*Blackbox Optimization:

- General Aim: Given a certain domain and target function defined over the domain find the minimum/maximum.

- If we know the analytic form → we can use gradient

If we don't know? x → problem → f(x) → Black box optimization!

Algorithm 9 *Random Search*

```
Best ← some initial random candidate solution
repeat
    S ← a random candidate solution
    if Quality(S) > Quality(Best) then
        Best ← S
until Best is the ideal solution or we have run out of time
return Best
```

Algorithm 4 *Hill-Climbing*

```
S ← some initial candidate solution
repeat
    R ← Tweak(Copy(S))
    if Quality(R) > Quality(S) then
        S ← R
until S is the ideal solution or we have run out of time
return S
```

Small modification/adjust. on current solution in order to explore nearby solutions

Algorithm 5 *Steepest Ascent Hill-Climbing*

```
n ← number of tweaks desired to sample the gradient

S ← some initial candidate solution
repeat
    R ← Tweak(Copy(S))
    for n − 1 times do
        W ← Tweak(Copy(S))
        if Quality(W) > Quality(R) then
            R ← W
    if Quality(R) > Quality(S) then
        S ← R
until S is the ideal solution or we have run out of time
return S
```

Algorithm 14 *Tabu Search*

```
l ← Desired maximum tabu list length
n ← number of tweaks desired to sample the gradient

S ← some initial candidate solution
Best ← S
L ← {} a tabu list of maximum length l        # first-in first-out
Enqueue S into L                              → kuyruğa ekle
repeat
    if Length(L) > l then
        Remove oldest element from L
    R ← Tweak(Copy(S))
    for n − 1 times do
        W ← Tweak(Copy(S))
        if W ∉ L and (Quality(W) > Quality(R) or R ∈ L) then
            R ← W
    if R ∉ L then
        S ← R
        Enqueue R into L
    if Quality(S) > Quality(Best) then
        Best ← S
until Best is the ideal solution or we have run out of time
return Best
```

L'nin elemanı dışında bir tweak bulmaya çalışıyoruz (R)

geri candidate listeye ekle

Algorithm 7 *Generate a Random Real-Valued Vector*

```
min ← minimum desired vector element value
max ← maximum desired vector element value

v⃗ ← a new vector ⟨v1, v2, ...vl⟩
for i from 1 to l do
    vi ← random number chosen uniformly between min and max inclusive
return v⃗
```

Algorithm 8 *Bounded Uniform Convolution*

```
v⃗ ← vector ⟨v1, v2, ...vl⟩ to be convolved
p ← probability of adding noise to an element in the vector
r ← half-range of uniform noise
min ← minimum desired vector element value
max ← maximum desired vector element value

for i from 1 to l do
    if p ≥ random number chosen uniformly from 0.0 to 1.0 then
        repeat
            n ← random number chosen uniformly from −r to r inclusive
        until min ≤ vi + n ≤ max
        vi ← vi + n
return v⃗
```

• Population Based Methods

- Idea: Evolve a population (multiset) of candidate solutions using the concepts of survival of the fitness, variation and inheritance

Initialization: Randomly generates the initial population of candidate solutions (String, real vectors)

Evaluation: Evaluates the population of candidate solutions using given fitness function

Selection: Selects promising solutions from current population by making more copies of better solutions

Variation: Processes selected solutions to generate new candidate solutions that share similarities with selected solutions but novel in some way

Replacement: Incorporates new candidate solutions into the original population

Algorithm 17 *An Abstract Generational Evolutionary Algorithm (EA)*

1: $P \leftarrow$ Build Initial Population
2: $Best \leftarrow \square$ ▷ $\square$ means "nobody yet"
3: **repeat**
4: AssessFitness(P)
5: **for** each individual $P_i \in P$ **do**
6: **if** $Best = \square$ or Fitness(P_i) > Fitness($Best$) **then** ▷ Remember, Fitness is just Quality
7: $Best \leftarrow P_i$
8: $P \leftarrow$ Join(P, Breed(P)) → Breed: mevcut P'den yeni bireyler doğurmak =)
9: **until** $Best$ is the ideal solution or we have run out of time
10: **return** $Best$

>> A Simple Genetic Algorithm:

- Generate the initial population

```
generate(n,N)
  P = new population of size N;
  for i=1 to N
    P[i] = generate_random(n);
  return P;
```

$P_1, \ldots, P_N$ (size n, size n)

- Evaluation

```
evaluate(P,f,n,N)
  fv = new array of N real numbers;
  for i=1 to N
    fv[i]=f(P[i],n);
  return fv;
```

N adet fv her bir P_i için

```
P = generate(n,N)
fv = evaluate(P,f,n,N)
S = selection(P,fv,n,N)
O = variation(S,P,n,N)
P = replacement(O,P,n,N)
```

n: # of bits
N: population size
f: objective function
pm: probability of mutation
pc: probability of crossover

```
binary_tournament_selection (P, fv, n, N)
  S = new population of Size N;
  for i=1 to N
    a=rand(1,N); → N adet random number
    b=rand(1,N-1); → N-1 adet random number
    if (b>=a) b++;
    if (fv[a]>fv[b])
      S[i]=P[a];
    else
      S[i]=P[b];
  return S;
```

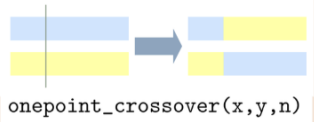
aynı bireyin iki kez seçilmesini önler

ex: a=1 b=1 → b=2
a=1 b=2 → b=3
a=1 b=3 → b=4

```
variation(S,n,N,pm,pc)
  O = new population of size N;
  shuffle(S,n,N); → randomly reorder strings in the population (P)
  for i=1 to N with step 2
    O[i] = mutation(S[i],n,pm);
    O[i+1] = mutation(S[i+1],n,pm);
    if (rand01()<pc) → rand01: random uniform (0,1)
      crossover(O[i],O[i+1],n);
  return O;
```

```
replace_all(O,P,n,N)
  new_P = new population of N strings;
  for i=1 to N
    new_P[i]=O[i];
  return new_P;
```

* crossover: combines bits and pieces of promising solutions



```
onepoint_crossover(x,y,n)
  cp=random(2,n);
  for i=cp to n
    exchange(x[i],y[i]);
```

```
twopoint_crossover(x,y,n)
  cp1=random(1,n);
  cp2=random(1,n);
  if (cp1>cp2)
    exchange(cp1,cp2);
  for i=cp1 to cp2
    exchange(x[i],y[i]);
```

* Mutation: makes small perturbations to promising solutions

```
mutation(x,n)
  y=new binary string of n bits;
  for i=1 to n
    if (rand01()<pm)
      y[i]=1-x[i];
    else
      y[i]=x[i];
```

>> Real-Coded Genetic Algorithms

- To extend the same principle to another coding, we need to implement new variation operators

- Real-Valued Operators: Candidate solutions are vectors of real values
  Random initialization generates a random number from an interval
  Operators must deal with real-valued parameters

- Arithmetic Crossover:

  Two real-valued parents: $x = (x_1, \dots, x_n)$, $y = (y_1, \dots, y_n)$

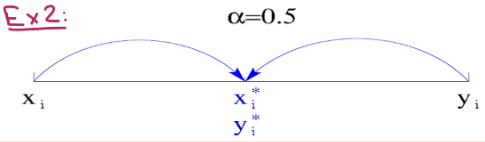
  Choose random variable $i \in (1, 2, \dots, n)$

  First child is: $o_1 = (x_1, \dots, x_{i-1}, \alpha y_i + (1-\alpha) x_i, y_{i+1}, \dots, y_n)$

  Second child: $o_2 = (y_1, \dots, y_{i-1}, \alpha x_i + (1-\alpha) y_i, x_{i+1}, \dots, x_n)$

Ex1: $\alpha = 0.25$

Ex2: $\alpha = 0.5$



- Simple Mutation:

  Change each variable with a fixed probability

  To change a variable, start by generating a random number from $[-\delta, \delta]$ where $\delta > 0$ is a small number (Add that number to the variable)

- Gaussian Mutation:

  Change is generated according to Gaussian Dist. $N(0, \sigma^2)$ → σ^2 variance of mutation steps (small number)

- Permutations:

  Ordering and sequencing problems form a special class

  Ex: Travelling Salesman Problem (TSP)

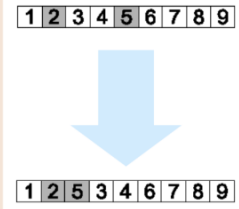
  - Given n cities, find a complete tour with min. length
  - Label the cities 1, 2, ..., n
  - One complete tour is one permutation (ex: n=4; [1 2 3 4] or [3 4 1 2] are Ok.)
  - Search space is huge → for 30 cities, 30! possible tours

  >> Can we apply genetic algorithm to TSP?

  YES. But crossover and mutation need to be adapted to the new representation

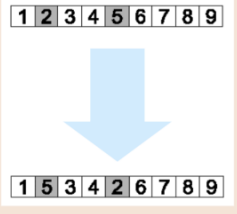
  → Single crossover and mutation operators lead to inadmissible solutions

- Insert Mutation for Permutations



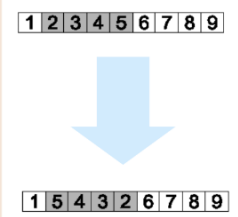
- Pick two allele (feature) values at random
- Move the second to follow the first, shift the rest
- This preserves most of the order and adjacency info

- Swap Mutation for Permutations



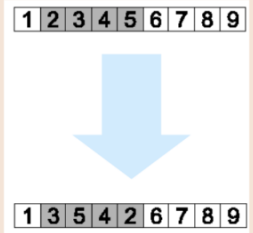
- Pick two alleles at random and swap their position
- Preserves most of adjacency info, disrupts order more

- Inversion Mutation for Permutations



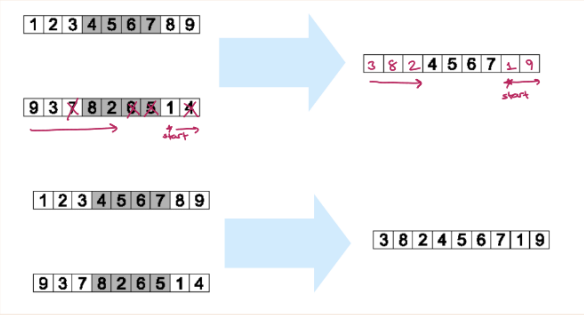
- Pick two alleles at random and invert the substring between them
- Preserves more adjacency info but disruptive of order info

- Scramble Mutation for Permutations



- Pick a subset of genes at random
- Randomly rearrange the alleles in those positions

• Order-1 Crossover for Permutations



- Choose an arbitrary part from first parent
- Copy this part to the first child
- Copy numbers that are not in the first parent to the first child
  (Start right from the cut point of the copied part, using the order of second parent and wrapping around at the end)
- Same for the second child with parent roles reversed.