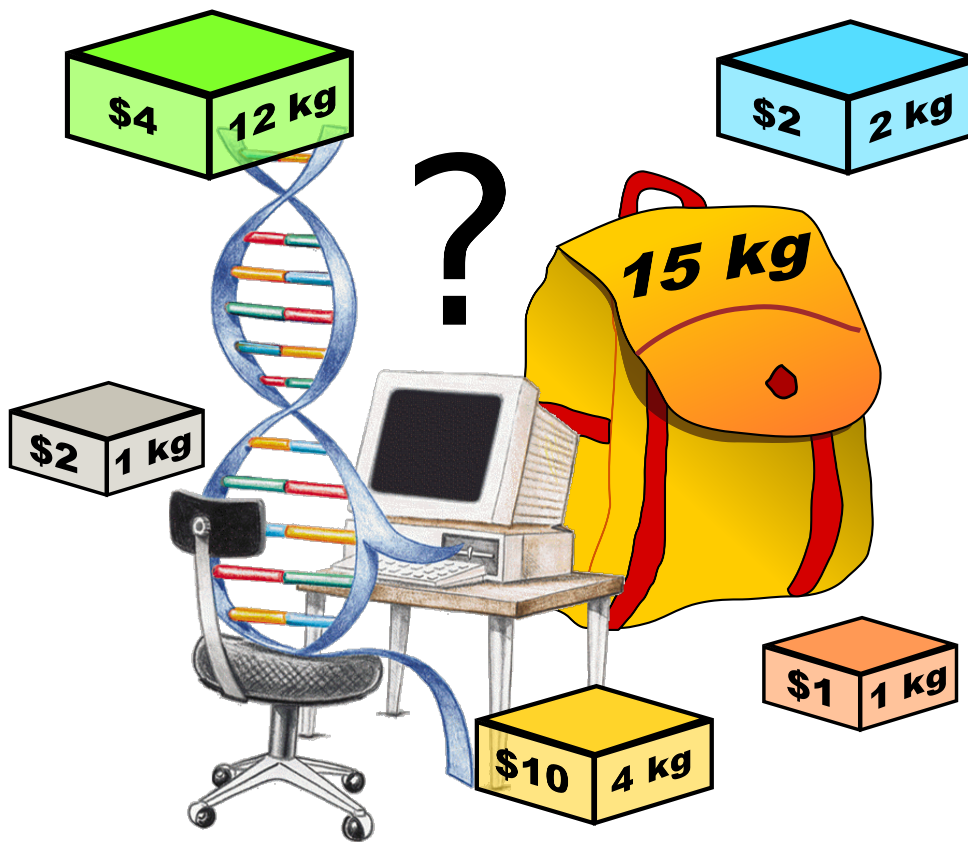
Solving Knapsack Problems with Evolutionary Computation



Introduction to Computer Science (Fall 2008/2009)
Assignment 5

Deadline: Sunday, February 22, 2009

1 Introduction

Some computational problems, such as sorting numbers and searching in an ordered list, are quite easy to solve. In this course we have introduced efficient algorithms to solve such problems—and implemented them in Java. However, many—if not most—problems in real life are hard and either can be *proved* to admit no easy solution (i.e., no polynomial-time algorithm), or shown to belong to classes of problems wherein *most likely* easy solutions do not exist. The knapsack problem is an example of a hard problem in computer science.

Suppose a thief has a knapsack with a certain capacity, i.e., the sack can hold at most a given weight. She looks around the house she has broken into and sees different items of different weights and values. She would like the items she chooses for her knapsack to have the greatest value possible yet still fit in her knapsack. More precisely: Given a set of items, each with a weight and a value, determine which items to include in a collection so that the total weight does not exceed a given limit and the total value is as large as possible. This is known as the *0-1 knapsack problem* since the number of copies of each kind of item is restricted to zero or one.

A simple algorithm immediately comes to mind. Thinking about the problem we realize that, basically, we have a set of items, and wish to find a subset of maximal value whose weight is less than or equal to the sack's capacity. So let's simply try each subset of items, and select the best one.

This simple algorithm is called exhaustive search, since it tries out *all* possible subsets. It is guaranteed to provide a correct answer. However, its runtime is very high: if the number of items (size of set) is n, then the number of subsets is 2^n (since each item can either belong or not to a subset). Thus, the runtime of this algorithm is exponential. And exponential is bad...

No polynomial-time algorithm is known for the knapsack problem. Does this mean we cannot solve it in a reasonable (i.e., less than exponential) amount of time?

Yes and No. If we are willing to give up the demand that our algorithm provide the *best* possible answer, and are willing instead to settle simply for a good solution (though not necessarily best), then the field of artificial intelligence (AI) provides us with many approximate, often stochastic (i.e., probabilistic) algorithms that can address problems such as knapsack.

In this assignment we will use a so-called **evolutionary algorithm**.

2 Evolutionary Algorithms

Nature is highly inventive: From the intricate mechanisms of cellular biology, to the sandy camouflage of flatfish; from the social behavior of ants to the diving speed of the peregrine falcon—nature has created versatile solutions, at variegated organizational levels, to the problem of survival. Many ingenious solutions were "invented", and still are, without any (obvious) intelligence directly creating them. This is perhaps the main motivation behind the field of evolutionary algorithms: creating the setting for a dynamic environment, in which solutions can be created and improved in the course of time, advancing in new directions, with minimal direct human intervention.

Evolutionary (genetic) algorithms are a family of search algorithms inspired by the process of (Darwinian) evolution in Nature [1–4]. Common to all the different flavors of evolutionary algorithms is the notion of solving problems by evolving an initially random population of candidate solutions, through the application of operators inspired by natural genetics and natural selection, such that in time "fitter" (i.e., better) solutions emerge. The field, whose origins can be traced back to the 1950s and 1960s, has come into its own over the past two decades, proving successful in solving multitudinous problems from highly diverse domains including (to mention but a few): optimization, electronic-circuit design, telecommunications, networks, finance, economics, image analysis, signal processing, music, and art.[1]

The genetic algorithm (GA) sets out with an initial population of individuals that is generated at random or heuristically. Every evolutionary step, known as a *generation*, the individuals in the current population are *decoded* and *evaluated* according to some predefined quality criterion, referred to as the *fitness*, or *fitness function*. To form a new population, which will constitute the next generation, individuals are *selected* according to their fitness, and then transformed via genetically inspired operators, of which the most well known are *crossover* and *mutation*. Crossover involves "mixing" two or more genomes to form novel offspring, and mutation randomly flips bits in the genomes. Continually iterating this procedure, and owing to the principle of selection of the fittest, the genetic algorithm may eventually find an acceptable solution, i.e., one with high fitness. Note that for hard problems an optimal solution is not guaranteed.

The basic structure of a genetic algorithm is:

1. produce an initial **population** of individuals, these latter being candidate solutions to the problem at hand (e.g., knapsack)
2. evaluate the **fitness** of each individual in accordance with the problem whose solution is sought
3. *while* termination condition not met *do*
    (a) **select** fitter individuals for reproduction
    (b) **recombine (crossover)** individuals
    (c) **mutate** individuals
    (d) **evaluate** fitness of modified individuals

Let us consider the following simple example, demonstrating the GA's workings. The population consists

[1] In the Department of Computer Science at BGU, Prof. Moshe Sipper, one of the lecturers in this course, leads the evolutionary algorithms group, which also includes three of the course's teaching assistants: Ilan Kadar, Michael Orlov, and Kfir Wolfson; more about the group at www.moshesipper.com.

of four individuals, which are binary-encoded strings (genomes) of length 10. For our example, suppose the fitness value simply equals the number of ones in the bit string. More typical values of the population size and the genome length are in the range 50–1000.

Crossover probability is $p_{\text{cross}} = 0.7$ and mutation probability is $p_{\text{mut}} = 0.05$.

Note that fitness computation in this case is extremely simple, since no complex decoding or evaluation is necessary. The initial (randomly generated) population might look as shown in Table 1.

Table 1. The initial population.

Label	Genome	Fitness
p_1	0000011011	4
p_2	1110111101	8
p_3	0010000010	2
p_4	0011010000	3

Having assigned a fitness value to each of the 4 individuals, by counting the number of ones in their genome (bit string), we now apply the first step of the genetic-algorithm loop: selecting fitter individuals. We will do this using a selection method known as tournament selection.

In tournament selection, we select parent individuals by randomly picking two individuals from the population, and choosing the one with higher fitness. Here, we must choose four individuals (two sets of parents), which means we hold four tournaments, each with two randomly selected individuals. In our example, suppose that the four tournament winners are p_2, p_4, p_1, and p_2 again, so that the two parent pairs chosen are $\{p_2, p_4\}$ and $\{p_1, p_2\}$ (note that individual p_3 did not get selected as our procedure is probabilistic and individual p_3 has a very low fitness value; also note that individual p_2 was selected twice).

Once a pair of parents is selected, crossover is effected between them with probability p_{cross}, resulting in two offspring. If no crossover is effected (with probability $1 - p_{\text{cross}}$), then the offspring are exact copies of each parent. Suppose, in our example, that crossover takes place between parents p_2 and p_4 at the (randomly chosen) third bit position:

$$\begin{array}{l} 111 \searrow 0111101 \\ 001 \nearrow 1010000 \end{array}$$

This results in offspring $p'_1 = 1111010000$ and $p'_2 = 0010111101$. Suppose no crossover is effected between parents p_1 and p_2, forming offspring that are exact copies of p_1 and p_2. Our interim population (after crossover) is thus as depicted in Table 2.

Next, each of these four individuals is subject to mutation with probability p_{mut} per bit. For example, suppose

Table 2. The interim (post-crossover) population.

Label	Genome	Fitness
p'_1	1111010000	5
p'_2	0010111101	6
p'_3	0000011011	4
p'_4	1110111101	8

Table 3. The resulting (post-crossover and post-mutation) population.

Label	Genome	Fitness
p''_1	1111010000	5
p''_2	0010101101	5
p''_3	0000011011	4
p''_4	1110111111	9

offspring p'_2 is mutated at the sixth position and offspring p'_4 is mutated at the ninth bit position. Table 3 describes the resulting population.

This population is that of the next generation (i.e., p''_i equals p_i of the next generation). As can be seen, the transition from one generation to the next is through application of selection, crossover, and mutation. Moreover, note that the best individual's fitness has gone up from 8 to 9, and that the average fitness (computed over all individuals in the population) has gone up from 4.25 to 5.75. Iterating this procedure, the GA will eventually find a perfect string, i.e., with maximal fitness value of 10.

We will now describe what you will need to write in order to implement a genetic algorithm for solving the knapsack problem.

3 Solving Subset Sum

Note. In this assignment, you are required to implement classes that have specific requirements regarding interfaces they implement, superclasses that they extend, and specific method signatures (see Section 5 for further details). Other than that, you have the freedom to define the necessary non-public fields, methods, and (possibly inner) classes. Details such as which sorting algorithm to use are left to your discretion. Moreover, Fig. 1 on page 6 contains the class diagram for Tasks 1–11.

To get a feeling of implementing an evolutionary algorithm, we will first implement a special case of the 0-1 knapsack problem: *subset sum*. You have probably become familiar with the subset sum problem earlier in the course. The variant we will solve is: given a set of integers $\mathcal{X}$ and the sum S, is there a subset $\mathcal{X}' \subseteq \mathcal{X}$ such that $\sum_{x \in \mathcal{X}'} x = S$? We would like to find such a subset, with its sum at least approximating S. Remember that for large sets $\mathcal{X}$, the standard recursive exhaustive search is not practical due to its exponential runtime.

Since the framework for the subset sum problem will be implemented using object-oriented principles, extending it to solve 0-1 knapsack and other search problems will consist of extending the existing hierarchy with a couple of new classes.

We will begin with implementing a population of individuals, where individuals know how to reproduce, and the population knows how to advance to the next generation. The population keeps individuals sorted based on their fitness, so that various selection methods can take advantage of the sorted order.

Task 1. Define an abstract class `Individual`.[2] It has a single floating-point field that keeps the fitness value. Fitness computation is the most time-consuming task in evolutionary algorithms, and depending on how they are managed, individuals do not necessarily compute their fitness. Thus, our individual will compute its fitness on-demand.

Add a default constructor that initializes the fitness field to a special "non-initialized" value.

```
public Individual()
```

Add a `getFitness` accessor method that computes fitness at most once during an object's lifecycle (saving the result in the fitness field).

```
public double getFitness()
```

To actually compute fitness, `getFitness` uses an abstract `evaluate` function that you should override in deriving classes. Note that since `evaluate` is used only in the hierarchy of `Individual`, its access level can be `protected`.

```
protected abstract double evaluate()
```

We need to be able to determine whether the individual is ideal, i.e., has the best possible fitness. Define a constant representing the ideal fitness 0, and use this constant in the predicate method `isIdeal`. *In this assignment, fitness is non-negative, and lower fitness is better.*

```
public boolean isIdeal()
```

Since individuals are kept sorted in the population, they need to be compared based on their fitness. We will use Java's `Comparable` interface for this purpose.

Task 2. Make `Individual` implement the `Comparable` interface. One individual precedes another in a sorted order if its fitness value is lower. Note that since `Individual` is abstract, it can implement an interface without defining the necessary methods. This is not what we want. `Individual` must conform to `Comparable`'s contract, so that deriving classes do not have to take care of that.

```
public int compareTo(Object obj)
```

[2]An abstract class is one in which some methods are abstract. An abstract method is one that is declared, but not defined—it has no body. A detailed explanation is available at `http://java.sun.com/docs/books/tutorial/java/IandI/abstract.html`.

We would now like to begin implementing the `Population` class, but face a problem. How will the population initialize a set of individuals? It cannot use `new`, since `Individual` is an abstract class. To solve this problem, we will make individuals cloneable, and the population will clone some prototype individual in its constructor. That is, in the initial population, all individuals will be equal to a given prototype.

Task 3. Make `Individual` implement the Java's `Cloneable` interface. Implement the necessary cloning method, so that it can be used by deriving classes in a chaining pattern. That is, each class can call its superclass' cloning method.

```
public Individual clone() {
    ... super.clone() ...
}
```

Pay attention that in the cloned object, the fitness should be reset to the "non-initialized" value. Note also that the cloning method can return a type that is more specific than `Object`, while still overriding `Object`'s cloning method.

Population should also be able to reproduce individuals. For this purpose, we define an interface.

Task 4. Define the `Selection` interface, containing a `reproduce` method that accepts an array of individuals as parameter, and returns a single individual (the selected one).

```
Individual reproduce(Individual[] pop)
```

In this assignment, reproduction is done together with selection.

Task 5. Define the `Population` class with a constructor that accepts the population size, the prototype individual, and the selection policy. The constructor should create an array of individuals, and fill it with individuals cloned off the prototype.

```
public Population(int size, Individual prototype,
                  Selection selection)
```

Define a `getBest` accessor that returns an individual with the highest fitness. Since the individuals array is kept in a sorted order, that would be the first individual in the array.

```
public Individual getBest()
```

We proceed from one generation to the next as follows. All the individuals are passed to the selection policy that returns a single individual every time. A new array is built that way, and can then replace the old generation.

Task 6. Implement the `nextGeneration` method in class `Population`. Remember to sort the new generation of individuals.

```
public void nextGeneration()
```

The sorting method should be private, and it will not be checked directly.

In object-oriented programming, we often encounter the problem of *coupling*—interdependency between classes. In our implementation as it is now, a subclass of `Selection` would need to know the concrete class of individuals in order to correctly perform reproduction (that includes crossover and mutation). It would be much easier if each individual had the capacity to mutate and to perform crossover with other individuals.

Task 7. Define the interface `Variable` that contains two methods. The first is `mutate`, returning a new mutated individual. The second is `crossover`, accepting another `Individual` as parameter, and returning a new individual which is the result of crossover.

```
Individual mutate()
Individual crossover(Individual other)
```

Make `Individual` implement the new interface. Since `Individual` is an abstract class, and we do not know how to perform variation on it, do not define the methods of the interface. That way, their implementation is delegated to the deriving classes.

In this assignment, crossover produces only one individual.

We can now write the class that drives evolution for a number of generations.

Task 8. Define the class `Evolution`. Its constructor should accept a population and the maximum number of generations. Provide a `getBest` accessor that returns the best individual in the current population.

```
public Evolution(Population population,
                 int maxGenerations)
public Individual getBest()
```

Write an `evolve` method that drives evolution until the ideal individual appears in the population, or until a maximum number of generations is reached. You should print the generation number and the best individual at each iteration (rely on the string conversion method of `Individual` being implemented in the concrete class). At iteration end, print the best individual, indicating whether it's a solution or an approximation (i.e., whether the best individual is ideal).

```
public void evolve()
```

Note that `evolve` does not deal with individuals directly, but uses the functionality of `population` to go from one generation to the next.

Upon completion of the tasks above, we have defined a "library" for evolutionary computation. Its abstract and concrete classes can be extended by problem-specific classes that implement only the necessary functionality, as shown in Fig. 1.

In the unified modeling language (UML) diagram, we see that to complete the implementation of subset sum, we need to extend the abstract `Individual` class, implement the `Selection` interface, and define a convenience class that extends `Evolution` and provides a constructor that creates the necessary population—with concrete individual prototype and selection policy.

Task 9. Define the `TournamentSelection` class that implements the `Selection` interface. The constructor accepts two parameters, p_{mut} and p_{cross}.

```
public TournamentSelection(double mutationProb,
                           double crossoverProb)
```

The `reproduce` method is implemented as follows.

```
public Individual reproduce(Individual[] pop)
```

1. Decide whether crossover will be performed, according to probability p_{cross}.
2. *Select* one (no crossover) or two (do crossover) individuals from the provided set, as described below.
3. If doing crossover, transform the two selected individuals into one via crossover.
4. Mutate the resulting individual with probability p_{mut}.

To *select* an individual, we perform the actual tournament selection: pick two random individuals from the population, and select the one with better (lower) fitness. If you want, you can assume that the population given to `reproduce` is sorted.

Remember that mutation and crossover operators return new individuals, and do not modify the existing ones. Regardless of whether crossover was used or not, `reproduce` returns one individual.

Task 10. Define the `SubsetSumIndividual` class that extends the abstract class `Individual`. Its constructor should accept an array of integer values and the desired integer sum. The constructor should create an array of boolean values of length that is the same as the length of values array. These boolean values are the genome, and indicate whether the corresponding

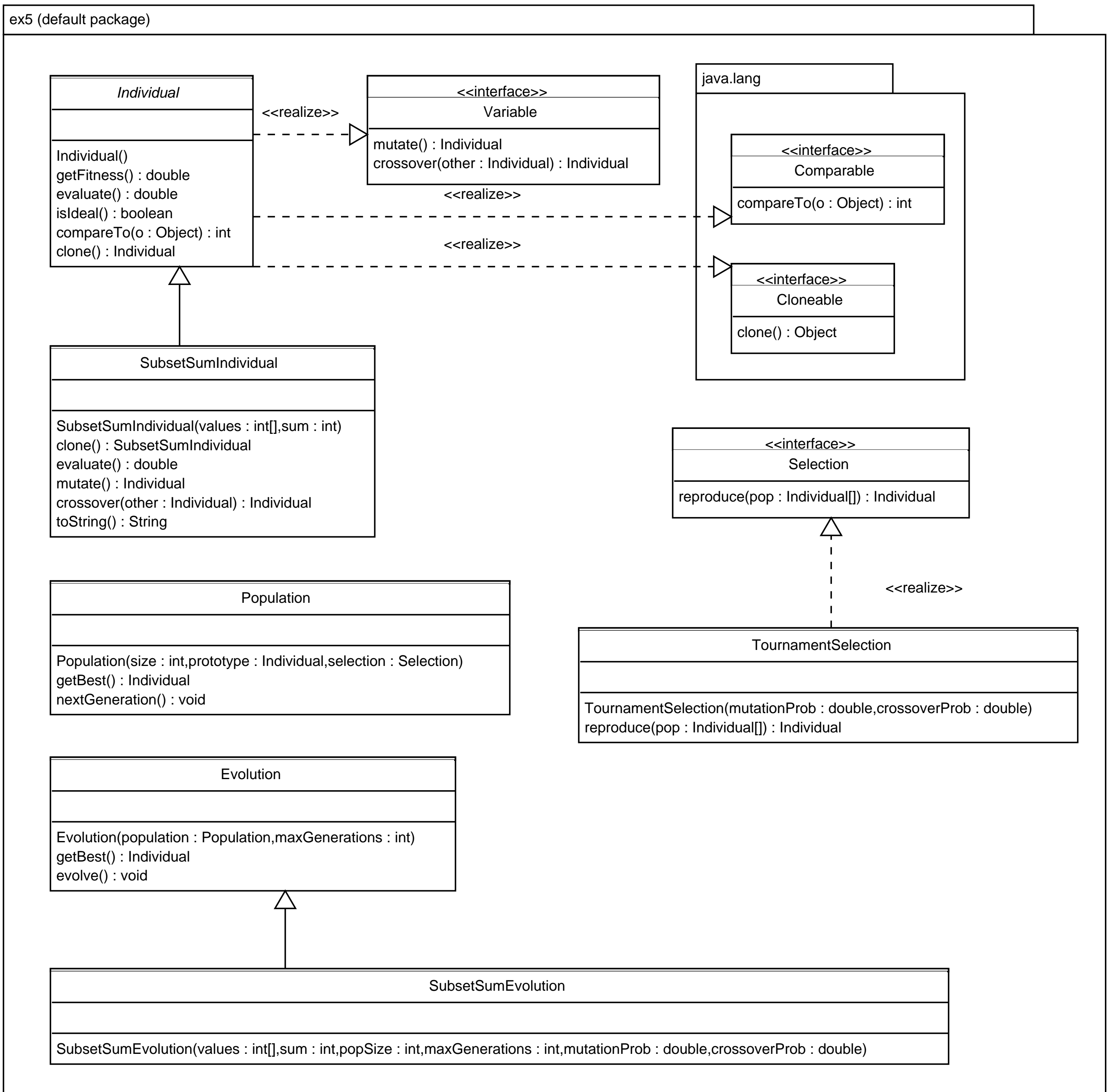


Figure 1. UML class diagram for subset sum solution implementation (Tasks 1–11). A dashed *«realize»* arrow corresponds to an interface implementation, and regular arrow corresponds to subclassing. Method signatures are in a format that's a bit different from Java. For example, the signature `foo(x: int): double` references the `double foo(int x)` method.

value in the values array is present in the subset.

public SubsetSumIndividual(**int**[] values, **int** sum)

Implement the abstract method inherited from `Individual`, by returning the absolute difference between the desired sum and the sum of values in the subset (as induced by the genome). Thus, a returned value of 0 indicates an ideal individual.

protected double evaluate()

Override the cloning method of `Individual`. Without the override, a cloned `SubsetSumIndividual` would contain references to the same values array (which is fine) and genome (which is not). Use `Individual`'s cloning method to create a clone, and then explicitly duplicate the booleans array in the cloned object.

public SubsetSumIndividual clone() {
 . . . **super**.clone() . . .

```
}
```

Implement the crossover and mutation methods, as required by the `Variable` interface. Remember that these methods are not destructive, and should produce new individuals (you can clone, or use the constructor). Mutation just flips a random boolean value in the genome. Crossover picks a random position *inside* the genome (between bits, not at the beginning or at the end), and copies the genome of individual given as parameter from the locus till genome's end (using same bit positions).

```
public Individual mutate()
public Individual crossover(Individual other)
```

For example, consider the individuals x with genome `0000`, and y with genome `1111` (where `0` and `1` represent boolean values *false* and *true*). A call to x`.mutate()` results in a new individual with four possible genomes: `1000`, `0100`, `0010` and `0001`. A call to x`.crossover(`y`)` results in a new individual with three possible genomes: `0111`, `0011` and `0001`.

Finally, override `Object`'s `toString` method, to return a string showing the values present in the subset, and the resulting fitness. This method will now be implicitly used in `Evolution`'s `evolve` method.

```
public String toString()
```

Task 11. Define the `SubsetSumEvolution` class that extends the class `Evolution`. It should contain just the constructor shown in Fig. 1. The constructor creates all the objects that are necessary to pass to the superclass constructor. Remember that a call to superclass' constructor must be the first line in the deriving class' constructor.

```
public SubsetSumEvolution(int[] values,
    int sum, int popSize, int maxGenerations,
    double mutationProb, double crossoverProb)
```

Finally, we can create a class with `main`, and evolve (after some debugging)! Figure 2 shows an example.

4 Solving Knapsack

We can now proceed to apply our class framework to the more general problem of 0-1 knapsack. In 0-1 knapsack, instead of a set of values and the sum, we are given a set of values with weights and the maximum weight, with the purpose of finding (or approximating) the maximal sum of values that fit under the maximum weight. Given a set of integer values $\mathcal{P}$, their integer weights $\mathcal{W}$, and the weight limit M, we would like to find a subset $\mathcal{P}' \subseteq \mathcal{P}$ such that $\sum_{p_i \in \mathcal{P}'} w_i \leqslant M$, whereas $\sum_{p_i \in \mathcal{P}'} p_i$ is maximized, or at least close to the maximum possible value.

```
public class SubsetSumMain {
  public static void main(String[] args) {
    int[]  values   = {7,3,5,10,13,17};
    int    sum      = 21;
    int    popSize = 100;
    int    maxGenerations = 1000;
    double mutationProb    = 0.2;
    double crossoverProb   = 0.8;

    Evolution evolution =
      new SubsetSumEvolution(
        values, sum, popSize,
        maxGenerations,
        mutationProb, crossoverProb);
    evolution.evolve();

    System.out.println("Result: " +
        evolution.getBest());
  }
}
```

Figure 2. A sample main class that takes advantage of the object-oriented evolutionary framework.

In general, once we opt to use the same selection method as before, we are expected to derive the knapsack individual class from `Individual`, and also derive a wrapper knapsack evolution class from `Evolution`—similarly to what was done in Tasks 10 and 11. However, it is more logical to implement crossover and mutation for knapsack individuals in the same way as for subset sum individuals. Even the string representation (fitness + list of values) suits the purpose. The only difference is in fitness evaluation.

Task 12. Define the `KnapsackIndividual` class that extends `SubsetSumIndividual`. The constructor should accept an array of integer values, array of integer weights, and the integer maximum weight. Note that the superclass constructor accepts a sum as its second argument. What value should be passed as the sum? Does it matter? Where is the sum used in `SubsetSumIndividual`?

```
public KnapsackIndividual(int[] values,
    int[] weights, int maxWeight)
```

Do we need to override the superclass' cloning method? If not overridden, how would it clone the fields of `KnapsackIndividual`? Does this behavior suit our purpose?

In Task 10, in mutation and crossover methods, you created new subset sum individuals using either cloning or constructor calls. It didn't matter then, since the outcome would be the same. Now that the same mutation and crossover methods are used for knapsack individuals as well, does it still not matter? How will the evolution progress in either case? If necessary, fix the implementation of `Variable` methods in `SubsetSumIndividual`.

We now need to define how knapsack individuals are evaluated. Fitness in 0-1 knapsack is a bit trickier than in subset sum. In subset sum, the goal was a fixed number, whereas in knapsack, we have a constrained optimization problem.

Task 13. In the `KnapsackIndividual` class, override the `evaluate` method of the superclass. Think of an appropriate fitness definition. Remember that *lower* fitness should indicate better fitness. Since it is hard to know whether an individual is ideal, fitness should always have a positive value (unless you are certain the solution is optimal—think when it is possible).

Note that you will probably need access to some fields in superclass, such as set of values and a set of corresponding boolean indicators. Escalate the access level of relevant fields in `SubsetSumIndividual` to `protected`.

Task 14. Define the `KnapsackEvolution` class. It should contain just the constructor that is similar to the one in `SubsetSumEvolution`, but instead of the second sum parameter, it receives an array of weights and the maximum weight.

public KnapsackEvolution(**int**[] values, **int**[] weights, **int** maxWeight, **int** popSize, **int** maxGenerations, **double** mutationProb, **double** crossoverProb)

Figure 3 shows an example program that evolves a knapsack solution, and stops when a solution is found (or when maximum number of generations has been reached).

5 Final Remarks

You can assume that all values, weights, subset sum, and maximal weight are positive.

You should submit a `ZIP` file with `Individual`, `Variable`, `Selection`, `Population`, `Evolution`, `TournamentSelection`, `SubsetSumIndividual`, `KnapsackIndividual`, `SubsetSumEvolution`, and `KnapsackEvolution` Java source files. Of course, you are free to define other classes.

The method signatures in the submitted files must be as described in the assignment, otherwise the submission system will reject the files. If this happens, look into the compiler errors returned by the system, and compare them with the test code that is compiled with the files (a link appears after the archive is sent).

All classes and interfaces must have `public`-level access. All methods and fields must be either `public`, `private` or `protected` (no *package*-level access). The strictest possible access level must be used for methods and fields. Moreover, fields must be declared `final` where possible.

```
public class KnapsackMain {
  public static void main(String[] args) {
    int[]  values    = { 4,2,2,1,10};
    int[]  weights   = {12,1,2,1, 4};
    int    maxWeight = 15;
    int    popSize   = 100;
    int    maxGenerations = 10000;
    double mutationProb   = 0.2;
    double crossoverProb  = 0.8;

    Evolution evolution =
      new KnapsackEvolution(
        values, weights, maxWeight,
        popSize, maxGenerations,
        mutationProb, crossoverProb);
    evolution.evolve();

    System.out.println("Result: " +
        evolution.getBest());
  }
}
```

Figure 3. A sample main class that takes advantage of the 0-1 knapsack extension of the object-oriented evolutionary framework. Note that inputs can have arbitrary length, and can be provided by the user. Here, the inputs are taken from the title illustration.

Do not define your own `Comparable` and `Cloneable` interfaces. You should implement the interfaces of the Java standard library.

Each submission will be checked both automatically and visually. Pay attention to proper indentation and comments.

Do not expect the assignment to be accepted by the submission system at the first time! This assignment is written in informal language—however, the compilation requirements are very strict, and are enforced by code that compiles with your files during submission process. You should allocate time for that, or suffer late submission penalty otherwise.

Good luck!

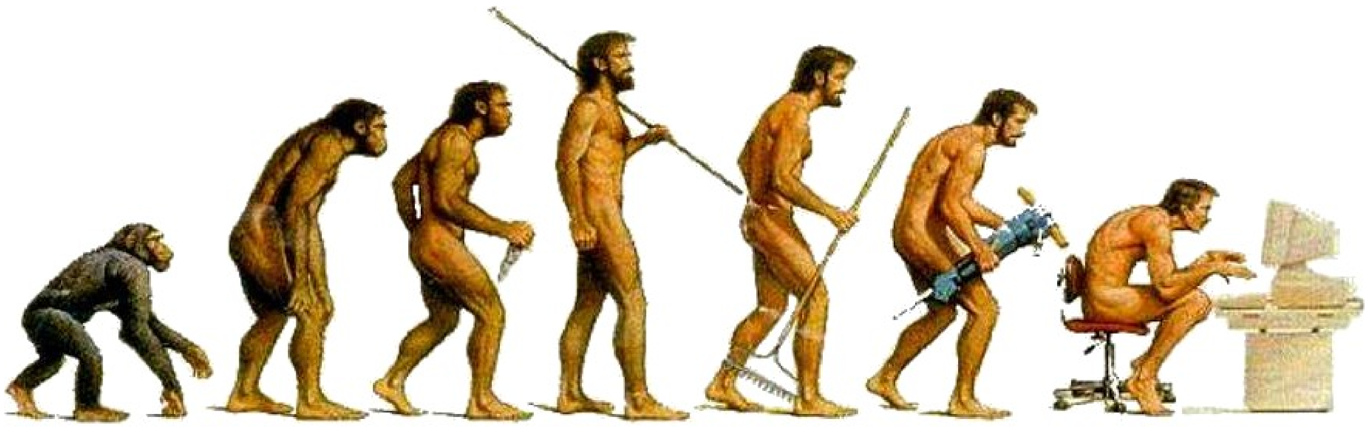


Figure 4. Evolution: The Rise and Fall of Man.

References

[1] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. The MIT Press, Cambridge, MA, USA, Dec. 1992. ISBN 0-262-11170-5.

[2] M. Sipper. *Machine Nature: The Coming Age of Bio-Inspired Computing*. McGraw-Hill, New York, July 2002. ISBN 0-071-38704-8. doi:10.1036/0071387048.

[3] A. Tettamanzi and M. Tomassini. *Soft Computing: Integrating Evolutionary, Neural, and Fuzzy Systems*. Springer-Verlag, Berlin / Heidelberg, Oct. 2001. ISBN 3-540-42204-8.

[4] D. Whitley. An overview of evolutionary algorithms: Practical issues and common pitfalls. *Information and Software Technology*, 43(14):817–831, Dec. 2001. doi:10.1016/S0950-5849(01)00188-4.