O. Optimization Problems

The notion of an optimization problem provides a structured way to think about solving lots of computational problems. Whenever you set about solving a problem that involves finding the biggest, the smallest, the most, the fewest, the fastest, the least inexpensive, etc., there is a good chance that you can map the problem onto a classic optimization problem for which there is a known computational solution.

In general, an optimization problem has two parts

- An objective function that is to be maximized or minimized. For example, the airfare between Boston and Istanbul.
- A set of constraints (possibly empty) that must be honored. For example, an upper bound on the travel time.

In this chapter we introduce the notion of an optimization problem, and give a few examples. We also provide some simple algorithms that solve them. In the next chapter we discuss more efficient ways of solving some classes of optimization problems.

The main things to take away from this chapter are:

- The notion that many problems of real importance can be simply formulated in way that leads naturally to a computational solution,
- Reducing a seemingly new problem to an instance of a well-known problem allows one to use pre-existing solutions,
- Exhaustive enumeration algorithms provide a simple, but often computationally intractable, way to search for optimal solutions,
- A greedy algorithm is often a practical approach to finding a pretty good, but not guaranteed optimal, solution to an optimization problem, and
- Knapsack problems and graph problems are classes of problems to which other problems can often be reduced.

As usual we will supplement the material on computational thinking with a few bits of Python and some tips about programming.

O.1. Knapsack Problems

It’s not easy being a burglar. In addition to the obvious problems (making sure that a home is empty, picking locks, circumventing alarms, etc.), a burglar has to decide what to steal. The problem is that most homes contain more things of value than the average burglar can carry away. What’s a poor burglar to do? She needs to find the set of things that provides the most value without exceeding her carrying capacity.

Suppose for example, a burglar who can carry away at most twenty pounds of loot breaks into a house and finds the items in Table Items.

---

1 For those of you too young to remember, a “knapsack” is a simple bag that people used to carry on their back—long before “backpacks” became fashionable. If you happen to have been in scouting you may remember the words of the “Happy Wanderer,” “I love to go a-wandering, Along the mountain track, And as I go, I love to sing, My knapsack on my back.”
The simplest way to find an approximate solution to this problem is to use a \textbf{greedy algorithm}. The thief would choose the best item first, then the next best, and continue until she reached her limit. Of course, before doing this, the thief would have to decide what “best” should mean. Is the best item the most valuable, the least heavy, or maybe the item with the highest value/weight ratio. If she liked highest value, she would leave with just the computer, which she could fence for $200. If she liked lowest weight, she would leave with the book, painting, radio, and vase, which would be worth $170. Finally, if she decided that best meant highest value/weight, she would depart with the vase, clock, radio, and book, which would have a value of $255.

Though greedy by density yields the best result of the three greedy methods tried here, there is no guarantee that a greedy by density algorithm always finds a better solution than greedy by weight or value. More generally, there is no guarantee that any greedy solution to this kind of knapsack problem will find an optimal solution.\footnote{There is probably some deep moral lesson to be extracted from this fact. As they used to say on Wall Street, “Bulls make money, bears make money, pigs get slaughtered.”} We will discuss this issue in more detail a bit later.

The code in Figure Greedy implements all three of these greedy algorithms. We first define class \texttt{Item}. Each item has a name, value, and weight. The only interesting part is the implementation of the function \texttt{greedy}. By introducing the parameter \texttt{keyFcn}, we make the function independent of the order in which the elements of the list are to be considered. All that is required is that \texttt{keyFcn} defines an ordering on the elements in \texttt{Items}. We then use this ordering to produce a sorted list containing the same elements as \texttt{Items}. We use the built in Python function \texttt{sorted} to this. We use \texttt{sorted} rather than \texttt{sort} because we want to generate a new list rather than mutate the list passed into the function. We use the \texttt{reverse} parameter to indicate that we want the list sorted from largest (with respect to \texttt{keyFcn}) to smallest.
class Item(object):
    def __init__(self, n, v, w):
        self.name = n
        self.value = float(v)
        self.weight = float(w)
    def getName(self):
        return self.name
    def getValue(self):
        return self.value
    def getWeight(self):
        return self.weight
    def __str__(self):
        result = '<' + self.name + ', ' + str(self.value) + ', ' + str(self.weight) + '>
        return result
    def weightInverse(item):
        return 1.0/item.getWeight()
    def density(item):
        return item.getValue()/item.getWeight()
def buildItems():
    names = ['clock', 'painting', 'radio', 'vase', 'book', 'computer']
    vals = [175,90,20,50,10,200]
    weights = [10,9,4,2,1,20]
    Items = []
    for i in range(len(vals)):
        Items.append(Item(names[i], vals[i], weights[i]))
    return Items
def greedy(Items, maxWeight, keyFcn):
    assert type(Items) == list and maxWeight >= 0
    ItemsCopy = sorted(Items, key=keyFcn, reverse = True)
    result = []
    totalVal = 0.0
    totalWeight = 0.0
    i = 0
    while totalWeight < maxWeight and i < len(Items):
        if (totalWeight + ItemsCopy[i].getWeight()) <= maxWeight:
            result.append((ItemsCopy[i]))
            totalWeight += ItemsCopy[i].getWeight()
            totalVal += ItemsCopy[i].getValue()
            i += 1
    return (result, totalVal)
def testGreedy(Items, constraint, getKey):
    taken, val = greedy(Items, constraint, getKey)
    print ('Total value of items taken = ' + str(val))
    for item in taken:
        print (item)
def testGreedyss():
    Items = buildItems()
    print('Items to choose from:')
    for item in Items:
        print(item)
testGreedy(Items, 20, Item.getValue)
testGreedy(Items, 20, weightInverse)
testGreedy(Items, 20, density)

What is the algorithmic efficiency of greedy? There are two things to consider. The time complexity of the built in function sorted, and the number of times through the while loop. The number of iterations of the while loop is bounded by the number of elements in Items, i.e., it is $O(n)$, where $n$ is the length of Items. However, the worst case time for Python’s built in
The sorting function is roughly \( O(n \log n) \), where \( n \) is the length of the list to be sorted. Therefore the running time of greedy is \( O(n \log n) \).

Suppose we decide that an approximation is not good enough, i.e., we want the best possible solution to this problem. Such a solution is call optimal. And, as it happens, we have been looking at an instance of a classic optimization problem, called the 0/1 knapsack problem, for which there is a nice computational solution.

The 0/1 knapsack problem can be formalized as follows:

1. Each item is represented by a pair, \(<value, weight>\).
2. The knapsack can accommodate items with a total weight of no more than \( w \).
3. A vector, \( I \), of length \( n \), represents the set of available items. Each element of the vector is an item.
4. A vector, \( V \), of length \( n \), is used to indicate whether or not each item is taken by the burglar.
   If \( V[i] = 1 \), item \( I[i] \) is to be taken. If \( V[i] = 0 \), item \( I[i] \) is not taken.
5. Find a \( V \) that maximizes

   \[ 5.1 \sum_{i=1}^{n} V[i] \times I[i].value \]

   subject to the constraint that

   \[ 5.2 \sum_{i=0}^{n-1} V[i] \times I[i].weight \leq w. \]

Let’s see what happens if we try to implement this formulation of the problem in a straightforward way:

1. Enumerate all possible combinations of items, this is called the power set,
2. Remove all of the combinations whose weight exceeds the allowed weight,
3. From the remaining combinations choose any one whose value is at least as large as the value of the other combinations.

This approach will certainly find the right answer. However, if the original set of items is large, it will take a very long time to run, because the number of possible combinations grows quickly with the number of items.

Suppose, for example, there were two items, call them \( a \) and \( b \). The possible combinations are: \{a\}, \{b\}, and \{a,b\}. Suppose there are three items. The possible combinations are \{a\}, \{b\}, \{c\}, \{a,b\}, \{a,c\}, \{b,c\}, \{a,b,c\}.

More generally, consider starting with a list of \( n \) items. We can represent any combination of items by a vector of 0’s and 1’s. The combination containing no items would be represented by a vector of all 0’s. The combination containing all of the items would be represented by a vector of all 1’s. The combination containing only the first and last elements would be represented by 100...001. Etc. For example, if we look at four items, the possible choices are shown in the table on the right.

---

\(^3\) Python uses an algorithm invented by C.A.R. Hoare known as quick sort or heap sort. An analysis of its complexity is beyond the scope of this book. However, you won’t go too far wrong if you treat it as \( O(n \log n) \).
If one looks at only the right hand column of the table, there does not seem to be a lot of rhyme or reason behind the order in which the sets of items appear. However, if one looks at the four leftmost columns, the system behind generating the sets becomes clear. We are simply enumerating in order all 16 four digit binary numbers.

When you first learned about decimal numbers, i.e., numbers base 10, you learned that a decimal number is represented by a sequence of the digits 0123456789. The rightmost digit is the $10^0$ place, the next digit towards the left the $10^1$ place, etc. So, for example, the sequence of digits 302 represents $3 \times 100 + 0 \times 10 + 2 = 5$. How many different numbers can be represented by a sequence of length $n$? $10^n$.

Binary numbers, i.e., base 2, work similarly. A binary number is represented by a sequence of the digits 0,1. The rightmost digit is the $2^0$ place, the next digit towards the left the $2^1$ place etc. So, for example, the sequence of digits 101 represents $1 \times 4 + 0 \times 2 + 1 \times 1 = 5$. How many different numbers can be represented by a sequence of length $n$? $2^n$.

Figure BruteForce contains a straightforward implementation of this brute force approach to solving the 0/1 knapsack problem. It uses some classes and functions defined in Figure Greedy.

It begins by repeatedly calling dtoB to generate a list of strings corresponding to each of the binary numbers in range(len(Items)). These strings are then used as templates to generate a list containing every possible combination (including the empty combination) of the items in Items. chooseBest then iterates through these combinations to choose an optimal one satisfying the weight constraint.

The complexity of this implementation is $O(n \times 2^n)$, where $n$ is the length of Items. genPset returns a list of lists of items. This list is of length $2^n$, and the longest list in it is of length $n$. Therefore the outer loop in chooseBest will be executed $O(2^n)$ times and the number of times the inner loop will be executed is bounded by $n$.

Many small optimizations can be applied to speed this program up. For example, genPset could have the header def genPset(Items, constraint, getVal, getWeight), and only return those combinations that meet the weight constraint. Alternatively, chooseBest could exit the inner loop as soon as the weight constraint is exceeded. While these kinds of optimizations are often worth doing, they don’t actually address the fundamental issue. chooseBest will still be $O(2^n)$, where $n$ is the length of Items, and will still take a long time to run when Items is large.

In a theoretical sense, the problem is hopeless. The 0/1 knapsack problem is inherently exponential in the number of items. In a practical sense, however, the problem is far from hopeless, as discussed in the chapter on Dynamic Programming.
When `testBest` is run, it prints:

```
Total value of items taken = 275.0
<clock, 175.0, 10.0>
<painting, 90.0, 9.0>
<book, 10.0, 1.0>
```

Notice that this solution is better than the solution found by any of the greedy algorithms. The essence of a greedy algorithm is making the best (as defined by some metric) local choice at each step. I.e., it makes a choice that is **locally optimal**. However, as this example illustrates a series of locally optimal decisions does not always lead to a **global optimum**.
Despite the fact that they do not always find the best solution, greedy algorithms are often used in practice. They are usually easier to implement and more efficient to run than algorithms guaranteed to find optimal solutions. As Ivan Boesky once said, “Greed is all right.”

There is a variant of the knapsack problem, called the fractional (or continuous) knapsack problem, for which a greedy algorithm is guaranteed to find an optimal solution. Suppose, for example, that our burglar found only three things of value in the house: a sack of gold dust, a sack of silver dust, and a sack of sand. In this case, a greedy by density algorithm will always find the optimal solution.

0.2. Graph Optimization Problems

Let’s think about another kind of optimization problem. Suppose you had a list of all the airline flights between each pair of cities in the United States, and what each flight cost. Suppose also that for all cities, A, B, and C, the cost of flying from A to C by way of B was the cost of flying from A to B plus the cost of flying from B to C. A few questions you might like to ask are:

- What is the shortest number of hops between some pair of cities?
- What is the least expensive airfare between some pair of cities?
- What is the least expensive airfare between some pair of cities involving no more than two stops?
- What is the least expensive way to visit some collection of cites?

All of these problems (and many others) can be easily formalized as graph problems.

A graph is a set of objects called nodes (or vertices) connected by a set of edges (or arcs). If the edges are unidirectional the graph is called a directed graph or digraph. Graphs are typically used to represent situations in which there are interesting relations among the parts. The first documented use of graphs in mathematics was in 1735 when the Swiss mathematician Leonhard Euler used what has come to be known as graph theory to formulate and solve the Königsberg bridges problem.

Königsberg, then the capital of East Prussia, was built at the intersection of two rivers that contained a number of islands. The islands were connected to each other and to the mainland by seven bridges, as shown below. For some bizarre reason, the residents of the city were obsessed with the question of whether it was possible to take a walk that involved crossing each of the bridges exactly once.

Euler’s great insight was that the problem could be vastly simplified by viewing each separate land mass as a point (think node) and each bridge as a line (think edge) connecting two of these points. The map of the town could then be represented by the graph below. Euler then reasoned that if a walk were to traverse each edge exactly once, it must be the case that each node in the middle of the walk (i.e., any node except the first and last node visited) must have an even number of edges to which is connected. Since none of the nodes in this graph has an even number of edges, he concluded that it is impossible to traverse each edge exactly once.

---

4 He said this, to enthusiastic applause, in a 1986 commencement address at University of California at Berkeley Business School. A few months later he was indicted for insider trading.

5 Computer scientists and mathematicians use the word “graph” in the sense used in this book. They typically use the word “plot” to denote the kind of graphs one learns about in secondary school.
Of greater interest than the Königsberg bridges problem, or even Euler’s theorem (which generalizes his solution to the Königsberg bridges problem) is the whole idea of formulating things as graph theoretic problems.

For example, only one small extension to the kind of graph used by Euler is needed to model the United State’s Interstate Highway system. If a weight is associated with each edge in a graph (or digraph) it is called a weighted graph. Using weighted graphs, the highway system can be represented as a graph in which cities are represented by nodes and the highways connecting them as edges, where each edge is labeled with distance between the two nodes. More generally, one can represent any roadmap (including those with one way streets) by a weighted digraph.

Similarly, the structure of the World Wide Web can be represented as a digraph in which the nodes are web pages and there is an edge from node A to node B if and only if there is a link to page B on page A. Traffic patterns could be modeled by adding a weight to each edge indicating how often is it used.

There are also many less obvious uses of graphs. Biologists use graphs to model things ranging from the way proteins interact with each other to gene expression networks. Physicists use graphs to describe phase transitions. Epidemiologists use graphs to model disease trajectories. Etc.

Figure Digraph contains classes implementing nodes, edges, and digraphs.

Having a class for nodes may seem like overkill. After all, the only computation in class Node is a conversion of name to a str. We introduced the class to give us the flexibility of deciding, perhaps at some later point, to introduce a subclass of Node with additional properties. Class Edge is also quite simple.

An important design decision is the choice of data structure used to represent a digraph. One common representation is an $n \times n$ adjacency matrix, where $n$ is the number of nodes in the graph. Each entry, $M_{i,j}$ contains information (e.g., weights) about the edges connecting the pair of nodes $<i, j>$. In the simplest case, each entry is True if there is an edge from $i$ to $j$ and False otherwise.

Another common representation is an adjacency list, which we use here. Class Digraph has two instance variables. The variable nodes is a Python set containing the names of the nodes in the digraph. The connectivity of the nodes is represented using an adjacency list. The variable edges is a dict that maps each node in the digraph to a list of the children of that node.
Class `Graph` is a subclass of `Digraph`. It inherits all of the methods of `Digraph` except `addEdge`, which it overrides. This is not the most space efficient way to implement graphs, since it stores each edge twice, once for each direction in the digraph.

```python
class Node(object):
    def __init__(self, name):
        self.name = str(name)
    def getName(self):
        return self.name
    def __str__(self):
        return self.name

class Edge(object):
    def __init__(self, src, dest):
        self.src = src
        self.dest = dest
    def getSource(self):
        return self.src
    def getDestination(self):
        return self.dest
    def __str__(self):
        return str(self.src) + '-' + str(self.dest)

class Digraph(object):
    def __init__(self):
        self.nodes = set([])
        self.edges = {}
    def addNode(self, node):
        if node in self.nodes:
            raise ValueError('Duplicate node')
        else:
            self.nodes.add(node)
            self.edges[node] = []
    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not(src in self.nodes and dest in self.nodes):
            raise ValueError('Node not in graph')
        self.edges[src].append(dest)
    def childrenOf(self, node):
        return self.edges[node]
    def hasNode(self, node):
        return node in self.nodes
    def __str__(self):
        res = ''
        for k in self.edges:
            for d in self.edges[k]:
                res = res + str(k) + '->' + str(d) + 'n'
        return res[:-1]

class Graph(Digraph):
    def addEdge(self, edge):
        Digraph.addEdge(self, edge)
        rev = Edge(edge.getDestination(), edge.getSource())
        Digraph.addEdge(self, rev)

Figure Graph
```

One of the nice things about formulating a problem using graph theory is that there are well known algorithms for solving many optimization problems on graphs. Some of the best known graph optimization problems are:

- **Shortest path.** For some pair of nodes, $N_1$ and $N_2$, find the shortest sequence of edges $<n_s, n_d>$ (source node and destination node), such that
• The source node in the first edge is N1
• The destination node of the last edge is N2
• For all \( i \) and \( j \), if \( e_j \) follows \( e_i \) in the sequence, the source node if \( e_j \) is the destination node of \( e_i \).

- **Shortest weighted path.** This is like the shortest path, except instead of choosing the shortest sequence of edges that connects the two nodes, we define some function on the weights of the edges in the sequence (e.g., their sum) and minimize that value. This is the kind of problem solved by Mapquest and Google Maps when asked to compute driving directions between two points.

- **Cliques.** Finds sets nodes such that there is a path (or a path with a maximum length) in the digraph between each pair of nodes in the set.  

- **Min cut.** Given two sets of nodes in a graph, a cut is a set of edges whose removal eliminates all paths from a node in one set to a node in the other. The minimum cut is the smallest set of edges whose removal accomplishes this.

### O.2.1. An Example of Applying Graph Theory

Figure TB\(^7\) contains a pictorial representation of a weighted graph generated by the Center for Disease Control (CDC) in 2003 in the course of studying an outbreak of tuberculosis in the United States. Each node represents a person, and each node is labeled by a color indicating whether the person has active TB, tested positive for exposure to TB (i.e., high TST reaction rate), tested negative for exposure to TB, or had not been tested. The edges represent connectivity among pairs of people. The weights, which are not visible in the picture, indicate whether the contact between people was “close” or “casual.”

![Figure TB](http://www.orgnet.com/TB_web.ppt)

---

\(^6\) This notion is quite similar to the notion of a social clique, i.e., a group of people who feel closely connected to each other and inclined to exclude those not in the clique. See, for example, the movie “Heathers.”

\(^7\) [http://www.orgnet.com/TB_web.ppt](http://www.orgnet.com/TB_web.ppt)
There are many interesting questions that can be formalized using this graph. For example,

- Is it possible that all cases stemmed from a single index patient? More formally, is there a node, \( n \), such that there is a path from \( n \) to every other node in the graph with a positive for TB label. The answer is “almost.” There is path from the node in the middle of the graph to each node except those nodes in the black circle on the right. Interestingly, subsequent investigation revealed that the person in the center of the black circle had previously been a neighbor of the index patient, and therefore there should have been a casual contact edge linking the two.

- In order to best limit the continued spread, which uninfected people should be vaccinated? This can be formalized as solving a min cut problem. Let \( N_I \) be the set of infected nodes and \( N_U \) be the set of uninfected nodes. Each edge in the minimum cut between these two sets will contain one infected and one uninfected person. The uninfected people are good candidates for vaccination.

### O.2.2. Another graph theory example

Social networks are made up of individuals and relationships between individuals. These are typically modeled as graphs in which the individuals are nodes and the edges representations. If the relationships are symmetric, the edges are undirected; if the relationships are asymmetric (e.g., parent of) the edges are directed. Some social networks model multiple relationships, in which case labels on the edges indicate the kind of relationship.

In 1990, the playwright John Guare wrote “Six Degrees of Separation.” The slightly dubious premise underlying the play is that, “everybody on this planet is separated by only six other people.” By this he meant that if one built a social network including every person on the earth using the relation “knows,” the shortest path between any two individuals involved passing through at most six other nodes.

A less hypothetical question is the distance using the “friend” relation between pairs of people on Facebook. For example, you might wonder if you have a friend who has a friend who has a friend who is a friend of Barack Obama. Let’s think about designing a program to answer such questions.

The friend relation (at least on Facebook) is symmetric, e.g., if Tom is a friend of Bertha, Bertha is a friend of Tom. We will, therefore, implement the social network using type Graph. We can then define the problem of finding the shortest connection between You and Barack Obama as,

- For the graph \( G \), find the shortest sequence of nodes, \( \text{path} = \{\text{You}, \ldots, \text{Barack Obama}\} \), such that

  - If \( n_i \) and \( n_{i+1} \) are consecutive nodes in \( \text{path} \), there is an edge in \( G \) connecting \( n_i \) and \( n_{i+1} \).

Figure ShortestPath contains a recursive function that finds the shortest path between two nodes in a Digraph. Since Graph is a special case of Digraph, it will work for our Facebook problem. The function findPath checks whether or not the nodes start and end are in graph. If so, it calls the recursive function shortestPath with appropriate initial values.

The algorithm implemented by shortestPath is an example of a recursive depth first search algorithm.

- The recursion terminates when \( \text{start} == \text{end} \).
• The recursive part begins by choosing one child of \texttt{start}. It then chooses one child of that node and so on, until either it reaches the node \texttt{end} or a node with no unvisited children.
  
  o The check \texttt{if str(node) not in path:}, prevents the program from getting caught in a cycle.
  
  o The check
    \begin{verbatim}
    if (shortest == None or len(path) < len(shortest)):
    \end{verbatim}
    is used to decide if it is possible that continuing to search this path might yield a shorter path than the best found so far.
  
  o If so, \texttt{shortestPath} is called recursively. If it finds a path to \texttt{end} that is shorter than the best found so far, \texttt{shortest} is updated.
  
  o When the last node on \texttt{path} has no children left to visit, the program \texttt{backtracks} to the previously visited node and visits the next child of that node.

• The function returns when all possible paths from \texttt{start} to \texttt{end} have been explored.

```python
from Graph import *

def shortestPath(graph, start, end, path, shortest):
    global numCalls
    numCalls += 1
    path = path + [str(start)]
    if start == end:
        return path
    for node in graph.childrenOf(start):
        if (str(node) not in path):
            if (shortest == None or len(path) < len(shortest)):
                newPath = shortestPath(graph, node, end, path, shortest)
                if newPath != None:
                    shortest = newPath
    return shortest

def findPath(graph, start, end):
    if not (graph.hasNode(start) and graph.hasNode(end)):
        raise ValueError('start or end not in graph')
    return shortestPath(graph, start, end, [], None)

Figure ShortestPath
```
D. Dynamic Programming

Dynamic programming was invented Richard Bellman in the early 1950’s. Don’t try and infer anything about the technique from its name. It was chosen to hide from governmental sponsors, “the fact that I was really doing mathematics… It was something not even a Congressman could object to.”

Dynamic programming is a method for efficiently solving problems that exhibit the characteristics of overlapping sub-problems and optimal substructure. Fortunately, many optimization problems exhibit these characteristics.

A problem has optimal substructure if a globally optimal solution can be found by combining solutions to local sub-problems. We’ve already looked at a number of such problems. Merge sort, for example, exploits the fact that a list can be sorted by first sorting sub-lists, and then merging the solutions. A problem has overlapping sub-problems, if an optimal solution involves solving the same problem multiple times. Again, sorting clearly exhibits this property.

It’s not immediately obvious, but the 0/1 knapsack problem exhibits both of these properties. Before looking at that, however, we will digress to look at a problem where the optimal substructure and overlapping sub-problems are more obvious.

D.1. Fibonacci Sequences

“They breed like rabbits,” has long been used to describe a population that the speaker thinks is growing too quickly. In the year 1202, the mathematician known as Fibonacci developed a formula designed to quantify this notion, albeit with some not terribly realistic assumptions.

Suppose a newly born pair of rabbits, one male and one female, are put in a pen. Suppose further that rabbits are able to mate at the age of one month and have a one month gestation period. Finally, suppose that these mythical rabbits never die, and that the female always produces one new pair (one male, one female) every month from its second month on. How many female rabbits will there be at the end of one year?

At the end of the first month (call it month 0), there will be one female. At the end of the second month, there will still be only female (now pregnant). At the end of the third month, there will now be two females, one of which is pregnant and one of which is not. Let’s look at this in tabular form.

<table>
<thead>
<tr>
<th>Month</th>
<th>Females</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
</tr>
</tbody>
</table>

Notice that in this table, for n > 1, \( \text{females}(n) = \text{females}(n-1) + \text{females}(n-2) \). This is not an accident. Each female that was alive in month \( n-2 \) will produce one female in month \( n \). The new females can be added to the females alive in month \( n-1 \) to get the females in month \( n \). The growth in population is described naturally by the recurrence:

\[
\text{females}(0) = 1 \\
\text{females}(1) = 1 \\
\text{females}(n + 2) = \text{females}(n-1) + \text{females}(n)
\]

Figure RecursiveFib contains a straightforward implementation of the Fibonacci recurrence.

---13---

Not surprisingly, given our rather unrealistic assumptions, the Fibonacci sequence does not actually provide a perfect model of the growth of rabbit populations. It does, however, have many interesting properties. For example, as $n$ approaches infinity, $\frac{\text{fib}(n)}{\text{fib}(n-1)}$ approaches the golden ratio (also known as the golden mean and as the divine ratio), a number that plays an important role in many geometric problems. Fibonacci numbers are also quite common in nature. For example, the number of petals on most flowers is a Fibonacci number. Black-eyed susans have 13 petals and most field daisies have 34 petals.

While the straightforward recursive implementation of the recurrence is correct, it is terribly inefficient. Try, for example, running $\text{fib}(120)$, but don’t wait for it to complete. The complexity of the above implementation is a bit hard to derive, but it is roughly $O(\text{fib}(n))$. I.e., its growth is proportional to the growth in the value of the result, and the growth rate of the Fibonacci sequence is substantial. For example, the 120th number in the sequence is 5,358,359,254,990,966,640,871,840.

Let’s try and figure out why this implementation takes so long. Given the tiny amount of code in the body of $\text{fib}$, it’s clear that the problem must be the number of times that $\text{fib}$ calls itself. As an example, look at the tree of calls that follows the invocation of $\text{fib}(6)$:

---

9 In 1859 Thomas Austin, an Australian farmer, imported twenty-four rabbits from England, to be used as targets in hunts. Ten years later, on the order of two million rabbits a year were shot or trapped in Australia, with no noticeable impact on the population. That’s a lot of rabbits, but not anywhere close to $\text{fib}(120)$.

10 If you are feeling especially geeky, try writing a Fibonacci poem. This is a form of poetry in which the number of syllables in each line is equal to the total number of syllables in the previous two lines. Think of the first line (which has zero syllables) as a place to take a deep breath before starting to read your poem.
Notice that we are computing the same values over and over again. For example \( \text{fib} \) gets called with 3 times, and each of these calls provokes four additional calls of \( \text{fib} \). It doesn’t require genius to think that it might be a good idea to remember the value returned by the first call and look it up rather than compute it each time it is needed. This is called **memoization**.

Figure FastFib contains an implementation if Fibonacci based on this idea. If you try running it, you will see that it is indeed quite fast, e.g., \( \text{fib}(120) \) returns almost instantly.

```python
def fastFib(x, memo):
    assert type(x) == int and x >= 0 and type(memo) == dict
    if x == 0 or x == 1:
        return 1
    if x in memo:
        return memo[x]
    result = fastFib(x-1, memo) + fastFib(x-2, memo)
    memo[x] = result
    return result

def testFastFib(n):
    assert type(n) == int and n >= 0
    for i in range(n):
        print ('fib of', i, '=', fastFib(i, {}))
```

**Figure FastFib**

What is the complexity of \( \text{fastFib} \)? It computes \( \text{fib}(n) \) exactly once for each value of \( 0 \leq n < x \). Therefore, under the assumption that dictionary lookup can be done in constant time, the time complexity of \( \text{fastFib}(x) \) is \( O(x) \).\(^{11}\)

### D.2. Dynamic Programming and The 0/1 Knapsack Problem

One of the optimization problems we looked at in the last chapter was the 0/1 knapsack problem. Recall that we looked at a greedy algorithm that ran in linear time, but was not guaranteed to find

\[^{11}\text{Though cute and pedagogically interesting, this is probably not the best way to implement Fibonacci. There is a simple linear time iterative implementation.}\]
an optimal solution. We also looked at a brute force algorithm that was guaranteed to find an
optimal solution, but ran in exponential time. Finally, we discussed the fact that problem is
inherently exponential in the size of the input. In the worst case, one could not find an optimal
solution without looking at all possible answers.

Fortunately, the situation is not as bad as it seems. Dynamic programming provides a practical
method for solving knapsack problems in a reasonable amount of time. As a first step in deriving
such a solution, we begin with a different approach to constructing an exponential solution based
on exhaustive enumeration. The key idea is to think about exploring the space of possible
solutions by constructing a rooted binary tree that enumerates all states that satisfy the weight
constraint.

A **rooted binary tree** is an acyclic directed graph in which:

- There is exactly one node (the root) with no parents,
- Each non-root node has exactly one parent, and
- Each node has at most two children. A childless node is called a **leaf**.

Each node in this tree is labeled with a quadruple that denotes a partial solution to the knapsack
problem:

1. A (perhaps partial) set of items to be taken.
2. The list of items for which a decision about whether or not to take each item in the list
   has not been made.
3. The total value of the items in the set of items to be taken. This is merely an
   optimization, since the value could be computed from the set.
4. The remaining space in the knapsack. Again, this is an optimization since this is merely
   the difference between the weight allowed and the weight of all the items taken so far.

The tree is built top down starting with the root.\[^{12}\] One element is selected from the still to be
considered items. If there is room for that item in the knapsack, a node is constructed that reflects
the consequence of choosing to take that item. By convention, we draw that as the left child. The
right child shows the consequences of choosing not to take that item. The process is then applied
recursively to non-leaf children. Because each edge represents a decision (to take or not to take
an item), such trees are called **decision trees**.\[^{13}\]

For example, here is a table describing a set of items and a decision tree for that set of items and a
knapsack with a maximum weight of five.

<table>
<thead>
<tr>
<th>Item</th>
<th>Value</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>d</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

The root of the tree has a label `<{ }, [a, b, c, d], 0, 5>`, indicating that no items have been taken, all items remain to be considered, the value of
the items taken is 0, and a weight of 5 is still available. Notice that for each leaf node, either the
second element is the empty list (indicating that there are no more items to consider taking) or the
fourth element is 0 (indicating that there is no room left in the knapsack).

---

\[^{12}\] It may seem odd to put the root of a tree at the top, but that is the way they are usually drawn. Perhaps it is evidence that computer scientists do not spend enough time contemplating nature.

\[^{13}\] Decision trees, which need not be binary, provide a structured way to explore the consequences of making a series of decisions. They are used extensively in many fields.
In Figure Decision Tree, the numbers above each node in the decision tree indicate one order in which the nodes could be generated. This particular ordering is called left-first depth first. At each node we attempt to generate a left node. If that is impossible, we attempt to generate a right node. If that too is impossible, we back up one node (to the parent) and repeat the process. Eventually, we find ourselves having generated both children of the root, and the process halts. When the process halts each reachable state has been generated, and a leaf node with the greatest value represents an optimal solution.

Unsurprisingly, the natural implementation of a tree search is recursive. maxVal returns two values, the set of items chosen and the total value of those items. It is called with four arguments corresponding to the labels of the nodes in the tree.

- **toTake.** Those items that nodes higher up in the tree (which corresponds to earlier calls in the recursive call stack) have decided to take.
- **toConsider.** Those items that nodes higher up in the tree have not yet considered.
- **val.** The total value of the items in toTake.
- **avail.** The amount of space available for the items in toConsider.
If you run this on any of the examples we have looked at you will find that it produces an optimal answer. In fact, it will always produce an optimal answer, if it get around to producing any answer at all.

The code in Figure TestMaxVal is a test harness that allows us to test maxVal on a large number of examples. It randomly generates a list of Items of a specified size. Try bigTest(10).

Now try bigTest(30). After you get tired of waiting for it to return, hit control c (or whatever you need to do to kill the program), and ask yourself what is going on.

Let’s think about the size of the tree we are exploring. Since at each level of the tree we are deciding to keep or not keep one item, the maximum depth of the tree is \( \text{len}(\text{items}) \). At level 0 we have only one node, at level 1 up to two nodes, at level 2 up to four nodes, at level 3 up to eight nodes. At level 29 we have up to \( 2^{29} \) nodes. No wonder it takes a long time to run!

```python
def maxVal(toTake, toConsider, val, avail):
    if toConsider == [] or avail == 0:
        result = (val, toTake)
    elif toConsider[0].getWeight() > avail:
        result = maxVal(toTake, toConsider[1:], val, avail)
    else:
        nextItem = toConsider[0]
        # Explore left branch
        withVal, withToTake = maxVal(toTake + (nextItem,), toConsider[1:],
                                      val + nextItem.getValue(),
                                      avail - nextItem.getWeight())
        # Explore right branch
        withoutVal, withoutToTake = maxVal(toTake, toConsider[1:],
                                                 val, avail)
        # Choose better branch
        if withVal > withoutVal:
            result = (withVal, withToTake)
        else:
            result = (withoutVal, withoutToTake)
    return result

def smallTest():
    Items = buildItems()
    val, taken = maxVal((), Items, 0, 20)
    for item in taken:
        print(item)
    print ('Total value of items taken = ' + str(val))

Figure MaxVal
```
What should we do about this? Let’s start by asking whether this program has anything in common with our first implementation of Fibonacci. In particular, is there optimal substructure and are there overlapping sub-problems?

Optimal substructure is visible in both Figure DecisionTree and in the code in Figure MaxVal. Each parent node combines the solutions reached by its children to derive an optimal solution for the sub-tree rooted at that parent. This is reflected in Figure MaxVal by the code following the comment #Choose better branch.

Are there also overlapping sub-problems? At first glance, the answer seems to be no. At each level of the tree we have a different set of available items to consider. This implies that if common sub-problems exist they must be at the same level of the tree. And indeed at each level of the tree each node has the same set of items to consider taking. However, we can see by looking at the labels that each node at a level represents a different set of choice about the items considered higher in the tree.

But think about what problem is being solved at each node. The problem being solved is finding the optimal items to take from those left to consider, given the remaining available weight. The available weight depends upon the total weight of the items in toTake, but not on which items are in toTake or even the total value of the items in toTake. So, for example, in Figure DecisionTree, nodes 3 and 9 are actually solving the same problem, deciding which subset of \{c,d\} should be added to toTake given that the available weight is 2.

The code in Figure FastMaxVal exploits the optimal substructure and overlapping sub-problems to provide a dynamic programming solution to the 0/1 knapsack problem. An extra parameter, memo, has been added to keep track solutions to sub-problems that have already been solved. It is implemented using a dict with a key constructed from the length of toConsider and the available weight. len(toConsider) is a compact way of representing the items still to be considered. This works because items are always removed from the same end (the front) of toConsider. The global variable numCalls is not part of the solution to the problem. It is there to make it easy examine how the length of the computation grows with the length of Items.
Since the members of Items are generated randomly, the number of calls will vary from run to run. Table FastMaxValPerformance shows the number of calls made during one representative run. The growth is hard to quantify, but it is clearly far less than exponential. But how can this be since we know that the 0/1 knapsack problem is inherently exponential in the number of items? Have we found a way to overturn fundamental laws of the universe? No, but we have discovered that computational complexity can be a subtle notion.  

---

14 OK, “discovered” may be too strong a word. People have known this for a long time.
The running time of `fastMaxVal` is governed by the number of distinct `<toConsider, avail>` pairs generated. This is because the decision about what to do next depends only upon the items still available and the total weight of the items already taken. The number of possible values of `toConsider` is `len(Items)`.

The number of possible values of `avail` is more difficult to characterize. It is bounded from above by the maximum number of items that the knapsack can hold. If the knapsack can hold at most `n` values, `avail` can take on at most $2^n$ different values. This could be a rather large number. However, in practice, it is not usually so large. If the weights of the items are chosen from a reasonably small set of possible weights, many sets of items will have the same total weight, greatly reducing the running time.

This algorithm falls into a complexity class called **pseudo polynomial**. A careful explanation of this concept is beyond the scope of this book. Roughly speaking, `fastMaxVal` is exponential in the number of bits needed to represent the possible values of `avail`.

To see what happens when the weights are chosen from a considerably larger space, change `maxWeight` to `1000`.

To see what happens when the weights are chosen from an enormous space, we can choose the possible states from the positive reals rather than the positive integers. To do this, replace the line:

```
Items.append(Item(str(i), random.randrange(1, maxVal),
                      random.randrange(1, maxWeight)))
```

by the line

```
Items.append(Item(str(i),random.randrange(1,maxVal),
                   random.randrange(1,maxWeight)*random.random()))
```

Don’t hold your breadth waiting for this last test to finish. Dynamic programming may be a miraculous technique in the original sense of the word, but it is not capable of performing miracles in the liturgical sense.