1. 이론

1 유전 알고리즘의 개요 및 구성

생물계 진화 과정과 유전 알고리즘

유전 알고리즘은 생물계의 진화 과정을 모방한 휴리스틱 해법입니다.

생물계의 진화 과정은 적자생존이라고 요약할 수 있음

환경에 적합한 유전자를 가진 개체는 살아남아 자식을 퍼뜨리지만, 그렇지 않은 유전자를 가진 개체는 자식을 퍼뜨리지 못함

환경에 적합한 유전자를 가진 개체가 교배해 자식을 만드는 과정에서 유전자가 섞이고 또 우연히 돌연변이가 탄생하기도 함

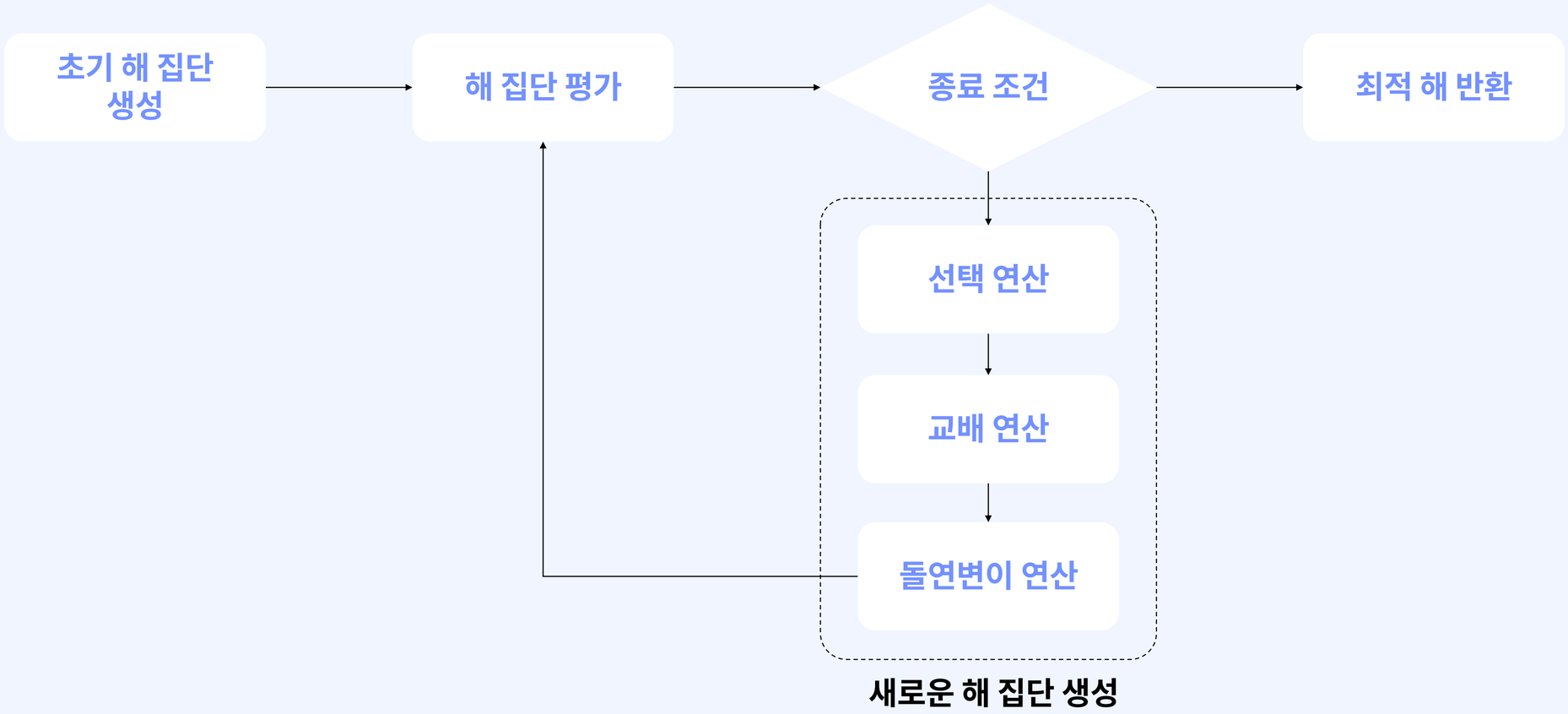
세대의 진화를 반복하며 다양한 유전자를 가진 개체가 태어나고 환경에 가장 적합한 유전자가 살아남

생물계 진화 과정과 유전 알고리즘 (계속)

유전 알고리즘은 생물계의 진화 과정을 모방한 휴리스틱 해법입니다.

생물계 진화 과정	유전 알고리즘
유전자	해
환경에 대한 적합성	적합도 함수(fitness function)
환경에 적합한 유전자가 살아남음	선택 연산: 적합도가 큰 해를 선택함
두 개체가 교배하여 자식을 낳음	교배 연산: 두 해를 섞음
돌연변이가 태어나기도 함	돌연변이 연산: 임의로 해를 일부 수정함

알고리즘의 구성



알고리즘의 구성
(상세)

(1) 초기 해 집단 생성: 임의로 여러 개의 초기 해를 만들어 집단을 구성합니다. 매 이터레이션에서의 해 집단을 세대(generation)라고 부릅니다.
즉, 초기 해 집단은 유전 알고리즘에서 첫 세대가 됩니다.

↓

(2) 해 집단 평가: 한 세대의 모든 해를 적합도 함수를 사용해 평가합니다. 즉, 각 해의 적합도를 계산합니다.

↓

(3) 종료 조건 평가: 최대 이터레이션 횟수에 도달하거나 해가 수렴하는 등의 종료 조건을 만족하는지 확인합니다.

만약 종료 조건을 만족하면 현재까지 탐색한 해 가운데 적합도가 가장 높은 해를 반환하고 알고리즘을 종료합니다. 그렇지 않으면 (4)로 갑니다.

↓

(4) 새로운 해 집단 생성: 선택 연산, 교배 연산, 돌연변이 연산을 사용해 새로운 해 집단을 만들고 (2)로 되돌아갑니다.

(4-1) 선택 연산: 다음 세대를 구성할 해를 선택합니다. 이때, 해의 다양성을 위해 적합도가 높은 해만 선택하지 않고 나쁜 해를 일부 선택하기도 합니다.

(4-2) 교배 연산: (4-1)에서 선택된 해 가운데 두 개를 임의로 선택하여 교배 연산을 적용하는 과정을 반복하여 새로운 해를 생성합니다.
세대별 해의 개수가 같도록 (이전 세대에 있는 해의 개수 – 선택한 해의 개수)만큼 생성하는 것이 보통입니다.

(4-3) 돌연변이 연산: 해의 다양성을 위해 교배 연산을 통해 만든 해 일부에 돌연변이 연산을 적용합니다.

특징 및 장단점

유전 알고리즘은 다른 최적화 알고리즘과 다르게, 동시에 여러 해를 탐색함

해 간 연산도 수행하므로 하나의 해를 탐색하는 알고리즘을 병렬적으로 실행하는 것보다 더 나은 결과를 기대할 수 있음

목적 함수가 미분 가능하지 않더라도 무리없이 적용할 수 있음

해를 표현하는 방법부터 교차 연산, 돌연변이 연산, 적합도 함수 등을 문제에 맞게 설계하거나 선택해야 함

유전 알고리즘은 매우 좋은 결과를 낼 수도 있고 매우 나쁜 결과를 낼 수도 있는 불안정한 알고리즘임

주요 하이퍼 파라미터

세대 수 및 해 집단 크기	• 탐색 공간이 크면 유전 알고리즘이 수렴하기는 매우 어렵기 때문에, 보통 최대 이터레이션 횟수(세대 수)에 도달하여 알고리즘이 종료됨 • 세대 수가 크면 클수록 시간은 오래 걸리지만 더 좋은 해를 찾을 가능성이 커짐 • 비슷한 논리로 해 집단에 속하는 유전자가 많으면 많을수록 시간은 오래 걸리는 대신에 더 좋은 해를 찾을 가능성이 커짐
선택 연산자	• 해마다 적합도 차이가 클 때 룰렛 휠을 사용하면 나쁜 해가 선택될 가능성이 적지만, 순위 선택을 사용하면 상대적으로 나쁜 해가 선택될 가능성이 큼 • 엘리트주의로만 해를 선택하면 나쁜 해가 선택될 가능성은 없음
교차 연산자	• 해를 얼마나 복잡하게 섞느냐에 따라 교차 연산자를 구분할 수 있음 • 해를 복잡하게 섞을수록 다양한 해를 탐색할 수 있어 기대치 못한 좋은 해를 찾을 가능성이 커지고 수렴할 가능성은 작아짐
돌연변이 연산자	• 돌연변이 연산자는 원래 유전자와 얼마나 다른 유전자를 만드느냐에 따라 구분할 수 있음 • 예를 들어, 비트 플립 돌연변이 연산자에서 각 요소를 선택할 확률이 클수록 다양한 해를 탐색하게 됨

1. 이론

2 유전자 표현 방법

이진 인코딩

이진 인코딩(binary encoding)은 가장 널리 사용되는 해 표현 방법으로, 모든 해를 0과 1, 혹은 False와 True로 구성된 이진 벡터로 표현합니다.

이진 인코딩

유전자 A	1	0	1	0	0	0	1	1	0	1
유전자 B	0	0	1	0	1	0	1	1	1	1

- 이진 벡터는 교배 연산과 돌연변이 연산을 적용하기에 좋은 구조임
- 그러나 표현할 수 있는 범위가 한정적이라는 단점이 있음
- 이진 인코딩은 특징 선택을 비롯한 조합 최적화(combinational optimization) 문제의 해를 표현하는 데 주로 활용함

이진 인코딩 구현

이진 인코딩(binary encoding)은 가장 널리 사용되는 해 표현 방법으로, 모든 해를 0과 1, 혹은 False와 True로 구성된 이진 벡터로 표현합니다.

이진 인코딩 구현 예제

```
1  import numpy as np
2  def binary_init(n, m, bool_type = False):
3      X = np.random.choice([0, 1], (n, m))
4      if bool_type:
5          X = X.astype(bool)
6      return X
```

- **라인 2:** 해 개수(n), 해의 길이(m), 부울을 반환할지(bool_type)를 입력받아 이진 인코딩 구조의 초기 해를 생성하는 binary_init 함수를 작성합니다.
- **라인 3:** np.random.choice를 이용해 0과 1 가운데 하나를 임의로 골라 (n, m) 크기의 배열을 만듭니다.
- **라인 4~5:** bool_type이 True라면 astype 메서드를 사용해 X의 데이터 타입을 bool로 변환합니다.

이진 인코딩 구현

이진 인코딩(binary encoding)은 가장 널리 사용되는 해 표현 방법으로, 모든 해를 0과 1, 혹은 False와 True로 구성된 이진 벡터로 표현합니다.

이진 인코딩 구현 예제

```
1  n = 5
2  m = 3
3  display(binary_init(n, m, bool_type = False))
4  display(binary_init(n, m, bool_type = True))
```

```
array([[1, 0, 0],
       [0, 1, 0],
       [0, 1, 1],
       [1, 0, 0],
       [1, 1, 1]])
```

```
array([[ True,  True,  True],
       [ True,  True,  True],
       [ True, False,  True],
       [ True,  True, False],
       [ True, False,  True]])
```

- 실행할 때마다 임의로 초기해가 만들어지므로 결과가 달라짐
- 시드를 고정하면 같은 결과가 나오겠지만, 결과를 재현하는 것이 중요하지 않고 어느 정도 임의성에 기대므로 시드를 고정하지 않음

순열 인코딩

순열 인코딩(permutation encoding)은 외판원 순회 문제와 같이 순서를 결정하는 문제에 주로 사용하는 해 표현 방법으로 각 해가 순서를 나타냅니다.

순열 인코딩

유전자 A	1	2	3	4	5	6	7	8	9	10
유전자 B	3	5	1	2	7	6	9	10	8	4

- 순서를 나타내므로 요소 간 중복이 없음
- 교차 연산과 변이 연산 등을 했을 때 실행 가능하지 않은 해가 만들어질 위험이 있음

순열 인코딩

순열 인코딩(permutation encoding)은 외판원 순회 문제와 같이 순서를 결정하는 문제에 주로 사용하는 해 표현 방법으로 각 해가 순서를 나타냅니다.

순열 인코딩 구현 예제

```
1  def permutation_init(n, m):
2      X = [np.random.permutation(m) for _ in range(n)]
3      X = np.array(X)
4      return X
```

- **라인 2:** np.random.permutation을 사용해 [0, 1, 2, ⋯, m-1]을 임의로 정렬한 배열 n 개를 만들어 X에 리스트로 저장합니다. 즉, X[i]는 임의로 정렬된 i번째 순열입니다.
- **라인 3:** 리스트를 ndarray로 변환합니다.

```
1  n = 2
2  m = 5
3  permutation_init(n, m)
```

```
array([[4, 2, 3, 1, 0],
       [0, 2, 3, 4, 1]])
```

값 인코딩

값 인코딩(value encoding)은 해를 실수 벡터 형태로 표현하는 방법으로, 해의 범위나 해가 따르는 확률 분포 등을 활용하여 해를 생성합니다.

값 인코딩

유전자 A	1.4	2.3	-0.2	0.8	4.1
유전자 B	12.1	-2.4	3.8	5.5	3.0

- 유전 알고리즘으로 신경망을 비롯한 머신러닝 모델의 가중치를 추정하는 데 사용하는 해 표현 방식이기도 함
- 이 표현 방법은 거의 모든 종류의 해를 표현할 수 있음
- 교차 및 돌연변이 연산을 정의하기 어려움

1. 이론

3 적합도와 선택 연산

적합도

적합도란 각 해가 얼마나 문제의 답으로 적합한지를 평가하기 위한 함수로, 목적 함수를 사용하는 것이 일반적입니다.

(예시)

- **목적 함수:** $\text{minimize } f(x_1, x_2) = x_1^2 + x_2^2$
- **적합도 함수:** $\frac{1}{f(x_1,x_2)+1}$

유전자 A	2	3	적합도 = 1/14
유전자 B	1	0	적합도 = 1/2
유전자 C	-1	1	적합도 = 1/3

np.apply_along_axis 함수

apply_along_axis는 이름 그대로 ndarray의 축에 따라 일괄적으로 함수를 적용하며, 판다스의 apply와 유사합니다.

주요 인자

인자	설명
func1d	ndarray의 각 요소에 일괄적으로 적용할 함수로, 이 함수의 입력은 1차원 배열입니다.
axis	함수를 적용할 축을 의미합니다.
arr	함수를 적용할 배열입니다.

np.apply_along_axis 함수 (계속)

일반적으로 1차원 ndarray로 해를 정의하고 2차원 ndarray로 해집합을 정의하므로, np.apply_along_axis를 이용해 각 해의 적합도를 평가합니다.

적합도 함수 예시

```
1  def fitness(x):
2      return sum(x * np.array([10, 1, 2, 5])) + 3
```

- 크기가 4인 이진 벡터 x와 (10, 1, 2, 5)의 내적에 3을 더한 값이 적합도입니다.

적합도 계산 예시

```
1  X = binary_init(5, 4, bool_type = False)
2  S = np.apply_along_axis(fitness, 1, X)
3  display(S)
```

- **라인 2**: 각 행에 함수를 적용하기 위해 axis를 1로 설정했습니다. axis는 계산 방향을 설정하는 인자로, axis = 0은 행 방향(위에서 아래 방향 ↓)으로 계산하고, axis = 1은 열 방향(왼쪽에서 오른쪽 →)으로 계산합니다.

```
array([ 8, 20, 10, 15,  8])
```

axis 인자에 대한 이해

axis 인자는 apply_along_axis 함수뿐만 아니라 넘파이와 판다스의 다양한 함수에서 사용되므로 반드시 그 내용을 이해하고 넘어가야 합니다.

(예시) sum 메서드

0	1	2
3	4	5
6	7	8
9	10	11
12	13	14

열방향:
x.sum(axis = 1)

3
12
21
30
39

행방향:
x.sum(axis = 0)

30	35	40

- axis = 1은 열 방향(왼쪽에서 오른쪽)으로 계산함을 나타냄
- axis = 0은 행 방향(위에서 아래 방향)으로 계산한다는 것을 나타냄
- 연산 결과가 1차원이라면 axis 값과 배열 방향이 같음.
  즉, axis = 0이면 행벡터 꼴의 결과가 나오며 axis = 1이면 열벡터 꼴의 결과가 나옵니다.

룰렛 휠 선택

룰렛 휠(roulette wheel) 선택은 각 해를 적합도 에 비례하여 확률적으로 선택합니다.

n 개의 후보 해 $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$의 적합도를 $s_1, s_2, \cdots, s_n$이라 한다면, $\boldsymbol{x}_i$가 선택될 확률 p_i는 다음과 같이 정의됨

$$p_i = \frac{s_i}{\sum_{i'=1}^{n} s_{i'}}$$

단, 같은 해가 중복해서 선택되는 것을 방지하기 위해 이미 선택된 해는 후보에서 제외함

다항 분포

다항 분포는 이산형 확률 분포로, n번의 독립 시행에서 k번째 값이 m(≤n)번 나올 확률을 정의합니다.

확률 질량 함수

$$\Pr(x_1, x_2, \cdots, x_k) = \frac{n!}{\prod_{i=1}^{k} x_i!} \prod_{i=1}^{k} p_i^{x_i}$$

- n: 시행 횟수
- x_i: 시행 중 i번째 값이 나온 횟수
- p_i: i번째 값이 나올 확률

np.random.multinomial 함수

- np.random.multinomial(n, probs)는 각 값이 선택될 확률이 probs일 때 n개의 값을 선택한 결과를 반환함
- np.random.multinomial(1, probs)과 argmax를 이용하면 하나의 값을 확률적으로 샘플링할 수 있음 [1)]

```
result1 = np.random.multinomial(5, [0.2, 0.3, 0.5]) # [1, 1, 3]
result2 = np.random.multinomial(1, [0.2, 0.3, 0.5]) # [0, 0, 1]
np.argmax(result2) # 2
```

[1)] 엄밀히 말해, 하나의 값만 샘플링하는 경우 카테고리컬 분포(categorical distribution)이라 함

룰렛 휠 선택 구현

룰렛 휠(roulette wheel) 선택은 각 해를 적합도에 비례하여 확률적으로 선택합니다.

룰렛 휠 선택 예제

```
1  def roulette_wheel(X, S, k):
2      selected_index = []
3      _S = S.copy()
4      for _ in range(k):
5          probs = _S / _S.sum()
6          x_idx = np.random.multinomial(1, probs).argmax()
7          selected_index.append(x_idx)
8          _S[x_idx] = 0
9      return X[selected_index]
```

- **라인 1:** 해 집단 X, 적합도 점수 목록 S, 선택할 해의 개수 k를 입력받는 함수 roulette_wheel을 작성합니다.
- **라인 2:** 선택된 인덱스 목록 selected_index를 빈 리스트로 초기화합니다. 이 리스트는 라인 9에서 X의 인덱스로 사용합니다.
- **라인 3:** S의 요소가 바뀌는 것을 방지하기 위해 copy 메서드를 이용해 S를 복사해서 _S에 저장합니다.
- **라인 5:** 점수 목록 _S를 _S의 합으로 나눠 각 해가 선택될 확률 probs를 계산합니다.
- **라인 6:** probs를 파라미터로 하는 다항 분포(multinomial distribution)로부터 하나의 해를 샘플링하고 argmax를 사용해 가장 큰 값인 1의 인덱스를 x_idx에 저장합니다. 즉, x_idx는 선택된 값의 인덱스라고 할 수 있습니다.
- **라인 7:** x_idx를 selected_index에 추가합니다.
- **라인 8:** 인덱스가 x_idx인 요소가 다시 선택되지 않도록 해당 해의 적합도 점수를 0으로 바꿉니다.

```
1  selected_X = roulette_wheel(X, S, 3)
2  display(selected_X)
```

- **라인 1:** 룰렛 휠 선택을 이용해 세 개의 해를 선택합니다.

```
array([[0, 0, 1, 1],
       [1, 0, 1, 1],
       [1, 0, 1, 0]])
```

순위 선택

순위 선택 방법은 적합도 점수를 기준으로 매긴 순위를 바탕으로 룰렛 휠 선택을 적용합니다. 즉, 적합도가 가장 큰 해가 n점을, 가장 낮은 해가 1점을 갖도록 점수를 순위로 변환하여 룰렛 휠 선택을 적용합니다.

n 개의 후보 해 $\boldsymbol{x}_1, \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n$의 적합도를 $s_1, s_2, \cdots, s_n$이라 한다면, $\boldsymbol{x}_i$가 선택될 확률 p_i는 다음과 같이 정의됨

$$p_i = \frac{\mathrm{Rank}(s_i)}{\sum_{i'=1}^{n} \mathrm{Rank}(s_{i'})}$$

단, 같은 해가 중복해서 선택되는 것을 방지하기 위해 이미 선택된 해는 후보에서 제외함

룰렛 휠 선택의 단점 해결

(1) 적합도에 비례해서 해를 선택하기에 적합도가 양수여야 함

(2) 하나의 적합도가 다른 적합도보다 훨씬 크다면 다른 해가 선택될 가능성이 거의 없음

순위 선택 구현

순위 선택 방법은 적합도 점수를 기준으로 매긴 순위를 바탕으로 룰렛 휠 선택을 적용합니다. 즉, 적합도가 가장 큰 해가 n점을, 가장 낮은 해가 1점을 갖도록 점수를 순위로 변환하여 룰렛 휠 선택을 적용합니다.

순위 선택 예제

```
1  from scipy.stats import rankdata
2  def rank_selection(X, S, k):
3      rank = rankdata(S)
4      return roulette_wheel(X, rank, k)
```

- **라인 3:** scipy.stats.rankdata 함수를 활용해서 S의 순위를 계산합니다.
- **라인 4**: 이전에 작성한 roulette_wheel 함수를 rank에 적용합니다.

```
1  display(rank_selection(X, S, 3))
```

- **라인 1:** 순위 선택을 이용해 세 개의 해를 선택합니다.

```
array([[1, 0, 1, 0],
       [1, 0, 1, 1],
       [0, 0, 0, 1]])
```

엘리트주의

엘리트주의(elitism)란 가장 좋은 k 개의 해가 선택되지 않는 것을 방지하기 위해 일부 해를 결정론적으로 선택합니다. 다시 말해, 적합도 점수가 높은 상위 $k \leq n$ 개의 해를 고른 다음, 나머지 해는 룰렛 휠이나 순위 선택 등을 통해 선택합니다.

엘리트 주의 예제

```
1  def elitism(X, S, k1, k2):
2      elite_index = (-S).argsort()[:k1]
3      not_elite_index = (-S).argsort()[k1:]
4      selected_X1 = X[elite_index]
5      selected_X2 = roulette_wheel(X[not_elite_index], S[not_elite_index], k2)
6      selected_X = np.vstack([selected_X1, selected_X2])
7      return selected_X
```

- **라인 1:** 엘리트주의를 이용해 선택할 해의 개수 k1과 다른 방법을 이용해 선택할 해의 개수 k2를 입력합니다.
- **라인 2:** argsort를 이용해 값이 큰 상위 k1 개의 인덱스를 elite_index에 저장합니다. argsort는 한 배열에서 값이 작은 인덱스를 먼저 배치하므로 (-S)를 사용해 값이 큰 인덱스를 먼저 배치했습니다.
- **라인 4:** elite_index를 사용해 상위 k1 개의 해를 selected_X1에 저장합니다.
- **라인 5:** roulette_wheel 함수를 사용해 선택되지 않은 X[not_elite_index]에서 k2 개의 해를 선택해 selected_X2에 저장합니다.
- **라인 6~7:** np.vstack을 이용해 selected_X1과 selected_X2를 행 방향으로 병합한 selected_X를 반환합니다.

```
1  display(elitism(X, S, 2, 1))
```

- **라인 1:** 엘리트주의로 2개의 해를, 그 외 방법으로 1개의 해를 선택합니다.

```
array([[1, 0, 1, 1],
       [1, 0, 1, 0],
       [0, 0, 0, 1]])
```

1. 이론

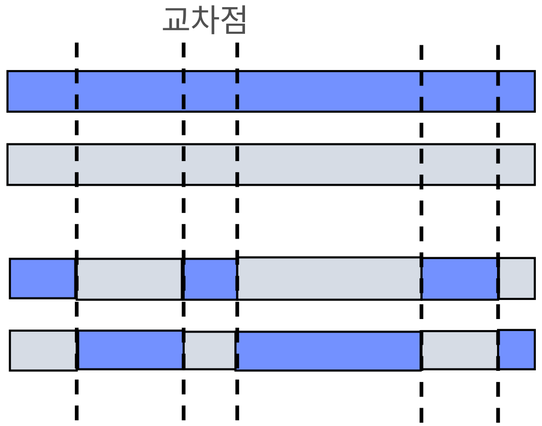
4 교차 연산과 돌연변이 연산

이진 인코딩에 대한 교차 연산자

이진 인코딩에 대한 교차 연산자는 임의로 선택한 교차점을 바탕으로 두 유전자를 섞습니다.

한 점 교차 연산자

두 점 교차 연산자



유니폼 교차 연산자

- 교차점을 임의로 생성하고 교차점을 기준으로 부모 유전자를 섞음
- 한 점 교차 연산은 교차점을 하나만 만들므로 상대적으로 단순하게 섞임
- 유니폼 교차 연산은 셋 이상의 교차점을 만들므로 복잡하게 섞임
- 즉, 한 점 교차 연산은 부모와 아주 비슷한 자식을 만들지만, 유니폼 교차 연산은 상대적으로 덜 비슷한 자식을 만듦

이진 인코딩에 대한 교차 연산자 구현

교차점의 개수를 입력받는 일반화된 연산자를 구현해보겠습니다.

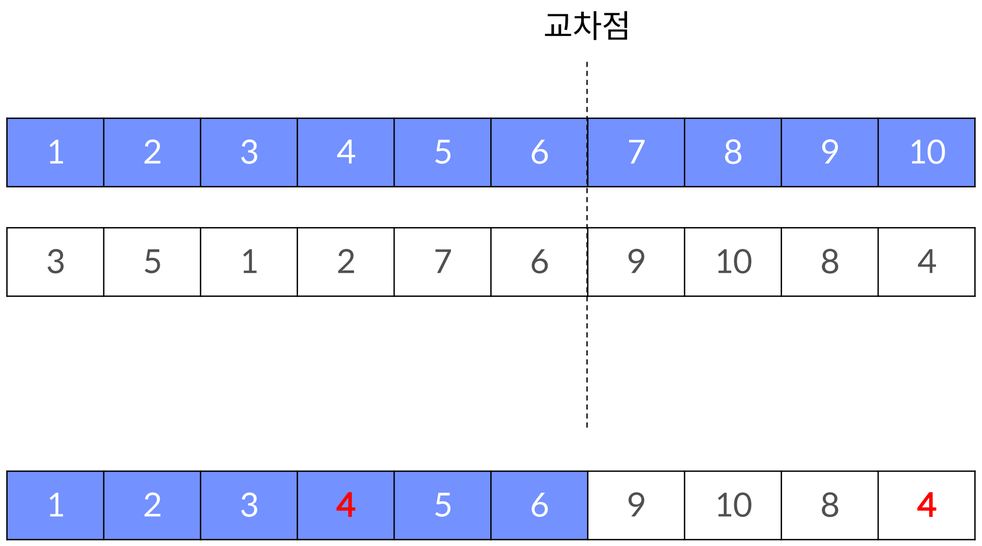
이진 인코딩에 대한 교차 연산 예제

```
1   def binary_crossover(X1, X2, num_points):
2       point_idx_list = np.random.choice(range(1, len(X1)), num_points, replace = False)
3       point_idx_list = np.insert(point_idx_list, 0, 0)
4       point_idx_list = np.insert(point_idx_list, num_points, len(X1))
5       point_idx_list.sort()
6       new_X = np.array([])
7       parent_idx = 0
8       for start_idx, end_idx in zip(point_idx_list[:-1], point_idx_list[1:]):
9           if parent_idx == 0:
10              new_X = np.hstack([new_X, X1[start_idx:end_idx]])
11          else:
12              new_X = np.hstack([new_X, X2[start_idx:end_idx]])
13          parent_idx = 1 - parent_idx
14      return new_X.astype(int)
```

- **라인 1:** num_points개의 교차점을 기준으로 부모 해 X1과 X2를 섞는 binary_crossover를 정의합니다.
- **라인 2:** 1부터 len(X1) 사이의 값 가운데 중복을 허락하지 않고 num_points개만큼의 수를 뽑아 point_idx_list에 저장합니다.
- **라인 3 - 4:** np.insert를 사용해 point_idx_list의 맨 앞에 0을, 맨 뒤에 len(X1)을 추가합니다.
- **라인 5:** sort 메서드를 이용해 교차점을 오름차순으로 정렬합니다.
- **라인 6~7:** 교차 연산을 통해 만들어지는 새로운 해 new_X를 빈 배열로, 선택할 부모의 인덱스 parent_idx를 0으로 초기화합니다.
- **라인 8:** zip 함수를 사용해 start_idx는 point_idx_list[:-1]을, end_idx는 point_idx_list[1:]을 순회합니다. 즉, start_idx가 point_idx_list[i]를 순회할 때 end_idx는 point_idx_list[i + 1]을 순회합니다.
- **라인 9~12:** parent_idx에 따라 X1과 X2를 선택하고 선택한 해의 start_idx부터 end_idx까지 슬라이싱한 배열을 new_X에 추가합니다.
- **라인 13:** 1 – parent_idx를 사용해 parent_idx가 0이면 1로, 1이면 0으로 변환합니다.
- **라인 14:** 연산 과정에서 자료형이 float으로 변환되므로 astype을 사용해 다시 자료형을 int로 바꿔줍니다.

순열 인코딩에 대한 교차 연산자

순열 인코딩으로 표현한 해에 대해 교차점 기반의 교차 연산자를 사용하면 실행 가능하지 않은 해가 생성될 수 있습니다.



- 4가 반복해서 등장함
- 7이 등장하지 않음

순열 인코딩에 대한 교차 연산자 (계속)

순열 인코딩으로 표현한 해에 대한 대표적인 교차 연산자로는 순서가 있는 교차 연산자(order crossover operator)를 들 수 있습니다. 순서가 있는 교차 연산자는 하나의 부모 유전자의 일부 배열을 그대로 가져오고 나머지 값은 다른 부모의 유전자 순서를 따르는 것입니다.

- 파란색 부모 유전자에서 [3, 6, 2, 5, 1]을 위치 그대로 가져옴
- 빈자리에는 파란색 부모 유전자에서 가져오지 않은 [8, 4, 7, 9, 0]을 채워야 함
- 하얀색 부모 유전자에서 [8, 4, 7, 9, 0]의 순서는 [0, 4, 7, 8, 9]이므로, 맨 앞 세 자리를 [0, 4, 7]로 맨 뒤 두 자리를 [9, 0]으로 채웁니다.

순열 인코딩에 대한 교차 연산자 구현

파이썬으로 순서가 있는 교차 연산자를 구현해보겠습니다.

순서가 있는 교차 연산 예제

```
1  def order_crossover(X1, X2):
2      start_idx = np.random.choice(range(0, len(X1)))
3      end_idx = np.random.choice(range(start_idx+1, len(X1) + 1))
4      new_X = np.empty(len(X1))
5      new_X[start_idx:end_idx] = X1[start_idx:end_idx]
6      new_X[~np.isin(X1, X1[start_idx:end_idx])] = X2[~np.isin(X1, X1[start_idx:end_idx])]
7      return new_X.astype(int)
```

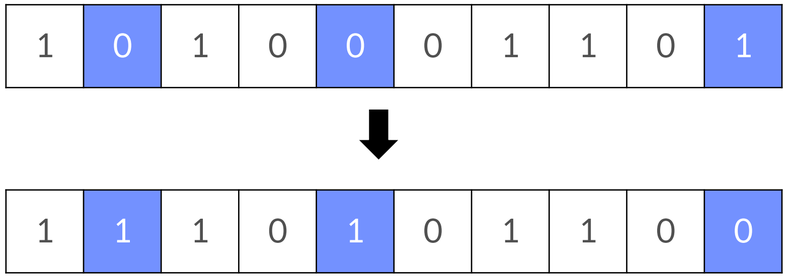
- **라인 2~3:** 선택된 유전 배열의 시작 인덱스인 start_idx와 종료 인덱스인 end_idx를 각각 설정합니다. 이때 end_idx가 start_idx보다 크게 설정합니다.
- **라인 4:** 자식 유전자를 길이가 len(X1)인 빈 배열로 초기화합니다.
- **라인 5:** start_idx부터 end_idx까지의 값은 X1의 값으로 채웁니다.
- **라인 6:** new_X의 요소 가운데 X1의 값으로 채워지지 않은 요소를 X2의 요소로 채웁니다.
- **라인 7:** 연산 과정에서 자료형이 바뀔 수 있으므로 int 형으로 자료형을 바꿔서 new_X를 반환합니다.

```
1  X1 = np.arange(1, 11)
2  X2 = np.arange(10, 0, -1)
3  display(order_crossover(X1, X2))
```

```
array([10,  9,  6,  5,  4,  3,  2,  7,  8,  1])
```

이진 인코딩에 대한 돌연변이 연산자

비트 플립(bit flip) 연산자는 유전자의 각 요소를 확률 p로 선택하여 0을 1로, 1을 0으로 바꿉니다.



- 2, 5, 10번째 요소를 확률 p로 선택했음
- 선택한 요소가 0이면 1로, 1이면 0으로 변경함

이진 인코딩에 대한 돌연변이 연산자 구현

비트 플립 돌연변이 연산자를 파이썬으로 구현해보겠습니다.

비트 플립 돌연변이 연산자

```
1  def bit_flip(x, p):
2      probs = np.random.random(len(x))
3      x[probs < p] = 1 - x[probs < p]
4      return x
```

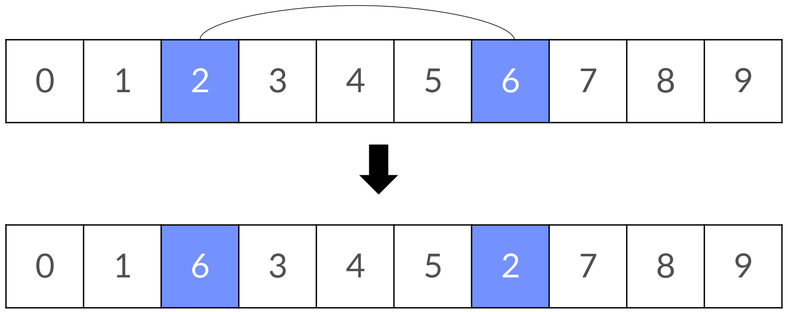
- **라인 2:** x와 길이가 같은 난수 배열 probs를 생성합니다.
- **라인 3:** 난수가 p보다 작을 확률은 p라는 점을 활용해 probs가 p 미만인 x의 요소를 1-x로 변경합니다. 즉, 0이면 1로, 1이면 0으로 변경합니다.

```
1  x = np.array([0, 1, 0, 1, 0, 1])
2  display(bit_flip(x, 0.5))
```

```
array([1, 1, 0, 0, 1, 1])
```

순열 인코딩에 대한 돌연변이 연산자

순서 변경(order changing) 돌연변이 연산자는 두 개의 요소를 임의로 선택하여 그 순서를 바꾸는 것입니다.



- 임의로 2와 6이 선택되어 두 요소의 위치가 변경됨

순열 인코딩에 대한 돌연변이 연산자 구현

순서 변경(order changing) 돌연변이 연산자는 두 개의 요소를 임의로 선택하여 그 순서를 바꾸는 것입니다.

순서 변경 돌연변이 연산자

```
1  def order_changing(x):
2      a, b = np.random.choice(range(len(x)), 2, replace = False)
3      (x[b], x[a]) = (x[a], x[b])
4      return x
```

- **라인 2:** 순서를 변경할 두 개의 인덱스 a와 b를 선택합니다.
- **라인 3:** 튜플을 이용해 x[a]와 x[b]의 값을 교체(swap)합니다.

```
1  x = np.array([0, 1, 2, 3, 4])
2  display(order_changing(x))
```

```
array([0, 2, 1, 3, 4])
```