Consider a restaurant that offers your family a free meal. The only condition is that you must spend exactly $1073.00. If you spend any other amount, you’ll need to pay, but if your bill is exactly $1073.00, it will be free. We face two questions: the first question is whether, given the menu of the restaurant, it is even possible to spend exactly $1073.00; the second question is which items we should order so that we’ll spend exactly $1073.00.

This problem is “the subset-sum” problem, and it is a quite famous (and surprisingly difficult) problem in computer science. Formally, subset-sum gives a set (here, the prices on the menu), $M$ and a total amount $t$, and asks whether

$$\exists S \subseteq M : \sum_{i \in S} i = t;$$

that is, it asks whether there exists a subset of $M$ named $S$ such that the sum of all elements in $S$ is $t$ (note that there is a small subtlety that states that it is not valid to spend $1073.00 by ordering the same menu item multiple times).
14.1 Brute-force

The brute-force solution to subset sum is fairly obvious: we simply try all subsets of \( M \) and check whether one of them sums to \( t \). We call the set of all subsets of \( M \) the “power-set” of \( M \). If \( M = \{m_0, m_1, \ldots, m_{n-1}\} \), then we can find the power set by enumerating every subset of \( M \) by how many items each subset contains:

\[
powerset(M) = \{
\{\},
\{m_0\}, \{m_1\}, \{m_2\}, \ldots, \{m_{n-1}\},
\{m_0, m_1\}, \{m_0, m_2\}, \ldots, \{m_1, m_2\}, \{m_1, m_3\}, \ldots, \{m_{n-2}, m_{n-1}\},
\ldots
M \setminus m_0, M \setminus m_1, M \setminus m_2, \ldots, M \setminus m_{n-1},
M
\}.
\]

In this manner, we need only iterate through every \( S \in powerset(M) \), as shown in Listing 14.1.

Listing 14.1: Solving subset-sum with brute force. The output is \texttt{False True}, indicating that 110 cannot be made by any subset of \( M \), but that 120 can be made by at least one subset of \( M \).

```python
import itertools

# from stackoverflow:
def powerset(items):
    s = list(items)
    return itertools.chain.from_iterable(itertools.combinations(s, r) for r in range(len(s)+1))

def powerset_subset_sum(M, t):
    for S in powerset(M):
        if sum(S) == t:
            return True
    return False

M = set([3, 5, 7, 9, 11, 13, 109, 207, 113, 300])

print powerset_subset_sum(M, 110)
```
14.2 Dynamic programming

We can improve this by using dynamic programming (mentioned in Chapter 7): here we will start our total at 0. Then, we will iterate through every value $M$ and add it to the total. But rather than add it deterministically, we will add both outcomes: we will add the outcome where the element $m_i$ is included in $S$ and we will add the outcome where the element $m_i$ is excluded from $S$. This is shown graphically in Figure [14.1].

As we move left to right, each layer of the graph will perform the Cartesian product between reachable nodes at the layer $i - 1$ and the set $\{0, m_i\}$. The current totals that can be reached in each layer can be stored in a sparse manner (via set or dictionary) or in a dense manner via array (as shown in Figure [14.1]).

When asking whether

$$\exists S \subseteq M : \sum_{i \in S} = t,$$

we need only determine whether there exists a path ending at the node with value $t$ (i.e., the node with a height of $t$) in the far right layer.

To figure out the cost of dynamic programming, we will restrict that each of the $m_i \leq k$ for some value $k$ representing the menu item with maximum price. Now we will compute the computational cost in terms of both $n = |M|$ and $k$. Note that these are related to our goal $t$: $t \leq n \cdot k$, because that would be the maximum meal attainable when ordering one of every menu

\[1\] Also called a “map”
After adding in possible contributions from

Figure 14.1: Dynamic programming for solving subset-sum on a set of integers. The total bill starts at 0 and each element of the set $M = \{m_0, m_1, m_2, \ldots, m_{n-1}\} = \{1, 3, 5, \ldots, 7\}$ is added in left-to-right. Each of these elements may be excluded from the set $S$ (moving horizontally without moving up) or included in the set $S$ (moving diagonally up and right, where we move up $m_i$ units). For example, after adding in $m_0 = 1$ the total is in $\{0, 1\}$. After adding in $m_1$ the total is in $\{0, 1, 3\}$. 
item (where each has price \( \leq k \)); as a result, sometimes you will see the computational complexity of dynamic programming for knapsack depicted in terms of \( n \) and \( t \) instead of \( n \) and \( k \).

At each left-to-right layer in the graph, the height of the next layer will increase by at most \( k \). The number of nodes in layer \( i \), \( h_i \) will therefore be \( h_i \leq k \cdot i + 1 \). Note that we add 1 because layer 0 has 1 node. Thus, creating these nodes will cost

\[
\sum_{i=0}^{n} h_i \leq \sum_{i=0}^{n} (k \cdot i + 1)
\]

\[
= \sum_{i=0}^{n} k \cdot i + \sum_{i=0}^{n} 1
\]

\[
= k \cdot \sum_{i=0}^{n} (i) + n
\]

\[
\in k \cdot \Theta(n^2) + n
\]

\[
= \Theta(k \cdot n^2).
\]

This does not yet show that we have found a solution in \( \Theta(k \cdot n^2) \). It only shows that we have built the nodes in that amount of time; however, the total runtime for solving subset-sum on a set of integers using this dynamic programming technique will not be substantially larger than the cost of initializing these nodes. The reason for this is that we add at most 2 edges out from every node (if the node was reachable, then we add an edge out excluding \( m_i \) from \( S \) and an edge out including \( m_i \) in \( S \)). Thus, the cost of adding edges and marking whether a node in the next layer is reachable (via a simple boolean) is no more than a constant factor greater than the total number of nodes. Thus, the total cost of our dynamic programming approach to subset-sum is \( \in \Theta(k \cdot n^2) \). This may also be written as

\[
\Theta(k \cdot n^2) = \Theta \left( \frac{t}{n} \cdot n^2 \right) = \Theta(t \cdot n).
\]

This approach is shown in Listing 14.2.

Listing 14.2: Solving subset-sum with dynamic programming. This method uses the same strategy as Figure 14.1. The output is \texttt{False True}, indicating that 110 cannot be made by any subset of \( M \), but that 120 can be made by at least one subset of \( M \).
```
def dynamic_programming_subset_sum(M, t):
    # start current_layer[0] as True
    current_layer = [True]

    for m_i in M:
        next_layer_size = len(current_layer) + m_i
        next_layer = [False]*next_layer_size

        for j in xrange(len(current_layer)):
            if current_layer[j]:
                next_layer[j + 0] = True
                next_layer[j + m_i] = True

        current_layer = next_layer

    # look in index t of current_layer and see whether or not it was
    # reachable
    return current_layer[t]

M = set([3, 5, 7, 9, 11, 13, 109, 207, 113, 300])
print dynamic_programming_subset_sum(M, 110)
print dynamic_programming_subset_sum(M, 120)
```

Note that this does not show that this is an algorithm that solves subset sum with runtime that is polynomial in the input parameters. The reason for this is that the value \( t \) requires only \( b = \lceil \log_2(t) \rceil \) bits\(^2\). When considering the number of bits given, the runtime is \( \in \Theta(2^b \cdot n) \). For this reason, we must be very careful when specifying whether or not an algorithm has a polynomial runtime\(^3\). Our dynamic programming approach to subset-sum has a runtime polynomial in \( n \) and \( k \), polynomial in \( n \) and \( t \), and at least exponential in \( b \).

\(^2\)We would also need bits to specify the values in the set \( M \), but here this is ignored for simplicity; however, each element is \( \leq k \), and so we need roughly \( n \cdot \lceil \log_2(k) \rceil \) bits additional for the set\(^4\).

\(^3\)We also may need some sort of delimeter to determine where one element’s bits end and where the next element’s bits begin.

\(^4\)To do so, we need to specify polynomial \emph{in what}?
14.3 Generalized subset-sum

We will denote the case where each individual has their own custom menu, from which they can order any items as the “generalized subset-sum” problem. In this case, we will continue to denote the maximum amount each individual can spend as \( k - 1 \) (i.e., there are \( k \) distinct values \( \{0, 1, 2, \ldots k - 1\} \) that each individual may be able to spend). Likewise the goal value \( t \) also does not need to have a fixed value; instead, it can be one of many allowed values specified by a set \( T \). Here we are considering whether the sums of each individual’s total purchases can reach any allowed goal \( t \in T \):

\[
m_0 + m_1 + m_2 + \cdots + m_{n-1} = t,
\]

where \( m_0 \in M_0, m_1 \in M_1, \ldots m_{n-1} \in M_{n-1}, \) where \( t \in T \), and where all \( M_i \subseteq \{0, 1, 2, \ldots k - 1\} \). For example, consider a possible generalized subset-sum problem with \( n = 2 \) and \( k = 6 \): individual 0 is able to spend any amount \( \in \{0, 1, 5\} \) and individual 1 is able to spend \( \in \{0, 1, 2, 4\} \). The total amount that can be spent by these two individuals is given by the Cartesian product between those sets:

\[
= \{i + j \mid \forall i \in M_0, \forall j \in M_1\} \\
= \{0, 1, 2, 3, 4, 5, 6, 7, 9\}.
\]

This generalized subset-sum problem can be solved using the same dynamic programming technique as the standard subset-sum problem; however, now the runtime is no longer dominated by the number of nodes in each layer, but by the number of edges in the graph.

The edges between layer \( i \) and layer \( i+1 \) will be the product of the number of nodes in layer \( i \) and the changes that each can introduce. The number of nodes in layer \( i \) will be roughly the same as before \( h_i \leq k \cdot i + 1 \); however, where before we had at most two edges coming from each node, we now have at most \( k \) edges coming from each node. Thus, the total number of edges
that we need to insert and process will be
\[
\sum_{i=0}^{n-1} h_i \times k = k \sum_{i=0}^{n-1} h_i

\leq k \sum_{i=0}^{n-1} (k \cdot i + 1)

= k \left( \sum_{i=0}^{n-1} k \cdot i + \sum_{i=0}^{n-1} 1 \right)

= k \left( \sum_{i=0}^{n-1} (k \cdot i) + n \right)

= k \left( k \sum_{i=0}^{n-1} (i) + n \right)

\in \Theta(k^2 \cdot n^2).
\]

Thus, we can use dynamic programming to solve the generalized subset-sum problem \( \in \Theta(k^2 \cdot n^2) \).

### 14.4 Convolution tree

It is an interesting question whether or not we can solve the generalized subset-sum problem faster than \( \Theta(k^2 \cdot n^2) \) used by the dynamic programming algorithm above. One thing that we can observe is that the dynamic programming starts efficient, but becomes slow as we progress and build the graph in a left-to-right manner. This is because the number of edges between a layer and the next layer will be bounded by the Cartesian product between the number of current nodes and the set \( \{0, 1, 2, \ldots k - 1\} \). The number of nodes in the leftmost layers (processed first) will be small, but as we progress, the Cartesian product becomes large.

One approach to attacking this problem is to merge arrays of similar size, thereby keeping them small as long as possible. For instance, the dynamic programming approach to summing the restaurant bill corresponds to placing the parentheses this way \(((M_0 + M_1) + M_2) + M_3\); instead, we can place the parenthesis in a manner that will merge pairs and then merge those merged
14.4. CONVOLUTION TREE

pairs: \((M_0 + M_1) + (M_2 + M_3)\). In this manner, we will merge collections of roughly equal size.

At first glance, this approach may appear promising; however, consider the final merger that will be performed: The final merger will be between \(\frac{n}{2}\) people’s subtotals and \(\frac{n}{2}\) people’s subtotals. Each will have subtotals \(\in \Theta(k \cdot \frac{n}{2})\). Performing the Cartesian product will have a cost

\[
\in \Theta \left( k \cdot \frac{n}{2} \times k \cdot \frac{n}{2} \right) = \Theta(k^2 \cdot n^2);
\]

therefore, we know that the cost of processing the full graph will be \(\in \Omega(k^2 \cdot n^2)\), which is no improvement.\(^5\)

The limiting factor is thus a single Cartesian product merger. Recall that when merging sets \(A\) and \(B\), we perform

\[
C = A \oplus B = \{i + j \mid \forall i \in A, \forall j \in B\}.
\]

This could be rewritten as a logical AND:

\[
i \in A \land j \in B \rightarrow i + j \in C.
\]

In turn, we could also rewrite this as a union of all possible ways to get to any sum \(i + j\):

\[
C = \bigcup_{i \in A, j \in B} \{i + j\}.
\]

Let us now shift from thinking of sets to thinking of equivalent vector forms:

\[
a_i = \begin{cases} 
1 & i \in A \\
0 & i \notin A 
\end{cases}
\]

\[
b_i = \begin{cases} 
1 & i \in B \\
0 & i \notin B 
\end{cases}
\]

\[
c_i = \begin{cases} 
1 & i \in C \\
0 & i \notin C 
\end{cases}
\]

\(^5\)It will actually yield a small constant speedup, but we are looking for more than a constant improvement.
From this, we have

\[ c_m = \bigcup_{i,j:i+j=m} a_i \land b_j \]
\[ = \bigcup_i a_i \land b_{m-i}. \]

Now, we can think of the union, \( \bigcup_i \), as a weaker form of a \( \sum_i \). That is, if we perform \( \sum_i \) on some vector of booleans, we can compute the union by testing whether or not the sum was > 0; therefore, we have

\[ d_m = \sum_i a_i \land b_{m-i}, \]
\[ c_m = d_m > 0. \]

Also note that with binary values \( a_i \) and \( b_j \), the logical AND is equivalent to multiplication: \( a_i \land b_j = a_i \cdot b_j \). Thus, we have

\[ d_m = \sum_i a_i \cdot b_{m-i}, \]
\[ = a \otimes b, \]
\[ c_m = d_m > 0, \]

where \( a \otimes b \) denotes the convolution between vectors \( a \) and \( b \).

In Chapter 13 we showed that a convolution between two vectors of length \( \ell \) can be computed in \( \Theta(\ell \log(\ell)) \) by using FFT. Thus, we can use FFT convolution to merge these sets and perform \( C = A \oplus B \) in \( \Theta(\ell \log(\ell)) \), where \( \ell = \max(|A|, |B|) \).

If we put this together with the merging pairs strategy above, we construct a tree data structure known as the “convolution tree”: In the first layer, there are \( n \) individuals (each described by a vector of length \( k \)) and \( \frac{n}{2} \) pairs. We merge each of these pairs using FFT in runtime \( \Theta(\frac{n}{2} \cdot k \log(k)) = \Theta(n \cdot k \log(k)) \). In the second layer, there are \( \frac{n}{4} \) vectors (each with length \( \leq 2k \)), which we can arrange into \( \frac{n}{4} \) pairs. The runtime of merging these pairs with FFT will be \( \in \Theta(\frac{n}{4} \cdot 2k \log(2k)) = \Theta(\frac{n}{4} \cdot k \log(2k)) \). We can continue in this
manner to compute the total runtime:

$$\sum_{i=0}^{\log_2(n)} \frac{n}{2^{i+1}} \cdot 2^i \cdot k \log_2(2^i \cdot k) = \sum_{i=0}^{\log_2(n)} \frac{n}{2} \cdot k \log_2(2^i \cdot k)$$

$$= \frac{n}{2} \cdot \sum_{i=0}^{\log_2(n)} k \cdot (\log_2(2^i) + \log_2(k))$$

$$= k \cdot \frac{n}{2} \cdot \sum_{i=0}^{\log_2(n)} (i + \log_2(k))$$

$$= k \cdot \frac{n}{2} \cdot \left( \sum_{i=0}^{\log_2(n)} i + \sum_{i=0}^{\log_2(n)} \log_2(k) \right)$$

$$= k \cdot \frac{n}{2} \cdot \log_2(n) \left( \frac{\log_2(n) (\log_2(n) + 1)}{2} + \log_2(k) \right)$$

$$\leq \Theta(k \cdot n \log(k \cdot n) \log(n)).$$

This runtime is “quasilinear” meaning it is only logarithmically slower than the linear $k \cdot n$ cost of loading the data that defines the problem.

Thus far, we have used convolution for addition between variables that can each take several values. Above, we did this in terms of sets. We can also do this in terms of probability distributions, which are identical to the vector forms of sets, except for the fact that probability distributions will be normalized so that the sum of the vector is 1. Such a normalized vector is called a “probability mass function” (PMF). An implementation of the convolution tree is shown in Listing 14.3. The successive merging of pairs is known as the “forward pass” of the convolution tree, and can be used to solve the generalized subset-sum problem in subquadratic time in both $n$ and $k$. The probabilistic version of whether the party could possibly reach any total bill $t \in T$ and is indicated by whether the “prior” vector, which is produced by the final merger, is nonzero at the index corresponding to any goal value. This corresponds to the forward pass denoted in Figure 14.2.

Listing 14.3: Solving generalized subset-sum with the convolution tree. The priors (accumulated by the forward pass) and the likelihoods (accumulated
Figure 14.2: Convolution tree, solving generalized subset-sum in sub-quadratic time. A convolution tree for \( n = 8 \) is depicted. The forward pass finds whether any element of \( T \) can be built as a sum of elements from \( M_0, M_1, M_2, \ldots \). The backward pass determines which elements of \( M_0 \), of \( M_1 \), etc., were used to achieve valid sums \( t \in T \).

by the backward pass) that are output are equivalent to the brute-force formulation.

```python
import numpy
from scipy.signal import fftconvolve

class PMF:
    def __init__(self, start_value, masses):
        self._start_value = start_value
        self._masses = numpy.array(masses, float)
        self._masses /= sum(self._masses)

    def size(self):
        return len(self._masses)

    def narrowed_to_intersecting_support(self, rhs):
        start_value = max(self._start_value, rhs._start_value)
        end_value = min(self._start_value + self.size() - 1,
                         rhs._start_value + rhs.size() - 1)
        masses = self._masses[start_value - self._start_value:]

        end_size = end_value - start_value + 1
        masses = masses[:end_size]
```

**14.4. CONVOLUTION TREE**

```python
return PMF(start_value, masses)

def __add__(self, rhs):
    return PMF(self._start_value + rhs._start_value,
               fftconvolve(self._masses, rhs._masses))

def __sub__(self, rhs):
    return PMF(self._start_value - (rhs._start_value+rhs.size()-1),
               fftconvolve(self._masses, rhs._masses[::-1]))

def __mul__(self, rhs):
    this = self.narrowed_to_intersecting_support(rhs)
    rhs = rhs.narrowed_to_intersecting_support(self)

    # now the supports are aligned:
    return PMF(this._start_value, this._masses*rhs._masses)

def support(self):
    return list(xrange(self._start_value, self._start_value+self.size()))

def support_contains(self, outcome):
    return outcome >= self._start_value and outcome < self._start_value +
    self.size()

def get_probability(self, outcome):
    return self._masses[outcome - self._start_value]

def __str__(self):
    result = 'PMF('
    for i in xrange(self.size()):
        mass = self._masses[i]
        result += str(self._start_value + i) + ':' + str(numpy.round(mass, 4))
        if i != self.size()-1:
            result += '	'
    return result

def __repr__(self):
    return str(self)

import itertools
# runtime is $\in \Omega(k^n)$
def brute_force_solve(prior_pmfs, likelihood_pmf):
```
# prior_pmfs = [X_0, X_1, ...]
# likelihood_pmf = Y|D

# compute prior of Y:
prior_supports = [pmf.support() for pmf in prior_pmfs]
all_joint_events = itertools.product(*prior_supports)
prior_of_y = [0.0]*likelihood_pmf.size()
for joint_event in all_joint_events:
    y = sum(joint_event)
    if likelihood_pmf.support_contains(y):
        probability = numpy.product([pmf.get_probability(event) for event, pmf in zip(joint_event, prior_pmfs)])
        prior_of_y[y-likelihood_pmf._start_value] += probability

prior_of_y = PMF(likelihood_pmf._start_value, prior_of_y)

# compute likelihoods X_1|D, X_2|D, ...
likelihoods = []
for i in range(len(prior_pmfs)):
    priors_without_i = [prior_pmfs[j] for j in range(len(prior_pmfs)) if j != i]
distributions = priors_without_i + [likelihood_pmf]
supports = [pmf.support() for pmf in priors_without_i] + [likelihood_pmf.support()]
all_joint_events = itertools.product(*supports)
result_i = [0.0]*prior_pmfs[i].size()
for joint_event in all_joint_events:
    y = joint_event[-1]
    sum_x_without_i = sum(joint_event[:-1])
    probability = numpy.product([pmf.get_probability(event) for event, pmf in zip(joint_event[:-1], distributions)])

    x_i = y - sum_x_without_i
    if prior_pmfs[i].support_contains(x_i):
        result_i[x_i - prior_pmfs[i]._start_value] += probability

result_i = PMF(prior_pmfs[i]._start_value, result_i)
likelihoods.append(result_i)

return likelihoods, prior_of_y
# runtime is \in \Theta(n k \log(n k) \log(n))

def convolution_tree_solve(prior_pmfs, likelihood_pmf):
    # prior_pmfs = \[X_0, X_1, \ldots\]
    # likelihood_pmf = Y|D
    n = len(prior_pmfs)

    # forward pass:
    layer_to_priors = []
layer_to_priors.append(prior_pmfs)

    while len(layer_to_priors[-1]) > 1:
        layer = []
        for i in xrange(len(layer_to_priors[-1])/2):
            layer.append( layer_to_priors[-1][2*i] + layer_to_priors[-1][2*i+1] )
        if len(layer_to_priors[-1]) % 2 != 0:
            layer.append(layer_to_priors[-1][-1])
layer_to_priors.append(layer)

    layer_to_priors[-1][0] =
    layer_to_priors[-1][0].narrowed_to_intersecting_support(likelihood_pmf)

    # backward pass:
    layer_to_likelihoods = [ [likelihood_pmf] ]

    for i in xrange(1, len(layer_to_priors)):
        # j is in \{1, \ldots \text{len(layer_to_priors)} - 1\}
        j = len(layer_to_priors) - i

        layer = []
        for k in xrange(len(layer_to_priors[j])):
            parent_likelihood = layer_to_likelihoods[-1][k]

        if 2*k+1 < len(layer_to_priors[j-1]):
            # this PMF came from two parents during merge step:
            lhs_prior = layer_to_priors[j-1][2*k]
            rhs_prior = layer_to_priors[j-1][2*k+1]

            lhs_likelihood = parent_likelihood - rhs_prior
            rhs_likelihood = parent_likelihood - lhs_prior

            lhs_likelihood =
            lhs_likelihood.narrowed_to_intersecting_support(lhs_prior)
            rhs_likelihood =
            rhs_likelihood.narrowed_to_intersecting_support(rhs_prior)
layer.append(lhs_likelihood)
layer.append(rhs_likelihood)
else:
    # this PMF came from one parent during merge step (because
    # previous layer was not divisible by 2):
    layer.append(layer_to_priors[j-1][2*k])

    # todo: adapt where not multiple of 2

layer_to_likelihoods.append(layer)
layer_to_likelihoods = layer_to_likelihoods[::-1]

# returns likelihoods X_0|D, X_1|D, ... and prior for Y
return (layer_to_likelihoods[0], layer_to_priors[-1][0])

A = PMF(3, [0.5,0,0.5])
print 'A', A
print ''

B = PMF(1, [1,0,0])
print 'B', B
print ''

C = PMF(0, [0,0.5,0.5])
print 'C', C
print ''

D = PMF(4, [0,0.333,0.0,0,0.333,0.333])
print 'D', D
print ''

print 'brute force:'
likelihoods, prior = brute_force_solve([A,B,C], D)
print 'prior D', prior
print 'likelihoods A,B,C', likelihoods
print

print 'convolution tree:'
likelihoods, prior = convolution_tree_solve([A,B,C], D)
print 'prior D', prior
print 'likelihoods A,B,C', likelihoods

Just as we performed probabilistic addition on the PMFs using FFT
(again, that is equivalent to the vector convolution performed above with
14.4. CONVOLUTION TREE

FFT), we can also perform probabilistic subtraction between random variables using FFT convolution. In this manner, the convolution tree can also be used to figure out which orders from each individual at your table yielded any allowed goal total bill \(t \in T\). That step is referred to as the “backward pass” and also has runtime \(\Theta(k \cdot n \log(k \cdot n) \log(n))\). The probabilistic version of whether the party could possibly reach an allowed total bill \(t \in T\) while a given individual also ordered a particular item is determined by the “likelihood” vector for that individual: where the likelihood vector for that individual is nonzero at the index corresponding to the price of the item, then it was possible for them to order the item and to reach a valid goal price \(t \in T\).

Importantly, the convolution tree also opens the door for us to weight the different menu items. A set \(A = \{0, 2\}\) is equivalent to the vector \(a = [True, 0, True]\), which in turn can be written as \(\text{pmf}_A = [0.5, 0, 0.5]\); however, the PMF form also allows us to weight these menu choices on a continuum: \(\text{pmf}_A = [0.9, 0, 0.1]\) would correspond to the same set \(A = \{0, 2\}\), but would indicate \(\Pr(A = 0) = 0.9\) and \(\Pr(A = 2) = 0.1\). This means we can use this form to encode information about how good a given solution would be. This will be useful for the “knapsack problem” in the next chapter.