<u>Evolutionary algorithms</u>

- **Simple genetic algorithms**
- **Evolutionary Strategies**
- **Genetic Programming**

Partially based on slides by Thomas Bäck

Heuristic Search

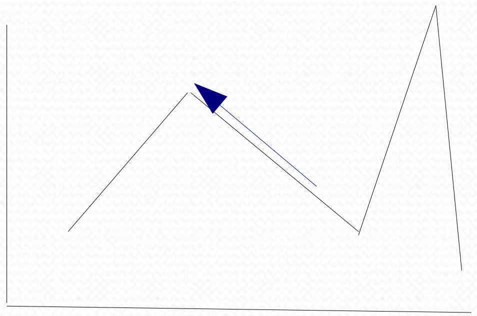
- SAT solvers, CP solvers, ILP solvers:
  - find exact solutions to discrete constraint optimization problems
  - can be time consuming
- Heuristic solvers:
  - employ “heuristics”: guidelines for finding good solutions quickly
  - don't find exact solutions
  - can be much faster
  - can deal with problems that are numerical and not in a “nice” form (eg., linear)

Examples in Fuzzy Logic

- When learning a fuzzy classifier from training data we need to find:
  - Parameters of membership functions
  - Attributes to put in rules

- When finding the parameters that maximize the output of a fuzzy system, we need to find numerical values

Hill-Climbing



- Hill-climbing is arguably the simplest heuristic algorithm

1. S = arbitrary candidate solution
2. S' = solutions in the neighborhood of S
3. **if** best solution in S' is not better than S **then** stop
4. let S be the best solution in S'
5. go to 2.

Neighborhood Search

- Important choice in hill-climbing: which neighborhoods to consider
  - Add a small value to each coordinate? Substruct a small value from each coordinate?

$$(x_1, x_2, \ldots, x_n) \rightarrow$$

$$(x_1 + \epsilon, x_2, \ldots, x_n)$$

$$(x_1 - \epsilon, x_2, \ldots, x_n)$$

$$(x_1, x_2 + \epsilon, \ldots, x_n)$$

$$(x_1, x_2 - \epsilon, \ldots, x_n)$$

$$\vdots$$

Large Neighborhood Search

- Iteratively select a random subset of variables of limited size, find an optimal assignment for these variables, assuming the others are fixed
  - Requires the availability of an algorithm to solve the intermediate problems optimally (linear programming, CP, ..)

$$(x_1, x_2, \ldots, x_n)$$
$$\downarrow$$
$$(?, ?, x_3, \ldots, x_n)$$
$$\downarrow$$
$$(x_1', x_2', x_3, \ldots, x_n)$$
$$\downarrow$$
$$(x_1', x_2', ?, x_4, \ldots, x_n)$$
$$\downarrow$$
$$(x_1', x_2', x_3', x_4, \ldots, x_n)$$

Other Well-known Heuristic Search Strategies

- Simulated annealing
- Tabu search
- Evolutionary algorithms
  - genetic algorithms
  - genetic programming
  - evolutionary strategies
- Artificial ants
- Particle swarms

Advantages of GAs

- Evolution and natural selection has proven to be a robust method
- A “black box” approach that can easily be applied to many optimization problems
- GAs can be easily parallelized and run on multiple machines

Some definitions

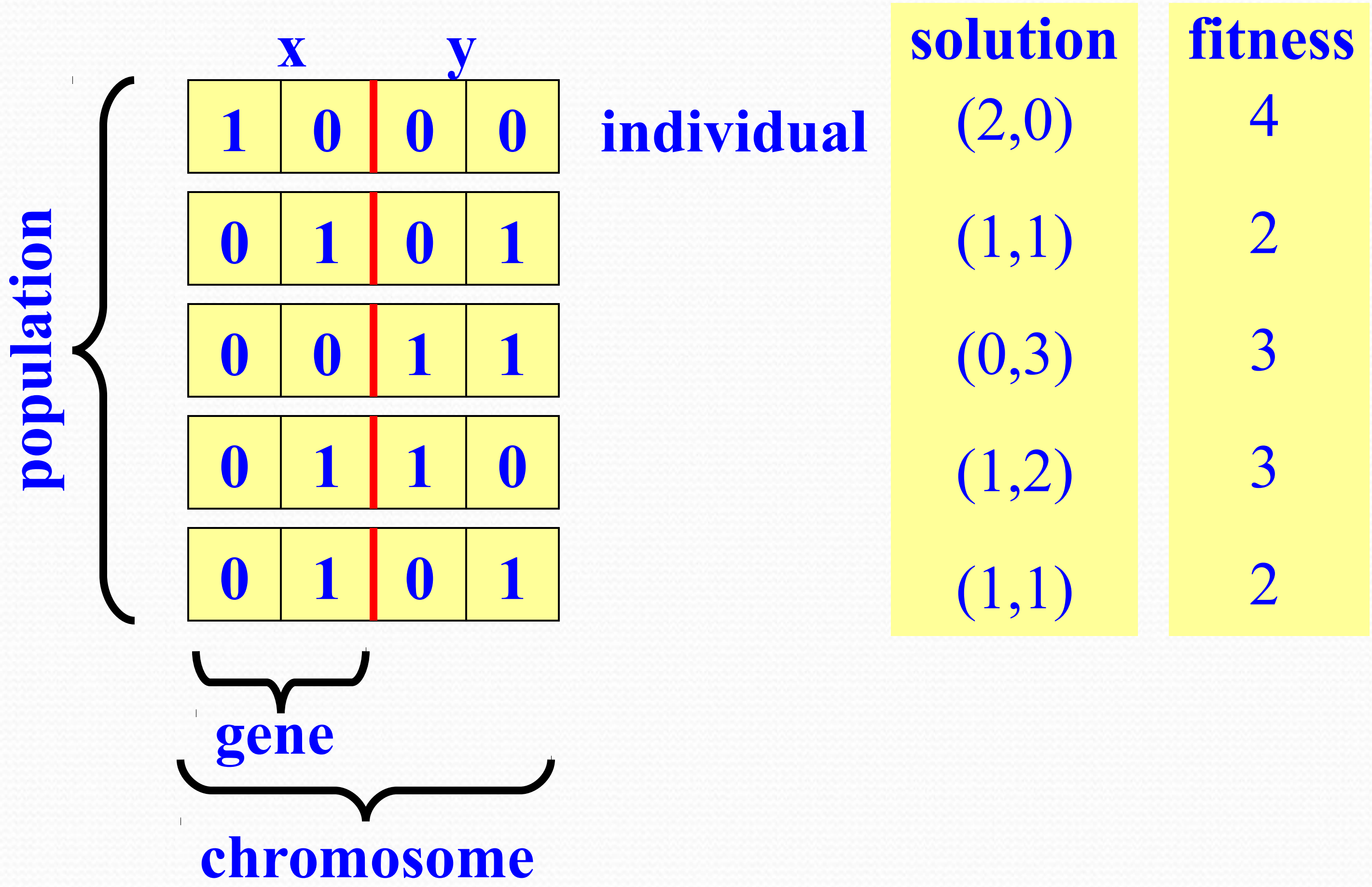
- **Population**: a collection of solutions for the studied (optimization) problem
- **Individual**: a single solution in a GA
- **Chromosome (genotype)**: representation for a single solution
- **Gene**: part of a chromosome, usually representing a variable as part of the solution

Some definitions

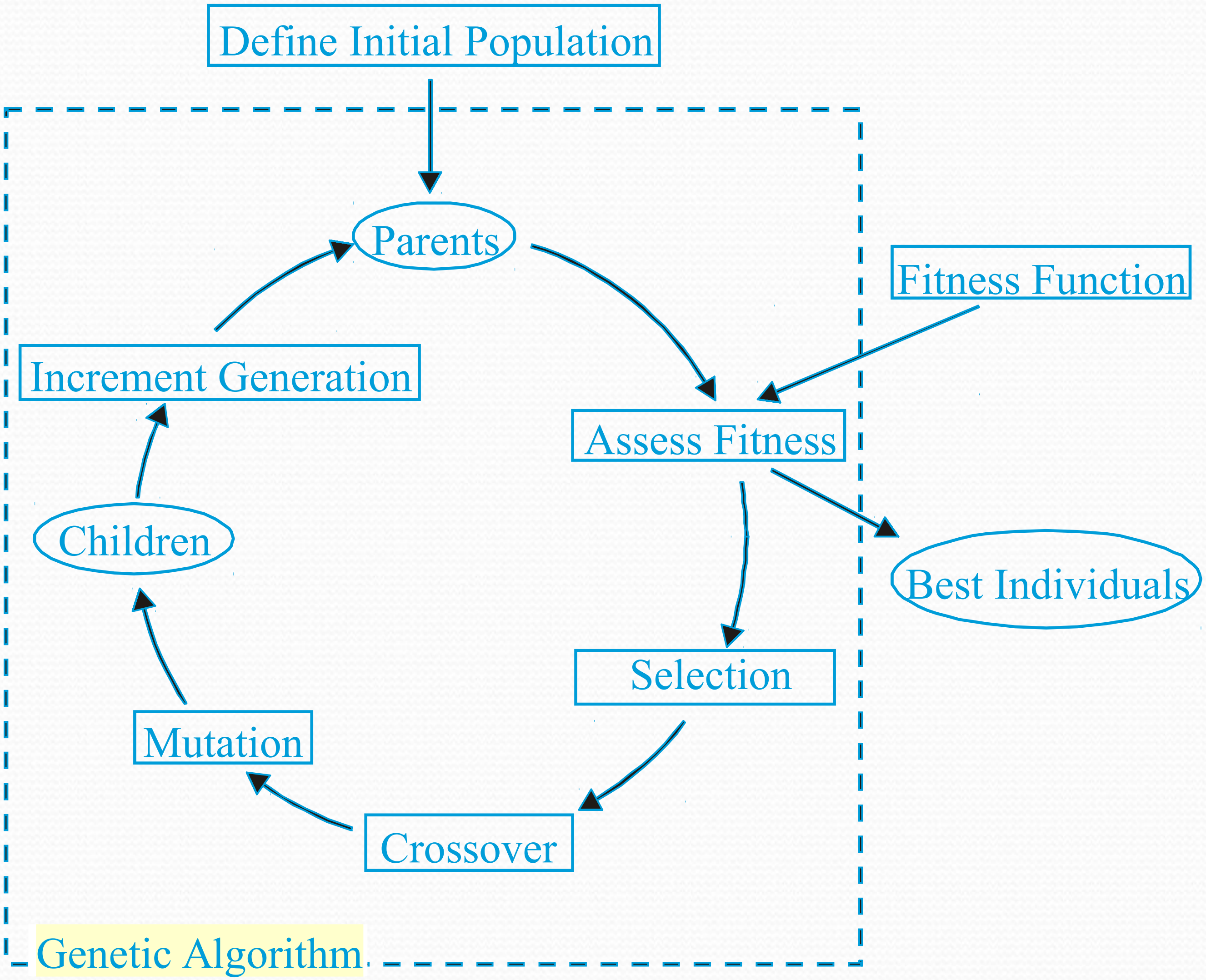
- **Encoding**: conversion of a solution to its equivalent representation (chromosome)
- **Decoding**: conversion of a chromosome (**genotype**) to its equivalent solution (phenotype)
- **Fitness**: scalar value denoting the suitability of a solution

GA terminology

Generation t



Genetic algorithm



Pseudo code

- Initialize population *P*:
  - E.g. generate random *p* solutions
- Evaluate solutions in *P*:
  - determine for all $h \in P$, Fitness(*h*)
- **While** terminate is FALSE
  - Generate new generation *P* using genetic operators
  - Evaluate solutions in *P*
- **Return** solution $h \in P$ with the highest Fitness

Termination criteria

- Number of generations
  (restart GA if best solution is not satisfactory)
- Fitness of best individual
- Average fitness of population
- Difference of best fitness (across generations)
- Difference of average fitness (across generations)

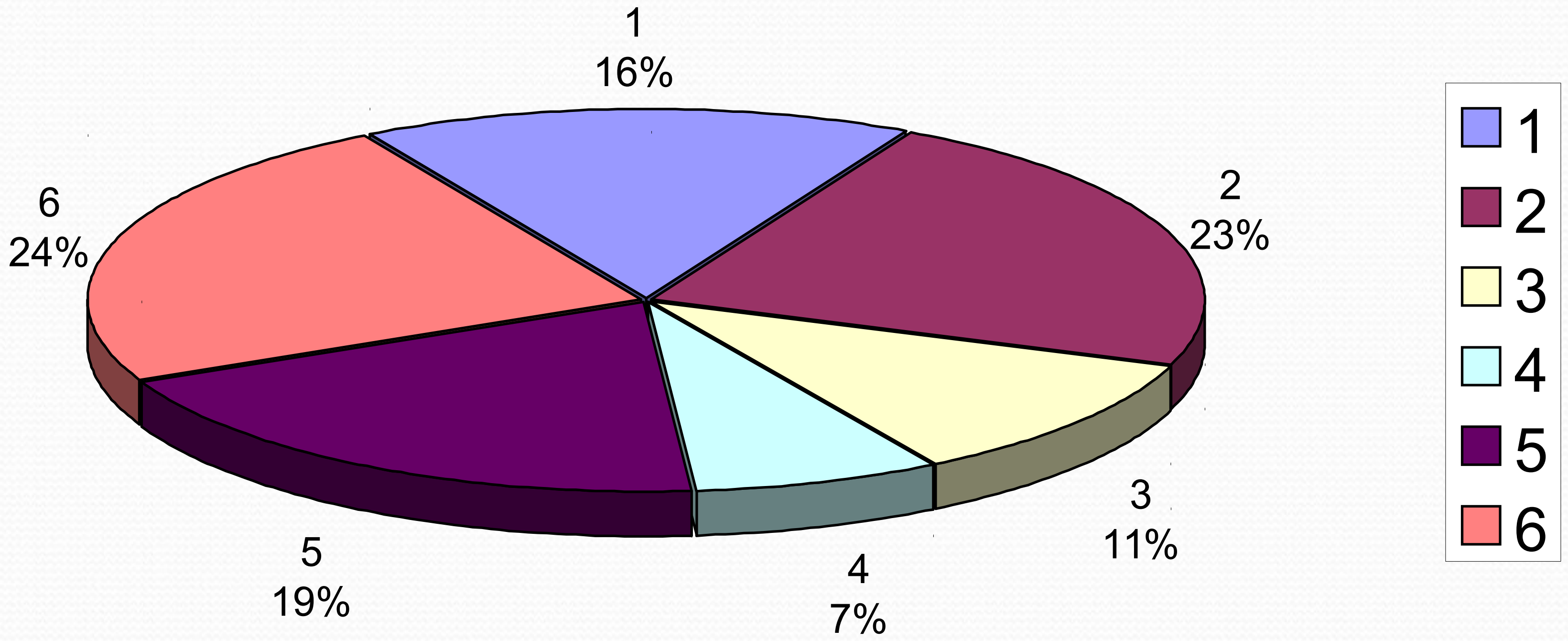
Reproduction

Three steps:

- Selection
- Crossover
- Mutation

In GAs, the population size is often kept constant. The programmer is free to choose which methods to use for all three steps.

Roulette-wheel selection



Roulette-wheel selection

	individuals	fitness		selection
p = 0.16	01100	34	→	01100
p = 0.23	10001	48		10001
p = 0.11	11010	23		10001
p = 0.07	00111	15		11000
p = 0.19	11000	41		10110
p = 0.24	10110	50		10110

Sum = 211

Cumulative probability: 0.16, 0.39, 0.50, 0.57, 0.76, 1.00

Tournament selection

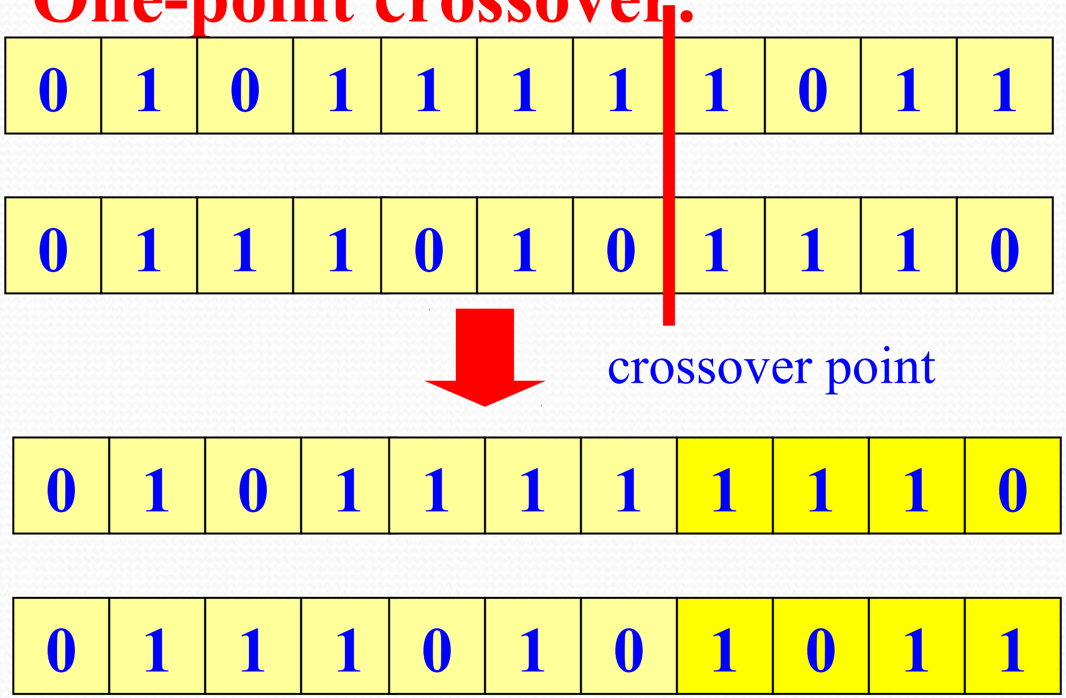
- Select pairs randomly
- Fitter individual wins
  - deterministic
  - probabilistic
    - constant probability that the better individual wins
    - probability of winning depends on fitness

Tournament selection can also be combined with roulette-wheel selection.

Crossover

- Exchange parts of chromosome with a crossover probability (p_c is usually about 0.8)
  - i.e., with probability $1-p_c$ no crossover takes place
- Select crossover points randomly

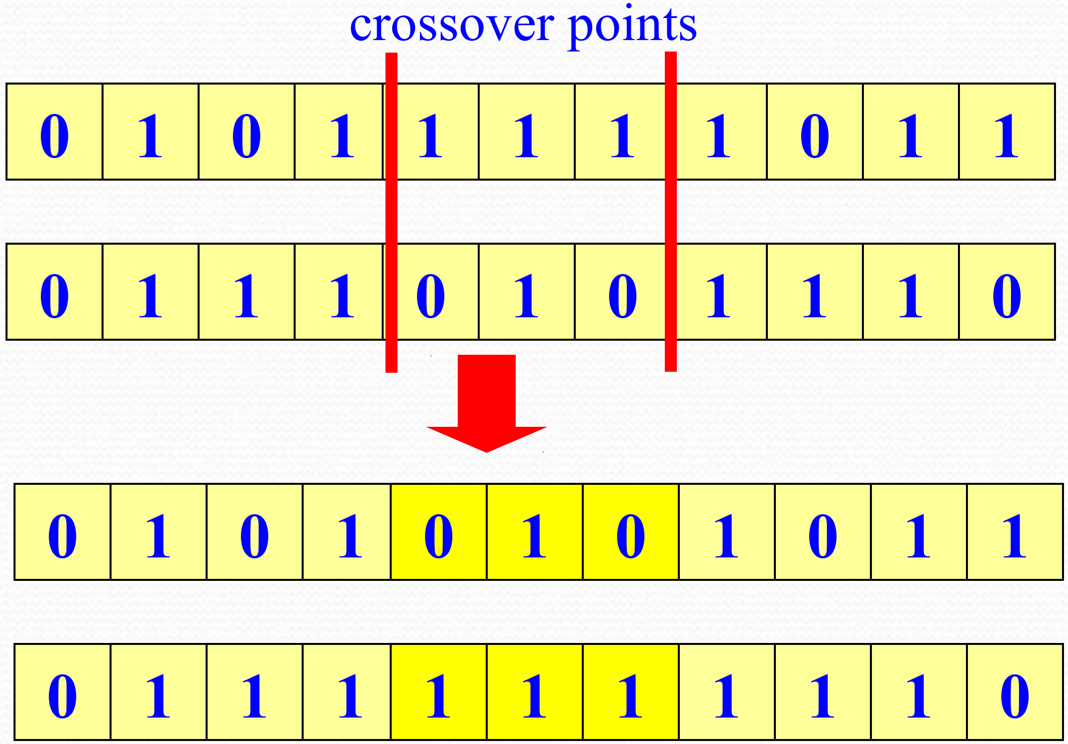
One-point crossover:



N-point crossover

- Select N points for exchanging parts
- Exchange multiple parts

Two-point crossover:



Uniform crossover

- Exchange bits using a randomly generated mask

0	1	0	1	0	0	1	0	0	1	1	mask

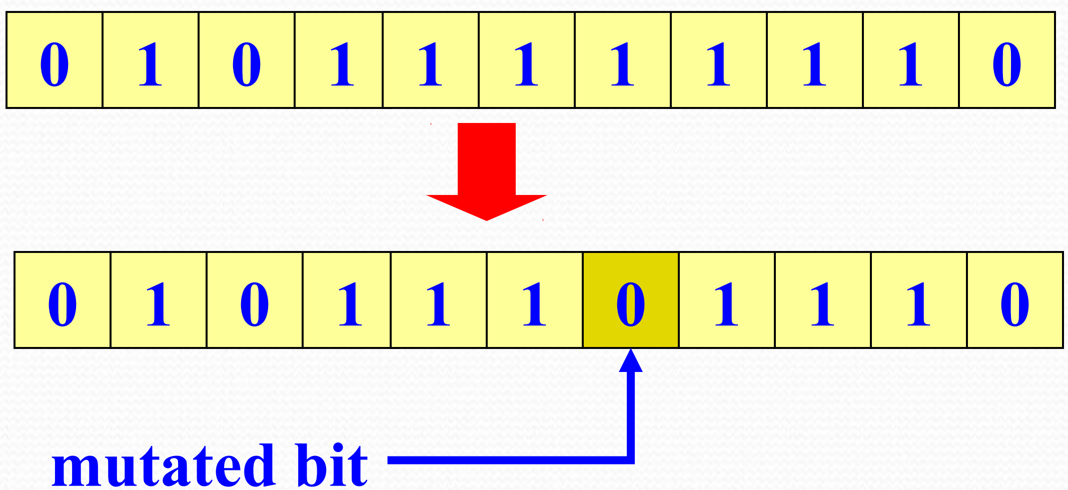
0	1	0	1	1	1	1	1	0	1	1
0	1	1	1	0	1	0	1	1	1	0

0	1	0	1	1	1	0	1	0	1	0
0	1	1	1	0	1	1	1	1	1	1

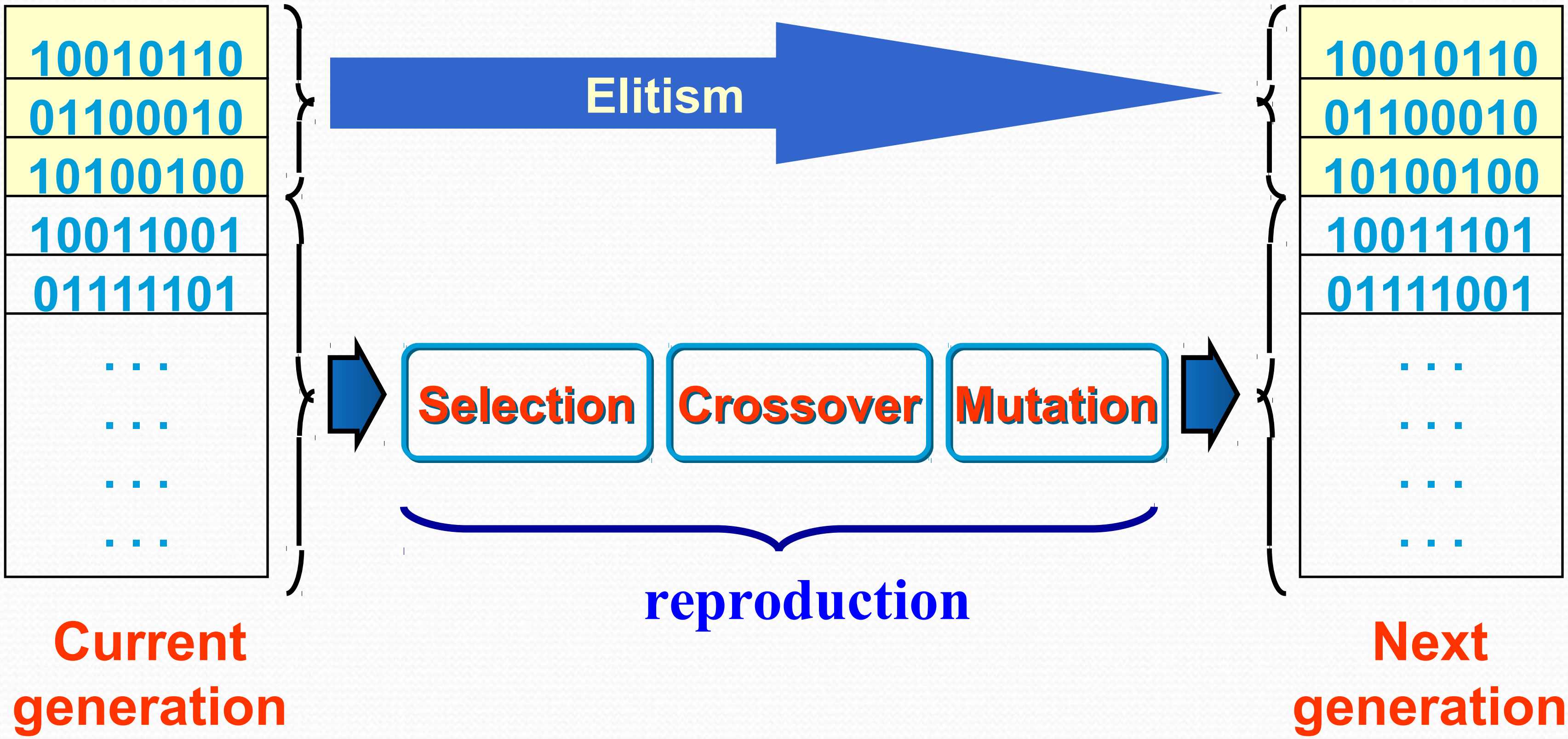
Mutation

- Crossover is used to search the solution space
- Mutation is needed to escape from local optima
- Introduces genetic diversity
- Mutation is rare (p_m is about 0.005)

Uniform mutation:



GA iteration



Encoding and decoding

- Common coding methods
  - “standard” binary integer coding
  - Gray coding (binary)
  - real valued coding (*evolutionary strategies*)
  - tree structures (*genetic programming*)

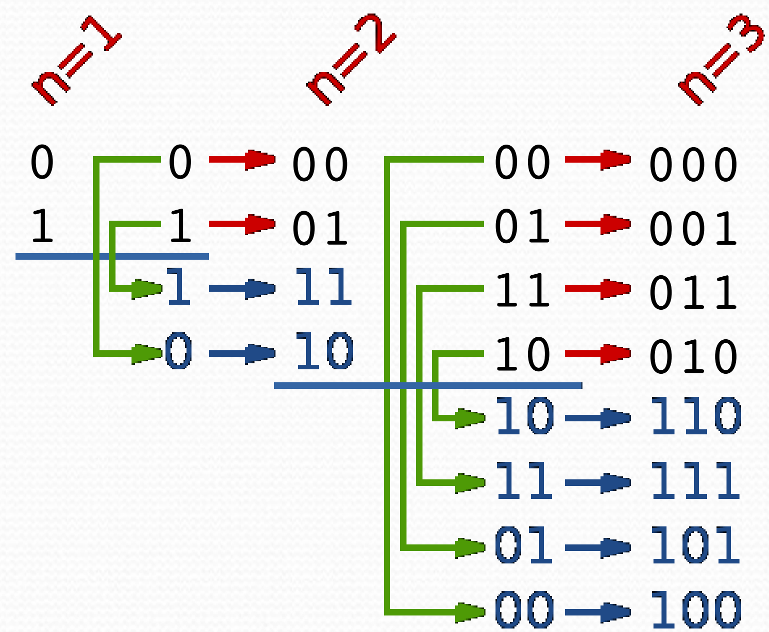
Gray Coding

- Aim: binary coding of integers such that integers x and y for which $|x-y|=1$ only differ in one bit

Dec	Gray	Binary
0	000	000
1	001	001
2	011	010
3	010	011
4	110	100
5	111	101
6	101	110
7	100	111

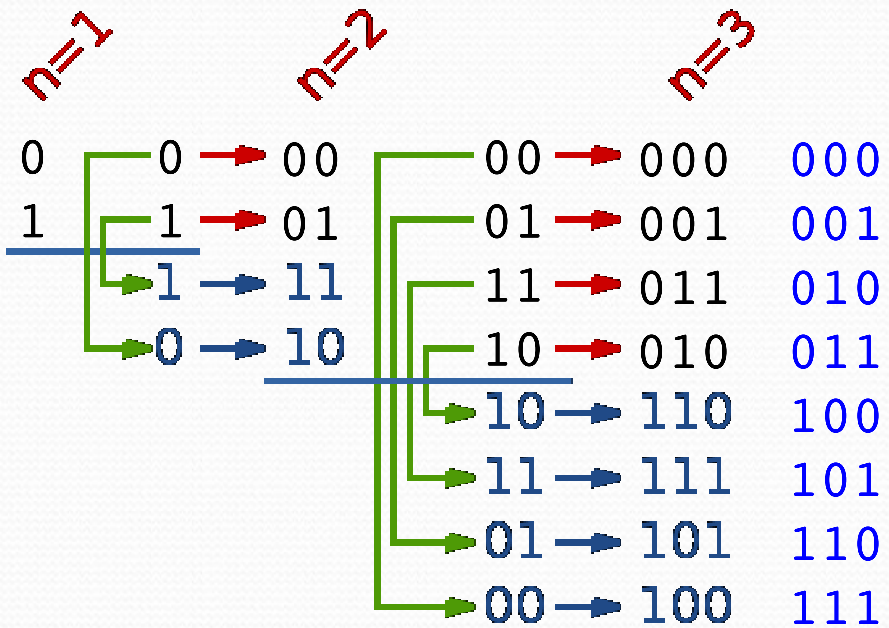
Gray Coding

- Codes for *n*=1: (i.e., integers 0, 1)
  0 1
- Codes for *n*=2: (i.e., integers 0, 1, 2, 3)
  Reflected entries for *n*=0:
  1 0
  Prefix old entries with 0:
  00 01
  Prefix reflected entries with 1:
  11 10
  Codes hence:
  00 01 11 10
- Codes for *n*=3: (i.e., integers 0, 1, 2, …, 7)
  Reflected entries for *n*=2:
  10 11 01 00
  Codes hence:
  000 001 011 010 110 111 101 100



Gray Coding

- Given a “normal” bit representation, how to calculate the Gray code?



bitstring → Gray

10100 → 11110

10101 → 11111

10110 → 11101

11001 → 10101

A bit flips in the Gray code iff the bit before it has value 1 in the original code.

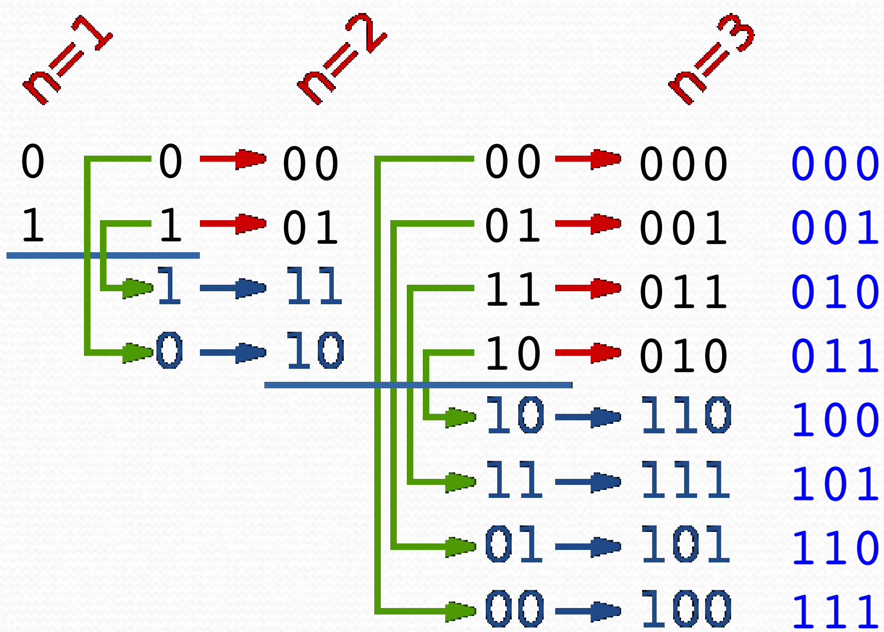
Gray Coding

- Source code in Python for calculating Gray code:

```
def binaryToGray(num):
        return (num >> 1) ^ num
```

Gray Coding

- Given a Gray code, how to calculate a "normal" bit representation?



bitstring → Gray
10100 → 11110
10101 → 11111
10110 → 11101
11001 → 10101

A bit flips in the "normal" code (as compared to the Gray code) iff the bit before it has value 1 in the "normal" code.

Gray Coding

- Gray coding does not avoid that integers far away from each other can have similar codes

  00000=0

  10000=31

  → Mutation can still change numbers a lot

- Gray coding only ensures that there always is a one-bit mutation to transform integer *x* into integer *x+1* or *x-1*.