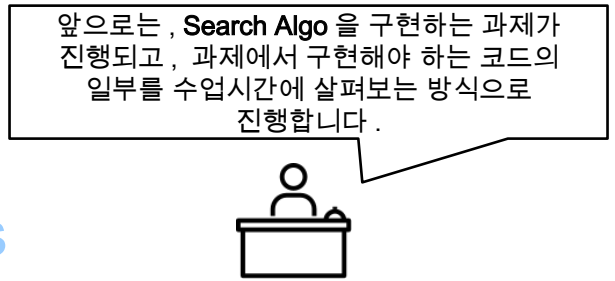
Search Algorithms: Object-Oriented Implementation (Part B)

구현 시작 !

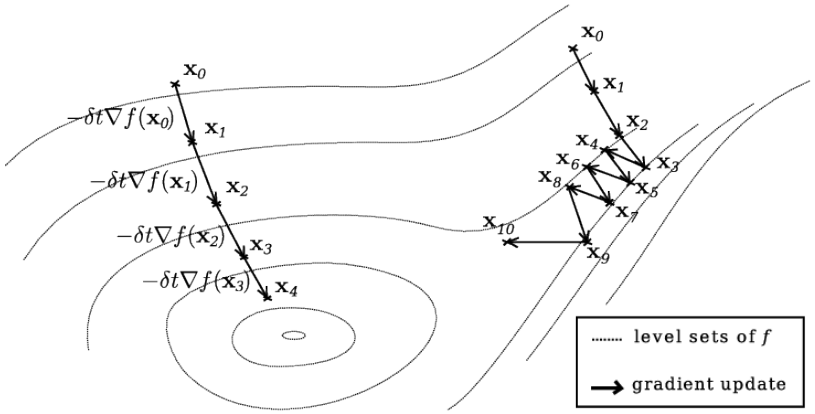
Contents

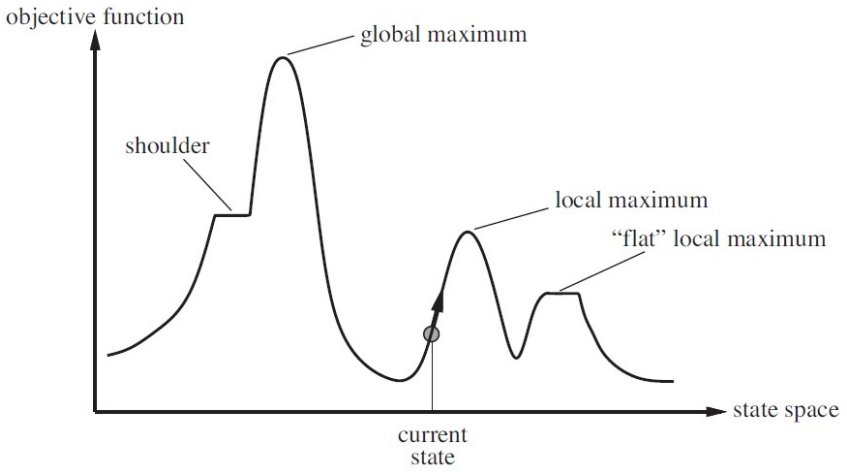
- Conventional vs. AI Algorithms
- Local Search Algorithms
- Implementing Hill-Climbing Algorithms
- Defining 'Problem' Class
- Adding Gradient Descent
- Defining 'HillClimbing' Class
- Adding More Algorithms and Classes
- Adding Genetic Algorithm
- Experiments



Implementation Reference Notes

- Search Algorithm
  - The techniques/algorithms covered in the class are methods for **iteratively finding the optimal solution**, and are called **Search Algorithms**.
  - However, the optimal solution found by the algorithm can be **global optimal** or **local optimal**.





Implementation Reference Notes

- Search Tool
  - In order to **execute a Search Algo.** (Algorithm), tasks such as **problem generation/loading and parameter setting** must be performed. // **Preprocessing** (or 'pre-work')
  - Additionally, a function to **display the Search Algo. execution results** is also necessary. // **Post-processing** (or 'post-work')
  - A program that performs the above steps: 1) Preprocessing, 2) Algorithm execution, and 3) Post-processing is called a **Search Tool**.

Implementation Reference Notes

- Search Tool Version
  - The Search Tool is implemented in various versions (iteratively expanded & improved).

Today's

  - v0 (강의 . Search Algo. B) : A version where **all the code to implement one algorithm is contained in a single file**
  - v1 (강의 . Search Algo. B) : A version implemented by **modularizing the logic common to different pieces of code into a separate file**, then importing and using that module/file (code refactoring & reuse)
  - v2 (강의 . Search Algo. C) : A version implemented using **classes** (v2.1, v2.2)
  - v3~v5 : Versions implemented using classes, with some algorithms added, leading to **iterative improvements in the code structure**
    - v3 : 강의 . Search Algo. D
    - v4 : 강의 . Search Algo. E
    - v5 : 강의 . Search Algo. F

Implementing Hill-Climbing Algorithms

- Our eventual goal is to implement an optimization tool that can run various search algorithms
- We begin to implement two hill climbing algorithms each for two different types of problems: (we implement a total of **four** hill climbing algorithms)
  - Steepest-ascent hill climbing for numerical optimization (SAHC-N)
  - First-choice hill climbing for numerical optimization (FCHC-N)
  - Steepest-ascent hill climbing for TSP (SAHC-T)
  - First-choice hill climbing for TSP (FCHC-T)

Ref:
- numerical optimization : The task of finding the variables that **maximize or minimize** the objective function/value given a mathematical expression/objective function. // **continuous**
- TSP : The problem of calculating the shortest distance to visit all given cities and return to the starting city. // **discrete**

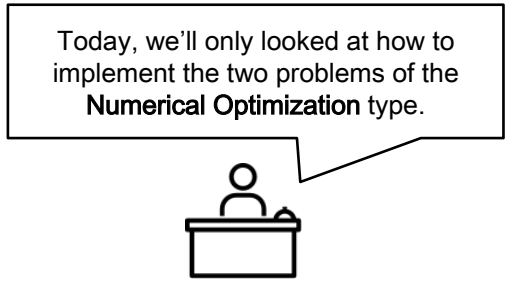
복습 : Hill Climbing Algo. (왼쪽) *vs* first-choice hill climbing (오른쪽)

[Steepest ascent version]

```
1: function HILL-CLIMBING( problem ) returns a state that is a local maximum
2:
3: current ← MAKE-NODE(problem.INITIAL-STATE)
4: loop do
5:     neighbor ← a highest-valued successor of current
6:     if neighbor.VALUE ≤ current.VALUE then return current.STATE
7:     else current ← neighbor
```

- If the goal is to maximize (maximize) the objective function, and there is no better solution (neighbor):
- **XXX.VALUE**: Refers to the **calculated value of the objective function** for the solution **XXX**.
- **Example**: If objective function$(x) = x\text{^}2$, current$(x = 4)$, then current.VALUE $= (4)\text{^}2$.

– First-choice (simple) hill climbing:
  - Generates successors randomly until one is found that is better than the current state
  - 후보 state를 무작위로 선택하고, current state 보다 좋다면 선택(계산이 간단)



Implementation by including all necessary logic in a single source code file

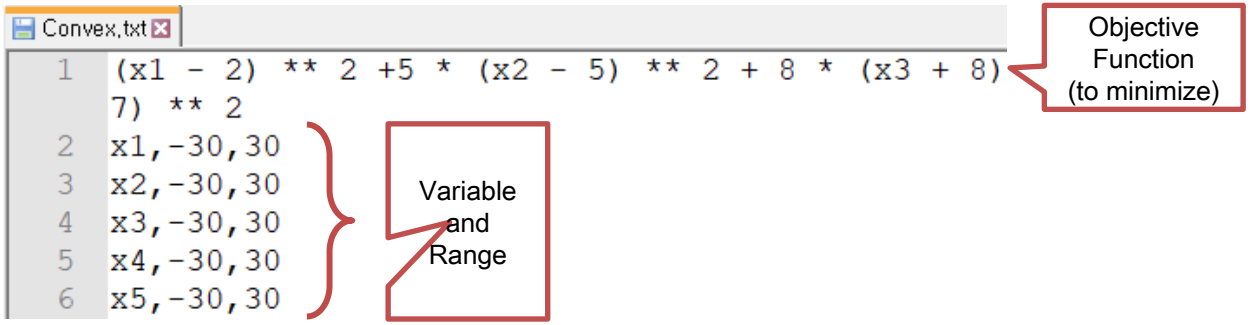
SEARCH TOOL V0

Numerical Optimization

- The problem of **finding the variable values** to **minimize or maximize a giv en mathematical expression**.

- Assume the problem is input via a **text file**, as shown below.
  - The problem is to find the variable values that **minimize the mathematical expre ssion** entered on the first line

  - Each variable has a minimum and maximum value range.
    - For example, the variables $x1 \sim x5$ below can only select values in the range of -30 to +30.



Numerical Optimization

- Execution Results (Example using the **Steepest-Ascent** Algorithm)

```
Enter the file name of a function: problem/Convex.tx

Objective function:
(x1 - 2) ** 2 + 5 * (x2 - 5) ** 2 + 8 * (x3 + 8) ** 2 + 3 * (x4 + 1) ** 2 + 6 * (x5 -
7) ** 2

Search space:
 x1: (-30.0, 30.0)
 x2: (-30.0, 30.0)
 x3: (-30.0, 30.0)
 x4: (-30.0, 30.0)
 x5: (-30.0, 30.0)

Search algorithm: Steepest-Ascent Hill Climbing

Mutation step size: 0.01

Solution found: (1.996, 5.003, -7.007, -1.005, 7.003)
Minimum value: 0.000

Total number of evaluations: 95,321
```

Input the name and path of the **.txt file defining the problem** (User Input).

Value goal

- * **Variable and Range:** The value of each variable can be changed within its specified range.
- * **Problem Goal:** Find the $x1, \ldots, x5$ values that make the **value of the objective function the smallest**.

The **calculated optimal solution**

The **result when the optimal solution is substituted into the objective function**.

Numerical Optimization

- Execution Results (Example using the **First-Choice** Algorithm)

```
Enter the file name of a function: problem/Convex.txt

Objective function:
(x1 - 2) ** 2 + 5 * (x2 - 5) ** 2 + 8 * (x3 + 8) ** 2 + 3 * (x4 + 1) ** 2 + 6 * (x5 -
7) ** 2

Search space:
 x1: (-30.0, 30.0)
 x2: (-30.0, 30.0)
 x3: (-30.0, 30.0)
 x4: (-30.0, 30.0)
 x5: (-30.0, 30.0)

Search algorithm: First-Choice Hill Climbing

Mutation step size: 0.01
Max evaluations with no improvement: 100 iterations

Solution found: (2.00
Minimum value: 0.000

Total number of evaluations: 37,336
```

Input the name and path of the **.txt file defining the problem** (User Input).

Value goal

- * **Variable and Range:** The value of each variable can be changed within its specified range.
- * **Problem Goal:** Find the $x1, \ldots, x5$ values that make the **value of the objective function the smallest**.

Main changes

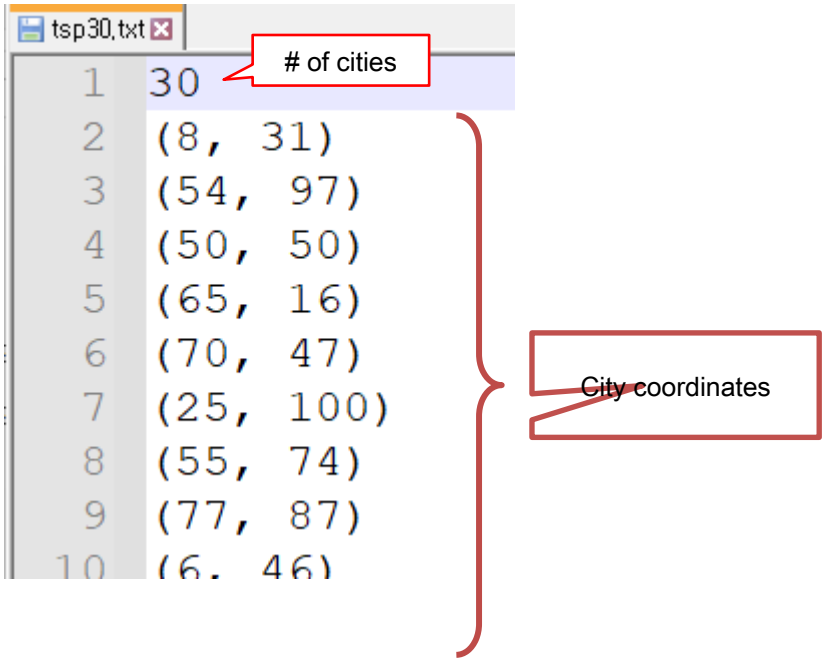
The **calculated optimal solution**

The **result when the optimal solution is substituted into the objective function**.

TSP Problem

- The problem of planning the **route (= order of visits)** to visit all cities with the **shortest total distance**.

- Assume the problem is input via a **text file**, as shown below.
  - The **number of cities** is on the first line.
  - The subsequent lines contain the **coordinates of each city.**



TSP Problem

- Execution Results (Example using the **First-Choice** Algorithm)

```
Enter the file name of a TSP: problem/tsp30.txt

Number of cities: 30
City locations:
     (8, 31)    (54, 97)    (50, 50)    (65, 16)    (70, 4?)
   (25, 100)    (55, 74)    (77, 87)     (6, 46)    (70, 78)
    (13, 38)   (100, 32)    (26, 35)    (55, 16)    (26, 77)
    (17, 67)    (40, 36)    (38, 27)     (33, 2)     (48, 9)
    (62, 20)    (17, 92)     (30, 2)    (80, 75)    (32, 3?)
    (43, 79)    (57, 49)    (18, 24)    (96, 76)    (81, ?)

Search algorithm: First-Choice Hill Climbing
Max evaluations with no improvement: 100 iterations

Best order of visits:
    4   26    2   24   16   27    0    8   10   12
   17   18   22   19    3   13   20   29   11   28
   23    7    9    6   25    1   14   15   21    5
Minimum tour cost: 491

Total numb    f evaluations: 677
```

Input the name and path of the **.txt file defining the problem** (User Input).

City starts from index 0...

The last city is City 29

The **Optimal Order of Visits**

The **Total Distance** traveled when following the optimal route.

TSP Problem

- Execution Results (Example using the **Steepest-Ascent** Algorithm)

```
Enter the file name of a TSP: problem/tsp30.txt

Number of cities: 30
City locations:
     (8, 31)    (54, 97)    (50, 50)    (65, 16)    (70, 47)
   (25, 100)    (55, 74)    (77, 87)     (6, 46)    (70, 78)
    (13, 38)   (100, 32)    (26, 35)    (55, 16)    (26, 77)
    (17, 67)    (40, 36)    (38, 27)     (33, 2)     (48, 9)
    (62, 20)    (17, 92)     (30, 2)    (80, 75)    (32, 36)
    (43, 79)    (57, 49)    (18, 24)    (96, 76)    (81, 39)

Search algorithm: Steepest-Ascent Hill Climbing

Best order of visits:
   11   29   20   13    3    4   28   23    9    1
    7    6    2   16   26   25   21    5    8   15
   14   10    0   12   27   24   17   19   18   22
Minimum tour cost: 620

Total num     of evaluations: 807
```

Input the name and path of the **.txt file defining the problem** (User Input).

The **Optimal Order of Visits**

The **Total Distance** traveled when following the optimal route.

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`main()`**:
  - Creates a problem instance by reading a function expression and its domain information from the file given by the user (**`createProblem`**)
    - Returns : `p = (expr, domain)`
    - `expr` : Objective Function
    - `domain` : minimization/maximization
  - Calls the search algorithm (**`steepestAscent`**) and obtains the results
  - Shows the specifics of the problem just solved (**`describeProblem`**)
  - Shows the settings of the search algorithm (**`displaySettings`**)
  - Reports the results (**`displayResults`**)

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`main()`**:
  - Creates a problem instance (**`createProblem`**)
  - Calls the search algorithm (**`steepestAscent`**) and obtains the results
  - Shows the specifics of the problem just solved (**`describeProblem`**)
  - Shows the settings of the search algorithm (**`displaySettings`**)
  - Reports the results (**`displayResults`**)

```
def main():
    # Create an instance of numerical optimization problem
    p = createProblem()   # 'p': (expr, domain)
    # Call the search algorithm
    solution, minimum = steepestAscent(p)
    # Show the problem and algorithm settings
    describeProblem(p)
    displaySetting()
    # Report results
    displayResult(solution, minimum)
```

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`createProblem()`**:
  - Gets a file name from the user and reads information from the file
    - Function expression – saved as a string # problem instance
    - Variable names – saved as a list of strings
    - Lower bounds of the variables – saved as a list of floats
    - Upper bounds of the variables – saved as a list of floats

    } # domain
  - Returns the problem instance as a list of expression and domain, where the domain is a list of the followings :
    - Variable names
    - Lower bounds
    - Upper bounds

the problem to be solved

Prints createProblem return value (=problem instance)

```
# Create an instance of numerical optimization problem
p = createProblem()   # 'p': (expr, domain)
print(p[0], end="")
print(p[1][0])
print(p[1][1])
print(p[1][2])
```

```
Enter the file name of a function: problem/Convex.txt
(x1 - 2) ** 2 + 5 * (x2 - 5) ** 2 + 8 * (x3 + 8) ** 2 + 3 * (x4 + 1) ** 2 + 6 * (x5 -
7) ** 2
['x1', 'x2', 'x3', 'x4', 'x5']  # var
[-30.0, -30.0, -30.0, -30.0, -30.0]  # lower
[ 30.0,  30.0,  30.0,  30.0,  30.0]  # upper
```

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`steepestAscent(p)`**:
  - Given a problem **`p`**, takes a random initial point (**`randomInit`**) as a current point and evaluates it (**`evaluate`**)
  - Repeats updating the current point:
    - Generate multiple neighbors (**`mutants`**)
    - Finds the best one (**`bestOf`**) among them
    - If it is better than current, update and continue
      Otherwise, stop
  - Returns the final current point and its evaluation value

> - "neighbor" = Solution adjacent to **current**
> - Adjacent solution = Solution obtained by slightly changing current solution

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`steepestAscent(p)`:**

```
def steepestAscent(p):
    current = randomInit(p) # 'current' is a list of values
    valueC = evaluate(current, p)
    while True:
        neighbors = mutants(current, p)
        successor, valueS = bestOf(neighbors, p)
        if valueS >= valueC:
            break
        else:
            current = successor
            valueC = valueS
    return current, valueC
```

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`randomInit(p)`**:
  - Given a problem **`p`**, returns a point randomly chosen within **`p`**'s domain

Generates a solution (point) randomly within the range that satisfies the specified minimum (lower bound) and maximum (upper bound) conditions.

- **`mutants(current, p)`**:

One of the methods for generating the adjacent (neighboring) successors of current

  - Returns $2n$ neighbors of **`current`** (n: # of variables)
    - i-th variable value **plus DELTA** n number of times
    - i-th variable value **minus DELTA** n number of times
  - Each neighbor is made by copying **`current`**, choosing an i-th variable of **`p`**, and then altering its value (**`mutate`**) by both adding and subtracting **`DELTA(d)`**
    - **`DELTA`** is a named constant representing the step size of axis-parallel mutation

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`mutate(current, i, d, p)`:**
  - Makes a copy of **`current`**, alter the value of i-th variable by adding **`d`** as long as the new value remains within the domain of **`p`**, and returns the resulting mutant

```
def mutate(current, i, d, p): ## Mutate i-th of 'current' if legal
    curCopy = current[:]
    domain = p[1]          # [VarNames, low, up]
    l = domain[1][i]       # Lower bound of i-th
    u = domain[2][i]       # Upper bound of i-th
    if l <= (curCopy[i] + d) <= u:
        curCopy[i] += d
    return curCopy
```

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`evaluate(current, p)`**:
  - Evaluates the expression of problem **`p`** after assigning the values of **`current`** to its variables, and returns the result
    - Global variable **`NumEval`** is used to count the number of evaluations

```
def evaluate(current, p):
    ## Evaluate the expression of 'p' after assigning
    ## the values of 'current' to the variables
    global NumEval

    NumEval += 1
    expr = p[0]          # p[0] is function expression
    varNames = p[1][0]   # p[1] is domain
    for i in range(len(varNames)):
        assignment = varNames[i] + '=' + str(current[i])
        exec(assignment)
    return eval(expr)
```

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`bestOf(neighbors, p)`**:
  - Evaluates each candidate solution in **`neighbors`** (**`evaluate`**), identifies the best one, and returns it with its evaluation value
- **`describeProblem(p)`**:
  - Shows the expression and the domain of problem **`p`**

```
def describeProblem(p):
    print()
    print("Objective function:")
    print(p[0])   # Expression
    print("Search space:")
    varNames = p[1][0] # p[1] is domain: [VarNames, low, up]
    low = p[1][1]
    up = p[1][2]
    for i in range(len(low)):
        print(" " + varNames[i] + ":", (low[i], up[i]))
```

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`desplaySetting()`**:
  - Shows that the steepest-ascent hill climbing has been used as the search algorithm
  - Displays the step size (**`DELTA`**) of the axis-parallel mutation

```
def displaySetting():
    print()
    print("Search algorithm: Steepest-Ascent Hill Climbing")
    print()
    print("Mutation step size:", DELTA)
```

```
DELTA = 0.01   # Mutation step size
```

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`displayResult(solution, minimum)`**:
  - Reports the result of optimization, which consists of
    - **`solution`** (the best solution found),
    - **`minimum`** (its evaluation value), and
    - the total number of evaluations
  - **`solution`** is transformed to a tuple (**`coordinate`**) before printing

- **`coordinate(solution)`**:
  - Rounds up **`solution`** and returns it (to the third decimal place)

Steepest-Ascent Hill Climbing for Numerical Optimization

- **`displayResult(solution, minimum)`:**
- **`coordinate(solution)`:**

```python
def displayResult(solution, minimum):
    print()
    print("Solution found:")
    print(coordinate(solution))  # Convert list to tuple
    print("Minimum value: {0:,.3f}".format(minimum))
    print()
    print("Total number of evaluations: {0:,}".format(NumEval))

def coordinate(solution):
    c = [round(value, 3) for value in solution]
    return tuple(c)  # Convert the list to a tuple
```

Steepest-Ascent Hill Climbing for Numerical Optimization

- The program is required to import ‘random.py’ and ‘math.py’

```
import random
import math
```

Algorithm change

First-Choice Hill Climbing for Numerical Optimization

- **`main()`**:
  - The only difference is that it calls **`firstChoice`** instead of **`steepestAscent`**

```
def main():
    # Create an instance of numerical optimization problem
    p = createProblem()   # 'p': (expr, domain)
    # Call the search algorithm
    solution, minimum = firstChoice(p)
    # Show the problem and algorithm settings
    describeProblem(p)
    displaySetting()
    # Report results
    displayResult(solution, minimum)
```

First-Choice Hill Climbing for Numerical Optimization

- **`firstChoice(p)`**:
  - Only one random successor is generated (**`randomMutant`**) to update the current solution
  - The algorithm stops if no improvement is observed for a certain consecutive number (**`LIMIT_STUCK`**) of iterations assuming that the search is stuck at a local minimum
    - **`LIMIT_STUCK`** is a named constant in this implementation

- Unlike the method of generating multiple **neighbor soln** and selecting the best among them, **only one random neighbor soln is selected.**
- If a worse solution appears consecutively for the number of LIMIT_STUCK iterations, the program stops.
- We assume LIMIT_STUCK = 100

First-Choice Hill Climbing for Numerical Optimization

- **`firstChoice(p)`:**

```
def firstChoice(p):
    current = randomInit(p)   # 'current' is a list of values
    valueC = evaluate(current, p)
    i = 0
    while i < LIMIT_STUCK:
        successor = randomMutant(current, p)
        valueS = evaluate(successor, p)
        if valueS < valueC:
            current = successor
            valueC = valueS
            i = 0              # Reset stuck counter
        else:
            i += 1
    return current, valueC
```

First-Choice Hill Climbing for Numerical Optimization

- **`randomMutant(current, p)`**:
  - Returns a mutant of **`current`**, which is made by randomly choosing a variable of **`p`** and then altering its value (**`mutate`**) by either adding or subtracting **`DELTA`**
  - The **randomMutant** function is implemented by calling the **mutate** function. When doing so, it must decide **which index's stored value to modify** ($\mathrm{d} = \mathrm{DELTA} = 0.01$).

```
def mutate(current, i, d, p): ## Mutate i-th of 'current' if legal
    curCopy = current[:]
    domain = p[1]          # [VarNames, low, up]
    l = domain[1][i]       # Lower bound of i-th
    u = domain[2][i]       # Upper bound of i-th
    if l <= (curCopy[i] + d) <= u:
        curCopy[i] += d
    return curCopy
```

First-Choice Hill Climbing for Numerical Optimization

- **`displaySetting()`**:
  - Shows that the search algorithm used is first-choice hill climbing

```
def displaySetting():
    print()
    print("Search algorithm: First-Choice Hill Climbing")
    print()
    print("Mutation step size:", DELTA)
```

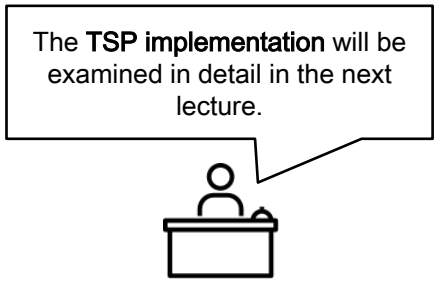
- The functions **`createProblem`**, **`randomInit`**, **`evaluate`**, **`mutate`**, **`describeProblem`**, **`displayResult`**, and **`coordinate`** are all reused without change

Hill Climbing for TSP (외판원 문제)

- For **`TSP`**, the structure of the problem and its implementation are **nearly identical** to the **`NumericalOptimization`** problem
  - However, the logic for generating `mutant` are different
  - `TSP` generate the `mutant` (adjacent solution) using `mutate by inversion` (refer to the previous lecture)

- `inversion :` Generating one adjacent solution
  - Reverese the items from index ***`i`*** to index ***`j`***

```
def inversion(current, i, j):  # Perform inversion
    curCopy = current[:]
    while i < j:
        curCopy[i], curCopy[j] = curCopy[j], curCopy[i]
        i += 1
        j -= 1
    return curCopy
```



Hill Climbing for TSP (외판원 문제)

- **`First-Choice (TSP) :`** generating a single **`mutant`**

```
def randomMutant(current, p): # Apply inversion
    while True:
        i, j = sorted([random.randrange(p[0])
                       for _ in range(2)])
        if i < j:
            curCopy = inversion(current, i, j)
            break
    return curCopy
```

- Variable $\mathbf{p[0]}$: $\mathbf{number\ of\ cities, N}$ (Read from the first line of the defined problem file).
- $\mathbf{random.randrange(N)}$: Returns a random integer from $0 \sim N-1$.
  - When the number of cities is N, each city is identified by a number from $\mathbf{0, 1, \ldots, N-1}$.
- The process:
  - **Two random numbers are generated and stored in a list, then the list is sorted.** (This ensures that the two values are in order, i.e., $i \leq j$).
  - **If i and j are different ($i < j$), they are treated as valid indices**, and the function calls $\mathbf{inversion(current, i, j)}$ to generate a single mutant. (The while True: loop continues until two different indices are selected such that $\boldsymbol{i < j}$).

Hill Climbing for TSP (외판원 문제)

- **`Steepest-Ascent (TSP) :`** Create a number of **`mutants`**

```
def mutants(current, p): # Apply inversion
    n = p[0]
    neighbors = []
    count = 0
    triedPairs = []
    while count <= n:  # Pick two random loci for inversion
        i, j = sorted([random.randrange(n) for _ in range(2)])
        if i < j and [i, j] not in triedPairs:
            triedPairs.append([i, j])
            curCopy = inversion(current, i, j)
            count += 1
            neighbors.append(curCopy)
    return neighbors
```

Code

- `Ref:`
  - The **code omitted from the lecture materials must be implemented in the homework.**