

# **Biologically Inspired Computing:**

## **Evolutionary Algorithms: Details, Encoding, Selection**

*'Biologically Inspired Computing'*

*Contents:*

*Steady state and generational EAs, basic ingredients,  
example encodings / operators*

# Recall that:

If we have a hard optimization problem, we can try using an Evolutionary Algorithm (EA) to solve it. To start with we need:

- a. An '**encoding**', which is a way to **represent** any solution to the problem in a computational form (e.g. it could be a string of integers, a 2D array of real numbers, basically any data structure ...)
- b. A **fitness function**  $f(s)$ , which works out a 'fitness' value for the solution encoded as 's'.

Once we have figured out a suitable encoding and fitness function, we can run an EA, which generally looks like this:

0. Initialise: generate a fixed-size population  $P$  of random solutions, and evaluate the fitness of each one.
1. Using a suitable selection method, **select** one or more from  $P$  to be the 'parents'
2. Using suitable '**genetic operators**', generate one or more children from the parents, and evaluate each of them
3. Using a suitable '**population update**' or '**reproduction**' method, replace one or more of the parents in  $P$  with some of the newly generated children.
4. If we are out of time, or have evolved a good enough solution, stop; otherwise go to step 1.

note: one trip through steps 1,2,3 is called a *generation*

# Some variations within this theme

Initialisation: Indeed we could generate ‘random’ solutions – however, depending on the problem, there is often a simple way to generate ‘good’ solutions right from the start. Another approach, where your population size is set to 100 (say) is to generate 10,000 (say) random solutions at the start, and your first population is the best 100 of those.

Selection: lots of approaches – however it turns out that a rather simple selection method called ‘binary tournament selection’ tends to be as good as any other – see later slides.

Genetic Operators: The details of these depends on the encoding, but in general there is always a choice between ‘mutation’ (generate one or more children from a single parent) and ‘recombination’ (generate one or more children from two or more parents). Often both types are used in the same EA.

Population update: the extremes are called: ‘steady state’ (in every generation, at most one new solution is added to the population, and one is removed) and ‘generational’ (in every generation, the entire population is replaced by children). Steady state is commonly used, but generational is better for parallel implementations.

# Encodings

We want to evolve schedules, networks, routes, coffee percolators, drug designs – how do we **encode** or **represent** solutions?

How you **encode** dictates what your **operators** can be, and certain constraints that the operators must meet.

It also influences the size of the search space

Examples at the end of the lecture notes

# Selection in EA

A **selection** method is a way to choose a parent from the population, in such a way that **the fitter an individual is, the more likely it is to be selected.**

*Selection* represents our strategy for deciding which areas of the search space to focus our efforts on.

**Pressure** refers to the level of prioritisation of fitter individuals

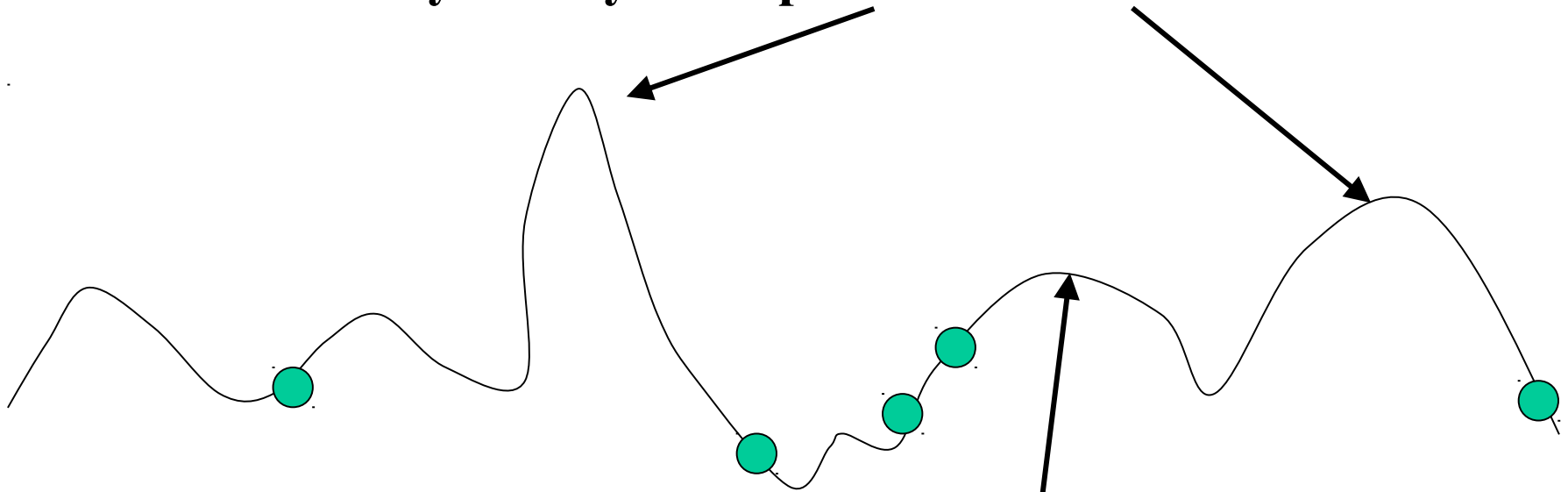
# Selection and Pressure

**Very low pressure selection (e.g. random)**

**No evolutionary 'progress' at all.**

**Suppose the green blobs indicate the initial population.**

**With a modest level of pressure.  
you may end up here or here:**



**With very high pressure  
(e.g. always select best), you will end up here**

# Selection vs. Diversity

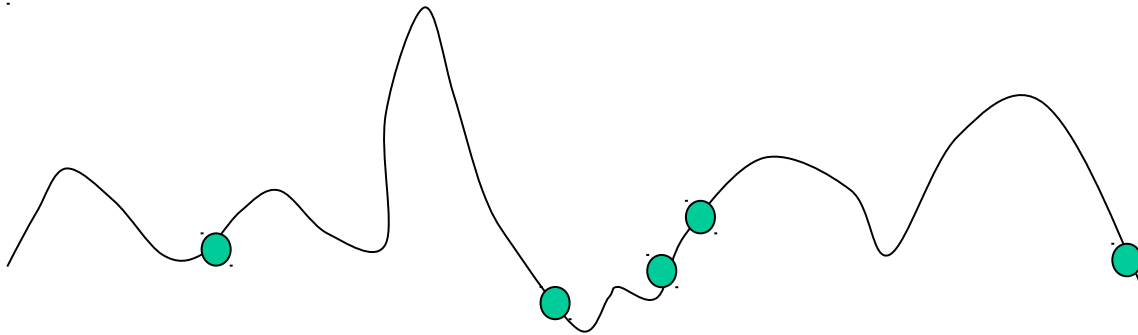
There are two competing factors that need to be balanced in the selection process:

The selection pressure and the genetic diversity

**Selective pressure** is the tendency to select only the best members of the current generation to propagate to the next. It is required to direct the EA to an optimum.

**Genetic diversity** is the maintenance of a diverse solution population. This also requires to ensure that the solution space is adequately searched, especially in the earlier stages of the optimization process.

## How we can ensure this diversity at the beginning of the optimisation process?



Too much selective pressure can lower the genetic diversity so that the global optimum is overlooked and the EA converges to a local optimum.

Yet, with too little selective pressure the EA may not converge to an optimum in a reasonable time. A proper balance between the selective pressure and genetic diversity must be maintained for the EA to converge in a reasonable time to a global optimum.



# Some Selection Methods

1.- Roulette Wheel Selection

2.- Tournament Selection

3.- Rank-based Selection

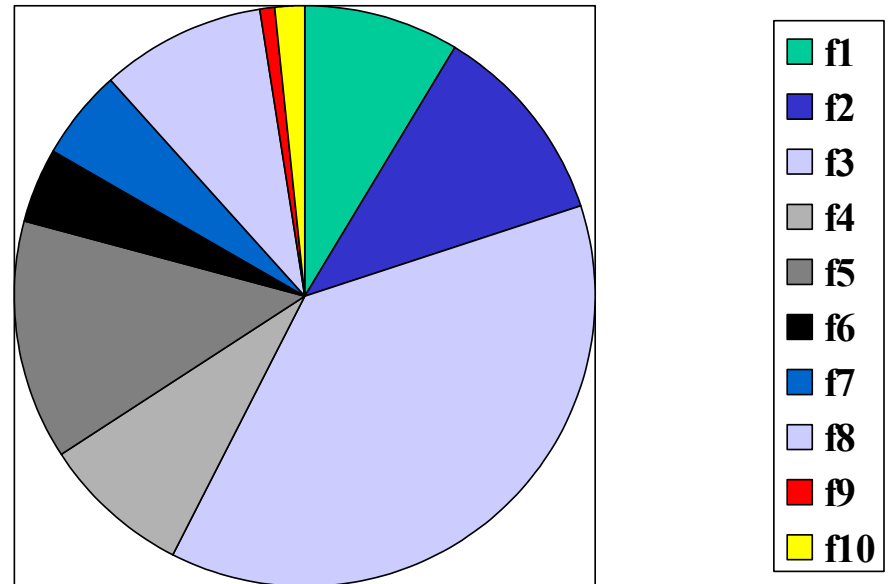
# Roulette Wheel selection

Also called *Fitness Proportionate Selection*

Suppose there are  $P$  individuals with fitnesses  $f_1, f_2, \dots, f_P$ ; and higher values mean better fitness. The probability of selecting individual  $i$  is simply:

$$\frac{f_i}{\sum f_k}$$

*This is equivalent to spinning a roulette wheel with sectors proportional to fitness*



# Problems with Roulette Wheel Selection

Having probability of selection directly proportional to fitness has a nice ring to it. It is still used a lot, and is convenient for theoretical analyses, but:

What about when we are trying to *minimise* the 'fitness' value?

What about when we may have negative fitness values?

We can modify things to sort these problems out easily, but fitprop remains too sensitive to fine detail of the fitness measure. Suppose we are trying to maximise something, and we have a population of 5 fitnesses:

100, 0.4, 0.3, 0.2, 0.1 -- the best is 100 times more likely to be selected than all the rest put together! But a slight modification of the fitness calculation might give us:

200, 100.4, 100.3, 100.2, 100.1 – a much more reasonable situation.

Point is: Fitprop requires us to be very careful how we design the fine detail of fitness assignment.

Other selection methods are better in this respect, and more used now.

# Tournament Selection

Repeat  $t$  times

choose a random individual from the pop  
and remember its fitness

Return the best of these  $t$  individuals (BTR)

Where tournament size =  $t$

This is very tunable, avoids the problems of superfit or superpoor solutions, and is very simple to implement

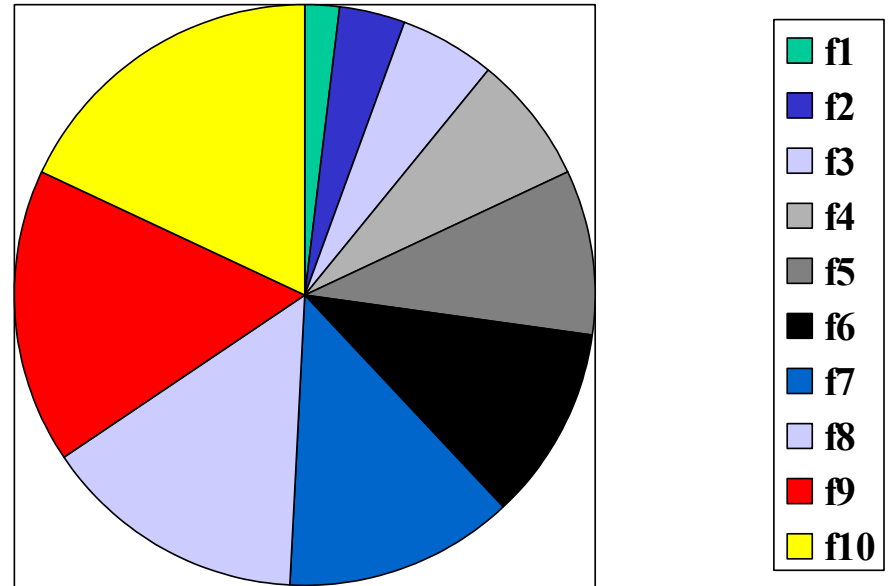
What happens to selection pressure as we increase  $t$ ?

What degree of selection pressure is there if  $t = 10$  and  
*popsize* = 10,000 ?

# Rank Based Selection

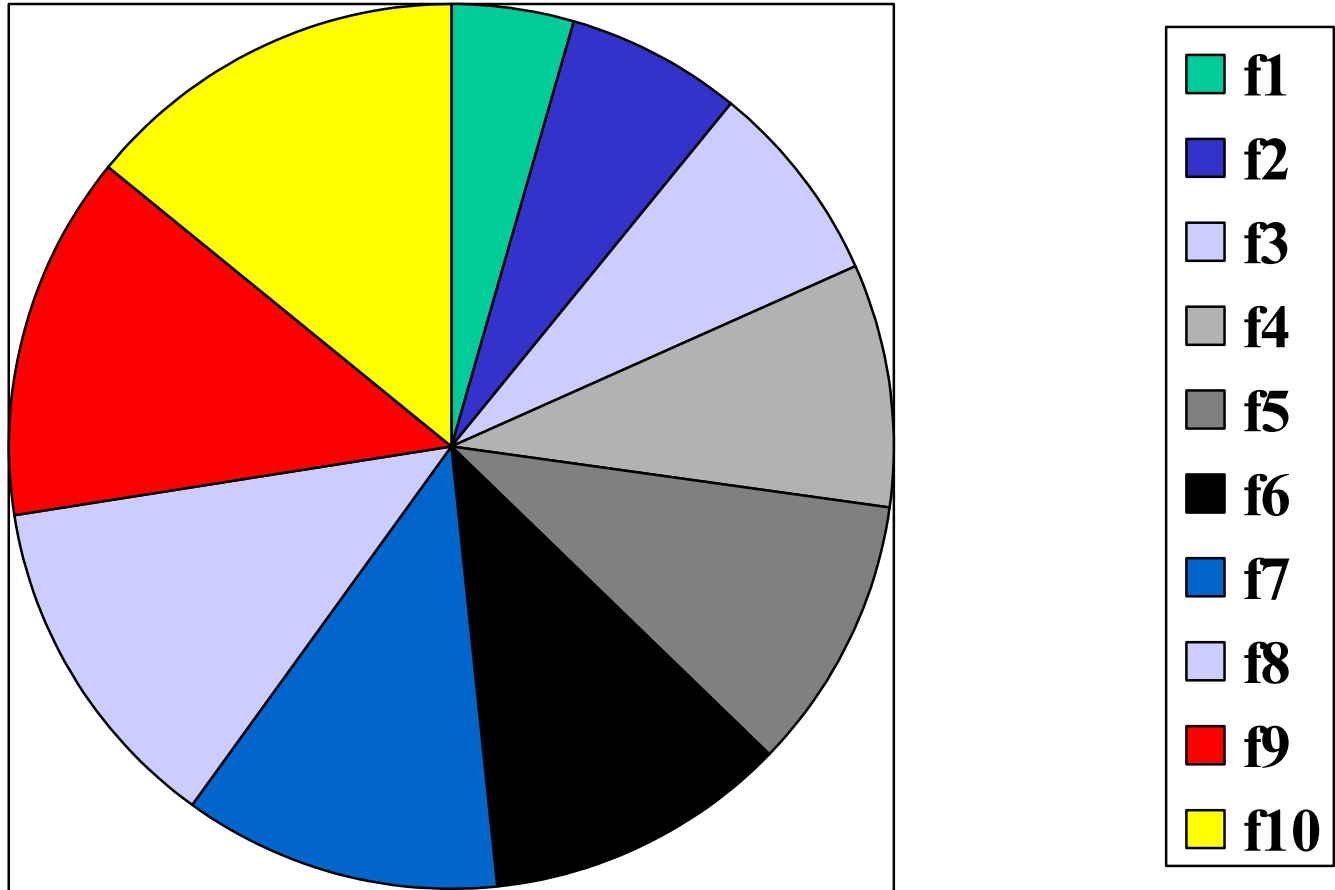
The fitnesses in the pop are *Ranked* (BTR) from *Popsize* (fittest) down to 1 (least fit). The selection probabilities are proportional to rank.

There are variants where the selection probabilities are a function of the rank.



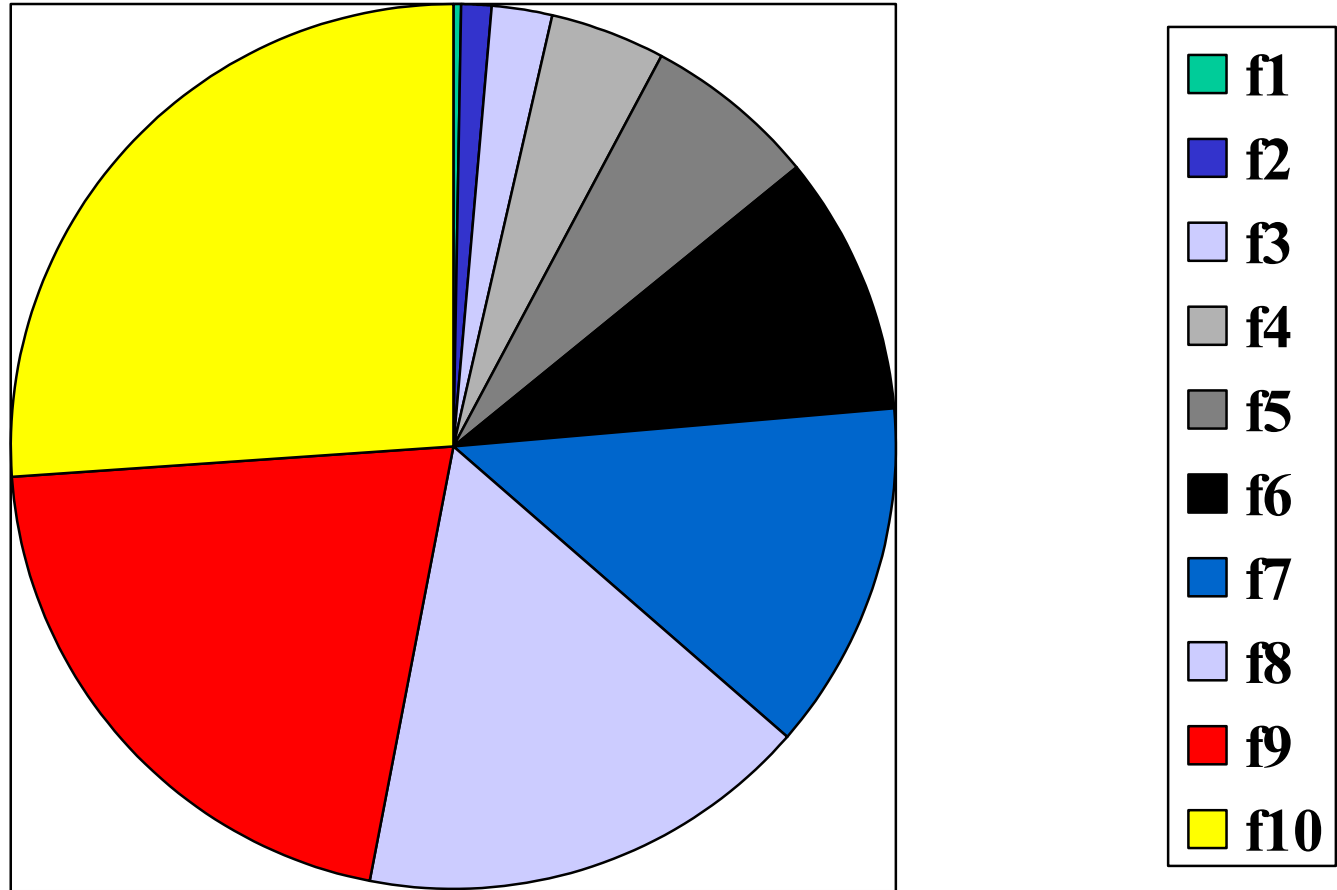
# Rank with low bias

Here, selective fitnesses are based  
on  $rank^{0.5}$



# Rank with high bias

Here, selective fitnesses are based  
on *rank*<sup>2</sup>



# Population Update

Here, the two extremes:

**Steady state:** population only changes slightly in each generation. (e.g. select 1 parent, produce 1 child, add that child to pop and remove the worst)

**Generational:** population changes completely in each generation. (select some [could still be 1] parent(s), produce popsize children, they become the next generation.)

What's commonly used are one of the following two:

**Steady-state:** as described above

**Generational-with-Elitism.**

‘elitism’ means that new generation always contains a fix number of the best solution from the previous generation. The remaining *popsize*-1 individuals are new children.



# Truncation selection

Applicable only in *generational* algorithms, where each generation involves replacing most or all of the population.

Parameter *pcg* (ranging from 0 to 100%)

Take the best *pcg*% of the population (BTR); produce the next generation entirely by applying variation operators to these.

For instance, select the 25% fittest from a population of 100, then we create four copies of each of them.

Less sophisticated method. It is less used than others

# Population update

A **population update** method is a way to decide how to produce the next generation from the merged previous generation and children.

- 1.- Replace-worst
- 2.- Replace randomly
- 3.- Parental replacement

*Note: in the literature, population update is sometimes called 'reproduction' and sometimes called 'replacement'.*

# Stopping Criterion

Different strategies can be used to stop the evolution of your population

- 1.- Maximum number of generations
- 2.- Maximum number of generations without improvement
- 3.- Maximum number of improvements

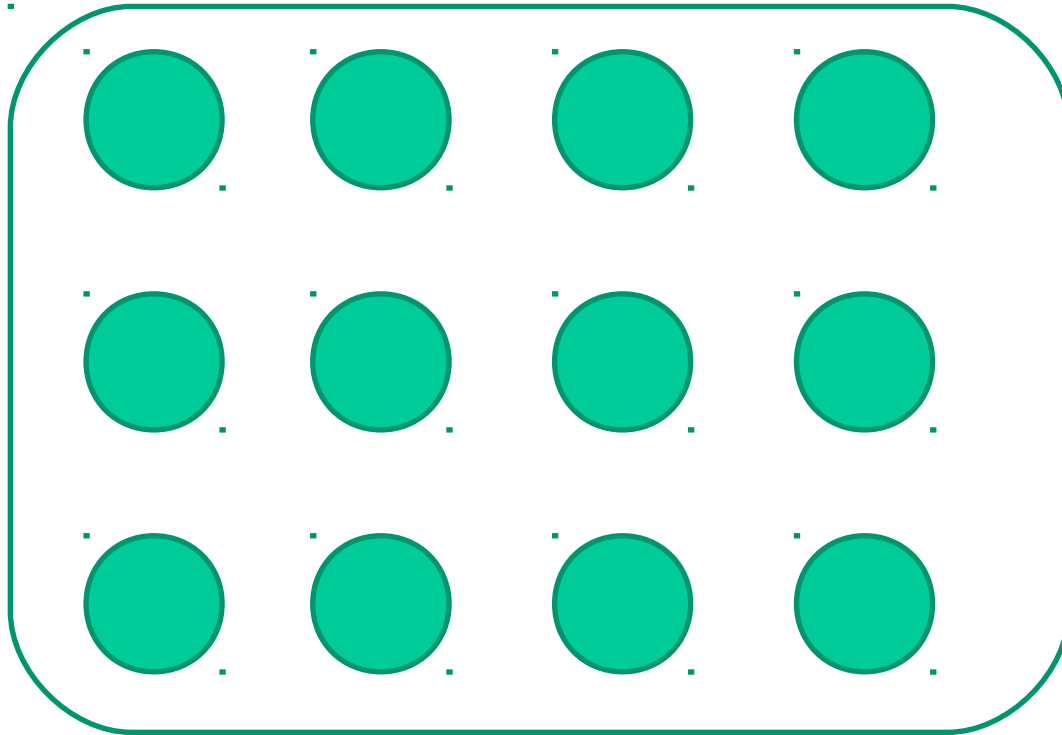
# *A steady state, mutation-only, replace-worst EA with tournament selection*

0. Initialise: generate a population of *popsiz*e random solutions, evaluate their fitnesses.
1. Run **Select** to obtain a parent solution *X*.
2. With probability *mutate\_rate*, mutate a copy of *X* to obtain a mutant *M* (otherwise *M* = *X*)
3. Evaluate the fitness of *M*.
4. Let *W* be the current worst in the population (BTR). If *M* is not less fit than *W*, then replace *W* with *M*. (otherwise do nothing)
5. If a termination condition is met (e.g. we have done 10,000 evaluations) then stop. Otherwise go to 1.

**Select:** randomly choose *tsiz*e individuals from the population. Let *c* be the one with best fitness (BTR); return *X*.

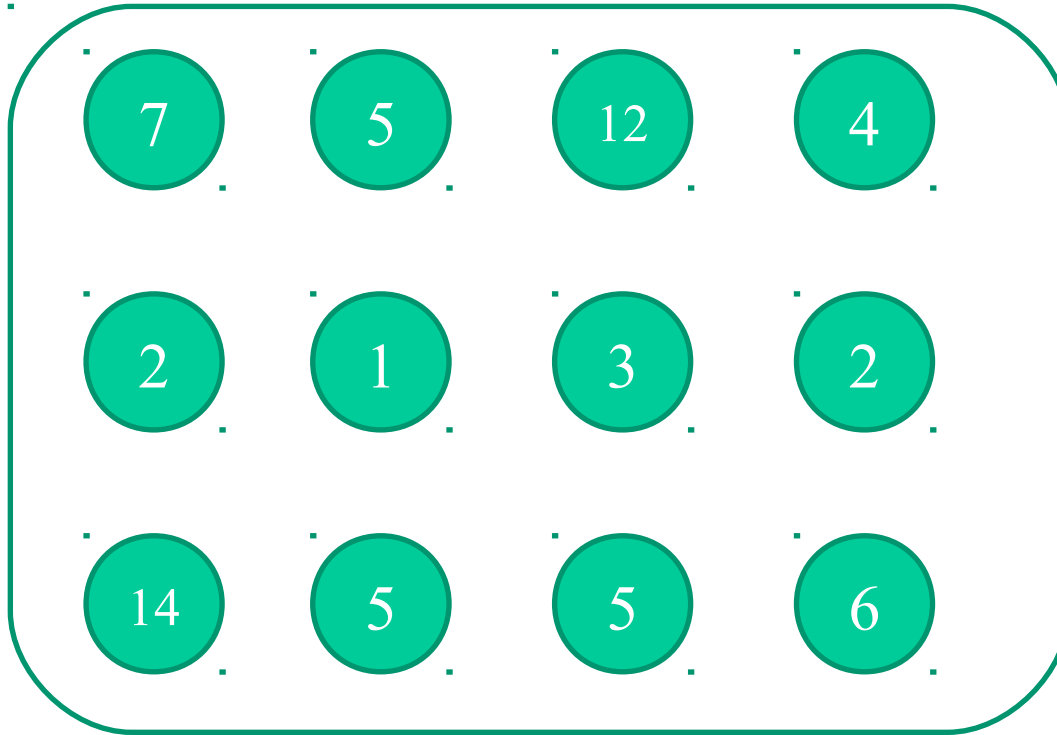
*Illustrating the 'Steady-state etc ...' algorithm ...*

## 0. Initialise population



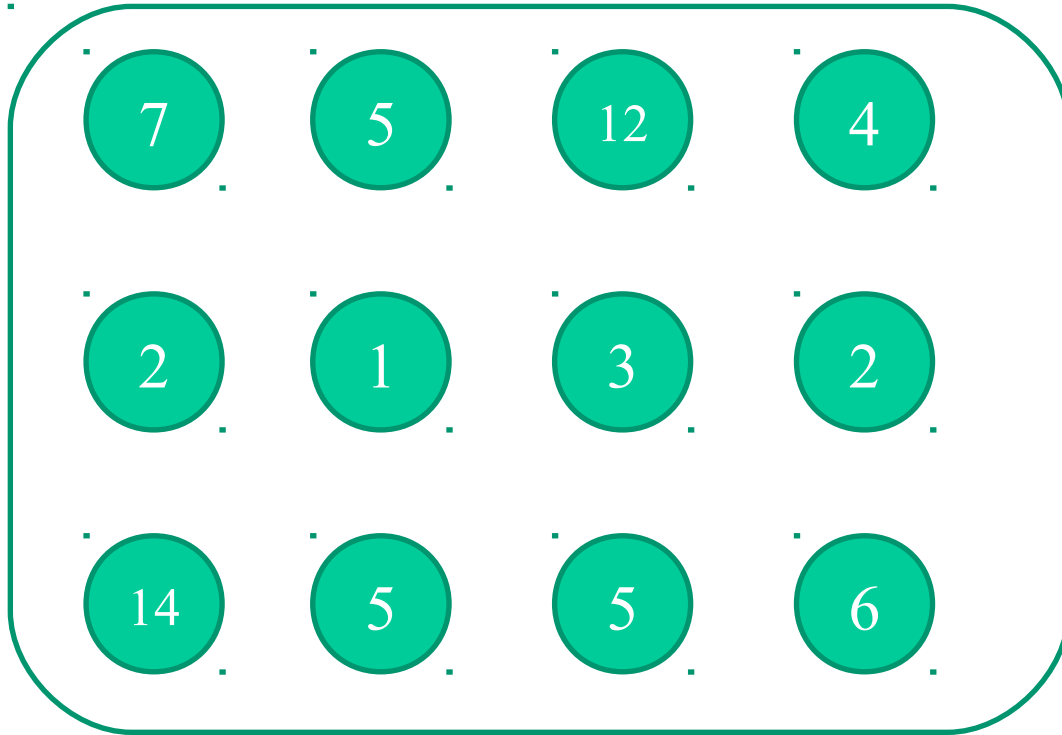
*Illustrating the 'Steady-state etc ...' algorithm ...*

0.... Evaluate their fitnesses



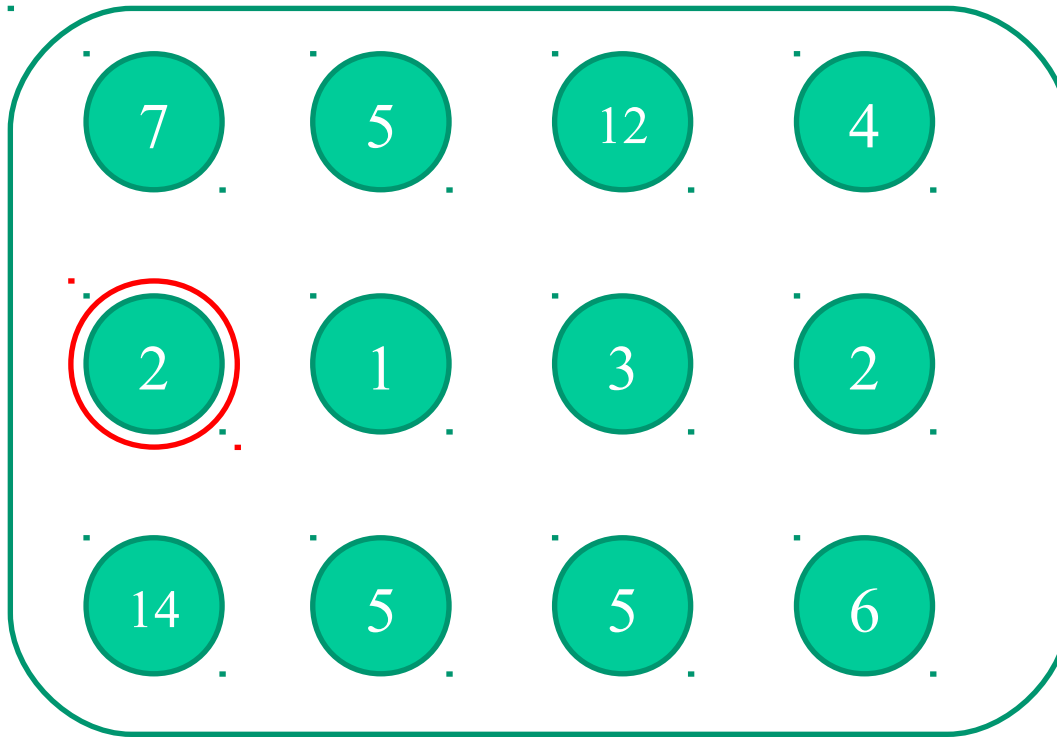
*Illustrating the 'Steady-state etc ...' algorithm ...*

## 1. Select



*Illustrating the ‘Steady-state etc ...’ algorithm ...*

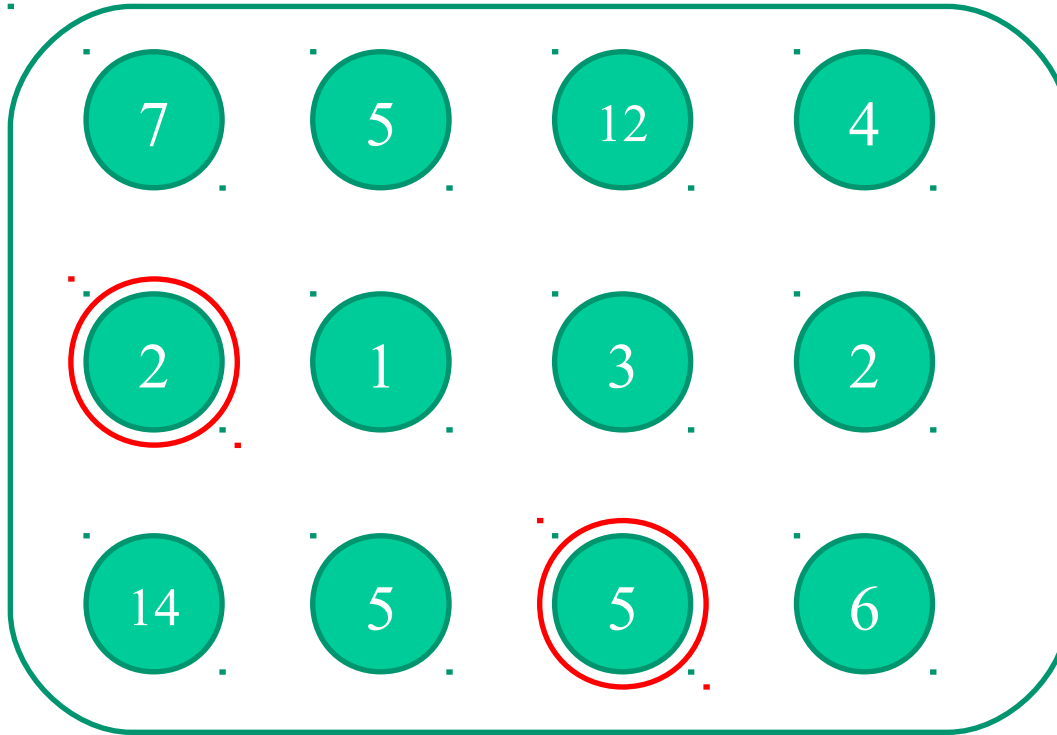
1... **Select:** choose one at random





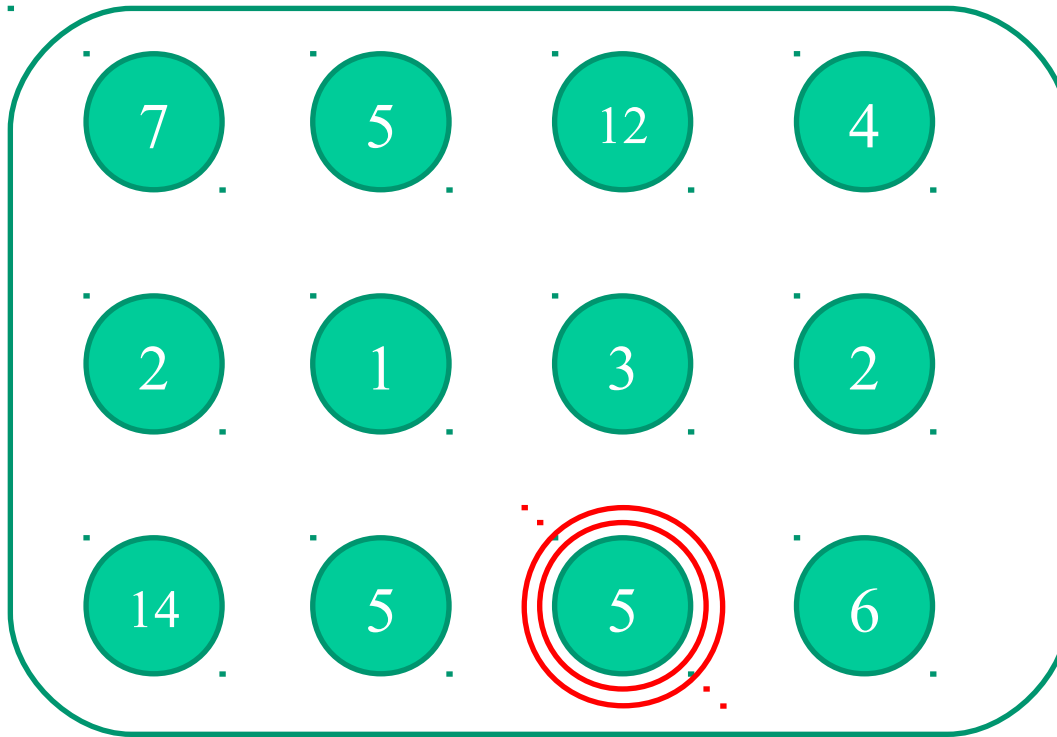
*Illustrating the ‘Steady-state etc ...’ algorithm ...*

1... **Select:** choose another one at random



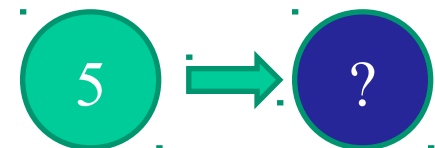
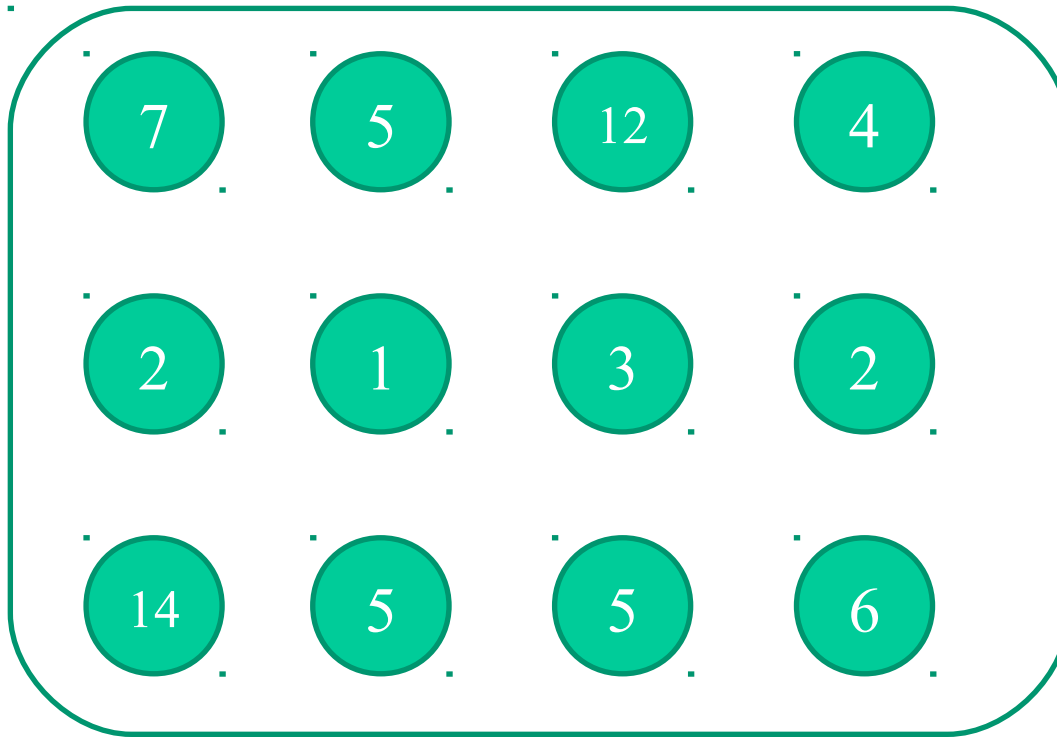
*Illustrating the ‘Steady-state etc ...’ algorithm ...*

1... **Select:** best of those two is the selected parent



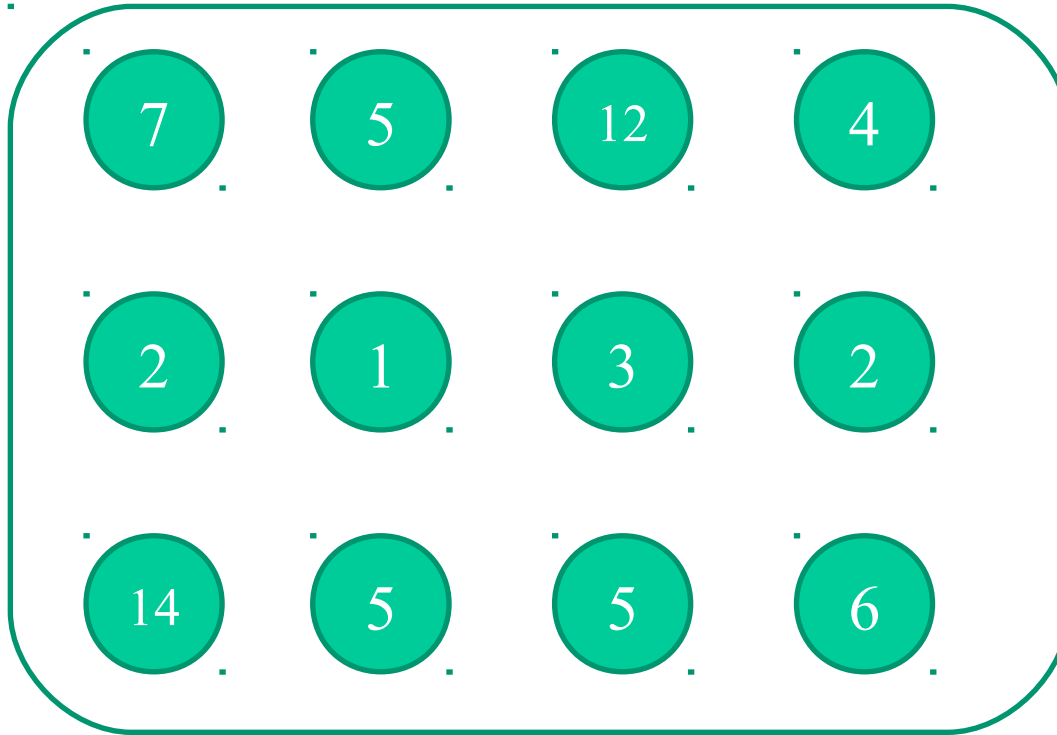
*Illustrating the 'Steady-state etc ...' algorithm ...*

2. **Mutate:** mutate a copy of the selected one



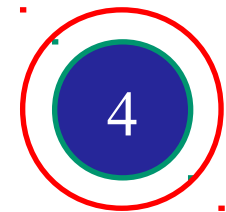
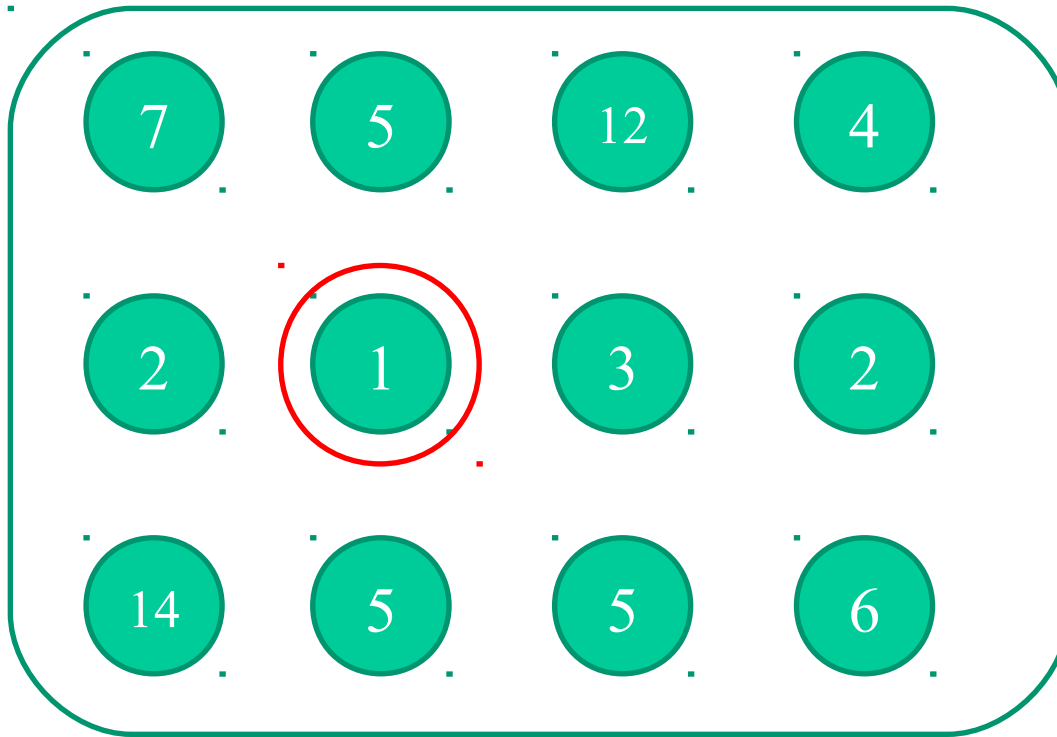
*Illustrating the 'Steady-state etc ...' algorithm ...*

### 3. **Mutate:** ... evaluate the mutant



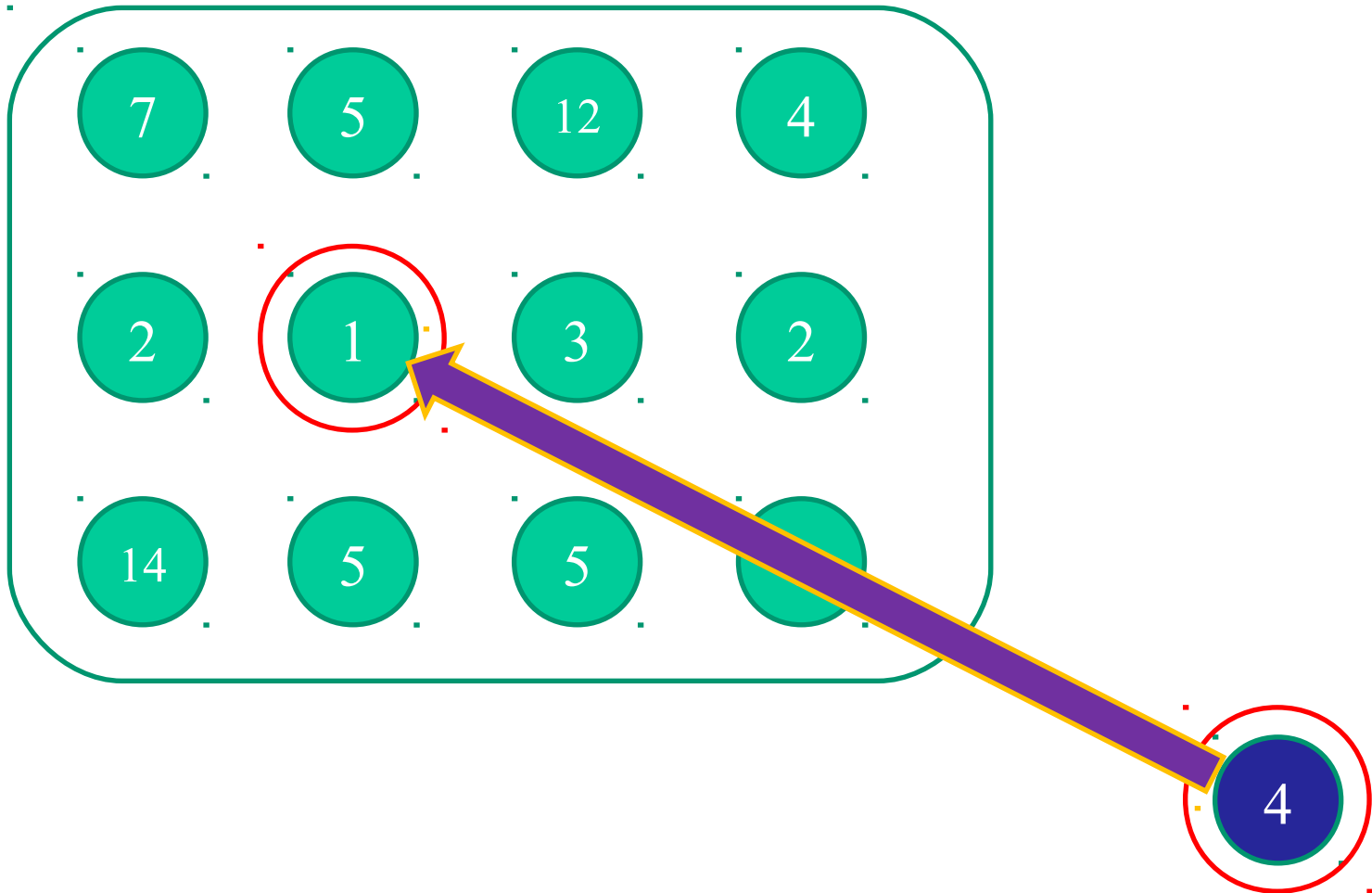
*Illustrating the 'Steady-state etc ...' algorithm ...*

4... **Replace**: check: is mutant better/eq worst in pop?



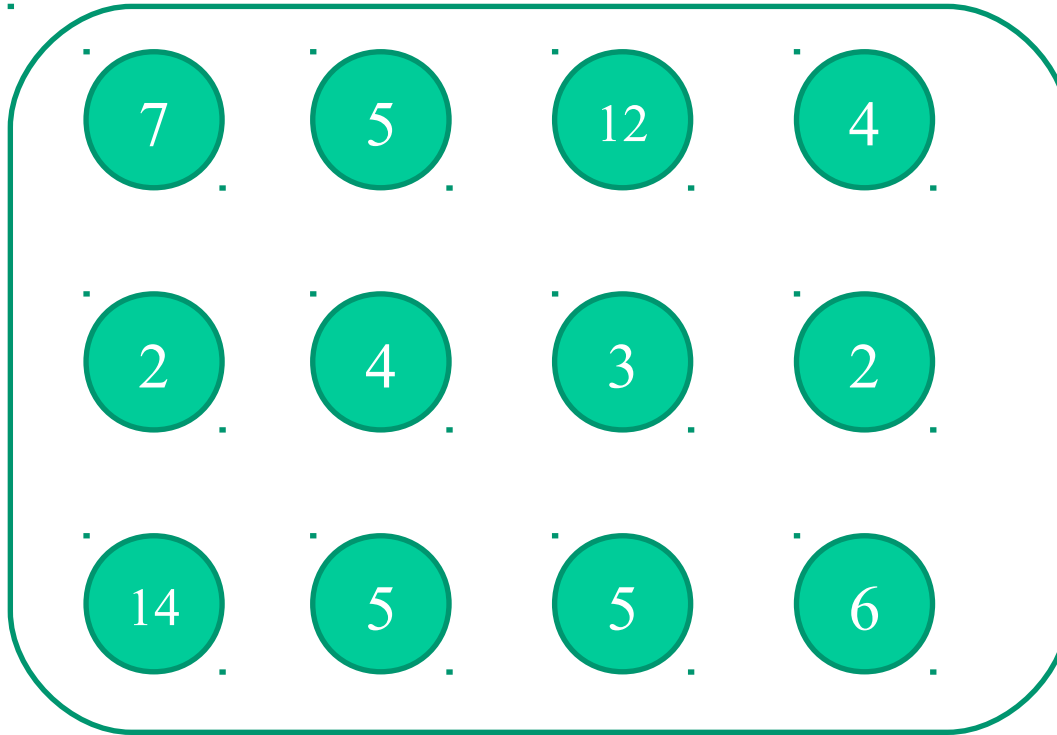
*Illustrating the 'Steady-state etc ...' algorithm ...*

4. **Replace:** it is, so it can go in, overwriting worst



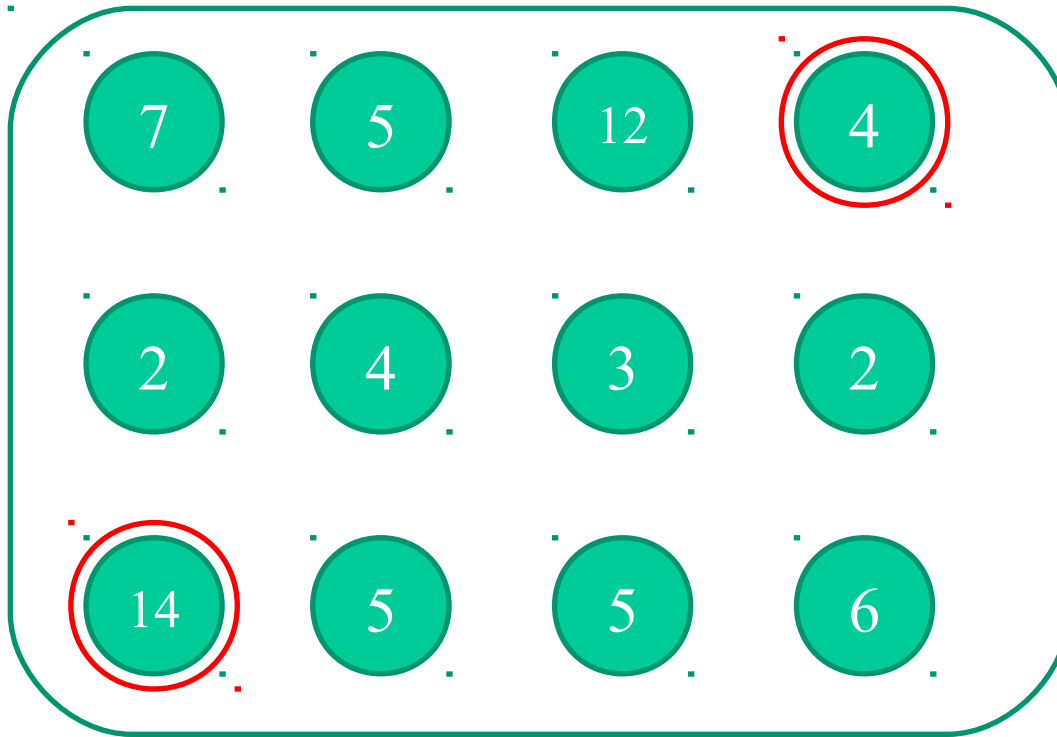
*Illustrating the 'Steady-state etc ...' algorithm ...*

**New generation:** and here we are...



*Illustrating the 'Steady-state etc ...' algorithm ...*

1. **Select** ... and so it goes on, until we're done





# *A generational, elitist, crossover+mutation EA with Rank-Based selection*

0. Initialise: generate a population  $G$  of *popsiz*e random solutions, evaluate their fitnesses.
1. Run **RankSelect**  $2 * (\text{popsiz}e - 1)$  times to obtain a collection  $I$  of  $2 * (\text{popsiz}e - 1)$  parents.
2. Randomly pair up the parents in  $I$  (into  $\text{popsiz}e - 1$  pairs) and apply **Vary** to produce a child from each pair. Let the set of children be  $C$ .
3. Evaluate the fitness of each child.
4. Keep the best in the population  $G$  (BTR) and delete the rest.
5. Add all the children to  $G$ .
6. If a termination condition is met (e.g. we have done 100 or more generations (runs through steps 1–5) then stop. Otherwise go to 1,

# A generational, elitist, crossover+mutation EA with *Rank-Based* selection, continued ...

**RankSelect:** sort the contents of  $G$  from best to worst, assigning rank  $popsiz$  to the best,  $popsiz-1$  to the next best, etc ..., and rank 1 to the worst.

The ranks sum to  $F = popsiz(popsiz+1)/2$

Associate a probability  $Rank_i/F$  with each individual  $i$ .

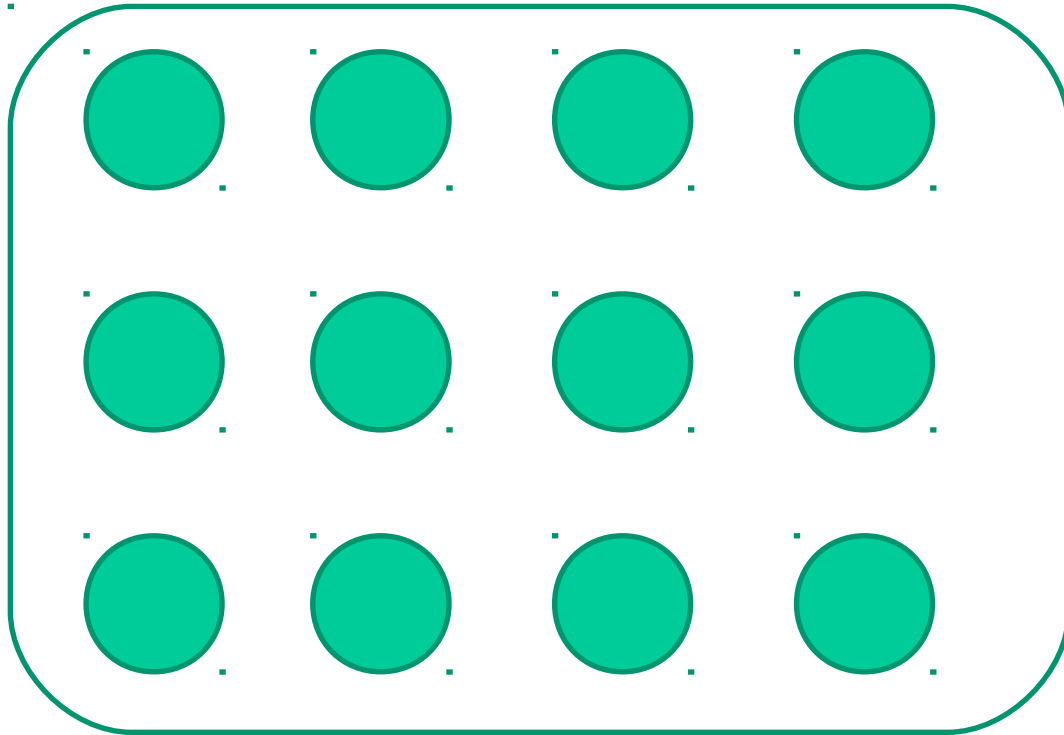
Using these probabilities, choose one individual  $X$ , and return  $X$ .

## **Vary:**

1. With probability  $cross\_rate$ , do a crossover:  
I.e produce a child by applying a crossover operator to the two parents. Otherwise, let the child be a randomly chosen one of the parents.
2. Apply mutation to the child.
3. Return the mutated child.

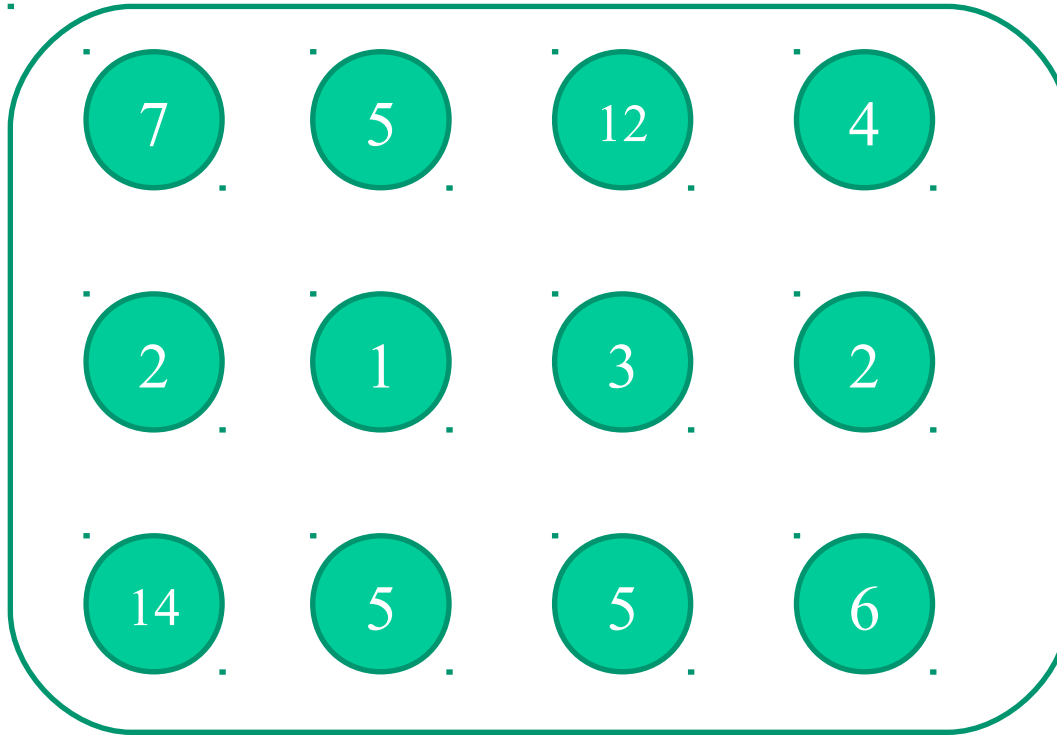
*Illustrating the ‘Generational etc ...’ algorithm ...\*

## 0. Initialise population



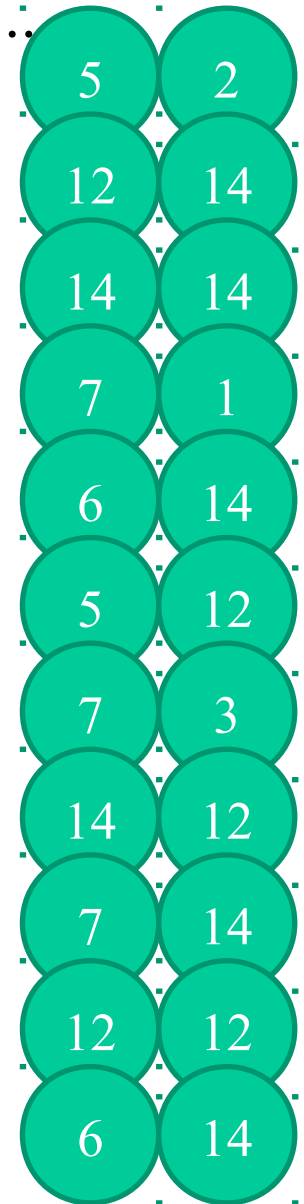
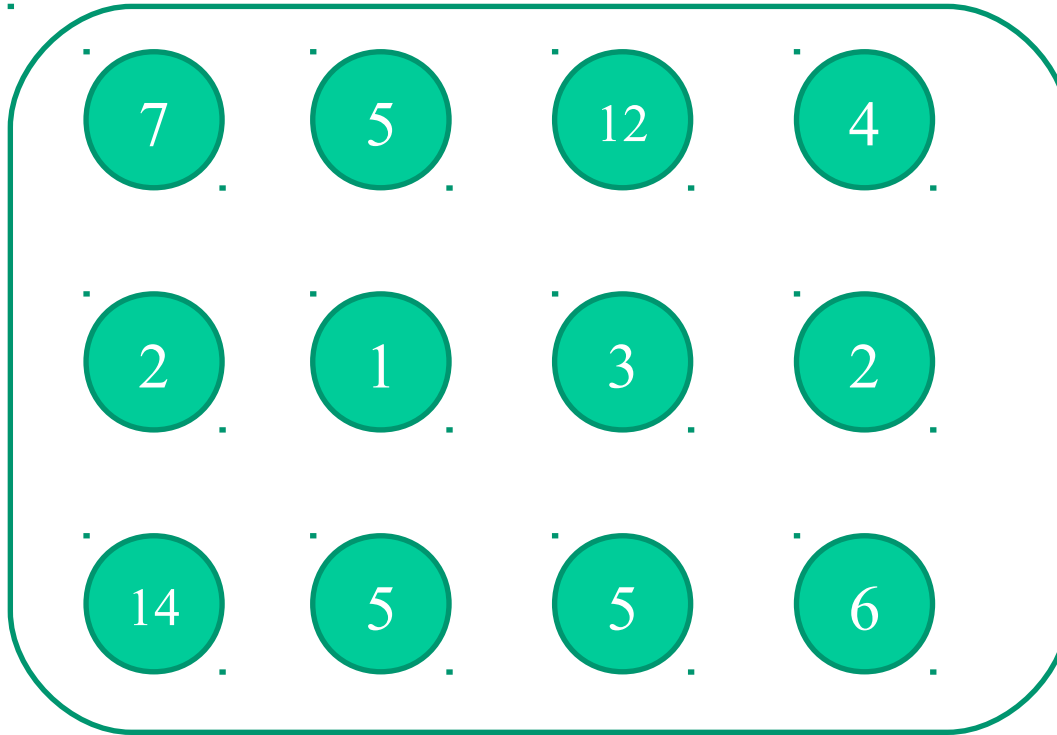
## *Illustrating the 'Generational etc ...' algorithm ...\*

0.... Evaluate their fitnesses



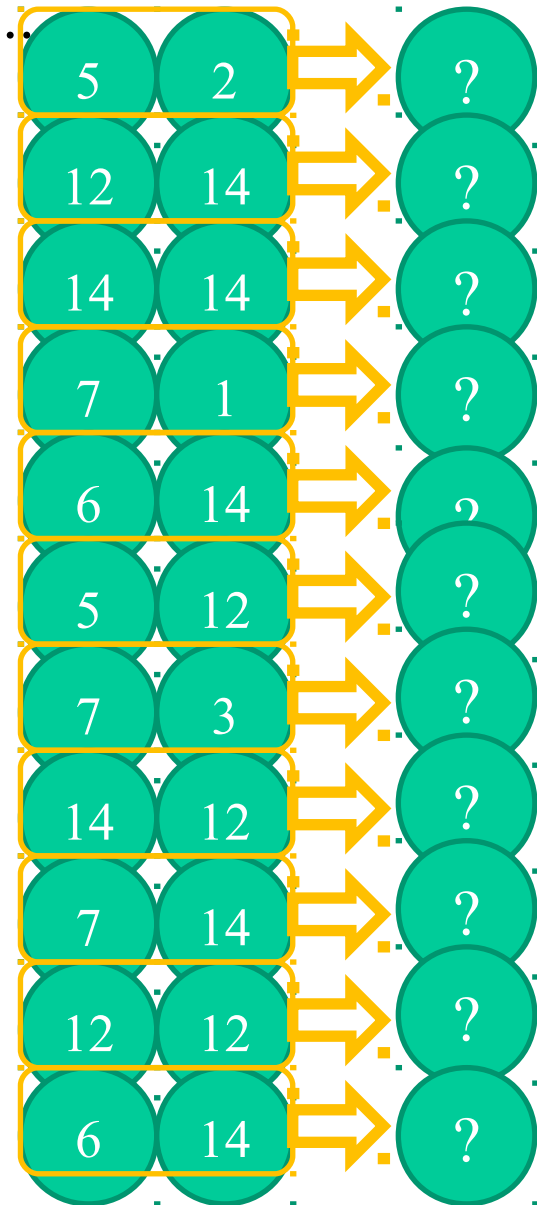
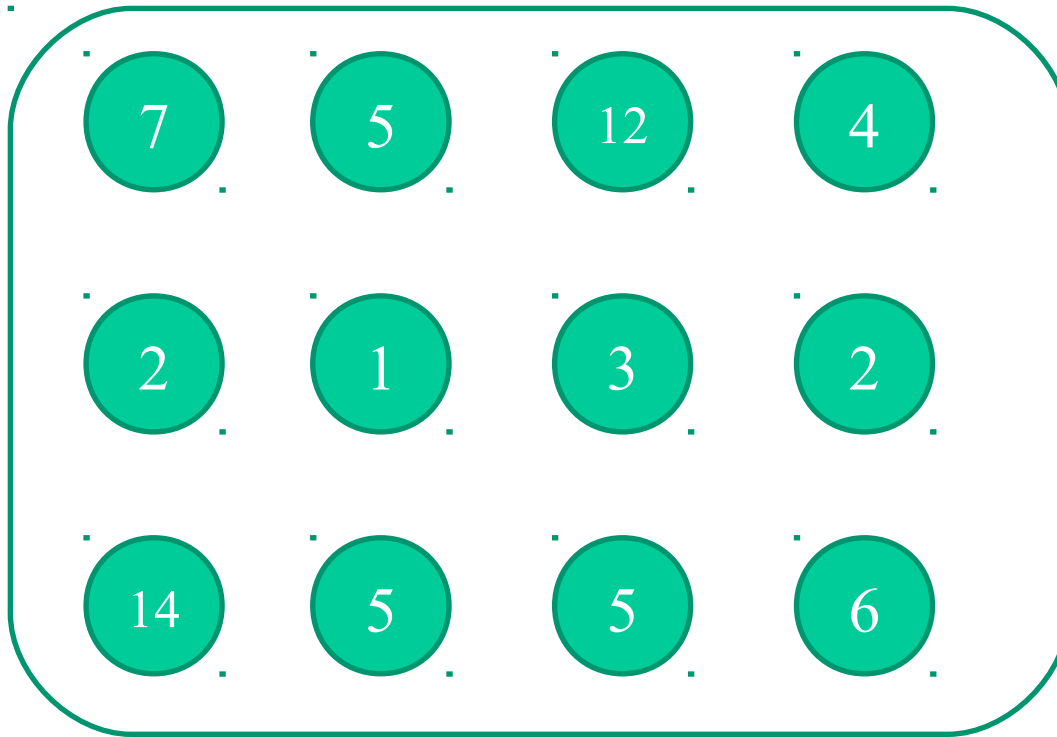
*Illustrating the 'Generational etc ...' algorithm ...*

## 1. Select



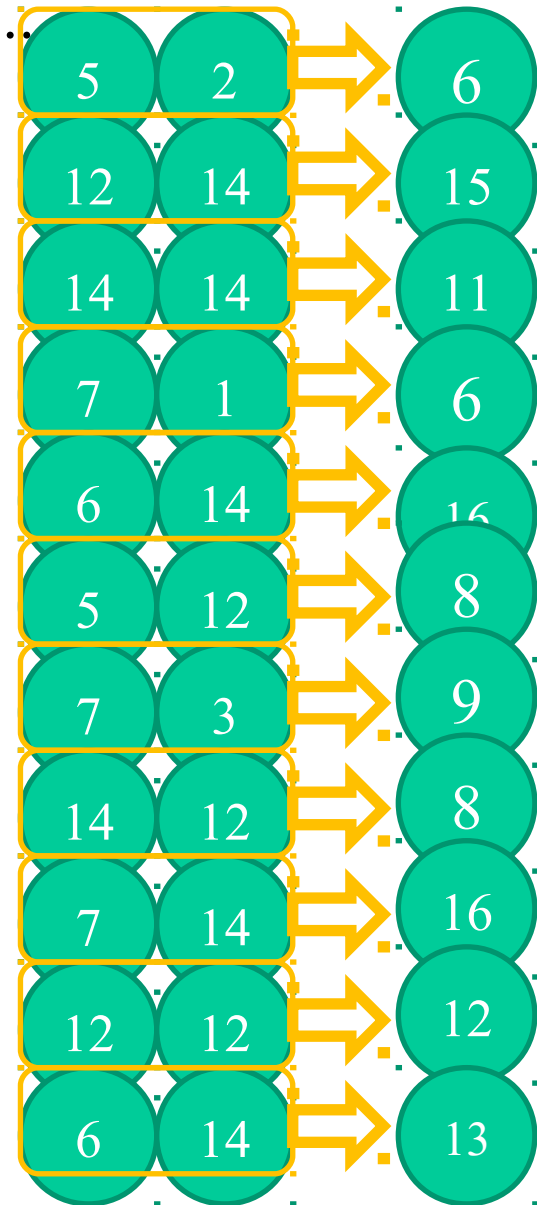
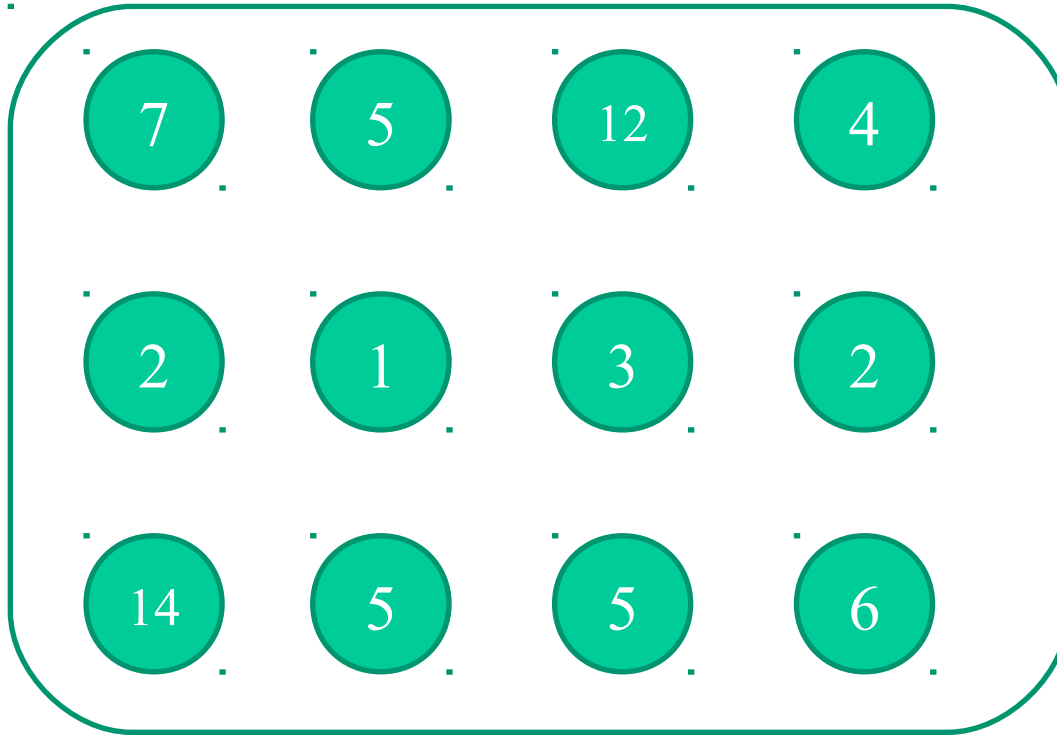
*Illustrating the 'Generational etc ...' algorithm ...*

## 2. Vary ...



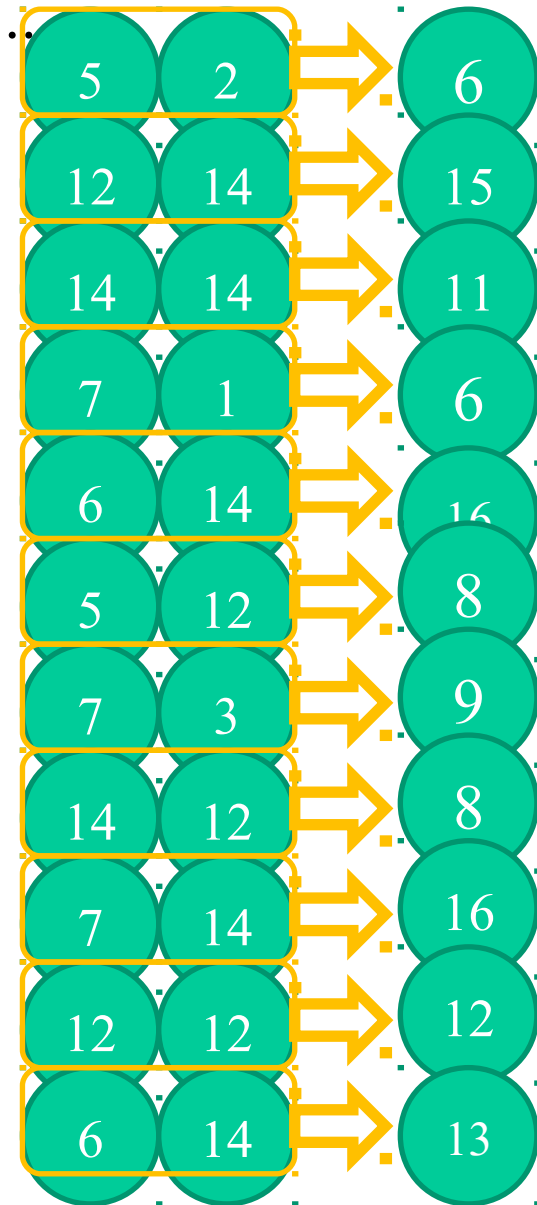
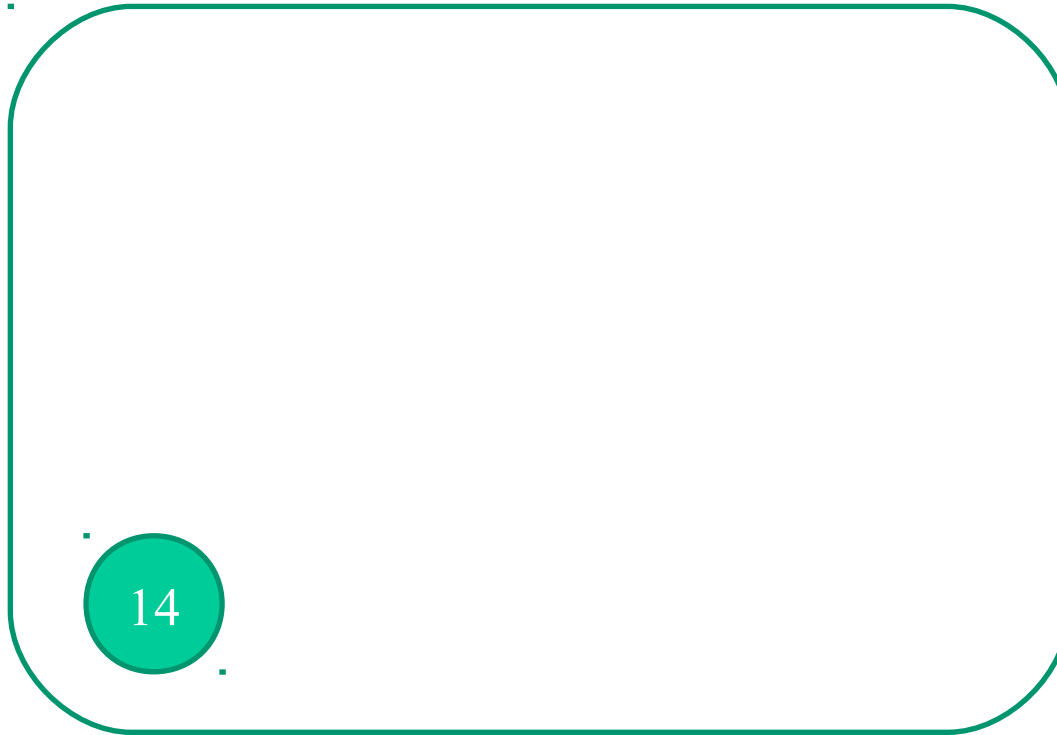
*Illustrating the 'Generational etc ...' algorithm ...*

### 3. Evaluate children ...



*Illustrating the 'Generational etc ...' algorithm ...*

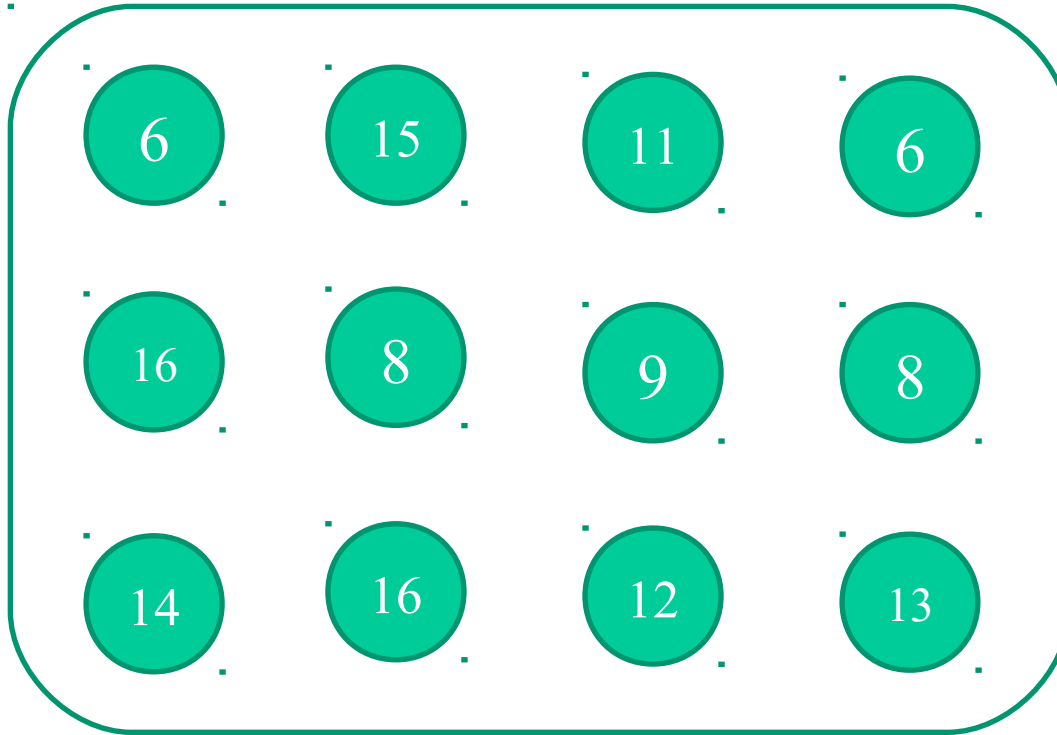
#### 4. **Keep best of old pop ...**





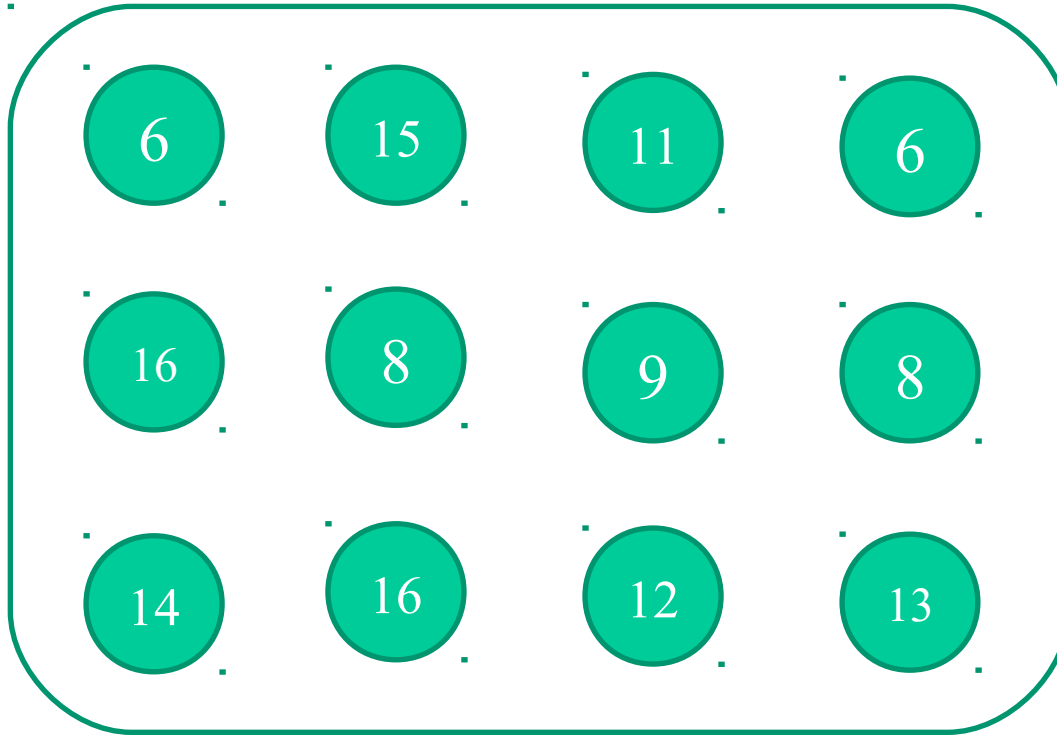
*Illustrating the ‘Generational etc ...’ algorithm ...*

## 5. Incorporate new children ...



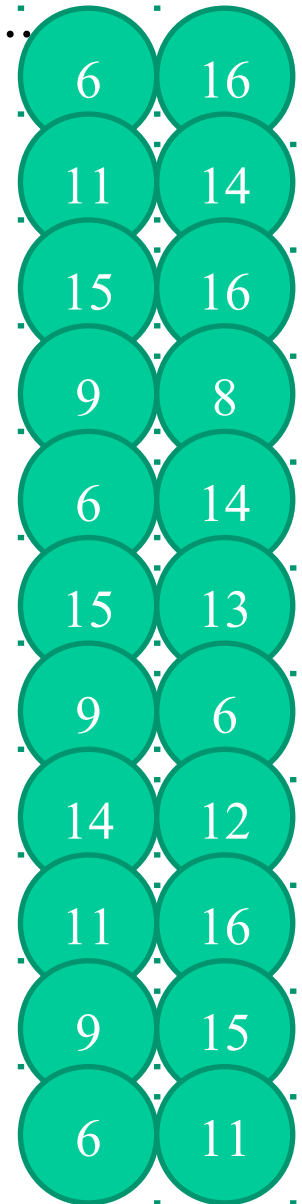
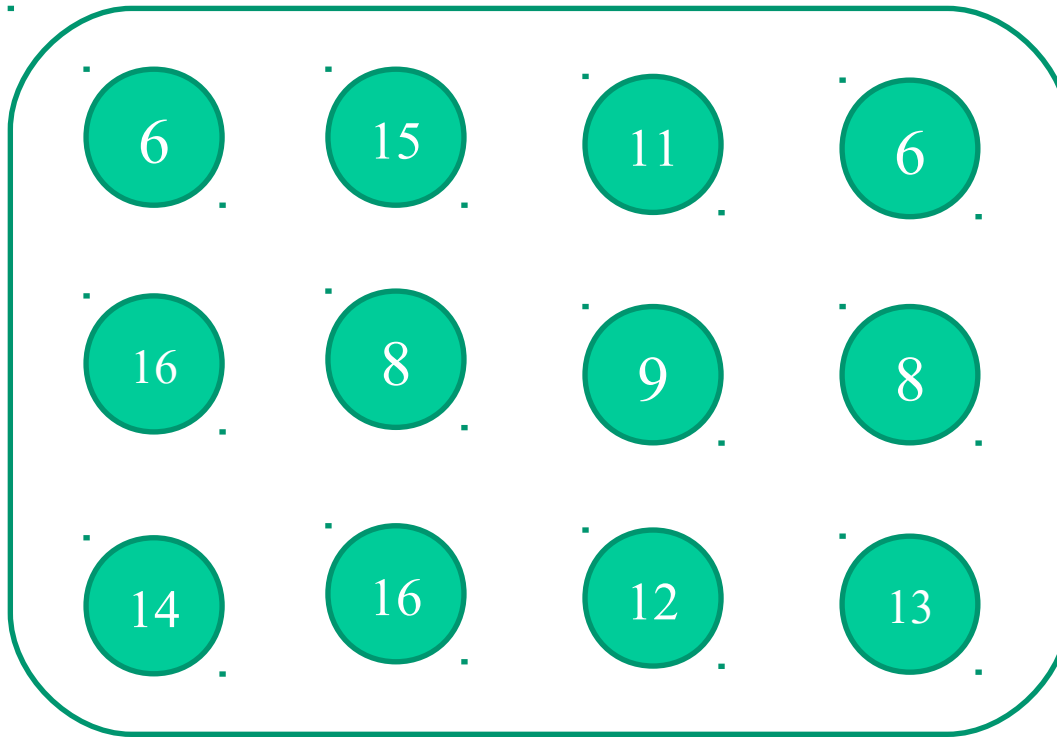
*Illustrating the ‘Generational etc ...’ algorithm ...*

## 6. if not finished, go to ...



*Illustrating the 'Generational etc ...' algorithm ...*

## 1. Select...and so on...



# ENCODING EXAMPLES

Example of an encoding (and associated operators)

# Bin-Packing

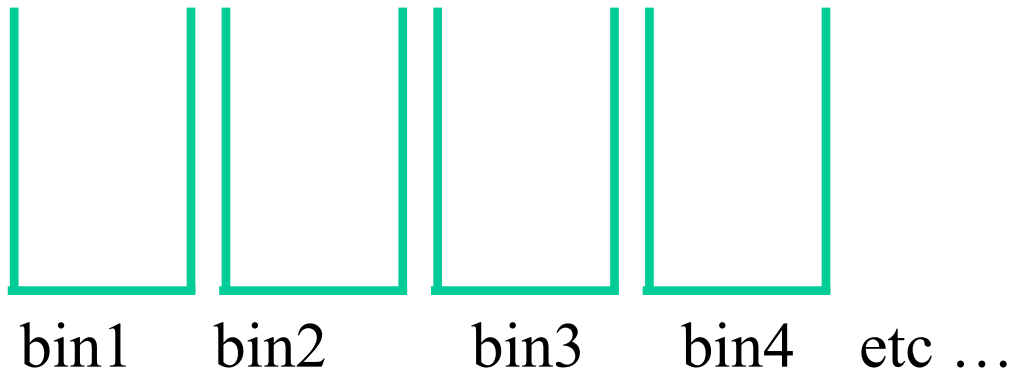
## The Bin-Packing Problem:

You have items of different sizes or weights:

e.g: 1 (30kg) 2 (25kg) 3 (10kg) 4 (20kg) 5 (15kg)



And you have several 'bins' with a max capacity, say 50kg.



Find a way to pack them into the smallest number of bins. This is, *in general*, a **hard** problem.

Example of an encoding (and associated operators)

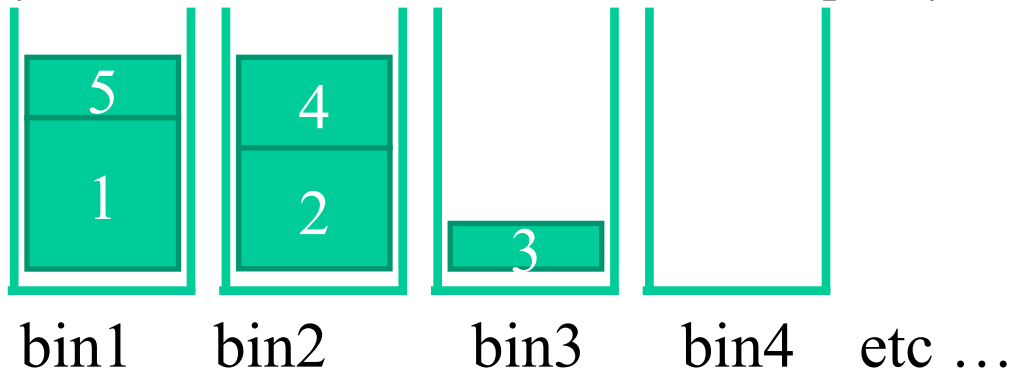
# Bin-Packing

## One packing

1 (30kg) 2 (25kg) 3 (10kg) 4 (20kg) 5 (15kg)



And you have several 'bins' with a max capacity, say 50kg.



*bin1 and bin2 have 45kg -- so the 10kg item 3 needs a new bin*

Example of an encoding (and associated operators)

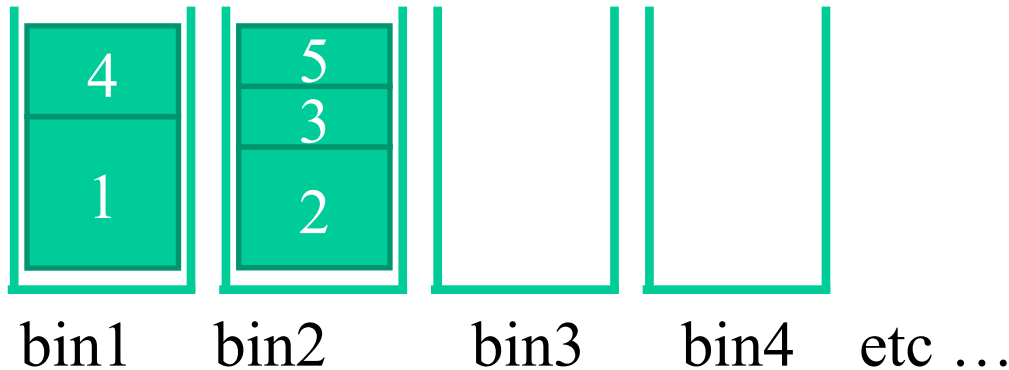
# Bin-Packing

## Another packing

1 (30kg) 2 (25kg) 3 (10kg) 4 (20kg) 5 (15kg)



And you have several 'bins' with a max capacity, say 50kg.



*by putting items 1 and 4 together, we can now fit the rest in one bin*

# A simple encoding for bin-packing

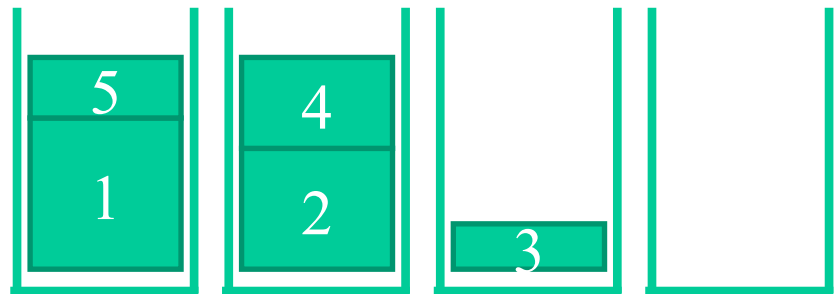
Suppose we have  $N$  items, and we have a maximum of  $B$  bins available,

A simple encoding is as follows:

- A solution is represented by a list of  $N$  integers, each of them being any number from 1 to  $B$  inclusive.
- the meaning of such a solution, like for example ‘**3**, **1**, **2**, **1**, **2** ....’ is ‘**the 1<sup>st</sup> item is in bin 3**, **the 2<sup>nd</sup> item is in bin 1**, **the 3<sup>rd</sup> item is in bin 2**, **the 4<sup>th</sup> item is in bin 1**, **the 5<sup>th</sup> item is in bin 2**, [and so on ...]’

So, in our simple example,

1,2,3,2,1 encodes this solution





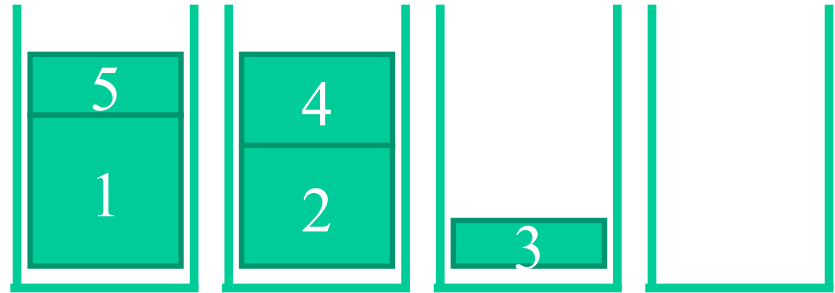
# Simple operators that work with this encoding

## A Mutation Operator:

choose a gene at random, and change it to a random *new* value

So, in our simple example,

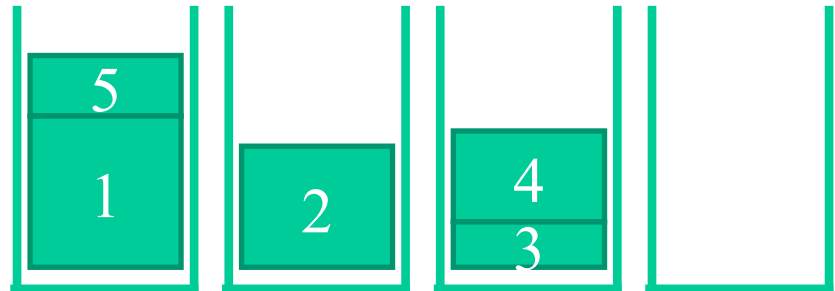
1,2,3,2,1 encodes this solution



Choose a gene at random: 1,2,3,**2**,1

Change it to a random *new* (but valid) value: 1,2,3,**3**,1

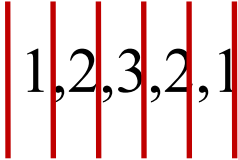
Our mutant is: 1, 2, 3, 3, 1  
which encodes this:



# Simple operators that work with this encoding

## A Crossover Operator:

We can use ‘one-point crossover’. In this case, any solution has 5 ‘genes’, which means we have 6 crossover points:

 1, 2, 3, 2, 1 ← here is an example solution with the crossover points indicated

In one-point crossover, we select two parents (P1 and P2), we then choose a crossover point  $X$  at random. We then build a child by copying P1 up to  $X$ , and then copying P2 from  $X$  onwards. E.g.

Crossover of **3,1,2,3,3** and **2,2,1,2,1** at 4th Xover point is: **3,1,2,2,1**

Crossover of **1,1,3,2,2** and **2,3,1,1,1** at 2<sup>nd</sup> Xover point is: **1,1,1,1,1**

# The TSP

## The Travelling Salesperson Problem

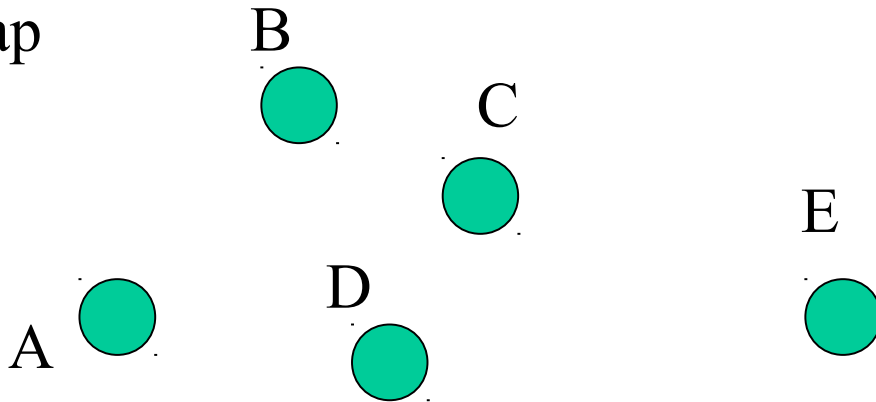
You have  $N$  ‘cities’, and an associated distance matrix. Find the shortest **tour** through the cities. I.e., starting from one of the cities (‘A’, for example), what is the quickest way to visit all of the other cities, and end up back at ‘A’ ?

Many many real problems are based on this (vehicle routing problems, bus or train scheduling), or exactly this (drilling thousands of holes in a metal plate, or ... an actual travelling Salesperson problem!).

Example of an encoding (and associated operators)

# The TSP

Map



	A	B	C	D	E
A		5	7	4	15
B	5		3	4	10
C	7	3		2	7
D	4	4	2		9
E	15	10	7	9	

Example of an encoding (and associated operators)

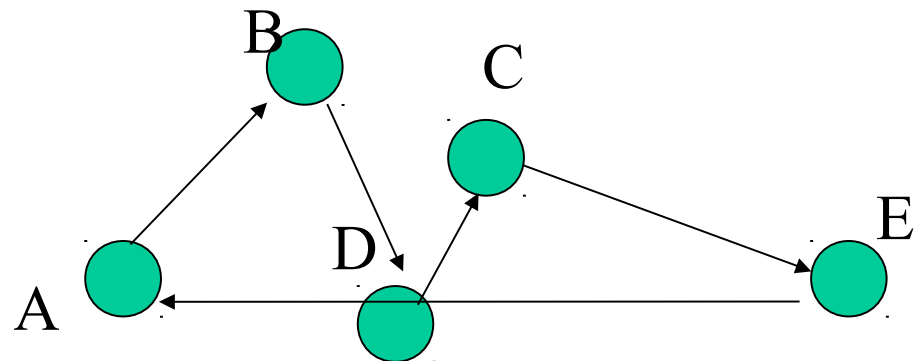
# The TSP: encoding

The encoding most commonly used for the TSP is a *Permutation encoding* (also generally called an *order-based* encoding).

If we have  $N$  cities,  $1, 2, 3, \dots, N$ , then any permutation of these  $N$  identifiers represents a solution.

e.g. if  $N$  is 8, then  $3, 4, 2, 8, 7, 6, 1, 5$  encodes a solution,  
but  $3, 3, 2, 1, 7, 3, 2, 4$  does not encode a solution – why?

In our previous example, we had cities A, B, C, D and E – so any permutation of these five letters represents a solution. E.g. ABDCE represents this:



Example of an encoding (and associated operators)

# The TSP: operators

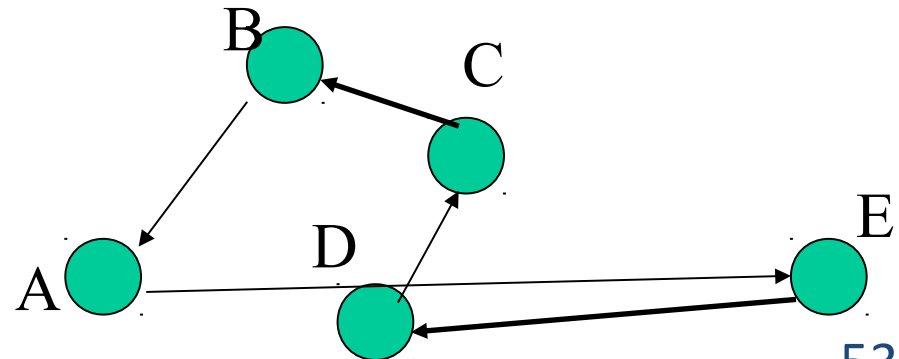
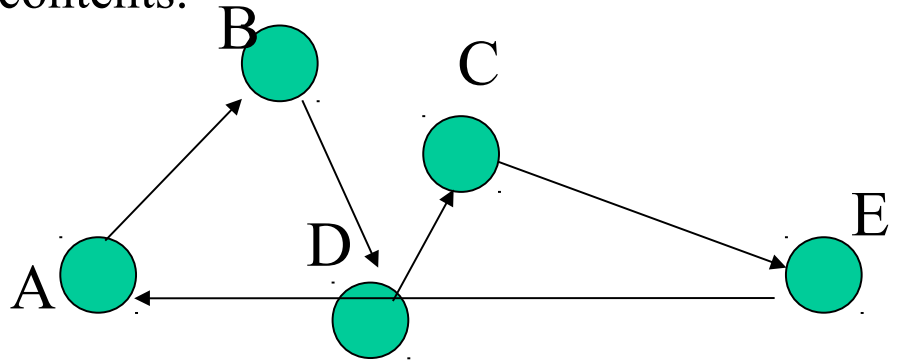
The mutation operator we described for the bin packing encoding would not be valid in this case. Why?

One valid mutation operator would be *random pair swap*. E.g. choose two different gene positions at random, and swap their contents.

Suppose we have solution CEABD

Let's randomly choose the 2<sup>nd</sup> and 4<sup>th</sup> genes and swap them.

We get CBAED, which is this solution:



Example of an encoding (and associated operators)

# The TSP: operators

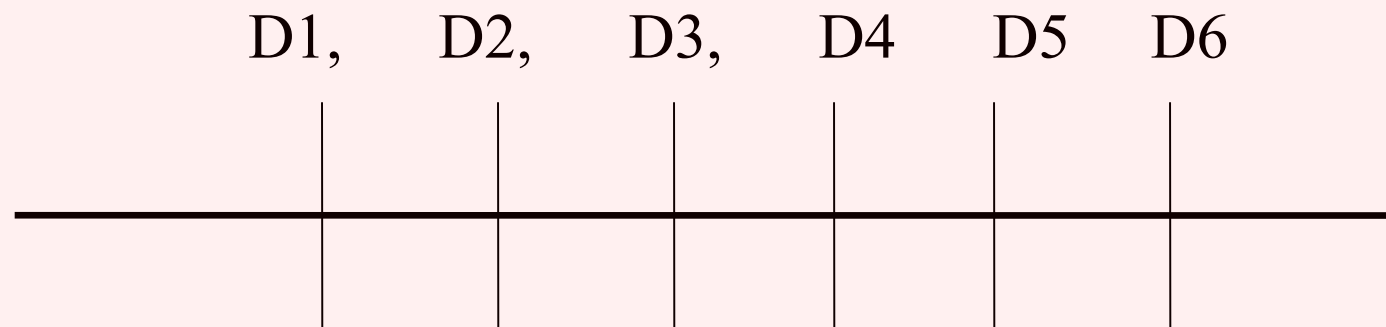
The crossover operator we described for the bin packing encoding would not be valid in this case. Why?

In general, recombination operators for order-based encodings are slightly harder to design, but there are many that are commonly used.

# Example real-number Encoding

Suppose we want to design the overall shape of a 2-phase jet-nozzle (this was the first ever EA application – mentioned in lecture 2)

We could do it by specifying a vector of 6 real numbers with the meaning indicated below:

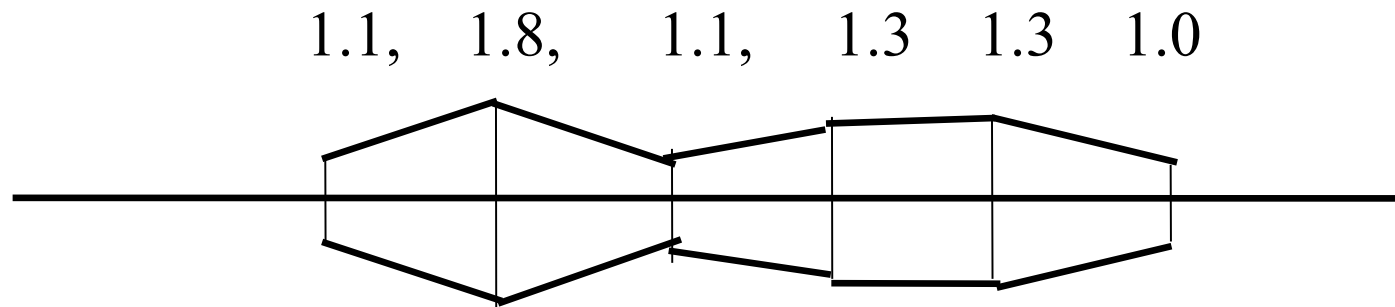
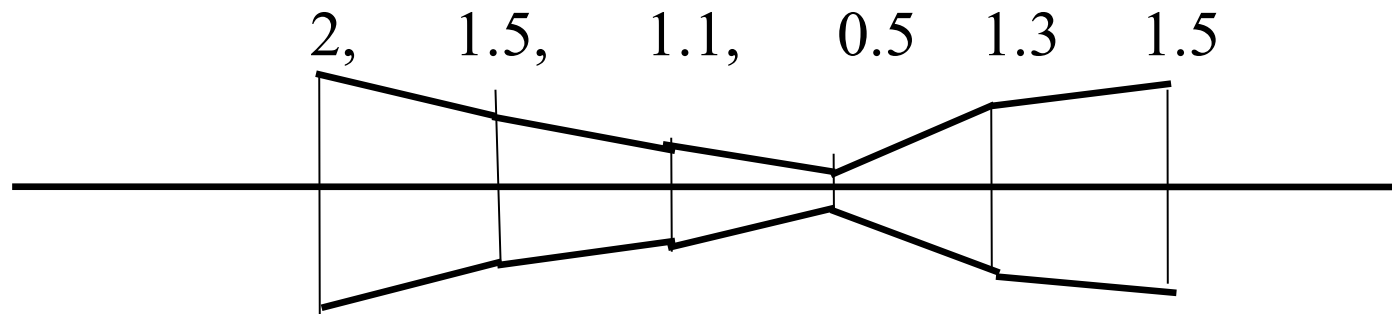
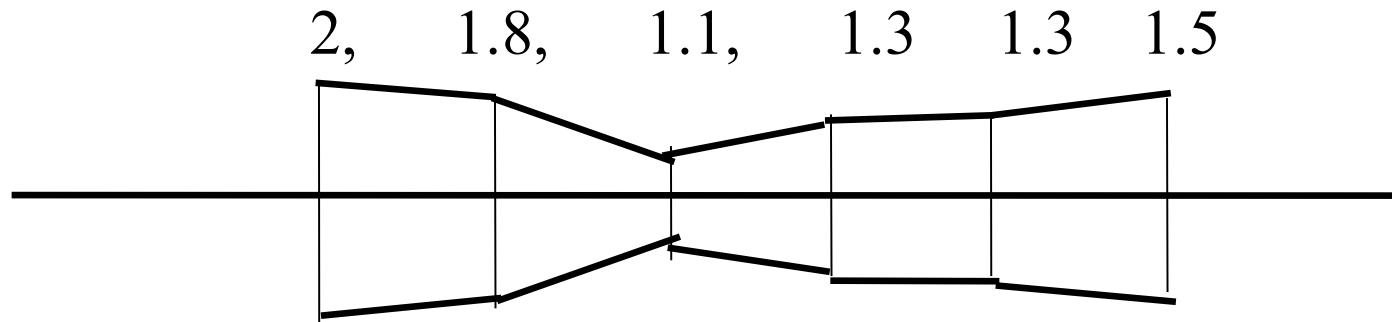


Fixed at six diameters, five sections

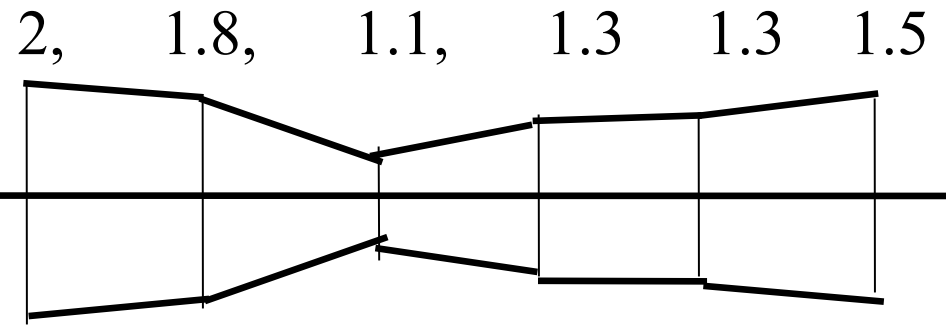




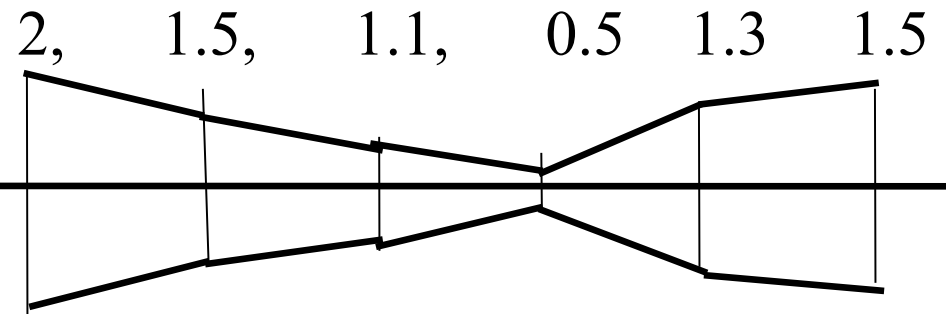
# examples



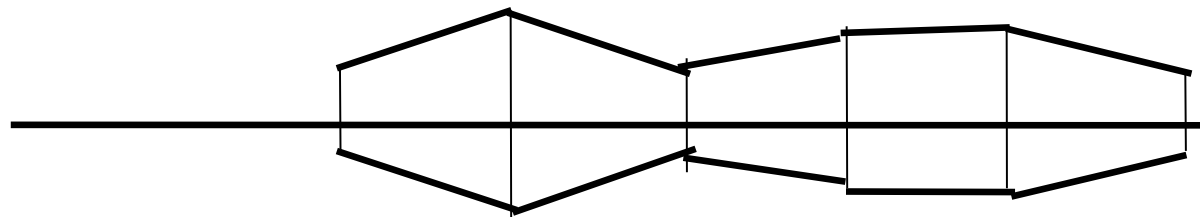
mutation and  
crossover –



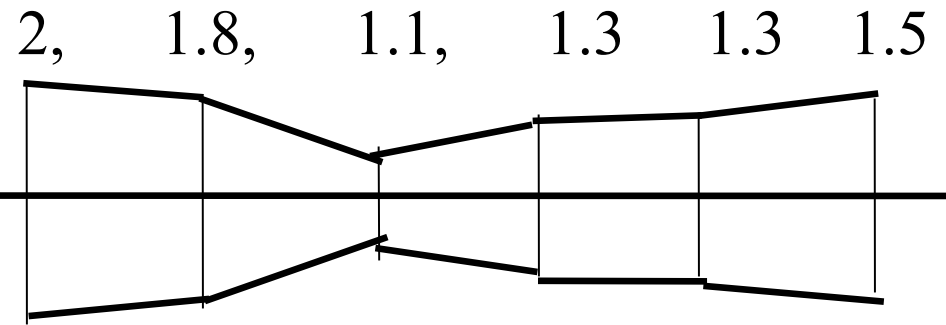
what do you  
suggest?



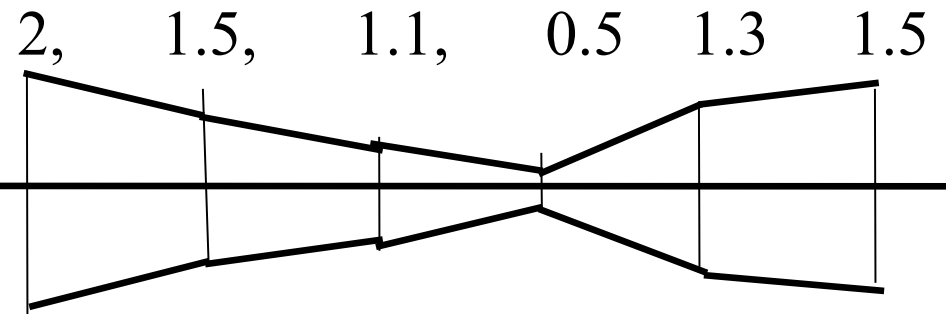
1.1, 1.8, 1.1, 1.3 1.3 1.0



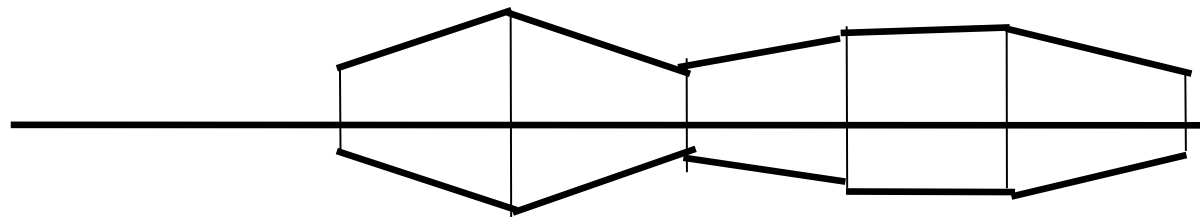
mutation and  
crossover –



what do you  
suggest?



1.1, 1.8, 1.1, 1.3 1.3 1.0



Or, could use a  
**binary** encoding



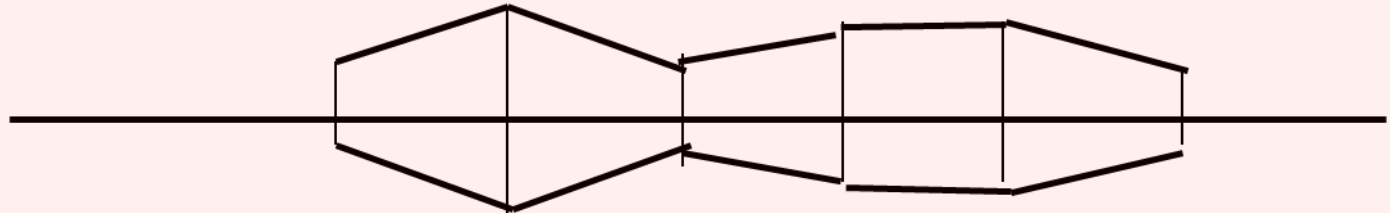
Encoded solution: 01101001 00110101 01101000 00001000 100...

Each chunk of 8 bits (0 to 255) scaled to be between 0 and 2

# Or, could use a **binary** encoding

Encoded solution: 01101001|00110101|01101000|00001000|100...

1.1, 1.8, 1.1, 1.3 1.3 1.0



Each chunk of 8 bits (0 to 255) scaled to be between 0 and 2

A more complex encoding– bigger search space,  
slower, but potential for innovative solutions

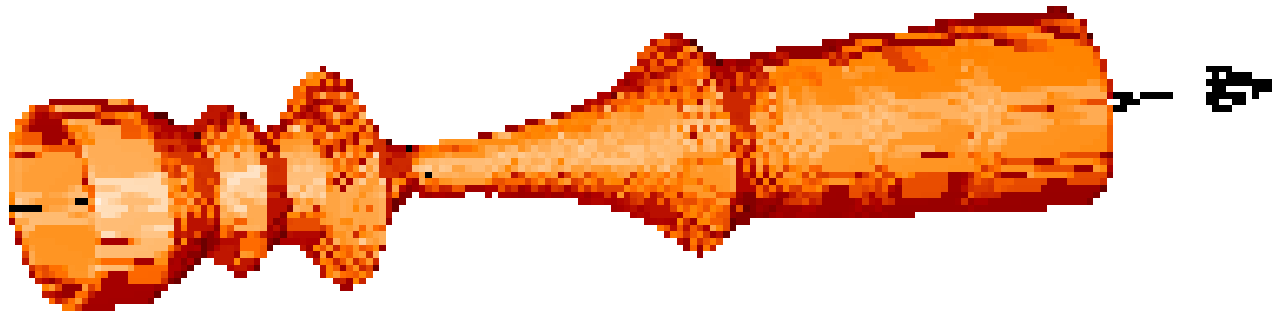
Num sections before smallest

Section diameters

$Z1, Z2,$

$D1, D2, D3 \ D_{small}..., D_n, D_{n+1}, ...$

Num sections after smallest



Mutations can change diameters, add sections,  
and delete sections