Chapter 39 – Heuristics

1. Polynomial, Non-Polynomial, NP-Complete
   In the strictest sense, algorithms are solutions to problems. This book has presented problems that have algorithmic solutions that grow at $O(n)$, $O(n \log_2 n)$, $O(n^2)$ and $O(n^3)$ in time. These algorithms are said to be polynomial time algorithms because the curve that describes their grown in time is generated by a polynomial expression. There are problems that only have known algorithmic solutions that grow in time at the rates of $O(n!)$, $O(2^n)$, and even $O(n^n)$. These algorithms are know as non-polynomial (or just NP) time algorithms.

   Needless to say, non-polynomial time solutions are to be avoided. Unfortunately, not all non-polynomial time solutions have known polynomial time equivalents. Worse, it is not possible to create a polynomial time solution for some problems. These problems and their algorithms are known non-polynomial time complete (or just NP-Complete).

2. Heuristic Algorithms
   A heuristic algorithm is an algorithm that using a strategy that does not examine all possible solutions to a problem. Heuristic algorithms make no attempt to find the perfect solution to the problem. Instead, heuristic algorithms look for a "good enough" solution in an acceptable amount of time. A heuristic algorithm is one that will provide a solution close to the optimal, but may or may not be optimal. The concept of heuristic solutions to problems normally solved via non-polynomial time algorithms has changed the way programmers regard NP and NP-Complete problems.

3. Maximizing Profit or Minimizing Cost
   Classic problems in computer science are usually stated in terms of a story. One such problem is that of maximizing profit or minimizing cost. For example, what units should be loaded into a shipping container in order to maximize profits at the destination of the container, given that there are more units to send then can be sent. Since this problem comes up in a variety of situations, this problem has been universally described as the Knapsack problem. This is described as follows:

   **The Knapsack Problem**

   *A thief breaks into a home. Wishing to maximize his take, the thief must find the combination of objects that will fit in his knapsack and give him the greatest profit. The time it takes to produce this solution is critical, as the owners are liable to show up and call the police. For this reason, the thief cannot try every possible arrangement of objects in his knapsack and compare the results. The absolute best solution would be nice, but a "good enough" solution, where the thief makes a profit close to the best possible, would do just as well, especially if choosing a "good enough" solution means that he doesn't run out of time.*
The Heuristic Solution to the Knapsack Problem

The heuristic solution to the knapsack problem is fairly straightforward. First, compute the profit / cost ratio for each object (time $O(n)$). Next, sort the objects into descending order on the profit / cost ratio using a divide and conquer sort (time $O(n \log n)$). Lastly, put the object with the largest profit / cost ratio into the knapsack, then the object with the next largest profit into the knapsack, etc. until the knapsack is full [sum of the costs (or size or weights, etc.) reaches maximum].

4. The Rational Knapsack

In the rational knapsack version of the knapsack problem, the objects that the thief must choose from can be subdivided into whatever fractions are needed. Since the knapsack can be filled with fractional units of objects, the knapsack can always be filled exactly. Even if the very first object costs more than the knapsack can accommodate, the knapsack can be filled with a fraction of it. When the rational knapsack is full (sum of the costs reaches maximum), the best possible solution has been found. The rational knapsack always returns the best possible solution.

Programming Exercise 39.1

Simulate the rational knapsack version of the knapsack problem with the following constraints.

1. maximum cost (size of knapsack) is 20
2. there are 100 objects to choose from, with randomly assigned profits between 10 and 90 and randomly assigned cost between 1 and 10

At the end of the simulation, output the contents of the knapsack, the total cost of the objects in the knapsack and the total profit of the objects in the knapsack.

5. The O/1 Knapsack

In the 0/1 knapsack version of the knapsack problem, the objects that the thief must choose from cannot be subdivided. While it is possible for the heuristic solution to the 0/1 knapsack to produce the optimal solution, it usually does not. The 0/1 knapsack may leave space in the knapsack.

Programming Exercise 39.2

Alter the program from programming exercise 39.1 so that a 0/1 knapsack is used.
**Programming Exercise 39.3**

The 0/1 knapsack often produces a solution that does not fill the knapsack because the last object considered may cost more than the space that remains in the knapsack. When this happens, a simple improvement to the 0/1 knapsack solution is to continue to search for an object that will fit in the remaining space. This process can be continued until the knapsack is full or until there are no more objects.

Add this improvement to the program of 39.2.

**Programming Exercise 39.4**

Even with the improvement used in 39.3, the 0/1 knapsack solution can often be improved using what is known as the *Monte Carlo* technique. In the Monte Carlo improvement, objects (usually one, two or three in a small knapsack) are removed from the knapsack via random selection. The knapsack is then filled with objects from the previously unselected list. If this version of the knapsack is an improvement over the original knapsack, this solution is kept. If this version of the knapsack is not an improvement over the original knapsack, the Monte Carlo version is discarded. (A process similar to finding a maximum or minimum value.) This process is usually repeated 20 to 50 times to see if an improvement to the original knapsack can be found.

Add the Monte Carlo improvement to the program of 39.3.