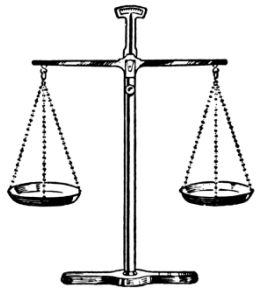


Genetic Algorithms

Genetic Algorithms - History

Originated in the fifties, but developed in the sixties by three distinct research groups:

- Pioneered by L. J. Fogel in the US
- J. H. Holland in the 1970's called his method a genetic algorithm
- I. Rechenberg and H.P. Schwefel introduced evolution strategies

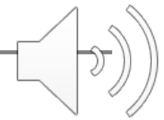
Got popular in the late 1980's

Based on ideas from Darwinian Evolution

Can be used to solve a variety of problems that are not easy to solve using other techniques

Evolution in the real world

- Each cell of a living thing contains ***chromosomes*** - strings of *DNA*
- Each chromosome contains a set of ***genes*** - blocks of DNA
- Each gene determines some aspect of the organism (like eye colour)
- A collection of genes is sometimes called a ***genotype***
- A collection of aspects (like eye colour) is sometimes called a ***phenotype***
- Reproduction involves recombination of genes from parents and then small amounts of ***mutation*** (errors) in copying
- The ***fitness*** of an organism is how much it can reproduce before it dies
- Evolution based on "survival of the fittest"



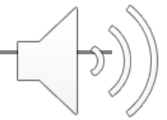
Genetic Algorithms

Generate a *set* of random solutions

Repeat

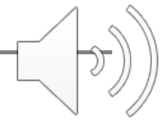
- Test each solution in the set (rank them)
- Remove some bad solutions from set
- Duplicate some good solutions
  - make small changes to some of them

Until best solution is good enough



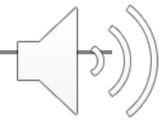
How do you encode a solution?

- Obviously this depends on the problem!
- GA's *often* encode solutions as fixed length "bitstrings" (e.g. 101110, 111111, 000101)
- Each bit represents some aspect of the proposed solution to the problem
- For GA's to work, we need to be able to "test" any string and get a "score" indicating how "good" that solution is

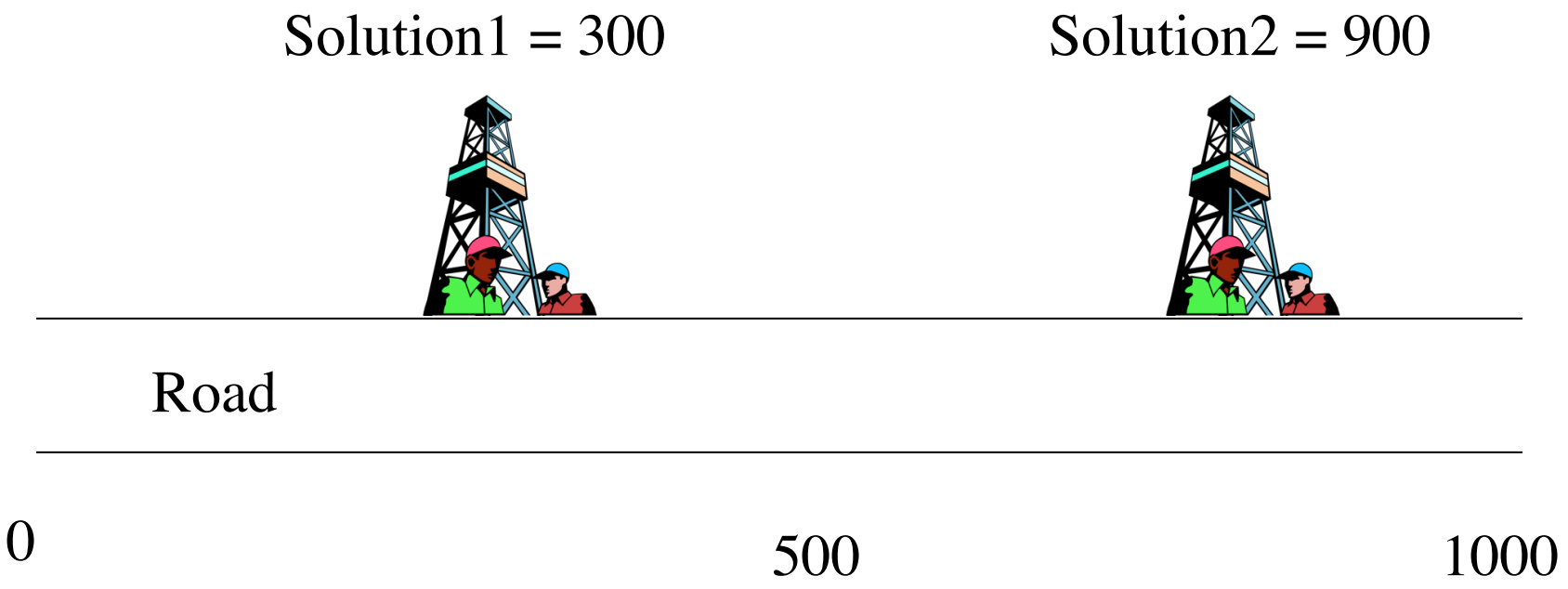


Silly Example - Drilling for Oil

- Imagine you had to drill for oil somewhere along a single 1km desert road
- Problem: choose the best place on the road that produces the most oil per day
- We could represent each solution as a position on the road
- Say, a whole number between [0..1000]

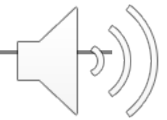


Where to drill for oil?



Digging for Oil

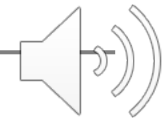
- The set of all possible solutions [0..1000] is called the *search space* or *state space*
- In this case it's just one number but it could be many numbers or symbols
- Often GA's code numbers in binary producing a bitstring representing a solution
- In our example we choose 10 bits which is enough to represent 0..1000



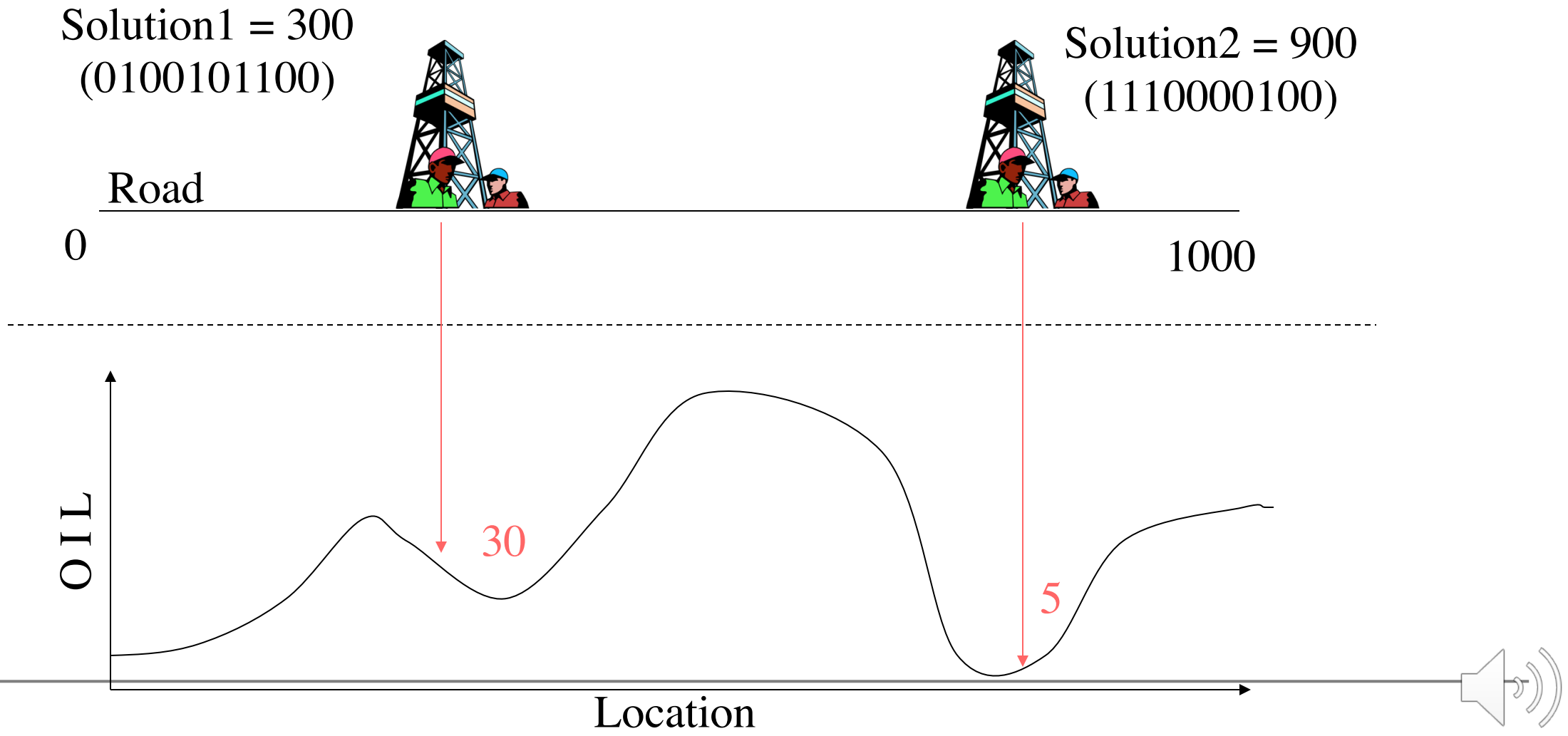
Convert to binary string

	512	256	128	64	32	16	8	4	2	1
900	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**1**	**0**	**0**
300	**0**	**1**	**0**	**0**	**1**	**0**	**1**	**1**	**0**	**0**
1023	1	1	1	1	1	1	1	1	1	1

In GA's these encoded strings are sometimes called "*genotypes*" or "*chromosomes*" *and the individual bits are sometimes called "genes"*



Drilling for Oil

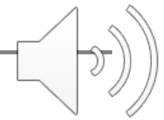


Summary

Our example is really optimisation over a function f(x) where we adapt the parameter x

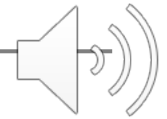
We have seen how to:

- represent possible solutions as a number and encode them in binary strings
- generate a score for each number given a *function* of "how good" each solution is - this is often called a *fitness function*

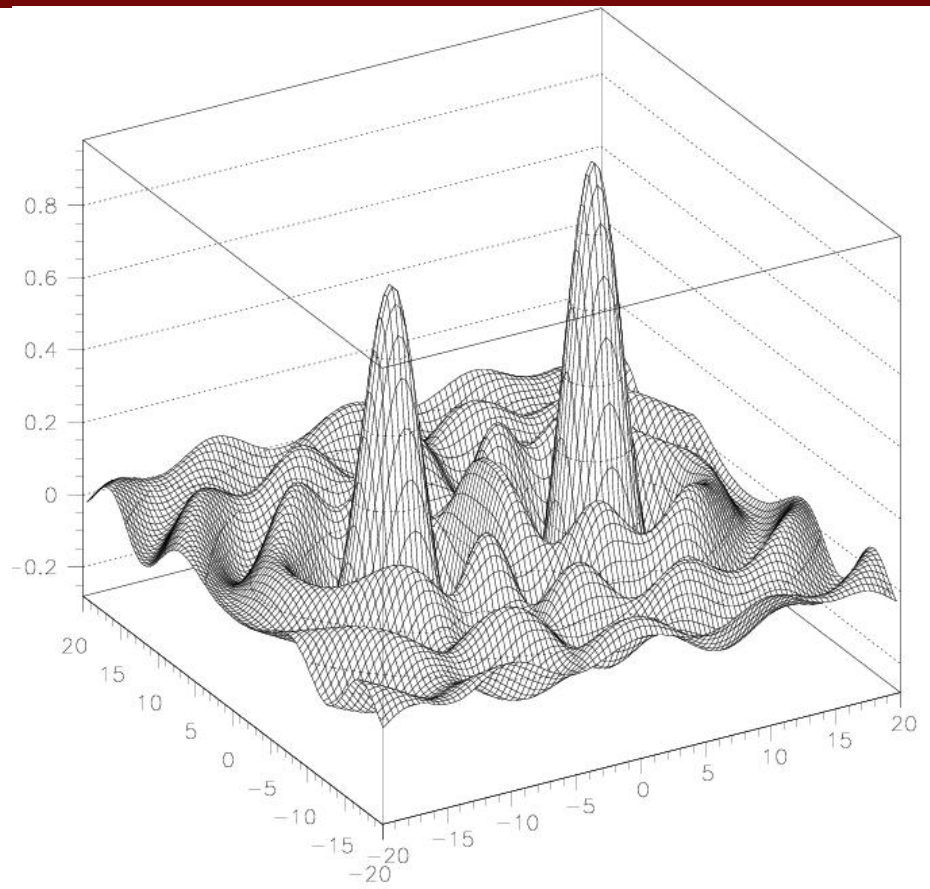


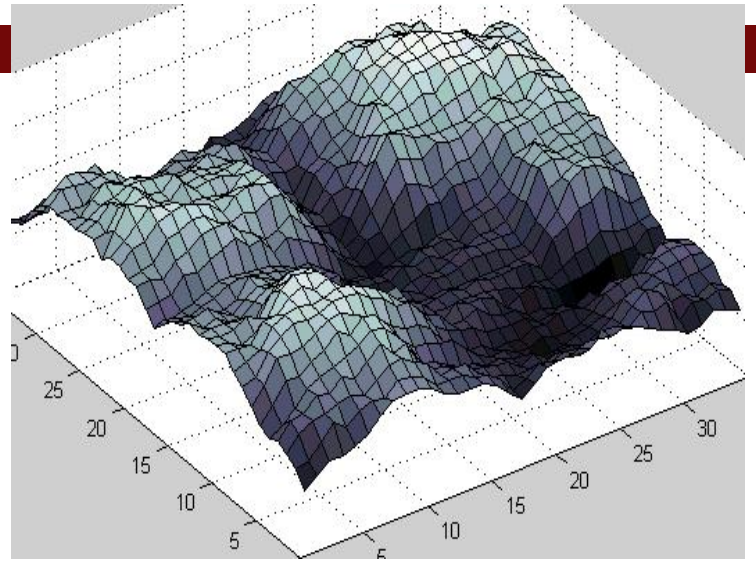
Search Space

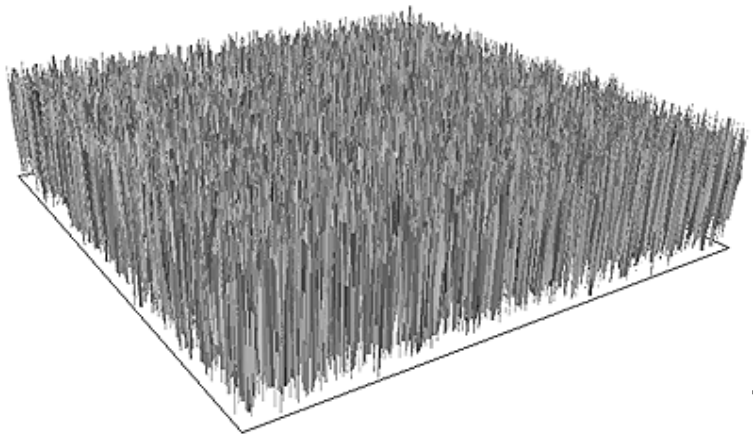
- For a simple function f(x) the search space is one dimensional.
- But by encoding several values into the chromosome many dimensions can be searched e.g. two dimensions f(x,y)
- Search space can be visualised as a surface or *fitness landscape* in which fitness dictates height
- Each possible genotype is a point in the space
- A GA tries to move the points to better places (higher fitness) in the space



Fitness landscapes

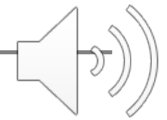






Search Space

- Obviously, the nature of the search space dictates how a GA will perform
- A completely random space would be bad for a GA
- Also GA's can get stuck in local maxima if search spaces contain lots of these
- Generally, spaces in which small improvements get closer to the global optimum are good



Back to the (GA) Algorithm

Generate a *set* of random solutions

Repeat

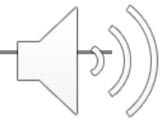
Test each solution in the set (rank them)

Remove some bad solutions from set

Duplicate some good solutions

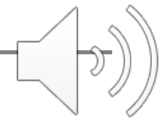
make small changes to some of them

Until best solution is good enough



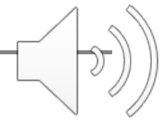
Adding Sex - Crossover

- Although it may work for simple search spaces our algorithm is still very simple
- It relies on random mutation to find a good solution
- It has been found that by introducing "sex" into the algorithm better results are obtained
- This is done by selecting two parents during reproduction and combining their genes to produce offspring



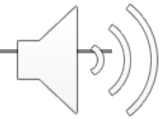
Adding Sex - Crossover

- Two high scoring "parent" bit strings (*chromosomes)* are selected and with some probability (crossover rate) combined
- Producing two new *offspring* (bit strings)
- Each offspring may then be changed randomly (*mutation*)



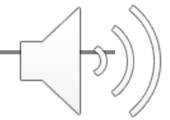
Selecting Parents

- Many schemes are possible as long as better scoring chromosomes more likely selected
- Chromosomes with highest *fitness* should have higher probability to be selected
- "Roulette Wheel" selection can be used:
  - Add up the fitness's of all chromosomes
  - Generate a random number R in that range
  - Select the first chromosome in the population that - when all previous fitness's are added - gives you at least the value R

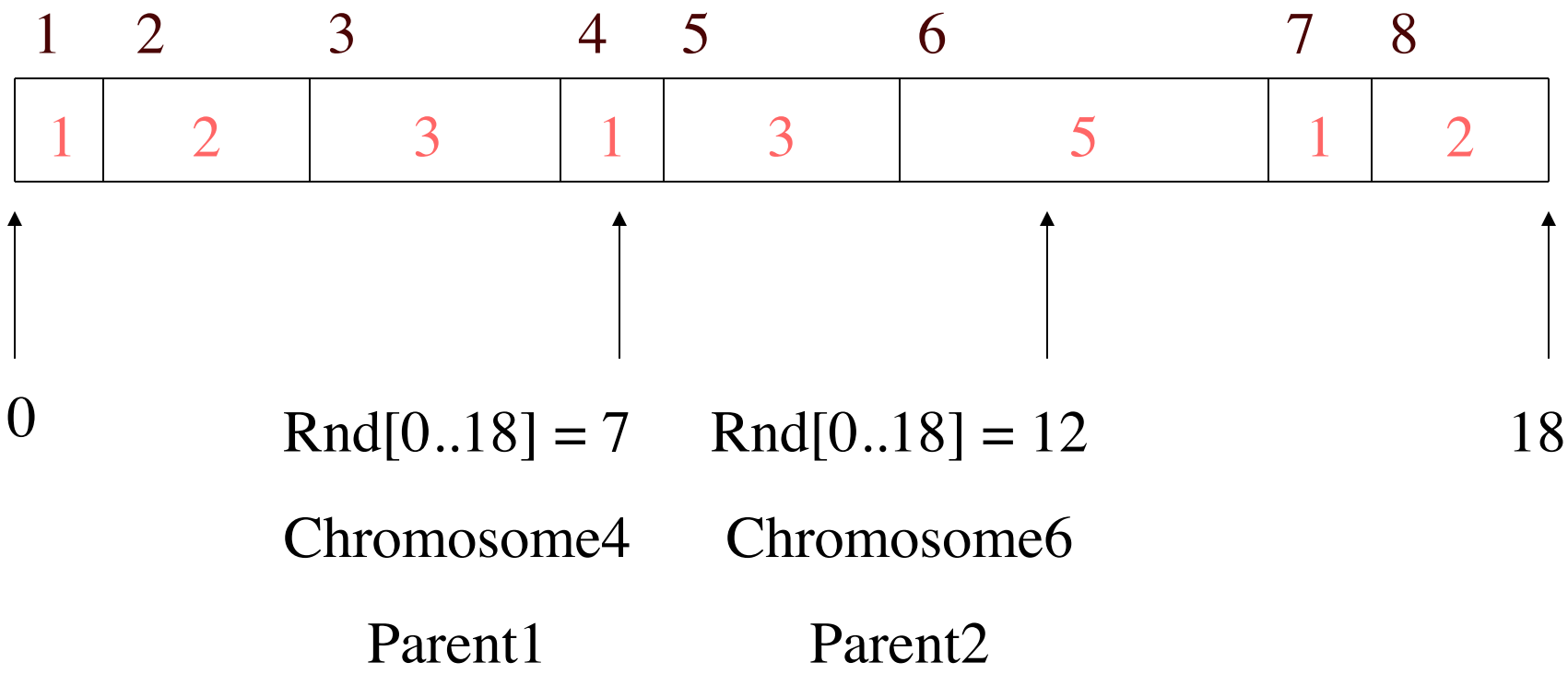


Example population

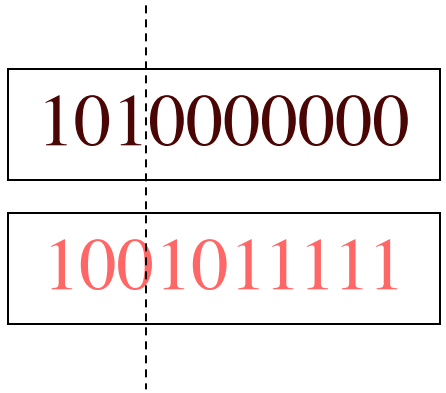
No.	Chromosome	Fitness
1	1010011010	1
2	1111100001	2
3	1011001100	3
4	1010000000	1
5	0000010000	3
6	1001011111	5
7	0101010101	1
8	1011100111	2



Roulette Wheel Selection



Crossover - Recombination



Parent1

Parent2

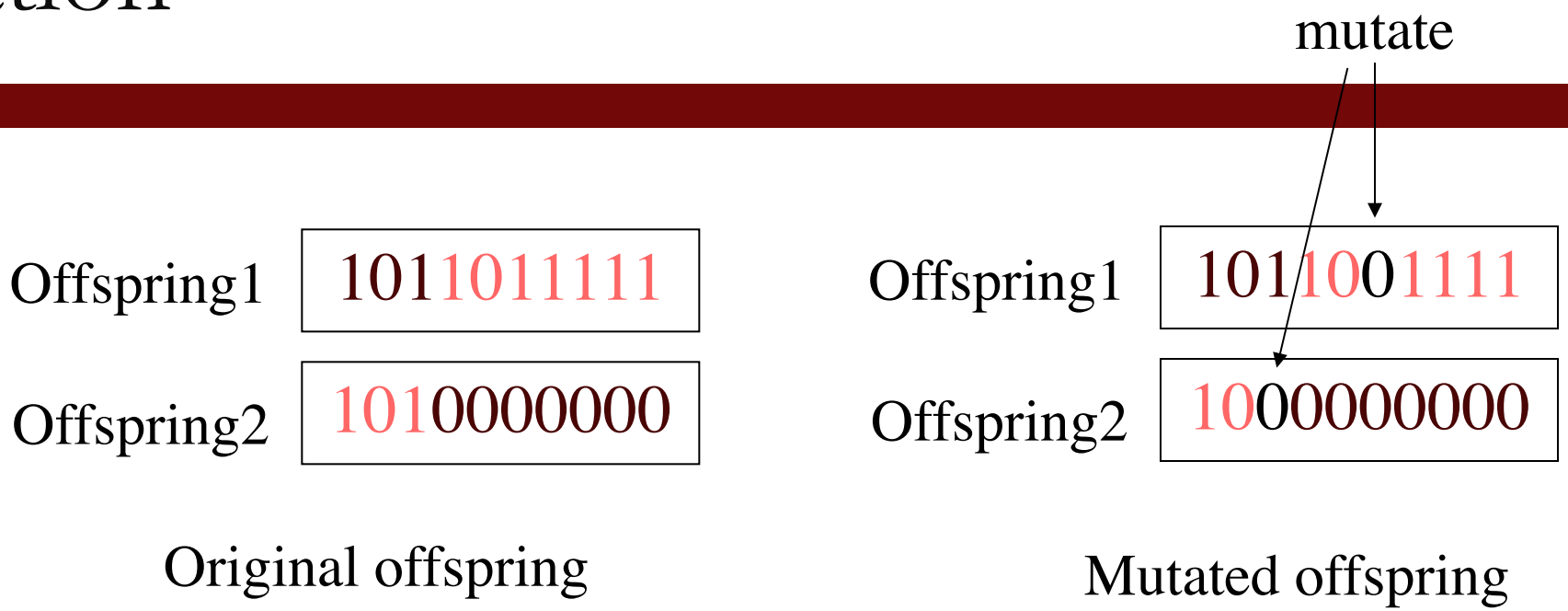
Offspring1 1011011111

Offspring2 1010000000

Crossover single point - random

With some high probability (*crossover rate*) apply crossover to the parents. (*typical values are 0.8 to 0.95*)

Mutation



With some small probability (the *mutation rate*) flip each bit in the offspring (*typical values between 0.1 and 0.001*)

Back to the (GA) Algorithm

Generate a *population* of random chromosomes

Repeat (each generation)

- Calculate fitness of each chromosome
- Repeat
  - Use roulette selection to select pairs of parents
  - Generate offspring with crossover and mutation
- Until a new population has been produced

Until best solution is good enough