2021

The "Knapsack Problem" Workbook: An Exploration of Topics in Computer Science

Steven Cosares
CUNY La Guardia Community College

The "Knapsack Problem" Workbook

An Exploration of Topics in Computer Science

Steven Cosares, LaGuardia Community College

The "Knapsack Problem" Workbook:
An Exploration of Topics in Computer Science

Steven Cosares, LaGuardia Community College

This workbook provides discussions, programming assignments, projects, and class exercises revolving around the "Knapsack Problem" (KP), which is widely a recognized model that is taught within a typical Computer Science curriculum. Throughout these discussions, we use KP to introduce or review topics found in courses covering topics in Discrete Mathematics, Mathematical Programming, Data Structures, Algorithms, Computational Complexity, etc. Because of the broad range of subjects discussed, this workbook and the accompanying spreadsheet files might be used as part of some CS capstone experience. Otherwise, we recommend that individual sections be used, as needed, for exercises relevant to a course in the major sequence. Each section, save for the Introduction, is written so that it can be presented independently of any other.

Spreadsheet files accompanying the document are included to illustrate topics presented in the sections. A compressed file containing the complete set can be found at:

https://www.dropbox.com/s/bu3xul5xdaynjif/Knapsack_CS_Workbook.zip?dl=0

Introduction: The Knapsack Problem Model

This section introduces the Knapsack Problem model (KP), which seeks to find the most beneficial subset of n items to be placed into a sack having capacity B. We present its most fundamental version, its parameters, and some of its variants. A simple "greedy" approach is described that identifies at least one feasible solution to KP that is "maximal", but not necessarily "maximum", i.e., optimal. Students are invited to find solutions to some sample problem instances and problem variants.

An Optimal Solution to the Knapsack Problem based on Recursion

This section covers enumerative approaches to finding an optimal solution to KP. It establishes a recursive "Dynamic Programming" formulation that may be applied to relatively small problem instances.

Heuristic Approaches to the Knapsack Problem

This section discusses efficient algorithmic approaches, that are designed to find reasonable solutions to KP, without guaranteeing that the solutions will be optimal, or even near-optimal. Students are invited to use their own knowledge of the problem and its structure to implement these approaches in a way that they believe will find good solutions quickly, when compared to some baseline approaches, like greedy algorithms or generating solutions at random.

Genetic Algorithms for the Knapsack Problem

When formulated as a "0-1 Mathematical Programming" problem, the Knapsack Problem and its variants seek to find an optimal bitstring of length n. By modeling feasible bitstrings as genetic code, we could describe finding "strong" solutions, or the optimal solution to KP, as an evolutionary process. Solutions are paired off to create "offspring" solutions. Some random elements are placed into the process, so that a specific pair of parents would not always produce the same offspring. Strong parents may (or may not) produce strong offspring. Occasional genetic mutations arise, some of which add strength to subsequent generations. If we seek to avoid complete enumeration of the bitstrings, we heuristically allow some members of the genetic pool to "die", i.e., to no longer generate offspring. Students are invited to supply their own approaches to implementing the details associated with this powerful optimization procedure.

Generating Knapsack Solutions with Java

In order to implement the algorithms described in this document using an Object-Oriented programming language like Java or C++, it is necessary to define appropriate classes and (abstract) data structures to house the relevant data and methods. When solution algorithms involve some form of enumeration, size limits on the problem instances or on the set of candidate solutions must be put into place. A method is described that encodes potential KP solutions as a single integer variable in order to save storage space. Students are invited to use their own development environment to assemble the code provided to generate "one-pass", "greedy" and "recursive" solutions to KP. The framework will allow them to add their own solutions methods.

Introduction: The Knapsack Problem Model

Accompanying File: https://www.dropbox.com/s/cy1skp4pfz2zezu/Knapsack_Intro.xlsm?dl=0

The Knapsack Problem (KP) tells a story about a person who is preparing for a long hike into the woods and must decide which items she wishes to carry with her in a knapsack. Clearly there are limits to how much she can carry, e.g., dictated by the total capacity of the knapsack. So, some potentially useful items might have to be left home; the hiker must select those items that provide the greatest total value without exceeding the size limit of the knapsack.

This story is used as a representation or "model" for a wide variety of similar problem-solving situations, e.g., in production, or in resource management, or in task scheduling, where someone must select the best subset from a list of items, subject to some limitation. The term "Knapsack Problem" is used as a label for the most fundamental version of this situation, which has specific rules regarding the problem parameters.

In the fundamental KP, there are n items under consideration. Each item has a fixed value representing the amount of benefit it provides and another value representing the amount of space it takes up in the knapsack, which can be thought of as the cost of including the item. An item can be either included or excluded. It is not possible to include a fraction of an item, nor can more than one copy of an item be selected. Items are selected independently from one another, i.e., the decision to include item A does not affect the benefit or cost value for item B. Item B is not automatically included or excluded based to the decision made about item A. (Item independence is important because the decisions can be taken in any order, even though this is not common in many "real-life" instances of the problem. For example, one might imagine that if the hiker decides to exclude shoes, she might find increased benefit from including socks!)

Example

Astro Investment company has $1,250,000 to available for some short-term prospective projects, listed below. Each project comes with the required investment amount and an expected profit. The bank wants to make as much profit as possible, but it cannot invest in every project. Which subset of these projects do you believe the company should choose?

Project	Investment Requirement	Expected Profit
A	$600,000	$170,000
B	$250,000	$125,000
C	$750,000	$200,000
D	$200,000	$150,000
E	$250,000	$90,000
F	$100,000	$70,000

It should be evident that this is a version of the KP, even though the context has nothing to do with camping. Participation in a project is all or nothing - it cannot be subdivided. The $1,250,000 budget limit

is analogous to the knapsack capacity; the total investment costs cannot exceed this value. It is not possible to engage in any project more than once; the decision to include or exclude a project does not affect the cost or profit of any other project.

In order to solve this problem, we would have to identify the particular subset of {A, B, …, F} whose total expected profit is as large as possible, but whose total required investment is less than or equal to \$1,250,000. A "brute force" algorithm would examine every one of the $2^6 = 64$ subsets to find the "optimal solution", i.e., the best of all possibilities. While this might be a reasonable approach for the small problem instance above, we point out that realistic versions of KP often have hundreds of items to consider. The number of subsets would be so large that it would take the fastest computer hundreds of years to find the optimal one.

Accompanying this document is an Excel spreadsheet file labeled, "Knapsack_Intro". In the sheet labeled, "Astro Solution" you will see the results from the brute force algorithm for the above instance. The best solution is located in row 61. It provides a total expected profit of \$515,000, by including projects A, B, D, and F, (as indicated by the variable value of 1 for these items). The total cost is \$1,150,000, which is within the budget.

You can use the sheet labeled, "Small Solver" to solve additional small instances of KP. The limit on the number of items is ten. If your problem instance has fewer than ten items to consider, place a value of 0 and a cost of 99999 in the slots for the unneeded items. Those "dummy" items would never be included in an optimal solution.

Exercise: Use the Small Solver to determine how much expected profit Project C would have to generate for it to be included in the optimal solution. What project(s) did it replace in your new solution?

Knapsack Problem Formulation

A problem formulation uses mathematical notation to describe the model. The variables $x_1, x_2, \ldots, x_n$ represent the decisions made about the items. Variable $x_i = 1$ if item i is included and 0 if it is excluded. We call the following an "Integer Programming" formulation because the variables are not allowed to take on fractional values. For the problem parameters, we let $b_1, b_2, \ldots, b_n$ represent the potential benefits for including the items and $c_1, c_2, \ldots, c_n$ to represent the costs. The parameter B represents the capacity of the knapsack, (which we often refer to as a budget constraint). The objective in the problem is to:

$$\begin{aligned} &\textit{Objective:} \quad \textit{Maximize} \quad b_1 x_1 + b_2 x_2 + b_3 x_3 + \ldots + b_n x_n \\ &\textit{subject to:} \qquad\qquad\quad c_1 x_1 + c_2 x_2 + c_3 x_3 + \ldots + c_n x_n \le B \\ &\qquad\qquad\qquad\qquad 0 \le x_i \le 1 \text{ and integer} \end{aligned}$$

One potential benefit from formulating the problem as an integer program is that it might be solved through the use of some integer programming application that is designed for more general problems, e.g., with multiple (capacity) constraints and variables allowed to take on more values than {0, 1}.

We use the term "Black Box" to represent a solution tool that takes, as input, the required parameters that define an instance of the problem, e.g., for KP, the number of variables, the knapsack capacity B, and the values for **b** and **c**. It produces, as output, the optimal solution to the problem. The box is black because

we cannot peek in to see how it does its job. Many times, when writing a computer program, we use a subroutine provided by someone else as a black box. The "applications programing interface" describes the data and format required as input and produced as output.

We should not expect a black box for Integer Programming to provide the most efficient way to solve a Knapsack Problem because it would likely not be designed to take advantage of the special properties associated with KP, like having only one constraint. However, there are likely to be situations in which this ready-to-use solution would be preferable to programming new routines from scratch.

Greedy Heuristic Algorithms

The "complexity" of a problem is a measure of its difficulty, relative to other model problems that we study in Computer Science. If we were to count the number of steps in an algorithm, (e.g., the additions, subtractions, multiplications, comparisons, square roots, etc.), to solve *any* problem instance of a particular size, we are likely to derive some (exponential, logarithmic, or polynomial) function in the size, s. The function is expressed using "Big-O" notation, (meaning Order of magnitude), that describes the shape of the function, (e.g., parabolic or $O(s^2)$, polynomial or $O(s^k)$ for some finite k, exponential or $O(b^s)$ for some b >1, or mixed like O(s log s)). Enumerative algorithms, e.g., examining every subset of a set, usually require an exponential number of steps, which is thought of as very inefficient, because they do not scale well for larger instances. There is greater interest in finding a polynomial algorithm to solve a class of problems, if possible.

According to the current beliefs regarding Computational Complexity, an important field in CS, there is no known polynomial algorithm to find the optimal solution to a general instance of KP. So, what do we do if we encounter a large problem instance of KP, where enumeration is not a reasonable option?

Answer 1: Try to develop an optimization method that is quicker than brute-force enumeration. There are some approaches that might provide performance improvements, but none that are currently guaranteed to do so in polynomial time for every possible instance. However, the problem instance under consideration may have special features that allow it to be solved to optimality relatively quickly.

Answer 2: Develop an efficient method to find a "good enough" solution, but maybe not the optimal one. We call such methods "heuristic algorithms". By applying common sense and some mathematical or logical study about the problem, we believe such a method could quickly identify a solution that is somewhat close to optimal. Over time, these methods cold be refined to provide improved performance and/or better solutions.

For the Knapsack Problem, we want the most total benefit for the least total cost. So, it would stand to reason that one would want to include into the knapsack the item with the highest benefit/cost index. (This index is sometimes referred to as "marginal value" because it measures the number of units of benefit per unit of cost. It is also called "bang for the buck" in more informal circles). If there is room after this first item is included, then it would make sense to try to fit the item with the second largest index, and so on. We use the term "greedy heuristic" to describe an algorithm like this, that orders items according to some criteria and in one pass makes its selection decisions for each variable.

Subroutine KP_Greedy(n, Budget)

Step 1: *For i=1 to n: Calculate index(i) = ben(i)/cost(i)*

Step 2: *Sort* ***index*** *into descending order.*
Let order(1) be the location of the item with the largest index.
Let order(n) be the location of the item with the smallest index.

Step 3: *For i=1 to n*
If (cost(order(i)) <= Budget
Set x(order(i)) to 1, i.e., the item is included
Budget := Budget – cost(order(i))
Else Set x(order(i)) to 0, i.e., the item is excluded
Return *x()*

The Excel spreadsheet file labeled "Knapsack_Intro" contains a sheet labeled "Greedy Solution". This sheet has a program that finds a greedy solution to any Knapsack Problem instance with 20 or fewer items. (Because this algorithm requires so few steps – most of the work is done during the sorting of the index values - it can be efficiently implemented for instances with a larger number of items).

Exercise: How does the greedy solution to Astro Investment's problem compare to the optimal solution?

Challenge: Develop a greedy heuristic for KP by using a different index for each item and use it to find your greedy solution to Astro Investment's problem. Give a reason why you believe in your approach. How does your greedy solution compare to the optimal solution?

A Variant of the Knapsack Problem Model

One could imagine that we might encounter a problem-solving situation that looks like the KP, i.e., optimally selecting from some slate of candidates, but where some of the rules governing the fundamental version might need to be suspended. In such cases, we may have to adjust our problem formulation or our algorithms to find an acceptable solution.

> *Baker tour company is planning a bus excursion for 125 people. They work with a bus company that supplies the drivers and has two different bus models to choose from: a big bus with a capacity for 55 people which costs $750 for the day and a smaller bus with a capacity for 30 people which costs $400 for the day. The tour company can order any combination of busses it needs for the trip. How many of each bus type should they order?*

In this version of the problem, we have only two different items. We seek the subset that minimizes the total cost, (rather than maximize a total benefit). Correspondingly, the capacity constraint is now a minimum, (rather than a maximum) – the amount of space we purchase must be at least 125. Clearly, in this problem instance, we need to be allowed to rent multiple busses of each type, as needed. As an integer program, this instance would be formulated as follows:

Let x_b = number of big busses needed
Let x_s = number of small busses needed

Objective: Minimize $750*x_b + 400*x_s$

Subject to: $55*x_b + 30* x_s >= 125$

$0<= x_b <= 3$, *integer*
$0<= x_s <= 5$, *integer*

Challenge: Suppose you have a "Black Box" solver designed for the traditional Knapsack problem that can quickly identify the optimal solution to any instance. Explain how you can use the black box to find the optimal solution to the above problem variant. **Hint:** You would need to model eight items. Think about which busses you will **not** need.

Based on your answer, use the enumerative solution generator, "Small Solver" in the Excel file, "Knapsack_Intro" as a black box to find a solution the above problem.

Other variations to the fundamental version of KP might not be solvable by using a black box for the fundamental KP, as described above. They add levels of complexity that not only make a hard problem even harder to solve, but would require new algorithms, even in cases where heuristic solutions are acceptable. Note that a black-box solver might still be useful as a subroutine for these cases.

Challenge: Think about how you might find a (heuristic) solution to the problem in cases where there are two capacity constraints. For example, consider the problem of stocking a truck with packages of varying sizes and weights. The set of items to include is limited by both a total weight capacity and a total volume capacity. Try your approach to find a reasonable solution to the following problem instance, where the total weight cannot exceed 1200 lbs. and the truck capacity is 5 cu. ft.:

Package	Weight (lbs)	Volume (Cu. Ins.)	Revenue
A	150	500	$200
B	265	1000	$300
C	750	2500	$700
D	300	950	$400
E	200	800	$250
F	450	700	$350
G	350	1000	$300

Image how much more difficult this problem would be if you were also required to get the packages to their destinations before some due date!

Challenge: Think about how you might find a (heuristic) solution to the problem in cases where there are conditional constraints on the items. For example, suppose that item B can only be considered for inclusion if item A is included, or if item C is included, then item D cannot be considered. Try your approach to find a reasonable solution to the following problem instance with a budget limit of $1,500,000:

Project	Considered only if:	Investment Requirement	Expected Profit
A		$600,000	$170,000
B	C in and F out	$500,000	$125,000
C		$750,000	$165,000
D	F in	$150,000	$80,000
E		$250,000	$90,000
F	A out	$100,000	$70,000

An Optimal Solution to the Knapsack Problem based on Recursion

Accompanying File: https://www.dropbox.com/s/msp5xpixcy602y5/Knapsack_Recursion.xlsx?dl=0

In an *enumerative* approach, we solve the Knapsack Problem of finding the subset of items with the highest total benefit, i.e., the "optimal" set, by identifying every subset of items. We remove from consideration those subsets that exceed the budget limit. We could also exclude from consideration those subsets that are not "maximal", i.e., subsets that leave room in the knapsack for at least one additional item. Then, we examine the remaining subsets to identify one that has the largest total benefit. Many solution approaches involve some form of *smart enumeration* to further cut down on the total amount of computational effort required to find the optimal set.

One way to perform an enumeration in an instance where there are n items, {1, 2, …, n}, is to perform a "branching" of the problem. This is where we create two smaller problems based on: *Branch A*: Assume that one of the items, say item n, will be *included* in the knapsack, and *Branch B*: Assume that item n will be *excluded* in the knapsack. As a pair, both branches ("subproblems") cover all of the possibilities of the original problem. Each subproblem now only has items {1, 2, …, n-1} to consider. In Branch A, the knapsack capacity is now ($B - c_n$) because item n uses up space in the knapsack. In the other branch, the capacity is still B.

If we were to further branch on item n-1 for both subproblems, then we would a total of four subproblems of size n-2, with items {1, 2, …n-2} to select from. If we were to further branch on item n-2 for the four subproblems, then we would a total of eight subproblems of size n-3, with items {1, 2, …, n-3} to consider. And so on.

In a smart enumeration, we may find that some of the subproblems we create cannot be optimal because they include some items with low relative benefit, (e.g., as measure by its index) and exclude some items of high relative benefit. By ruling out those subproblems before we try to solve them, we can save a lot of work. A method called "Branch and Bound" is an implementation of this approach.

In a *recursive* technique, we assume the existence of a "Black-Box" problem-solver, S(k, C). This problem finds the optimal solution to the subproblem with items {1, 2, …, k} and a knapsack with remaining capacity C. We do not know how such a black box operates, but for very small cases, we could anticipate the results it would provide.

For example, S(0, C), which represents a Knapsack subproblem with *no* items, must produce a solution of an empty subset and a total benefit of 0.

For example, S(k, 0), which represents a Knapsack subproblem with items {1, 2, …, k} but no remaining capacity in the knapsack, must produce a solution of an empty subset and a total benefit of 0.

These ("base") solutions could be used to determine the solution to a problem with more items or more space. Then those solutions could be used to find the solution to even larger instances. In particular, the

solution to the original problem, S(n, B), could be determined from the solutions to the subproblems created by branching on item n.

The solution from S(n, B) is the better of:

(A) the solution from S(n-1, B - c_n) with item n included and additional benefit b_n, or
(B) the solution from S(n-1, B) with item n excluded.

The total benefit (optimal value) of the solution, V(), follows the following relationship:

$$V(n, B) = \text{Max}(V(n-1, B),\ b_n + V(n-1, B - c_n))$$

In general, the recursive equation is: $V(k, C) = \text{Max}(V(k-1, C),\ b_k + V(k-1, C - c_k))$, where

$V(k, C) = 0$ for all k, when $C <= 0$
$V(0, C) = 0$ for all C

Thus, it is possible to find the optimal value for the Knapsack problem by populating a two-dimensional array (matrix) for V with (B+1) rows for the potential intermediate capacity values and (n+1) columns for considering 0…n items. First place the "base" values into the matrix and then use the recursion equation to populate additional locations until a value is found for V(n, B). Any method for the problem must also keep track of whether the optimal solution for V(k, C) includes item k or not.

The spreadsheet file labeled "Knapsack_Recursion" that accompanies this document contains a sheet that populates the V matrix for the following instance of the Knapsack Problem with *n=7* and *B=15*. The optimal total benefit for this instance is 10.

Item #	1	2	3	4	5	6	7
Cost	3	2	5	1	3	6	5
Benefit	5	7	8	1	6	8	9

Challenge: Find the optimal subset of items to include in the knapsack for this problem instance.

Challenge: Find the optimal subset and total benefit associated with the following instance of KP with *n=10* and *B=20*:

Item #	1	2	3	4	5	6	7	8	9	10
Cost	3	2	5	1	3	6	5	4	7	8
Benefit	5	7	8	1	6	8	9	6	10	15

Challenge: Write a computer program that uses recursion to solve any instance of KP with 25 or fewer items and a knapsack capacity of 100 or less. Make sure the program identifies both the optimal total benefit and which items to include in the knapsack.

Heuristic Approaches to the Knapsack Problem

Accompanying File: https://www.dropbox.com/s/p2pks3cq2zfq5kt/Knapsack_Heuristics.xlsm?dl=0

Heuristic algorithms are efficient methods used to find a solution to the Knapsack Problem, e.g., that avoid enumeration. These methods cannot guarantee finding the *best* or *optimal* solution, nor can most make claims about how close they come to optimality. However, if enough thought is put into the process, there is reason to believe that such methods may find a subset that has a "good enough" total benefit for the amount of time allotted for it. (One might expect that the more work an algorithm does, the better the quality guaranteed in the solution obtained, but this is often not the case. Finding an algorithm for a problem that hits the "sweet spot" that bests trades quality of solution with total computation time is an important and interesting challenge to researchers and practitioners of Computer Science).

Heuristics take on different approaches to finding a reasonable solution to a model problem like KP. When taking on a new problem model, some relatively simple method is devised to be used as a "baseline" against which to measure the efficiency or solution quality of more complex algorithms. For any algorithm that provides a solution to the problem, we usually measure the "worst-case complexity", which gives the minimum number of steps necessary to guarantee that a solution can be found for any problem instance. This is usually expressed as a "Big-O" function of the problem size. Since a heuristic is designed with the goals of being relatively fast and providing reasonably good solutions, we also measure performance by running the heuristic and the baseline on a large set of sample instances, using the same computer, and compare their running times and solution values. Any heuristics that consistently provides better solutions in shorter timeframes might take on the role of the new baseline for the problem.

For KP, we will use a "Greedy" heuristic as the baseline. This is a "one-pass" method, which means that, in a single loop, each item is considered in some order. If the item can fit in with those previously included, it is added. If it cannot fit in, it is excluded. In the Introduction section, we chose a greedy method where the items in are sorted in decreasing order of marginal benefit, i.e., (b_i / c_i). For KP problem instances with n items the worst-case complexity of a one pass algorithm would be O(n) if the items are taken in the order in which they are input. Our Greedy algorithm, however, has complexity O(n log n), because we choose to sort the items first. We call this a "Greedy by Index" method.

It should be clear that the number of possible heuristics for KP, or any other complex problem class, is virtually unlimited. As a discipline, Computer Science perpetually invites people to provide better algorithms or better analysis of existing algorithms for a very large set of model problems and their variants. As a result, we continually develop a deeper understanding of the problems and the application of computers to solve them. We describe just a few of these approaches, when applied to the Knapsack Problem.

Random Problems and Random Solutions

The spreadsheet file labeled, "Knapsack_Heuristics" accompanying this document includes a sheet called "Random Solutions" that allows the user to create a problem instance at random that has 20 or fewer items. Since the variable for each item can take on a value of either 1 (if included) or 0 (if excluded) we can

virtually flip a coin, ("heads" for a 1; "tails" for a 0), to decide whether an item is included. The sheet provides 25 different solutions to KP obtained this way. As you will see, the set of solutions is not likely to include a particularly good one. Many solutions would select to include too many items to fit in the knapsack, (these are called "Infeasible"), and many others would include too few items, leaving room in the knapsack for additional items, thus not providing maximal benefit. This approach could be thought of as a heuristic version of enumeration, which considers only 25 solutions out of a possible 2^{20}, which is over a million. The sheet provides a solution using the Greedy-by-Index method, for the purposes of comparison.

Challenge: Think of some smarter ways to use randomization, e.g., coin flips, to generate a better set of 25 solutions, i.e., solutions that are less likely to include too many or too few items. **Hint:** You might want to mix in some "greediness" or other heuristic approach before virtually flipping the coins. A more advanced approach involves flipping a "biased" coin, where the probability of heads is larger for items with a larger index. (We will discuss this approach in more detail later). You can evaluate and compare your solutions by entering them into the sheet labeled, "Evaluate Solutions".

Iterative Improvement Methods

The motivation for developing heuristics is to find a thoughtful solution in a way that would not take up too much computation time, e.g., by enumerating through every possible solution. In some commercial application programs, e.g., for Cargo Consolidation or Vehicle Routing or Capacity Planning, we may need to solve hundreds or thousands of instances of KP as a subroutine. Systems Architects might conduct studies to determine how much computation time or memory space to allocate to a single run of the KP routine, so that the overall application could provide a solution in a reasonable amount of time.

Suppose in such a case, we can quickly identify some solution that is reasonable, e.g., by applying some greedy heuristic or at random, but we have some extra time available to find something better. We can use this current solution as a start toward finding an even better solution. Then we can use that better solution to find an even better one. We call such approaches, "Iterative Improvement" methods. In a loop, the method tries to improve and improve until some "Stopping Conditions" are met. This might be after some time limit or iteration count limit, or if the current solution appears to be "good enough" and has little chance of being further improved upon. One would hope that the last solution found is much better than the starting solution.

An example of an improvement method for KP is to select some item(s) that are included in the current solution and remove them. Then we fill the space created in the knapsack with some of the excluded items. If this solution has a greater total benefit, then it becomes the new current solution. There are many ways to implement this approach. Any heuristic you develop must describe: 1) how to select the items to remove, 2) how to order the excluded items for potential placement into the knapsack, and 3) when to stop.

In the spreadsheet file, "Knapsack_Heuristics" the sheet labeled "Improvement" contains a program the implements this method as follows:

Repeat the following:

1) *Select an item that is currently included in the knapsack. Remove it from consideration.*
2) *Use the Greedy-by-Index method, (e.g., where index = benefit / cost), on the excluded items to see if any can now fit into the knapsack. If the new solution has greater total benefit than the current solution, make it the new current solution.*

Stop when no item removal from the knapsack results in any improved solution.

You may notice that the Greedy solution is a fairly good starting point, so in many cases, further improvement through this method is unlikely. This, of course, does not mean that the Greedy solution or any solution you obtain at the end of the improvement process is optimal. You may find that starting the process with a different first solution may lead to a different, and perhaps better, ultimate solution.

Challenge: Modify the code in the improvement heuristic to use any feasible solution as the starting point. Keep track of how often or how rarely the last solution from this approach is better than the Greedy solution.

Challenge: Write a computer program that modifies the above method by repeatedly selecting *pairs* of items, (i.e., subsets of size 2), in the knapsack to remove from consideration. Notice that this would increase the computational complexity because there are $O(n^2)$ pairs. Does this method appear to find better solutions?

Note: If we were to implement this algorithm by looping through *all possible subsets* and taking them from consideration, then we are guaranteed to find the optimal solution. However, such an algorithm would require multiple enumerations of the subsets, so it would not be efficient.

Look-Ahead Methods

Rather than take the approach of finding one solution, only to do work find a better one, we may like to have a single algorithm that is smart enough to find a single, good solution - one that is not likely to be easily improved upon. Greedy methods are simple and clever, but have the flaw that, in many KP instances, including what appears to be the best item early in the process takes up room in the knapsack that might be better used by items are considered too late and can no longer fit. As an alternative, "Look-Ahead" methods consider some of the future consequences of including an item before making the decision about whether to include it.

For example, in a "One-Step Greedy" Look-Ahead method we loop through each item, say item k, and identify the total benefit associated with a Greedy solution, (or other type), that includes item k, (conditionally), as the first decision. Thus, we have generated n different solutions. If the best of these – one with the largest total benefit – is associated with choosing item k first, then we decide that item k is

the first to be permanently included in the knapsack. Now the knapsack has a residual capacity of ($B - c_k$). Next, we loop through the remaining n-1 items and identify the (Greedy) solutions to the KP instance having the revised capacity. The item associated with the best of these solutions is permanently included next into the knapsack; the residual capacity is further reduced, based on this decision. We continue in this fashion for n-2 additional phases, or we can stop early when no remaining items can fit into the knapsack with its reduced residual capacity.

In a "Two-Step" Look-Ahead method, we would start by looping through each *pair* of items, conditionally assume that they are included, and generate $O(n^2)$ (Greedy) solutions over the remaining items. One of the items from the pair associated with the best of these solutions is permanently included into the knapsack. Its residual capacity is then updated. In the next phase, all pairs of the remaining items are used to generate additional solutions that help us select the next item to permanently include. So, as its name implies, in this method we look two steps ahead before making any decision about which item to permanently include. The spreadsheet file, "Knapsack_Heuristics" contains a sheet labeled "Two-Step Greedy" that demonstrates the method by listing the item included after each phase of the heuristic.

Notice, like before, that the heuristic often gives a result that is equivalent to a Greedy solution. This is because Greedy-by-Index heuristic often finds a solution that leaves little room for the types of improvements described in this document. We have to wonder whether it is worth the extra computational effort to occasionally do better. When a better solution might lead to hundreds of thousands of dollars in additional profits, the answer is most assuredly yes! Note also that if we were to apply these heuristic approaches to variants of the problem, e.g., where there are conditions constraints regarding which items can be include or excluded, then the difference between a greedy approach and a more sophisticated one might become more evident.

Challenge: Use a One-Step Look-Ahead approach to find a reasonable solution to the following variant problem instance with a total budget limit of $1,500,000:

Project	Considered only if:	Investment Requirement	Expected Profit
A		$600,000	$170,000
B	C in and F out	$500,000	$125,000
C		$750,000	$165,000
D	F in	$150,000	$80,000
E		$250,000	$90,000
F	A out	$100,000	$70,000

Genetic Algorithms for the Knapsack Problem

Genetic algorithms are methods to generate solutions to a problem like KP in a way that mimics the biological process of evolution. To survive and thrive, the members of a species change over time to better adapt to their environment. The more robust members tend to survive longer and create more progeny, while, over the generations, the traits of those less adapted would tend to fade out in the species' gene pool. Every so often, a member is born with a genetic mutation. If the mutation is beneficial to the species the member is more likely to pass it on the new trait to the next generations. If not, members with the trait are less likely to have opportunity to pass it on, so it will eventually die out.

For the Knapsack Problem, a solution can be thought of as a member of a species. Its genetic code can be expressed as a bitstring of length n, i.e., a sequence of 0s and 1s where location k is 1 when item k is included in the knapsack and 0 if it is excluded. If a solution is feasible and it has a relatively high total benefit, then we consider it to be a "strong" member. In the algorithm, we maintain some set of candidate solutions; it can be thought of as the current gene pool for the species. The goal is to let the gene pool evolve until some very strong members (good solutions) are identified. If we let the process go on long enough and keep the gene pool large enough, we will eventually find the optimal solution, but for practicality's sake this need not be our goal.

In the spirit of evolutionary processes like *survival of the fittest* and *random mutation,* a genetic algorithm to solve KP needs a way to measure the strength or "fitness rating" of a solution. This rating should be a function of both the total benefit of the solution and the total cost, which in some cases may be too high to fit in the knapsack. The stronger solutions might be given more opportunities to "mate" with other solutions to create new solutions for the pool. If we place limits on the total size of the pool, then we must periodically identify some solutions to remove from consideration. Every now and then, some new solution is randomly generated in the pool, (i.e., a "mutation"), to see if it has unexpected traits that might add strength to its progeny and further improve the pool of candidate solutions.

Initialization

In the first phase of a genetic algorithm for KP, a set of binary string starting solutions are generated to initiate the gene pool. This can be done using some simple heuristics or in a random manner. We calculate the fitness rating associated with each of these candidate solutions, as described below. The candidates can then be sorted by their ratings, so that a solution with a higher fitness rating will have a higher probability of being selected to "mate", i.e., combine with another solution create offspring solutions. The offspring are new solutions that resemble their "parents" and are hoped to have high fitness ratings of their own.

If necessary, e.g., to keep the size of the gene pool within some reasonable limit, the best of these solutions can replace some of the weaker solutions in the gene pool. Although larger gene pools with many different solutions may be harder to maintain, they are often associated with a greater diversity in the offspring, which can sometimes lead to better solutions over time. For efficiency's sake, it would make sense to

perform periodic maintenance of the gene pool to make sure that the number of identical solutions is kept to a minimum.

There are a variety of approaches you can use to assign a fitness rating to the candidate solutions in the pool. This must be reflective of the fact that solutions with too few items included, like (1001000010) are not "maximal"; they leave room in the knapsack for additional items and additional benefit, so they should not have a strong rating. Solutions with too many items included, like (1011110111) are "infeasible"; the total cost of the included items is larger than the capacity of the knapsack, so the total benefit is unrealistic, and the rating should penalize this overage.

An example of such a rating method could be defined by the function:

$$\text{Fitness Rating} = \alpha * \text{Total Benefit} - \beta * \text{MAX}(0, \text{Total Cost} - B)$$

This function gives a higher rating to solutions with high benefit but removes rating points if the total cost is larger than the capacity of the knapsack. The multiples α and β, are used to balance the size of benefit parameters with the size of the cost values.

For example, suppose we must solve the following instance of KP, with *n=10* and *B=20*:

Item #	1	2	3	4	5	6	7	8	9	10
Cost	3	2	5	1	3	6	5	4	7	8
Benefit	5	7	8	1	6	8	9	6	10	15

Suppose based on the values and the size of the knapsack, we decide to set α to 1 and β to 2.5.

The solution (1001000010) would have a rating of

$$(5+1+10) - 2.5 * \text{Max}(0, 3+1+7 -20) = 16.$$

The solution (1011110111) would have a rating of

$$(5+8+1+6+8+6+10+15) - 2.5 * \text{Max}(0, 3+5+1+3+6+4+7+8 -20) = 16.5$$

Challenge: Find some feasible, *maximal* solution to the problem instance, e.g., by using a greedy method, and calculate its fitness rating.

Challenge: Develop an alternative function to measure the fitness rating of a candidate solution. Give some reasons why you believe such a function would do a good job in rating solutions.

Selection of Parents

In the next phases, which we call a "generation", parents generate offspring, mutations arise, and weaker solutions leave the gene pool. In a genetic algorithm, the gene pool evolves from one generation to the next, until some "stopping conditions" are met.

In this part, some pair of "parent" solutions are selected from the pool to "mate" and generate offspring solutions. This selection can be done in a variety of different ways, for example:

- *The two candidates with the highest values can serve as parents*. This approach allows the better solutions to create (maybe) better offspring but would often select the same pair of parents repeatedly. This may also lead to many identical offspring solutions, wasting time and further reducing the diversity in the gene pool.
- *Any two candidates can be selected as parents at random*. When every candidate solution is equally likely to serve in the role of parents, we are likely to generate a diverse set of solutions. However, if weaker parents are believed to produce weaker offspring, then much of the work done in this case might be dedicated to generating solutions that are predisposed to be relatively weak.
- *Develop a compromise between the two approaches*. For example, mate a randomly selected solution with a strong solution. One could also modify the random selection method with a "Monte Carlo" method, where stronger solutions are assigned a stronger probability of being selected as a parent. The details of this approach are presented in an Appendix to this section.

Generation of Offspring solutions

Once a pair of solutions is selected as parents, they "mate" to generate offspring. Like an actual species in the biosphere, the offspring should resemble their parents in some way. As one would expect, there are a variety of ways to generate offspring from a pair of solutions to KP.

Note: It is important to include some form of randomization in the decisions made in this phase to assure that the same set of parents are likely to generate a variety of offspring. We refer to the Appendix for details about how to accomplish this.

Some approaches to generating offspring include:

- In a "crossover", some position k is selected at random, then the child gets the sequence of values from parent A in positions 1…k and the values from parent B in positions k+1 …n.

- If both parents have a "1" (or a "0") in position k of the solution, then the child can take on that same value in position k. This "deterministic" approach can be replaced by a *random* selection for the value at position k, e.g., where a biased coin (more likely to show a head) is flipped. If it shows a Head, then match the parents' value, otherwise take the opposite value. If the value at position k differs for the two parents, then flip a fair coin to select the value for the child in position k. This approach leads to greater diversity among the offspring.

Random Mutation

Every now and then it is important increase the diversity of the gene pool to generate new solutions that may not resemble those generated through the usual mating process. A novel mix of traits may lead to unexpected improvements to the quality of the solutions. This could be done by generating a new solution

at random, or by selecting some strong single parent solution in the gene pool from which to mutate. For each position k, flip a (biased) coin, if it shows a Head match the value of the parent, otherwise choose the opposite value, (1 - parent value).

Pool-Size Reduction

If we are not willing to expend the effort to find the optimal solution to KP, we would want to keep the size of the gene pool reasonable, so It would be necessary to identify weaker solutions that we believe unlikely to have the optimal solution among its progeny. In this part, some of the weaker candidate solutions are removed from the gene pool. For example, if the size of the pool exceeds some fixed threshold, then the extra candidates must be removed. Some approaches for this include:

- The candidates with the lowest fitness ratings are removed, or
- Older candidates that have already generated many children can be removed, or
- Candidates are removed at random, or
- The candidates with lower fitness ratings have a greater probability of being removed.

Stopping Criteria

Finally, when it is time to end the loop, the best solution in the gene pool is selected. This could be after a predetermined number of generations, or when a limit to the amount of computation time is reached, or if it appears that the method is not generating any additional strong solutions from one generation to the next.

Appendix: A Monte Carlo Random Selection Method

Suppose we have a set of k items and we must select one at random. In the case where k=2, we can complete this task with a simple coin-flip: if the coin shows Head, then select item 1; if it shows Tail, the select item 2. Assuming the coin is fair, (i.e., the probability of Head is 0.5), then either item has an equal likelihood of being selected. For other values of k, we can image using some (fair) k-sided die to help us make this selection. When using a computer program to make this selection, we need some form of "random number generator" to help us virtually flip a coin or toss a die. Most programming languages implement random number generation function in at least one of the following ways:

- *Return a random between integer a and integer b, inclusive.* This function returns a single value from the range {a, a+1, a+2, …, b-1, b}. By using the range 1 to k, the function can help you select the item from the set.
- *Return a random floating-point value between 0.0 and .999999.* This function is more difficult to use, but provides greater flexibility, especially when you want to make selections that respect some probability distribution. We will assume that this *rand()* function is available for our needs.

For the task of randomly selecting among k items using the *rand()* function:

$$\text{Selection} = INT(1 + rand() * k),$$

where the *INT()* function returns the integer part of a floating point number, (i.e., where the decimal part is truncated).

Now suppose you want to make a random selection but want certain items to have a greater likelihood of being selected. For example, suppose k=2 and you wish to flip an "unfair" coin, where the probability of showing a Head is p, where p > 0.5. The probability of showing a Tail is hence (1-p) < 0.5. This can also be accomplished by using the *rand()* function. If the function returns a value that is less than p, then the virtual coin shows Head; if it is p or greater, then the result is Tails

More generally, in a genetic algorithm we may wish to design the parent selection process in a way that is biased toward the stronger solutions in the pool. In a "Monte Carlo" approach, the selection process would respect some discrete "probability distribution", as the following example illustrates:

Item	Probability of Selection	Random # Range
1	35%	0.0 <= *rand()* < 0.35
2	10%	0.35 <= *rand()* < 0.45
3	25%	0.45 <= *rand()* < 0.70
4	20%	0.70 <= *rand()* < 0.90
5	10%	0.90 <= *rand()*

The total probability must sum to 100%. In this case, it is possible for any of the items to be randomly selected but the likelihood of selecting item 1 is higher than selecting item 5. This is achieved by using the random number generator as described in the third column. For example, if the *rand()* function returns a value of 0.72354 then we select item 4; if it returns a value of 0.24183 then we select item 1. (Notice

that the upper bounds on the ranges in the third column correspond to the cumulative probabilities in the second column: 35%, 45%, 70%, 90%, 100%).

One approach to setting the probability values in a way that depends on the ratings would be to let *R* equal the total of all the item ratings. For each item k, set the probability to *rating(k) / R*. This is a valid (discrete) probability distribution because the probabilities are non-negative, and the sum of the probabilities is 1.0 or 100%.

Activity: Genetic Algorithm Solutions using Excel

File: https://www.dropbox.com/s/5f89px4i83tmviv/Knapsack_Genetic.xlsm?dl=0

You will develop your own scheme to implement a genetic algorithm to find a "good" solution to the Knapsack Problem. The Excel file labeled "Knapsack_Genetic" contains a sheet called "Solutions By-Hand" that contains a sample instance to KP, with n=10, and an initial gene pool of 20 candidate solutions. (For the purpose of grading, your instructor may assign a different instance or different initial pool). You can use this sheet to update your gene pool and compare your Genetic Algorithm solution to a solution generated by a "Greedy" method. The sheet will automatically calculate the total benefit and total cost for each solution in the pool.

Guidelines:

- You are provided space in the spreadsheet to place your choice of formula for the "Fitness Rating" associated with each solution in the gene pool. Recall that this formula should reward a large total value of the solution but should discount going over the cost capacity. (In the text we suggested a function of the form: *Fitness = Total Value – β * MAX(0, Total Cost – B)*, where you find an appropriate value for β). The spreadsheet will help you identify the solution with the highest rating and whether the solution is feasible.
- You will run your algorithm for five "generations" where, in each generation, some offspring are created and some weaker solutions are removed, leaving a total pool size of 20. You may want to keep a copy of the gene pool at the end of each generation so you can observe the improvements from one generation to the next. In particular, you should keep track of the best solution in your gene pool. If your method works well, then this solution would improve somewhat from that of the prior generation.
- The sheet labeled "Monte Carlo Selection" can help you to make a random selection among 20 different items. If you choose complete randomness, each item should be given an equal value. Otherwise, you may assign a valid probability for each item. (If you want to select from among fewer items, set the probability for the unneeded items to 0%). For your convenience, a column is reserved to store an item "value". You can then set the probabilities as a function of this column, as described in the Appendix.
- The sheet labeled "Monte Carlo Selection" also has a feature that allows you to virtually flip a coin based on some desired probability of showing a "head".
- In each generation:
    1. You will select four weak candidate solutions to remove from the gene pool. Describe how you made this selection.
    2. You will select a pair of parents from the remaining pool. Describe how you selected this first set of parents.
    3. You will "mate" the two parents to create two new offspring to add to the gene pool. You should use some random elements in an attempt to generate offspring that are somewhat different from their parents and their siblings. Describe your method.

4. You will select a second pair of parents from the pool. Describe how you selected this second set of parents.
5. You will "mate" these two parents in the same way that you used in Step 3.
6. (Optional) Use your method from Step 1 to remove one additional candidate solution from the gene pool. Replace this solution with a mutation, e.g., derived from some stronger solution in the gene pool. Describe how you selected the solution and how you performed the mutation.

Generating Knapsack Solutions with Java

The Object-Oriented paradigm for code development, e.g., using Java, dictates that classes be defined to organize the data and methods needed to generate solutions to the Knapsack Problem. Some conventions are applied to make these classes as simple as possible, but no simpler. In particular, we identify the different "entities" (or objects) that interact within our processes. Then we identify or create a Java class for each type of object, which contains the data components that identify and distinguish the individual objects and the methods that govern how they evolve, act, and interact.

For KP, the objects include the "problem instances" that need to be solved. For this discussion, we focus on the instances that follow the most fundamental version of the KP model. Variants to the problem would likely require extended classes with specialized methods. The "potential solutions" to an instance each have a meaning, (e.g., the set of included items, the total cost, and total value), that must be modeled in our code. We can think of a "solver" as a separate entity whose job is to apply some approach to a problem instance and generate one (or more) solutions. A solver may need access to some advanced data structures to support the algorithms it contains.

The key activities in the life of an object involve receiving data, e.g., to be born (or constructed) with, to evolve over time, and to die (or deconstruct). Along the way it also produces data for other objects and for the user. For the Knapsack Problem, which seeks a number-based answer to a number-based question, we assume that the life of the objects is lived within some "Main" program on a "console" which is initiated by the user on a keyboard, interacts with the user through a display, and stores its results where it is directed before the program ends.

The Problem Instance Class

A problem instance is a specific Knapsack Problem that needs to be solved. It is characterized by the number of items that are candidates for inclusion, the size of the knapsack and the cost and benefit value associated with each item. The number of items, *size*, must be an integer; the set of items are indexed, {0,1,2,…,size-1}. For the purposes of this discussion, the value of *size* can be in the hundreds, but if we seek optimal solutions then it has much lower limits. The remaining values could be either double or integer, but we note that the "Recursive" solution approach requires integer costs and an integer capacity on the knapsack. In a Java class the data can be defined as follows:

```
private int size;
private double capacity;
private double cost[] = new double[SIZE_LIMIT];
private double benefit[] = new double[SIZE_LIMIT];
```

The methods required for this class are limited to the constructor that populates this data and the accessor or "get" functions that allow other objects to use this data.

Exercise: Write the code for the "KP_Instance" class.

The Solution Class

A solution to KP is a specific set of values that dictate, for some instance of the problem, which items are included in the knapsack and which are excluded. A solution could be modeled as an array of Boolean values, where *included[k]* is true if item k is included and false if excluded. The solution might be infeasible if the total cost of the included items exceeds the capacity of the knapsack. Thus the data for the "KP_Solution" class can be defined as follows:

```
private KP_Instance problem;
private Boolean[] included = new Boolean[SIZE_LIMIT];
```

The variable *problem* is a pointer to the specific instance of KP being solved by the solution.

When a new solution object is constructed, we can assume that all of the items are excluded and the total cost and total benefit of the solution is 0. Throughout the life of the solution object, the algorithms may reset the Boolean values in the *included[]* array, which in turn modify the values of the total cost and total benefit. The following "mutator" methods would be necessary for the object:

```
public void include(int itemLoc) {
       included[itemLoc] = true;
}

public void exclude(int itemLoc) {
       included[itemLoc] = false;
}

public double totalBenefit() {
       int n = problem.getSize();
       double val=0;
       for(int i=0; i<n; i++)
              if(included[i])  val+=problem.getBenefit(i);
       return val;
}

public double totCost() {
       //  Insert your code here
}
```

The following method determines whether the current contents of a solution object is feasible:

```
public boolean feasible() {
       return (totCost() <= problem.getCapacity());
}
```

A Solver Class for the Greedy Heuristic

An algorithm that finds some solution for some problem instance is placed in a class whose job is to make the connection between the two. This is neither the job of a Problem_Instance object nor the job of a Solution object. We first illustrate with a class called "Greedy_Solver" which applies a "One-Pass" algorithm that examines each item in its order and includes it if it fits.

```
public class GreedySolver {
	public KP_Solution onePass(KP_Instance problem) {
		KP_Solution gSol = new KP_Solution(problem);  // Empty solution
		n = problem.getSize();
		for(int i=0; i<n; i++) {
			gSol.include(i);           // Put the item in
			if (!gSol.feasible()) gSol.exclude(i);  // Can't fit; take it out
		}
		return gSol;
	}
}
```

Challenge: Add a second method to this class, *public KP_Solution greedyByIndex(KP_Instance prob)* that orders the variables by (benefit / cost) before placing them into the knapsack.

Running the Main Console Program

It is the job of the "Main" routine, with the help of the user, to create a Problem Instance object and populate it with the problem parameter values.

```
KP_Instance problem = new KP_Instance(n,capacity,cost,benefit);
```

Then it creates a Solver object that applies the algorithm and produces a solution.

```
GreedySolver solver = new GreedySolver();
KP_Solution solut1 = solver.greedyByIndex(problem);
```

Then it uses the "get" functions in the solution object *solut1* to place the results onto the screen or into a system file.

Challenge: Write an application, e.g., using Java, that finds the Greedy by Index solution to the following problem instance, with a budget limit of $1,250,000:

Budget	Investment Requirement	Expected Profit
A	$600,000	$170,000
B	$250,000	$125,000
C	$750,000	$200,000
D	$200,000	$150,000
E	$250,000	$90,000
F	$100,000	$70,000

A Solver Class for the Recursive Optimizer

Since finding an optimal solution to KP might require an algorithm that performs an exponential number of steps, we must be careful to set appropriate limits on the problem size. The Recursive Algorithm described in Section I requires storing the potential solutions in a 2-dimensional array of size (n+1)*(B+1), where B is the budget. Thus we can only solve problems where the Java Machine can handle a structure of size: (n+1)(B+1)(# bytes in a solution object).

This array is stored in a *RecursiveSolver* object that supplies the algorithm. Thus, the data for this object would be defined as follows:

```
private KP_Instance problem;
private KP_Solution[][] matrix = new KP_Solution[SIZE_LIMIT][BUDGET_LIMIT+1];
```

Since the algorithm is recursive, we need one method to set up the problem and another method that performs the recursion.

```
public KP_Solution solve(KP_Instance p) {
       problem=p;
       int n = problem.getSize();
       int K = problem.getCapacity();
       for(int i=0; i<n; i++) matrix[i][0]=new KP_Solution(problem);

       for(int i=0; i< n; i++)
       for(int j=1; j<=K; j++) {
              matrix[i][j] = null;
       }

       populate(n-1, K);               // Call the recursive routine
       return matrix[n-1][K];
}
```

This method initializes the 2-dimensional array *matrix[][]*. It asks the *populate()* method to populate the rest of the matrix, e.g., using recursion, and it returns the desired solution stored at the appropriate location.

Before performing any recursive subroutine, the *populate* method checks the array to see if a prior call already populated the desired position in the array.

```
private double populate(int i, int B) {
	double value=0;
	int cost1 = (int) problem.getCost(i);
	if(i==0) {                  // Basic instance with one item
		matrix[0][B]=new KP_Solution(problem);
		if(B >= cost1) {
			matrix[0][B].include(0);  // Include item 0 if it fits
			value = problem.getBenefit(0);
		}
	}

	else {                      // Not a basic instance; multiple items
		if (matrix[i][B] == null) {      // Not already solved
			value = populate(i-1, B);
			matrix[i][B] = matrix[i-1][B].sCopy();
			if(B >= cost1) {
				double val2 = problem.getBenefit(i)
                                        + populate(i-1, B-cost1);
				if (val2 > value) {
					value=val2;
					matrix[i][B] = matrix[i-1][B-cost1].sCopy();
					matrix[i][B].include(i);
				}
			}
		}
		else value = matrix[i][B].totBenefit();// Already solved
	}
	return value;
}
```

Challenge: Write an application, e.g., using Java, that finds the optimal solution to the following problem instance, with a budget limit of $1,250,000:

Budget	Investment Requirement	Expected Profit
A	$600,000	$170,000
B	$250,000	$125,000
C	$750,000	$200,000
D	$200,000	$150,000
E	$250,000	$90,000
F	$100,000	$70,000

Programming Activity: Genetic Algorithm Implementation

Write the code for a *GA_Solver* class that contains algorithms to find a "good" heuristic solution to KP using a genetic algorithm. Your instructor will provide the instance(s) you are to solve with your program. The Solver class will likely need the following data elements to support the algorithm and its subroutines:

```
private KP_Instance problem;
private KP_Solution[] pool = new KP_Solution[POOL_SIZE];
private double[] fitness = new double[POOL_SIZE];
```

where *problem* is the instance of KP to be solved, *pool[]* represent the set of solutions currently in your gene pool and *fitness[]* stores the ratings provided to the solutions to help determine which are stronger (and hence more likely to produce offspring), and which are weaker (and more likely to be removed from the gene pool).

There is a fair amount of flexibility in how you implement the Genetic Algorithm. Parameters that control how many solutions to consider might be set as follows:

```
public static final int POOL_SIZE = 1000;
public static final int NUM_GENS = 500;
public static final int NUM_OFFSPRING = 10;
public static final int NUM_MUTATIONS = 0;
```

Since the genetic algorithm is a heuristic, the value of n, (the number items), can be fairly large. The pool size dictates how many of the 2^n possible solutions will be put under consideration at the start of the routine. The pool size should be larger for large n. The remaining values dictate how many additional (hopefully better) solutions will be considered in the program, i.e.,

NUM_GENS * (NUM_OFFSPRING + NUM_MUTATIONS)

Your instructor will guide you in selecting these values for the instances assigned in the project.

The solution method in the Solver class should be structured as follows:

```
public KP_Solution GA_Solution(KP_Instance p) {
          problem = p;
          pool = new KP_Solution[POOL_SIZE];

          initialPool();

          for(int k=0; k<NUM_GENS; k++) {
                calcFitness();
                removeSolutions();
                offSpring();
                mutations();
          }
          return bestSolution();
}
```

It is your responsibility to develop the code for the subroutines, subject to the following guidelines:

(Note: Your instructor may supply the code for some of these routines).

- The subroutine *initialPool()* is where you populate the *pool[]* array with a set of starting solutions. These can be generated at random, or by using some of the heuristic methods described in this document.
- The subroutine *calcFitness()* is where you evaluate each solution and store their fitness ratings in *fitness[]*. For example, *Fitness = Total Value – β * MAX(0, Total Cost – B)*, where you set the value for β.
- Based on the fitness values, your routine *removeSolutions()* identifies (NUM_OFFSPRING + NUM_MUTATIONS) weak solutions to remove from the pool to create space for the new offspring to be generated in the next phases. We suggest that you swap the locations of these solutions within the *pool[]* array so that the newly emptied slots are at the end.
- New "offspring" solutions are placed into the empty slots in the pool. Within *offspring()* there is a loop where a pair of strong "parent" solutions are selected and then one (or two) "child" solutions are generated. This is repeated until NUM_OFFSPRING solutions have been added.
- If desired, NUM_MUTATIONS additional solutions can be added to the pool. This could be accomplished by selecting (at random) a single "parent" solution and randomly modifying it to create a "mutant" solution to add to the pool.
- The subroutine *bestSolution()* returns the reference to the KP_Solution object from the pool that has the highest total benefit while not exceeding the knapsack's capacity.

The random elements in your algorithm my involve flipping a (biased) coin to determine whether to include an item into the knapsack. We recommend creating the following class for this purpose:

```
import java.util.Random;
public class Coin {
	public Coin(double b){
		bias=b;
	}
	public Coin(){
		bias=0.5;
	}
	public boolean heads(){
		return (new Random().nextDouble() < bias);
	}

	private double bias;
}
```

Advanced Programming Activity: A Branch and Bound Method for the Knapsack Problem

In an enumerative approach, you solve the Knapsack Problem of finding the subset of items with the highest total benefit, i.e., the "optimal" set, by evaluating every subset of items. We can remove from consideration those subsets whose total costs exceed the budget limit. We could also exclude from consideration those subsets that are not "maximal", i.e., subsets that leave room in the knapsack for at least one additional item – these would clearly not be optimal. Then, we can examine the remaining subsets to identify one that has the largest total benefit. The "Branch and Bound" method attempts to perform a "smart" enumeration to rule out solutions and reduce the total amount of computational effort required to find the optimal set. It requires storing, e.g., in a Stack, Queue, or Heap, (the "Storage structure"), a set of "candidate" solutions. This set can get quite large, even for a reasonably sized number of items, so any guarantee of optimality in the final result would be surrendered if the Storage structure fails to accept any candidates due to capacity limitations.

Branching

One way to perform an examination of the subsets of items {0,1,2…n-1} to include and exclude is to perform a "branching". This is where we partition the set of 2^n possible solutions based on:

> **Branch A:** Assuming that item 0 is *included* in the knapsack, and
> **Branch B:** Assuming that item 0 is *excluded* in the knapsack.

These branches are associated with a pair of "partial" solutions: (1,*,*,*,…,*) and (0,*,*,*,…,*), where a 1 indicates the item is included, a 0 indicates that it is excluded, and a star * indicates that no decision has yet been made about the item. If a partial solution is deemed a viable candidate, through a process called "bounding", (see below), then it stored in the Storage structure for additional branching later. Otherwise it (and all of its potential sub-branches) are discarded from consideration (i.e., the branch is "pruned").

A "leaf" is partial solution with all of its components resolved to 1 or 0. Since this would be a complete solution, it can be modeled in Java using the *KP_Solution* class described in the text.

The Java class for a *PartialSolution*, e.g., (1,1,0,1,1,0,0,*,*,*,*), can be modeled as an extension to KP_Solution class. The data elements would be:

```
private KP_Instance problem;
private Boolean[] included = new Boolean[n];
private int depth;
```

Where the additional variable *depth* represents the number of items for which an inclusion/exclusion decision has been made, which is 7 for partial solution (1,1,0,1,1,0,0,*,*,*,*).

Bounding

In the partial solution with depth 0, i.e., (*,*,*,…,*), no decisions have been mode about the items, i.e., this represents the original problem instance. Since we seek to find a feasible solution with the maximum

possible ("optimal") total benefit, we should notice that the total benefit from *any* feasible solution, e.g., one obtained through a one-pass heuristic, provides a lower bound on the optimal total benefit, i.e.,

(Best so far) Feasible Total Benefit <= Optimal Total Benefit <= Sum of All Benefits

As we encounter better feasible solutions, we can get "tighter" bounds on the optimum value, i.e., closer to the optimal, so more binding. Since the optimal solution is likely to exclude some items (because of the knapsack's limited capacity), the optimal total benefit would be no greater than the sum of the benefits from *all* of the items. This provides a "loose" upper bound on the optimum value.

As the Branch and Bound algorithm progresses, we hope make updates so that the range between the lower bound and the upper bound gets smaller and smaller until the bounds converge to the optimal value. We would also like to see many branches be pruned to cut down on the total work, time, and space required to find a solution.

A feasible partial solution that is stored in the data structure, e.g., (1,1,0,1,1,0,0,*,*,*,*), can be converted to a feasible complete solution by simply excluding all of the starred items, i.e., setting the * components to 0. This is implemented in the following *PartialSolution* method:

```
public double LowerBound() {
      double val=0;
      for(int i=0; i<= depth; i++)
            if(included[i]) val+=problem.getBenefit(i);
      return val;
}
```

Since this generates a feasible solution, it can be compared to the current lower bound to see if it is a better solution. However, if the depth is low and there are a lot of 0 components, the solution is not likely to be optimal. (It is not even likely to be "maximal", a necessary condition for optimality). So, there are likely better ways to obtain a feasible complete solution from a feasible partial solution, e.g., perform a one-pass greedy heuristics on the starred items to fit in as many additional items as possible.

The upper bound associated with a partial solution measures the best total benefit that can be hoped for in any of its branches. Obviously, if this value is less than any lower bound that we obtain throughout the run of the routine, then the partial solution and all of its branches can be removed from further consideration. The overall performance of the Branch and Bound algorithm for KP depends being able to do the pruning of sub-optimal branches while the depth of the partial solution is low so fewer of its branches would be unnecessarily stored and examined. Thus, it would be best to find a method that provides as tight an upper bound to a partial solution as possible, i.e., one that is much better than the following:

```
public double UpperBound() {
      int n = problem.getSize();
      double val=0;
      for(int i=0; i<depth; i++)
            if((included[i]) val+=problem.getBenefit(i);
      for(int i=depth; i<n; i++)
            val+=problem.getBenefit(i);
      return val;
```

```
}
```

Notice that this trivial (loose) upper bound simply obtains the value from including all of the starred items, i.e., it sets the * components to 1 without regard to the knapsack's capacity.

Note: A "tight" upper bound to KP can be obtained using the following algorithm:

Subroutine KP_UpperBound(n, Budget)

Step 1: *For i=1 to n: Calculate index(i) = ben(i)/cost(i)*

Step 2: *Sort* ***index*** *into descending order.*
Let order(1) be the location of the item with the largest index.
Let order(n) be the location of the item with the smallest index.

Step 3: *For i=1 to n*
Set x(order(i)) to Min(1, Budget / cost(order(i)))
Budget := Budget – cost(order(i)) x(order(i))*
Return *x()*

This algorithm applies to the original instance, where all of the items are starred *. You can modify it to find a tight upper bound on a partial solution and potentially increase the number of times a branch with a less promising solution is pruned.

The BB_Solver Class

The Solver class contains the algorithms needed to find a solution and the data they need to perform that task. There are a number of reasonable choices for the data structure that stores the partial solutions. For example, you can use a "Heap" if you want to keep the solutions associated the best upper bounds (and hopefully the solutions with the most promise to be optimal) closer to the top. A "Stack" or "LIFO Queue" would have its benefits if the items were numbered in their (Benefit / Cost) index order. This could be implemented in an array as follows:

```
private KP_Instance prob;
private PartialSolution[] solStack = new PartialSolution[STACK_LIMIT];
private int stackPtr = 0;

private boolean overflowCheck = false;
```

The variable *stackPtr* gives the location of the top partial solution, which is initially (*,*,*,…,*). When it is popped it will be replaced by partial solutions (0,*,*,…,*), which we call the "0-child", and possibly (1,*,*,…,*), which we call the "1-child".

If at any point, the array limit is met and a candidate solution cannot be pushed onto the stack, *overflowCheck* would indicate that the final solution may not be optimal.

The following pseudo-code provides a framework for the *BB_Solution()* routine which identifies the best solution observed by the Branch and Bound approach:

```
while(Stack is not empty) {
        pop a partial solution from the Stack
        if(UpperBound > BestFeasible) AND (Partial Solution in not a leaf) {
                Generate the 1-child, whose depth is one greater
                if(oneChild is feasible) {
                        Update BestFeasible if necessary
                        Push 1-child onto Stack (if not full)
                }

                Generate the 0-child, whose depth is one greater
                Push 0-child onto Stack (if not full)
        }
}
```

Assignment: Write the code for a *BB_Solver* class that contains the algorithms to find a solution to KP using a branch and bound scheme. Your instructor will provide additional instruction and the instance(s) of KP you are to solve with your program.