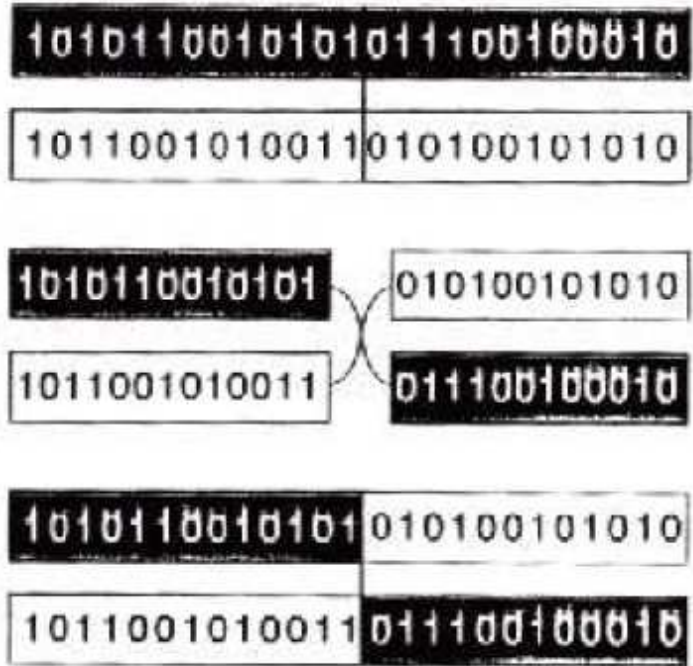


Crossover (1-point)

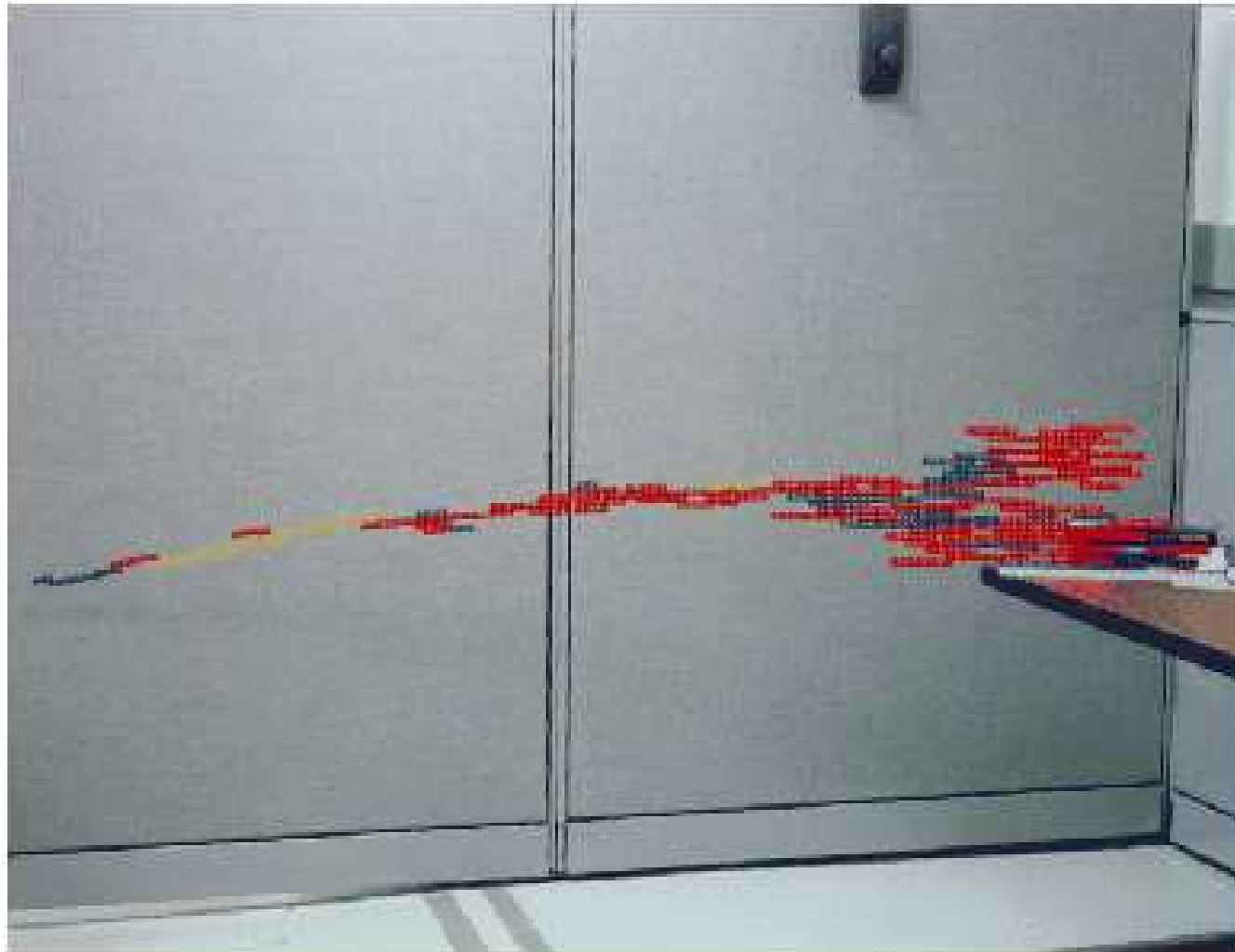


CROSSOVER is the fundamental mechanism of genetic rearrangement for both real organisms and genetic algorithms.

Chromosomes line up and then swap the portions of their genetic code beyond the crossover point.

In GAs each individual usually just has one chromosome (string of genes) in its genotype, so the crossover (recombination) is between the entire genotypes of two individuals.

Evolved Lego Structures (Pablo Funes)



Evolved Lego Structures (Pablo Funes)



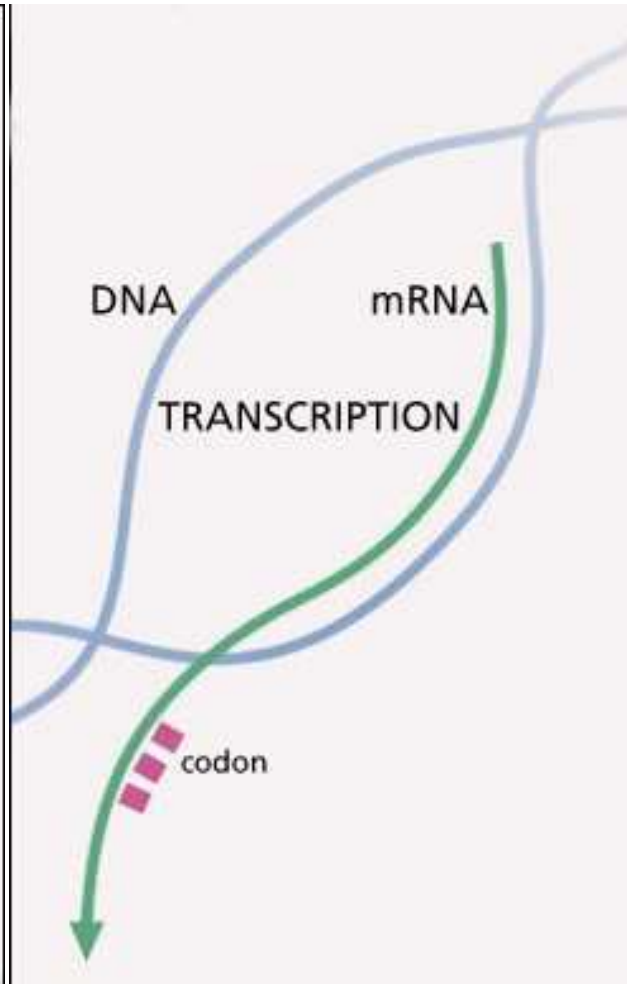
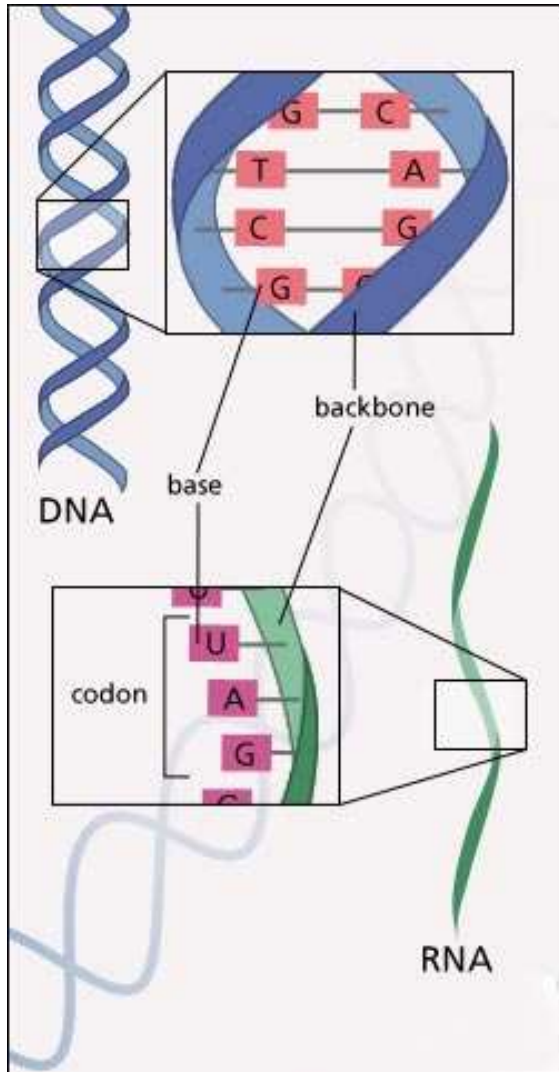
Genes, loci, alleles, bits, encoding (1)

“Gene” was initially an abstract unit of inheritance.

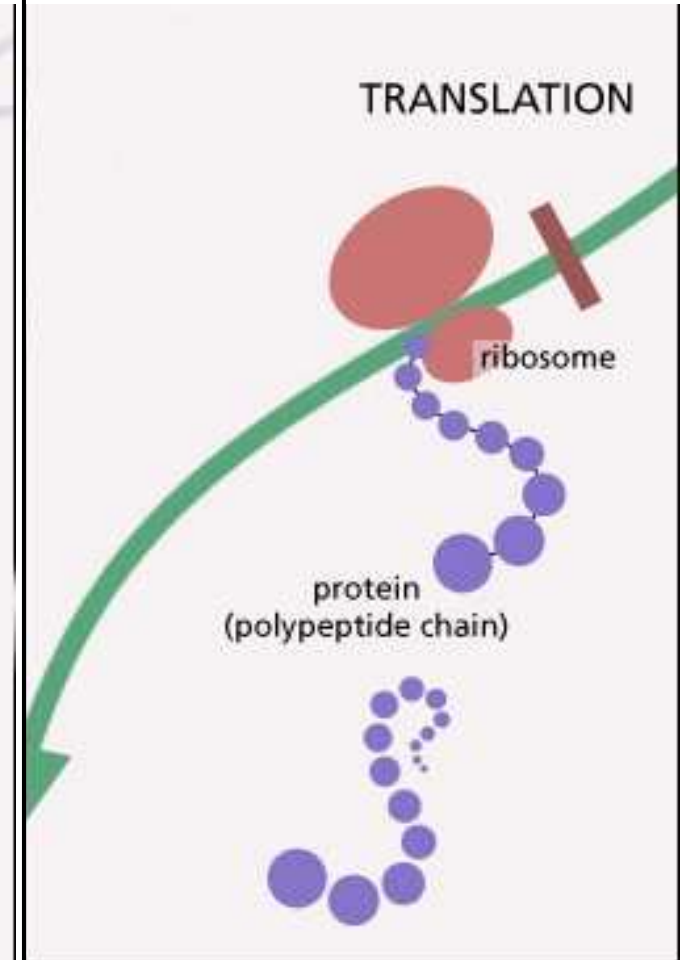
Now “gene” tends to be used to mean a *cistron*: a sequence of codons (groups of 3 base pairs) in the DNA chain of the chromosome that codes for the production of a particular protein, that ends up affecting the phenotype somehow.

The possible different genes at a particular position (locus) are the alleles.

↓ Inside Nucleus ↓



Outside Nucleus



Genes, loci, alleles, bits, encoding (2)

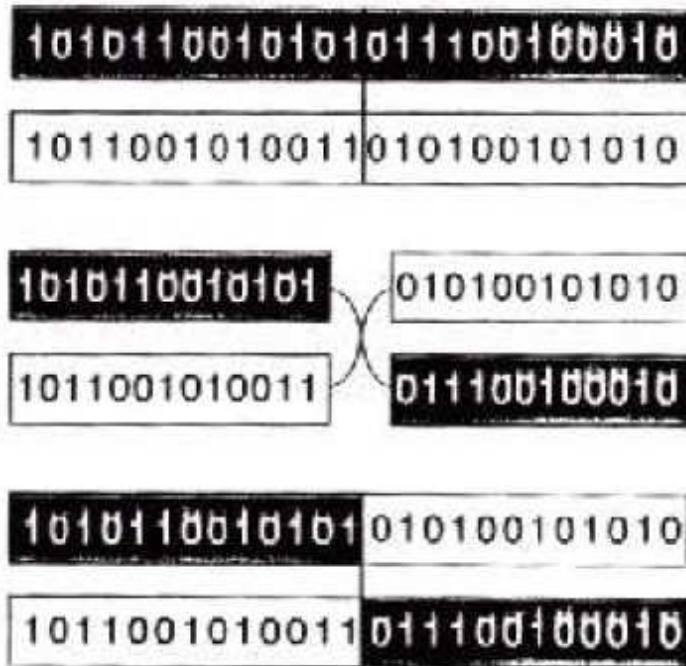
In the Glasgow online demo, a phenotype $x=4.179$, $y=2.486$ has genotype 41792486.

Do we call “7” a gene? Sometimes people would, but in EAs “gene” could mean various things: need to be clear.

Traditional GAs always used a binary genotype (bitstring) eg 1011101100...and mutation was just bit-flips.

It's often better to choose an encoding that more naturally fits your problem, then to design variation operators (like mutation) thoughtfully.

Crossover (1-point)



CROSSOVER is the fundamental mechanism of genetic rearrangement for both real organisms and genetic algorithms.

Chromosomes line up and then swap the portions of their genetic code beyond the crossover point.

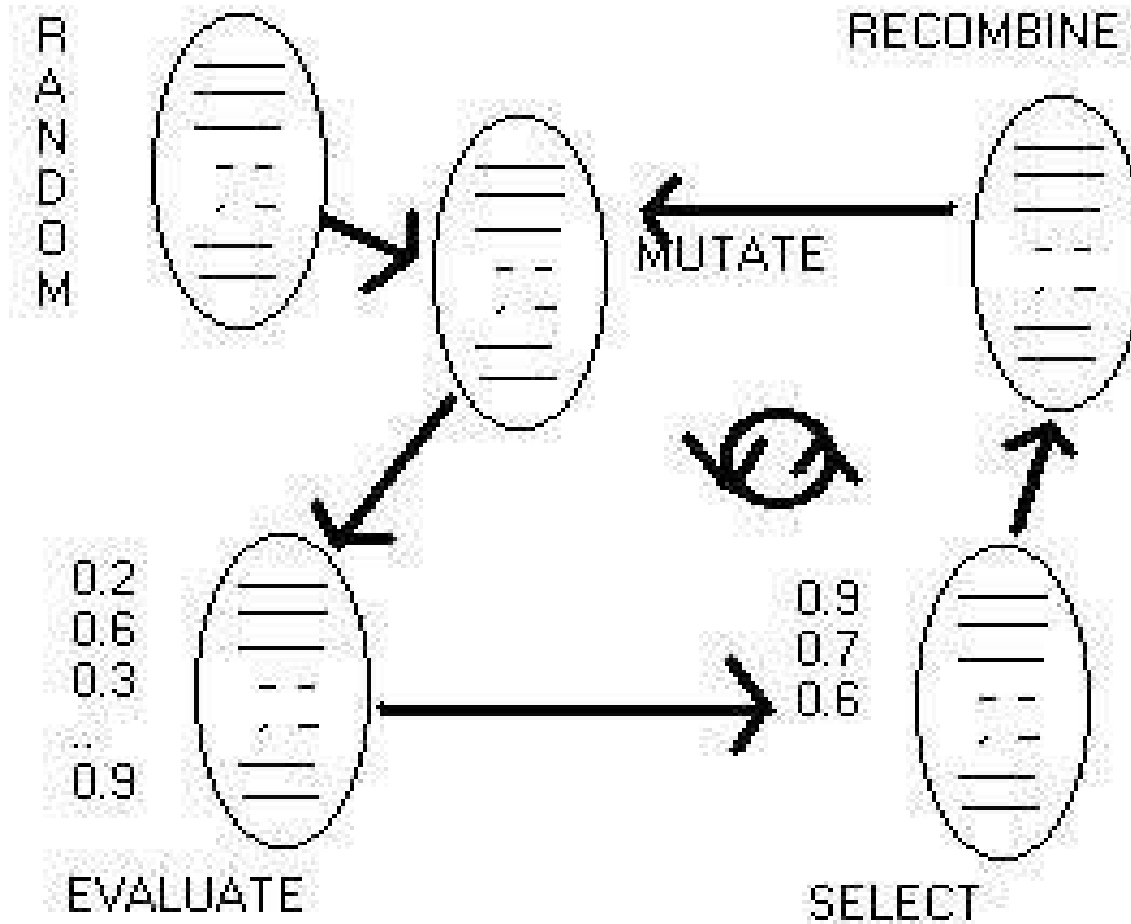
Theories of EA Crossover (Sexual Recombination)

1. In classic bitstring GAs, the schema theorem suggested that crossover allows parallel processing of the fitness contributions of every (partial) bit pattern present in the pop.
2. Combining disparate parents “casts a net” across the fitness landscape
3. Shuffling of “building blocks” found in different individuals.
4. Macromutation
5. Repair from deleterious mutation
6. Accelerate propagation of a beneficial mutation through the pop

1, 2 and to some extent 3: rely on the individuals being dissimilar, which they aren't after a some 10's-100's of generations unless the population is huge or special steps are taken.

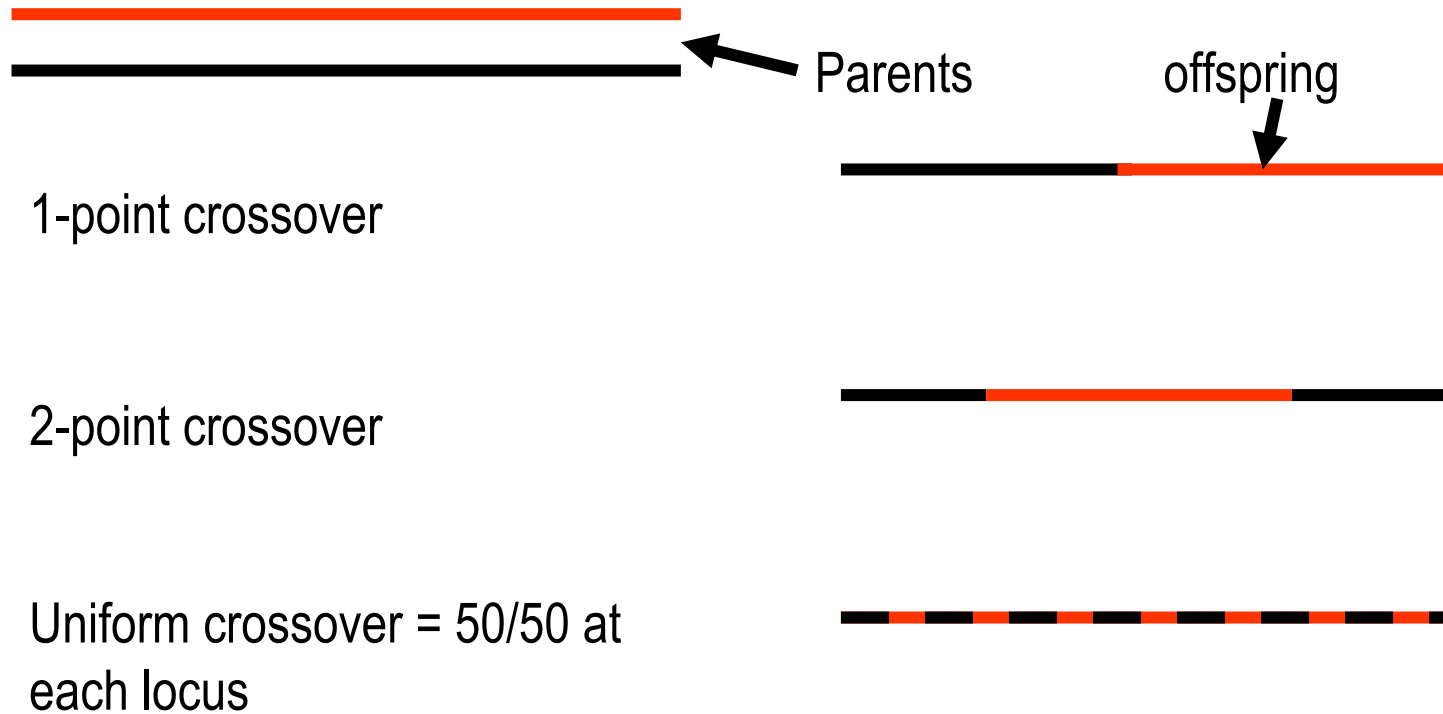
GA overview

John Holland "Adaptation in Natural and Artificial Systems" 1975



EA Recombination

Typically 2 parents recombine to produce an offspring



Mutation

After an offspring has been produced from two parents (if crossover is chosen, or just copied from one parent if not):



Mutate at randomly chosen loci with some probability



Or, equivalently, mutate every locus with some low probability.

Fitness & Selection

- In biology “fitness” = “number of offspring”
- In EAs, “fitness” = performance.
 - number of offspring is some function of this:

Eg:

- **Truncation**: Draw a line between “good” and “bad” and select randomly from the “good”. Animal breeders do this.
- probability of selection is **proportional to *fitness*** (common but can make selection too strong/weak)
- probability of selection is **proportional to the *rank*** of the individual in a sorted list of fitness.
Recommended. Maintains constant selective pressure through run.

Generational vs Steady State GA

- **Generational** breeds a whole new generation of offspring, then replaces parents en masse (eg. Glasgow online demo)
- A **steady-state** GA breeds one child, inserts it into the population, then just iterates
 - that's rather like a $(\mu+1)$ ES, but we can use crossover in the GA paradigm
- Both can work well

Easy Steady-State GA

1. Initialise random pop (eg. data structure:
`int pop[POPSIZE][GENLEN];`)
2. Pick 2 inds at random & evaluate them:
parent $p1$ is the better
3. Pick 2 inds at random & evaluate them:
parent $p2$ is the better
4. Pick a random ind d to die
5. Do crossover $p1 \times p2$ to give an offspring o in
 d 's place
6. Mutate o
7. Until success or give up, goto 2.

That was a steady-state GA with tournament selection (tournament size 2).

We could have also:

- Chosen p_1 & p_2 with probability proportional to their fitness or rank using a roulette wheel.
 - Actually, tournament selection does select proportional to rank, but noisily (the greater the tournament size, the less noise)
- Chosen d inversely proportional to fitness or rank (possibly just killing the worst individual)
- Not performed crossover every time

An example generational GA

1. Initialise random pop P
2. Evaluate entire pop
3. Allocate memory for a temporary pop P' (eg. another array, just like P . Perhaps P' is just set up once at the beginning.)
4. Copy best ind (the “elite”) from P to P' unaltered (“elitism”) --- this is optional but often helps
5. With some probability *select* two parents from P and form an offspring o in P' from crossover, otherwise form offspring o as a direct copy of one *selected* parent
6. Mutate o
7. Until P' is full, goto 5
8. Replace P with P'
9. Until success or give up, goto 2

A typical GA

- Population 100 (could be 5,10, 200000...)
- Elitism: make sure one of your equal-best individuals (chosen at random) is copied unaltered to the next generation.
- Parents selected with probability in proportion to their *rank* in the list of sorted fitnesses.
- 70% of offspring the result of crossover applied to two parents, the others just mutated copies of one parent
- Mutation applied to offspring with a per-locus probability such that on average about 1 change that really matters is made per individual (at any time some genes may be “junk” so the total number you aim for may be >1 . Try 3?)

Appendix: Implementing a roulette wheel

1. Loop through all the slots to find the sum S of their widths. (The widths could be from fitness, normalised fitness, rank, whatever your criterion is.)
2. Generate a random number p between 0 and S .
3. Loop through the slots again, keeping a running sum r of their width, until $r \geq p$
4. Return whatever individual corresponds to the current slot

Advanced Programming Trick for a roulette wheel for rank selection

- We know in advance that the slot widths will always be $0, 1, 2, 3, 4, \dots, \text{POPSIZE}-1$
- This forms an arithmetic series, and we can calculate its sum, S .
- We generate a random number p in $[0.0, S)$
- By inverting the formula for the sum of an arithmetic series, and solving the resulting quadratic equation, we can directly calculate which slot in the roulette wheel p is pointing at.
- The following is C code for this: *don't just use it blindly!* Must use an algorithm you can understand and test.
- If the parameter `inverse` is true, it chooses inversely proportional to rank

```
int rank[POPSIZE];
// rank[] should hold the indices of the inds in the pop in
// fitness-sorted rank order. Eg, population_array[rank[0]]
// is the fittest individual and
// population_array[rank[POPSIZE-1]] is the least fit

int select_individual(int inverse)
{
    double s, p;
    int n, ind;

    s = (POPSIZE*(POPSIZE-1))/2.0;
    p = random_real()*s;
    // Where random_real gives a real from 0 to 1.0
    n = (1+sqrt(1+ 8*p))/2.0;
    if (inverse)
        ind = rank[n];
    else
        ind = rank[POPSIZE-1-n];
    return ind;
}
```