of all items that weighs exactly S pounds.

We only care about the existence of a path, so we can use BFS or DFS on the graph. Alternatively, we can assign the same weight of 1 to all edges, and run the DAG shortest-paths algorithm. Both approaches yield the same running time, but the latter approach maps better to the dynamic programming solution.

K-Sum

A further restriction of the Subset Sum problem is that the backpack has K pockets, and you can fit exactly one gold bar in a pocket. You must fill up all the pockets, otherwise the gold bars will move around as you walk, and the noises will trigger the vault's alarm. The problem becomes: given n gold bars of weights $s_0 \dots s_{n-1}$, can you select exactly K bars whose weights add up to exactly S?

Decisions and State

A solution to a problem instance looks the same as for the Knapsack problem, so we still have n Boolean decisions $d_0 \dots d_{n-1}$.

However, the knapsack structure changed. In the original problem, the knapsack capacity placed a limit on the total weight of the items, so we needed to keep track of the total weight of the items that we added. That was our state. In the k-Sum problem, the knapsack also has slots. When deciding whether to add a gold bar to the knapsack or not, we need to know if there are any slots available, which is equivalent to knowing how many slots we have used up. So the problem state needs to track both the total weight of the bars in the knapsack, and how many slots they take up. The number of slots equals the number of bars in the backpack.

Dynamic Programming Solution

$dp[i][j][k]$ is TRUE (1) if there is a k-element subset of the gold bars $i \dots n-1$ that weighs exactly j pounds. $dp[i][j][k]$ is computed considering all the possible values of d_i (the decision at step i, which is whether or not to include bar i):

1. Add bar i to the knapsack. In this case, we need to choose a $k-1$-bar subset of the bars $i+1 \dots n-1$ that weighs exactly $j - s_i$ pounds. $dp[i+1][j-s_i][k-1]$ indicates whether such a subset exists.

2. Don't add item i to the knapsack. In this case, the solution rests on a k-bar subset of the bars $i+1 \dots n-1$ that weighs exactly j pounds. The answer of whether such a subset exists is in $dp[i+1][j][k]$.

Either of the above avenues yields a solution, so $dp[i][j][k]$ is TRUE if at least one of the decision possibilities results in a solution. The recurrence is below.

$$dp[i][j][k] = \max \left(\begin{array}{l} dp[i+1][j][k] \\ dp[i+1][j-s_i][k-1] \quad \text{if } j \geq s_i \text{ and } k > 0 \end{array} \right)$$

The initial conditions for this problem are $dp[n][0][0] = 1$ (TRUE) and $dp[n][j][k] = 0$ (FALSE) $\forall 1 \leq j \leq S, 0 \leq k \leq K$. The interval $n \dots n-1$ contains no items, the corresponding knapsack is empty, which means the only achievable weight is 0.

The answer the original problem is in $dp[0][S][K]$.

A topological sort can be produced by the following pseudo-code.

```
DPTOPSORT(n, S)
1  for i in {n − 1, n − 2 . . . 0}
2      for j in {0, 1 . . . S}
3          for k in {0, 1 . . . K}
4              print (i, j, k)
```

Once again, the topological sort can be combined with the Dynamic Programming recurrence to obtain the full solution pseudo-code in a straight-forward manner.

```
KSUM(n, K, S, s)
 1  for i in {n, n − 1 . . . 0}
 2      for j in {0, 1 . . . S}
 3          for k in {0, 1 . . . K}
 4              if i == n // initial conditions
 5                  if j == 0 and k == 0
 6                      dp[i][j][k] = 1
 7                  else
 8                      dp[i][j][k] = 0
 9              else
10                  choices = []
11                  APPEND(choices, dp[i + 1][j][k])
12                  if j ≥ s_i and k > 0
13                      APPEND(choices, dp[i + 1][j − s_i][k − 1])
14                  dp[i][j][k] = MAX(choices)
15  return dp[0][S][K]
```

DAG Shortest-Path Solution

The reduction to shortest-paths in a DAG offers a different avenue that leads to the same solution, and helps build intuition.

Let's start with the reduction of Subset Sum. We have a graph of vertices (i, j), and want a path from $(0, 0)$ to (n, S) that picks up exactly k bars.

Let's attempt to use edge weights to count the number of bars. Edges from (i, j) to $(i + 1, j)$ will have a cost of 0, because it means that bar i is not added to the knapsack. Edges from (i, j) to $(i + 1, j + s_i)$ have a cost of 1, as they imply that one bar is added to the knapsack, and therefore one more slot is occupied. Figure 4 shows an example.

Given the formulation above, we want to find a path from $(0, 0)$ to (n, S) whose cost is exactly K. We can draw upon our intution built from previous graph transformation problems, and realize that we can represent K-cost requirement by making K copies of the graph, where each copy k is a layer that captures the paths whose cost is exactly k.

Therefore, in the new graph, a vertex (i, j, k) indicates whether there is a k-cost path to vertex (i, j) in the Subset Sum problem graph. Each 0-cost edge from (i, j) to $(i + 1, j)$ in the old graph

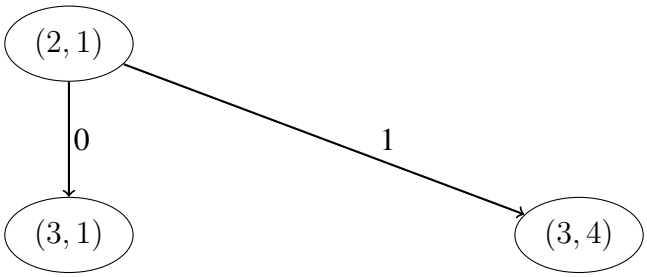

Figure 4: Edges coming out of the vertex $(2, 1)$ in a k-Sum problem instance where gold bar 2 has weight 3. If bar 2 is added to the knapsack, the new total weight will be $1 + 3 = 4$, and the number of occupied slots increases by 1.

maps to K edges from (i, j, k) to $(i + 1, j, k)$ in the new graph. Each 1-cost edge from (i, j) to $(i, j + s_i)$ maps to K edges (i, j, k) to $(i, j + s_i, k + 1)$. A comparison between figures 5 and 4 illustrates the transformation.

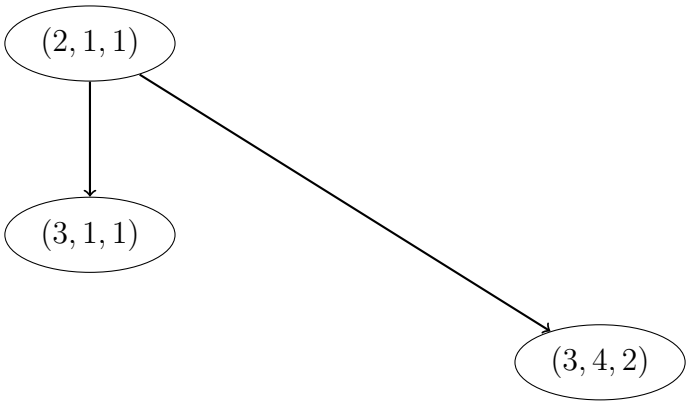

Figure 5: Edges coming out of the vertex $(2, 1, 1)$ in a k-Sum problem instance where gold bar 2 has weight 3. If bar 2 is added to the knapsack, the new total weight will be $1 + 3 = 4$, and the number of occupied slots increases by 1.

In the new graph, we want to find a path from vertex $(0, 0, 0)$ to vertex (n, S, K). All edges have the same weight.

Convince yourself that the DAG described above is equivalent to the dynamic programming formulation in the previous section.

Running Time

The Dynamic Programming solution solves $O(K \cdot ns)$ sub-problems, and each sub-problem takes $O(1)$ time to solve. The solution's total running time is $O(KnS)$.

The DAG has $K + 1$ layers of $O(nS)$ vertices (vertex count borrowed from the Knapsack problem), and K copies of the $O(nS)$ edges in the Knapsack graph. Therefore, $V = O(K \cdot S)$ and $E = O(Kc \cdot S)$. The solution's running time is $O(V + E) = o(KnS)$

On your own: Multiset k-Sum

How does the previous problem change if there is an endless supply of copies of each bar so, effectively, each bar can be added to the knapsack more than once?

Compare the $O(nK \cdot S)$ solution to the $O(n^{\frac{k}{2}})$ solution in Quiz 1 (Problem 6: Shopping Madness).