

GENETISCHE ALGORITHMEN

Grundidee

- ▶ "Survival of the fittest"
- ▶ Bildet die Natur nach
- ▶ Die besten Individuen haben bessere Überlebenschance und setzen sich durch
- ▶ D.h. sie vermehren sich auch = geben ihre Gene weiter ("selective breeding")
- ▶ Dabei kann sich Genstruktur verändern
- ▶ basiert auf Zufall
- ▶ ➔ man findet durch "gezieltes Würfeln" die beste Lösung

Anwendungsgebiete

- ▶ Richtige Lösung ist nicht bekannt
- ▶ Zu viele Möglichkeiten → man kann nicht alle ausprobieren

Voraussetzung:

- ▶ Es ist bekannt, ob eine Lösung besser ist als eine andere
- ▶ Man kann Lösung als **Array von Zahlen** beschreiben
 - Rucksackproblem
 - Travelling Salesman
 - 8-Damen-Problem

Begriffe

Folgende Begriffe werden in den folgenden Folien erklärt:

- ▶ Individuum
- ▶ Population/Generation
- ▶ "Änderung im Erbgut"
 - Crossover
 - Mutation
- ▶ Fitness
- ▶ Selektion
 - Roulette Wheel
 - Tournament
- ▶ Eliten

Individuum

- ▶ Ein Individuum besteht aus Genen
- ▶ Jedes Gen beschreibt eine Eigenschaft
- ▶ Gene werden an Nachfolger weitergegeben
- ▶ Codierung notwendig
- ▶ Jedes Gen entspricht einer Zahl → **List<int>**

22	73	55	49	50	63	81	20	47	72	75	91	42
----	----	----	----	----	----	----	----	----	----	----	----	----

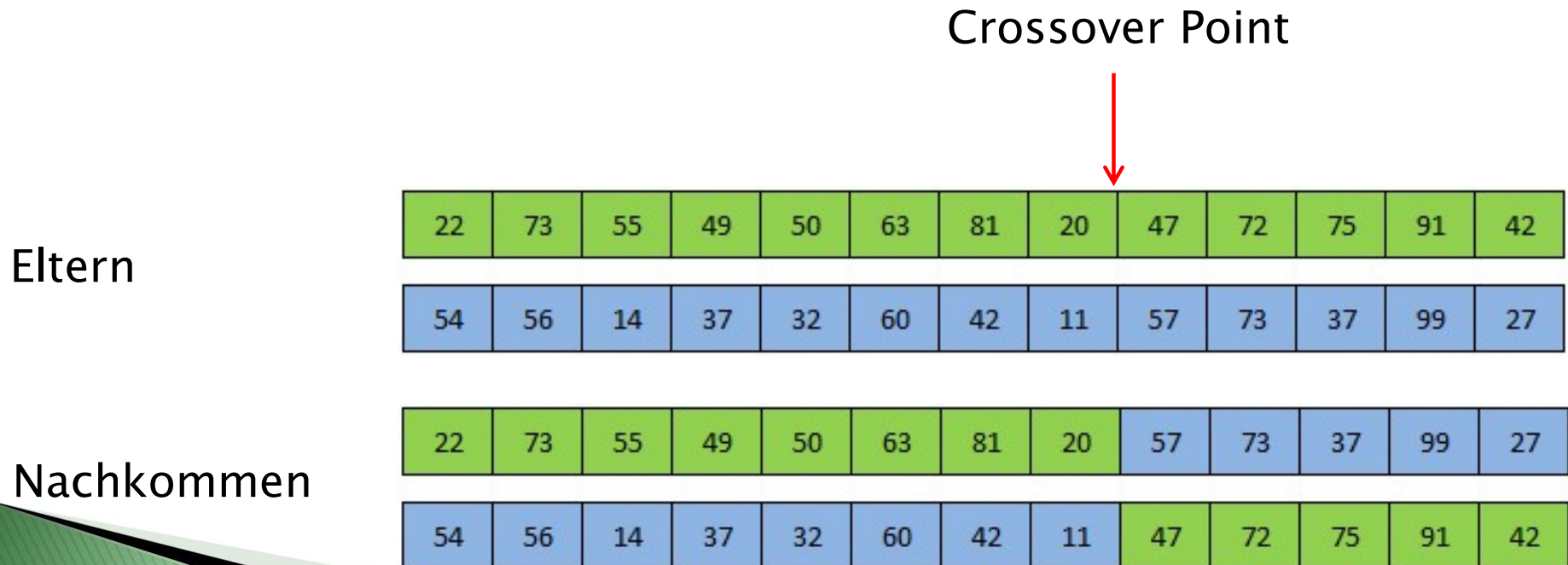
- ▶ Bei Rucksackproblem:
 - Zahlen nur 0/1 → **List<bool>**

Generation / Population

- ▶ Alle Individuen bilden eine Generation
- ▶ Die besten Individuen pflanzen sich fort → **Selektion**
- ▶ Einige besonders gute Individuen können auch überleben (**Eliten**)
- ▶ Man braucht Vielfalt, damit man nicht bei lokaler Lösung hängenbleibt
- ▶ Bei der Fortpflanzung kann sich das Erbgut **verändern**.
 - Crossover
 - Mutation

Crossover

- ▶ Zwei Elternteile tauschen Gene aus
- ▶ und zwar an einem "Crossover point"
- ▶ Es entstehen zwei Kinder
- ▶ ➔ muss zwei List<int> an Index tauschen



Mutation

- ▶ Wichtig, um Vielfalt zu erhalten
- ▶ Ein zufälliges Gen wird zufällig verändert
- ▶ oder: mit einem anderen getauscht
- ▶ ➔ in List<int> zwei Indizes tauschen

vorher

22	73	55	49	50	63	81	20	47	72	75	91	42
----	----	----	----	----	----	----	----	----	----	----	----	----

mutiert

22	73	55	49	72	63	81	20	47	50	75	91	42
----	----	----	----	----	----	----	----	----	----	----	----	----

Fitness

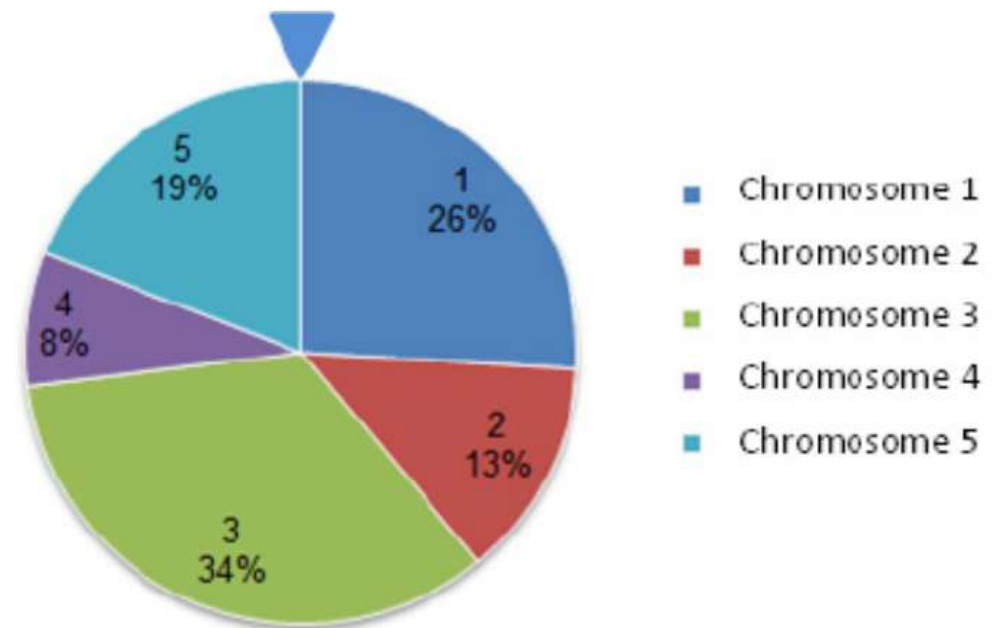
- ▶ Um beste Individuen zu finden → man muss diese **bewerten**
- ▶ Das nennt man Fitness
- ▶ Muss durch eine Funktion berechnet werden, die eine Zahl liefert
- ▶ Wird sehr oft aufgerufen → muss effizient sein
- ▶ Ist das **einzigste Bewertungskriterium!**

Selektion

- ▶ Individuen für die nächste Generation bzw. Fortpflanzung werden ausgewählt
- ▶ Je fitter, desto wahrscheinlicher ist die Auswahl
- ▶ nicht zu eng wählen, weil sonst Lösungsraum eingeschränkt wird – brauche Vielfalt
- ▶ Beim Programmieren gibt es zwei gängige Varianten

Roulette Wheel Selection

- ▶ Es wird die **relative Fitness** berechnet
- ▶ Also $p = \text{Fitness} / \text{Gesamtfitness}$
- ▶ je fitter, desto höher die Wahrscheinlichkeit
- ▶ Wie Rouletterad:
 - man dreht
 - beim wem Zeiger steht, der wird ausgewählt



Tournament Selection

- ▶ Zufällig N Individuen auswählen (unabhängig von Fitness)
- ▶ "treten in einem Turnier gegeneinander an"
- ▶ der Beste davon wird ausgewählt (also der **Fitteste**)
- ▶ ist schneller als Roulette Wheel Selection

Elitism

- ▶ Optional kann man die **Eliten überleben** lassen
- ▶ Man sortiert nach Fitness und übernimmt die N besten in die nächste Generation

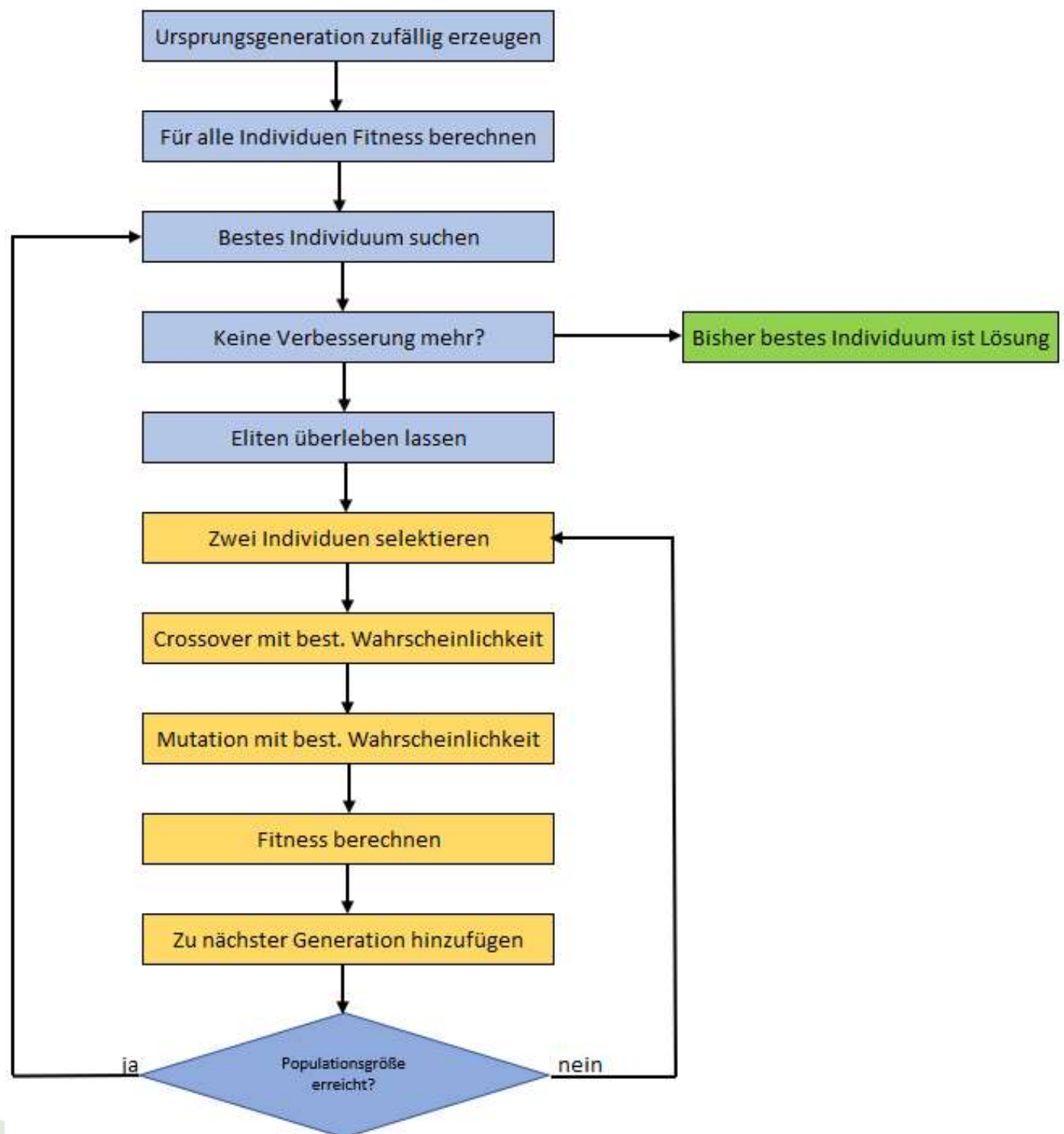
Abbruchkriterium

- ▶ Man kennt in der Regel die beste Lösung nicht
- ▶ ➔ man kann nicht mit dieser vergleichen
- ▶ Daher Abbruch, wenn über mehrere Generationen keine Verbesserung mehr

Vorbereitung

- ▶ Klasse für das Individuum schreiben
- 1. Parameter als Gene darstellen → **List<int>**
- 2. Konstruktor erzeugen, der zufällige Gene erzeugt
- 3. Methode **Mutate()** programmieren
- 4. Methode **Crossover()** programmieren
- 5. **Fitnessfunktion** programmieren
- 6. Unplausible Individuen "reparieren"

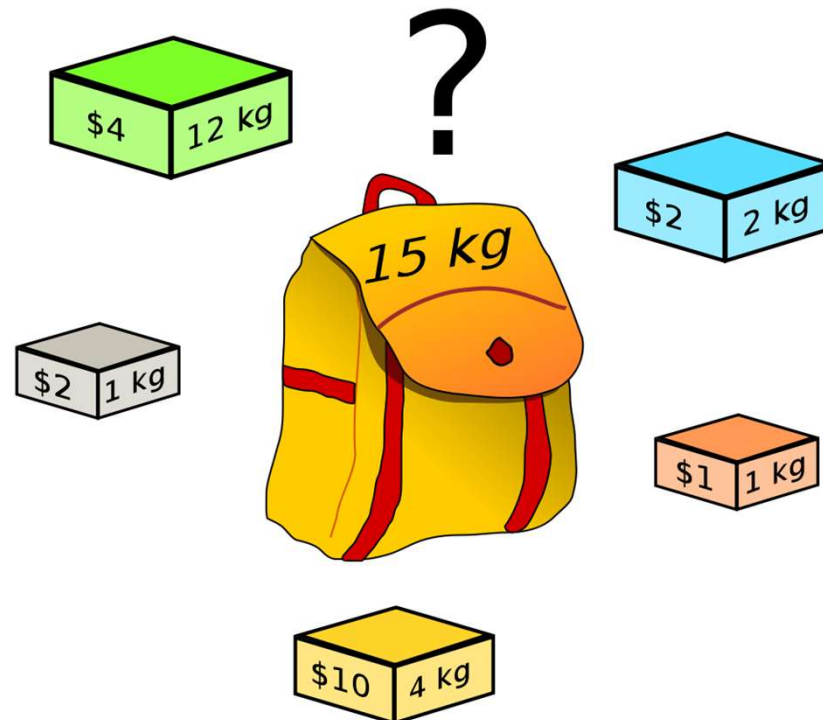
Ablauf



RUCKSACKPROBLEM

Problembeschreibung

- ▶ Man hat Dinge mit Größe und Wert
- ▶ Rucksack hat nur bestimmte Kapazität
- ▶ Welche Dinge muss man einpacken, damit man den größten Wert erreicht?

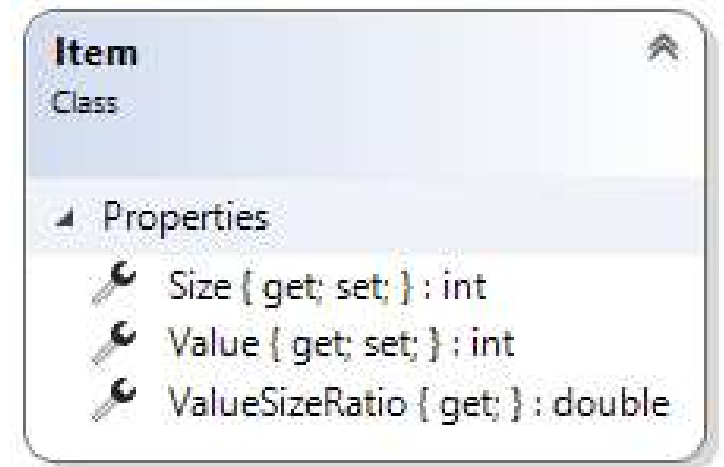


konventionell

- ▶ 3 Möglichkeiten
 - Der Größe nach einpacken bis voll
 - Die Wertvollsten zuerst
 - Die relativ Wertvollsten der Reihe nach

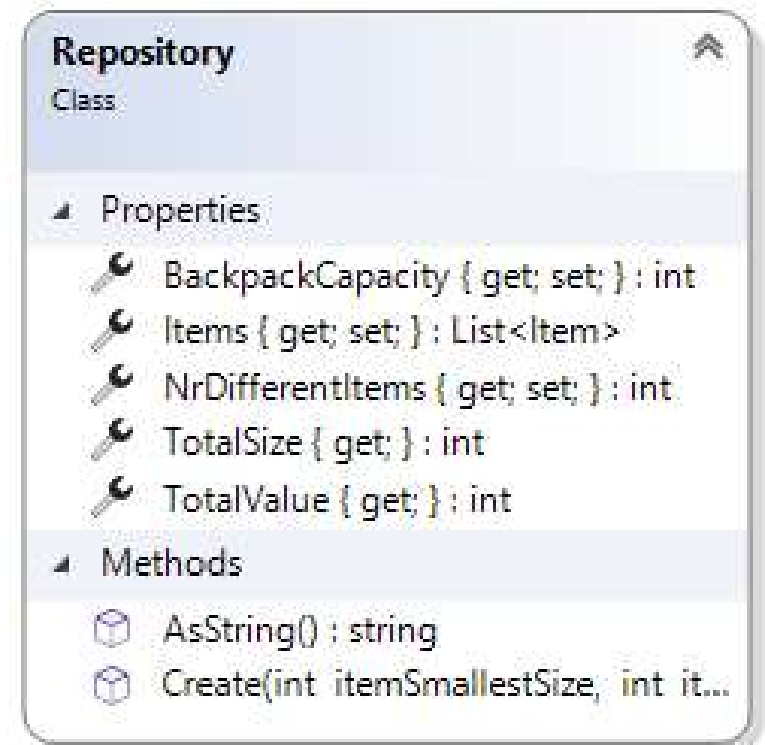
Klasse Item

- ▶ Properties
 - Größe
 - Wert
- ▶ Calculated Property
 - Relativer Wert



Klasse Repository

- ▶ Properties
 - List von Items
 - Kapazität des Rucksacks
 - Anzahl verfügbarer Items
 - Gesamtgröße
 - Gesamtwert
- ▶ Methoden
 - Zufällig erzeugen



The image shows a UML Class Diagram for the 'Repository' class. The class is titled 'Repository' and is labeled as a 'Class'. It has two main sections: 'Properties' and 'Methods'. The 'Properties' section lists five attributes: 'BackpackCapacity' (type 'int'), 'Items' (type 'List<Item>'), 'NrDifferentItems' (type 'int'), 'TotalSize' (type 'int'), and 'TotalValue' (type 'int'). Each attribute is preceded by a wrench icon, indicating it is a mutable property. The 'Methods' section lists two methods: 'AsString()' (return type 'string') and 'Create(int itemSmallestSize, int it...)' (return type 'int'). Each method is preceded by a cube icon, indicating it is a static method.

```
classDiagram
    class Repository {
        BackpackCapacity : int
        Items : List<Item>
        NrDifferentItems : int
        TotalSize : int
        TotalValue : int
        AsString() : string
        Create(int itemSmallestSize, int it...) : int
    }
```

```
public static void Create(
    int itemSmallestSize, int itemBiggestSize,
    int itemSmallestValue, int itemBiggestValue)
```

Klasse Randomizer

- ▶ Zufallszahlen sind nicht wirklich zufällig
- ▶ Selber Startwert → garantiert gleiche Zufallszahlen
- ▶ → Random-Objekt nicht immer selbst erzeugen
- ▶ Sondern Singleton verwenden



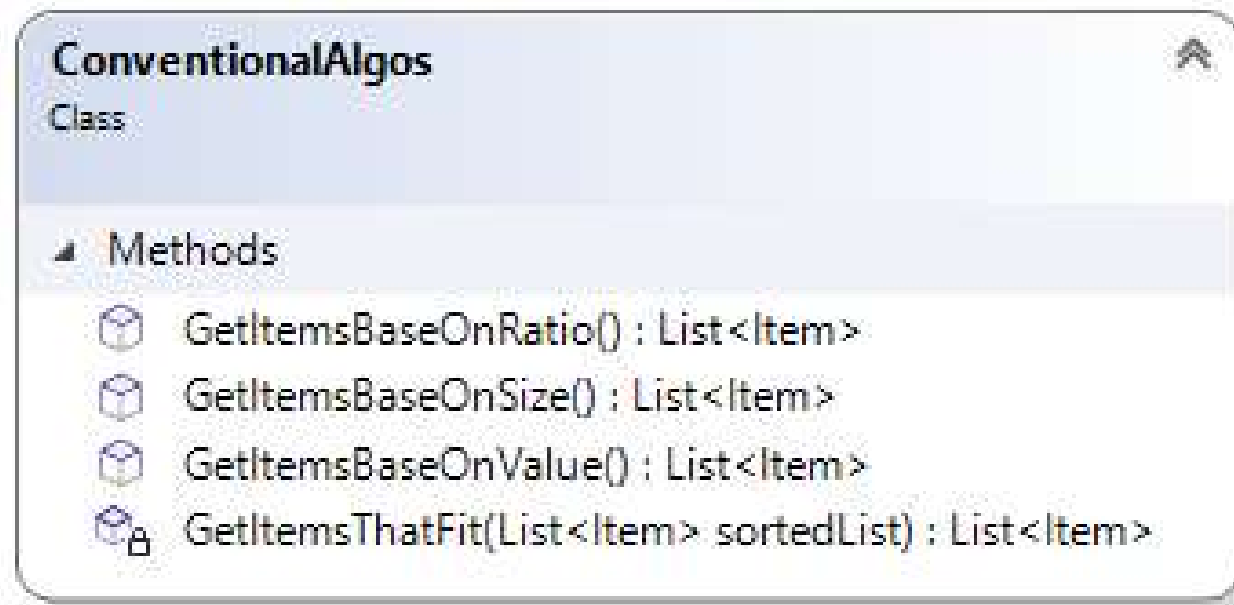
```
class Randomizer
{
    private static Random random = null;
    public static Random Rnd
    {
        get
        {
            if (random == null) random = new Random(666);
            return random;
        }
    }
    private Randomizer() { }
}
```

Klasse ConventionalAlgos

► Konventionell lösen

- Der Größe nach einpacken bis voll
- Die Wertvollsten zuerst
- Die relativ Wertvollsten

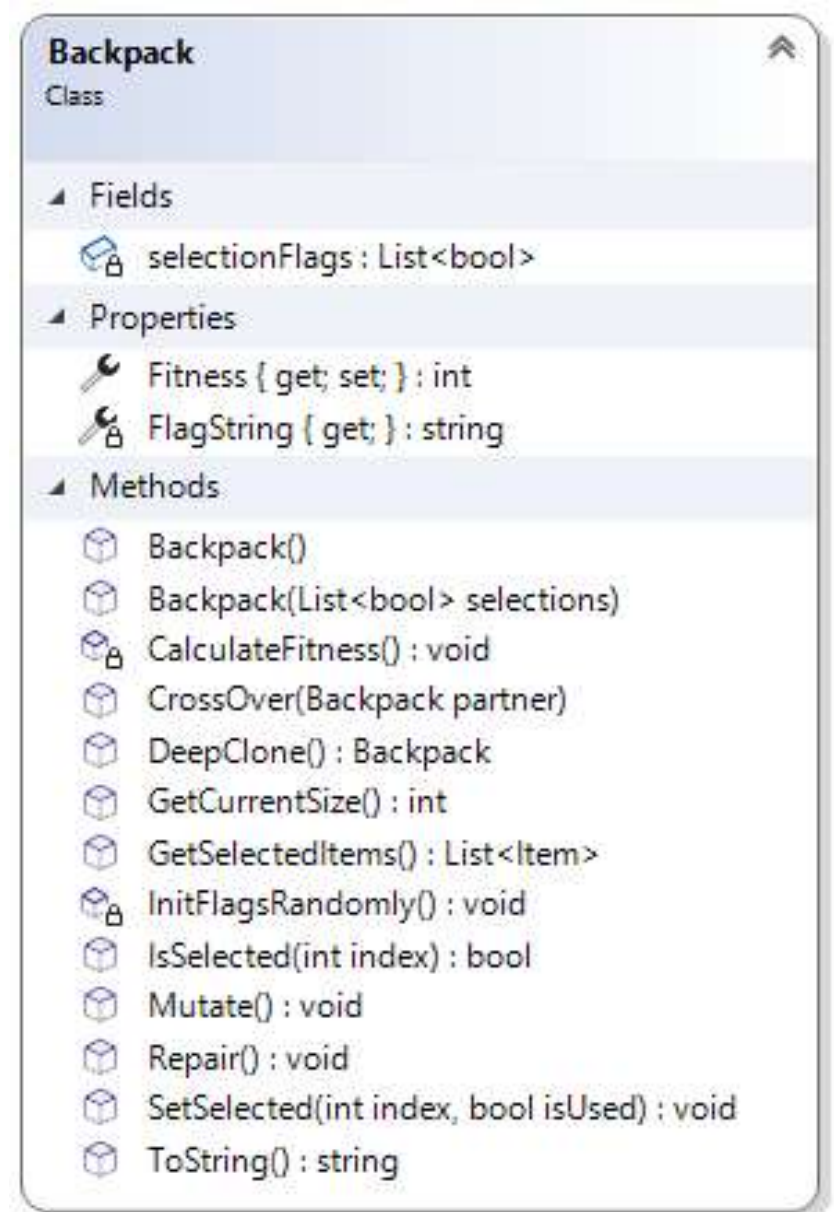
► Daher:



```
private static List<Item> GetItemsThatFit(List<Item> sortedList)
```


Klasse Backpack

- ▶ Variable
 - Item #i einpacken?
→ List<bool>
- ▶ Properties
 - Fitness
- ▶ Methoden
 - helpers: GetCurrentSize,...
 - CalculateFitness
 - Mutate
 - Crossover
 - Repair (nach Crossover bzw. Mutate)















The screenshot shows a Java IDE window titled "Backpack" with a "Class" tab. The class structure is displayed in a tree view with the following sections:

- Fields**
 - selectionFlags : List<bool>
- Properties**
 - Fitness { get; set; } : int
 - FlagString { get; } : string
- Methods**
 - Backpack()
 - Backpack(List<bool> selections)
 - CalculateFitness() : void
 - CrossOver(Backpack partner)
 - DeepClone() : Backpack
 - GetCurrentSize() : int
 - GetSelectedItems() : List<Item>
 - InitFlagsRandomly() : void
 - IsSelected(int index) : bool
 - Mutate() : void
 - Repair() : void
 - SetSelected(int index, bool isUsed) : void
 - ToString() : string

```
public (Backpack, Backpack) CrossOver(Backpack partner)
{
    Backpack child1 = this.DeepClone();
    Backpack child2 = partner.DeepClone();
}
```

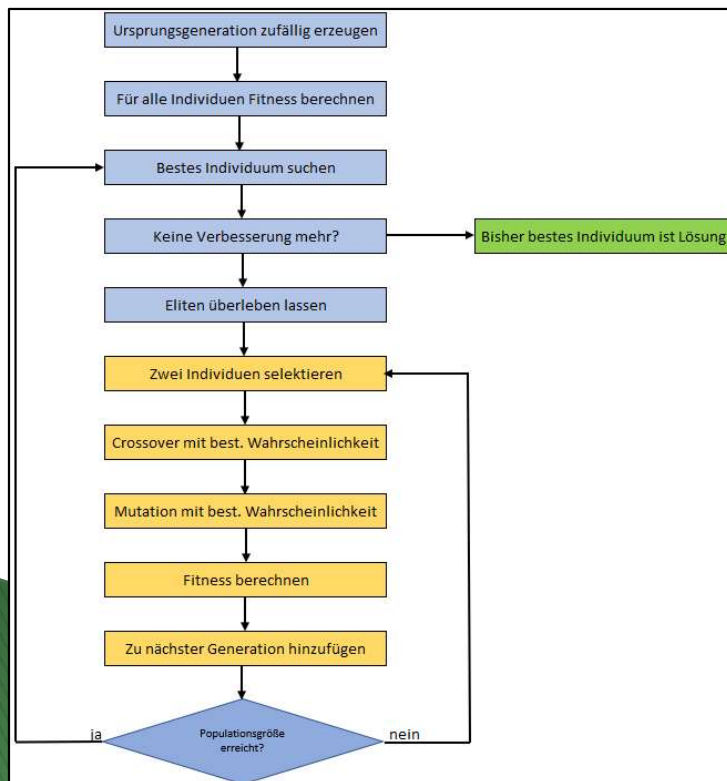

Algorithmus Variable

- ▶ Konfiguration/Wahrscheinlichkeiten
- ▶ aktuelle Generation
- ▶ Abbruchkriterium
- ▶ beste Lösung

	bestBackpack : Backpack
	bestFitnessAllTime : int
	crossoverRate : double
	currentGeneration : List<Backpack>
	currentTotalFitness : int
	elitismRate : double
	generationNrOfBest : int
	mutationRate : double
	nrGenerationsChangeLimit : int
	populationSize : int
	SelectCandidate : Func<Backpack>
	tournamentSize : int
	UseTournamentSelection : bool

Algorithmus Methoden

- ▶ Hauptmethode: **FindBest()**
- ▶ Ursprungsgeneration: **SeedGenerationZero()**
- ▶ Bestes Individuum: **GetBestAndCalculateFitness()**
- ▶ Abbruchcheck: **ShouldTerminate()**
- ▶ Eliten überleben: **AddElite()**
- ▶ Fortpflanzung: **CreateNextGeneration()**



- 🔒 AddElite(List<Backpack> nextGeneration) : void
- 🔒 CreateNextGeneration() : List<Backpack>
- 🔒 FindBest() : Backpack
- 🔒 GeneticAlgorithmEngine(int populationSize, int crossoverPercent, double r
- 🔒 GetAverageFitnessCurrentSolution() : double
- 🔒 GetBestAndCalculateTotalFitness() : Backpack
- 🔒 SeedGenerationZero() : void
- 🔒 SelectCandidateByRouletteWheel() : Backpack
- 🔒 SelectCandidateViaTournament() : Backpack
- 🔒 ShouldTerminate(int generationNr, Backpack bestOfThisGeneration) : bool