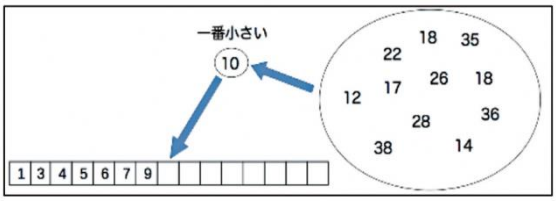
エキスパート活動　課題 A　「選択ソート」

選択ソートはリスト内のデータから最小値を探索し，最小値から順に取り出すことで並べ替えを実現するアルゴリズムである。（図表８参照）



図表８　選択ソートの概念

イメージ図

＜プログラム例＞

コード例を以下に示す。a はソート対象のリストである。なお a[i]と a[j]の値を入れ替える際には変数 temp を使って以下のように行っている。

1　a[i]の値を一時的に temp に入れておく(temp=a[i])
2　a[i]の値を a[j]の値に置き換える(a[i]=a[j])
3　a[j]の値を temp の値に置き換える(a[j]=temp)

＜プログラム例＞

```
##　選択ソート

arr = [64, 25, 12, 22, 11]          # 例：配列 [64, 25, 12, 22, 11] をソートする
n = len(arr)                        # 配列の長さを取得
temp = 0                            # 変数 temp の初期値を設定

for i in range(n):                  # 配列全体を走査するための外側のループ
   min_index = i                    # 未ソート部分の最小値のインデックスを現在の要素に初期化
   for j in range(i + 1, n):        # 未ソート部分の残りの要素を走査するための内側のループ
      if arr[j] < arr[min_index]:   # 現在の要素が最小値よりも小さい場合
         min_index = j              # 最小値のインデックスを更新

   temp = arr[i]                    # arr[i] の値を一時避難
   arr[i] = arr[min_index]          # arr[i] に arr[min_index] を上書き
   arr[min_index] = temp            # 避難させていた値を arr[min_index] に入れる

print("ソート済み配列:", arr)        # ソートされた配列を出力
```

確認①. 上記のコードにおいて，『各ステップの並び替えが終了したときにリストの内容が表示』されるようにプログラムを更新した。これを実行し、数値の並び替えがどのようになっているのかを確認しよう。

確認②. 上記のコードにおいて，『データを入れ替えた回数を表示』できるように変数 swap_count を追加してプログラムを更新した。これを実行し、入れ替えた回数を確認しよう。

発問①. ソート前が[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]となる数列に変更し、『各ステップの並び替え』と『並び替え回数』を確認しよう。

エキスパート活動　課題Ｂ　「バブルソート」

バブルソートは、隣り合う要素を比較・交換し、大きな（または小さな）データを順に端へ送ることでデータを整列させる、最も単純なソートアルゴリズムである。

基本的なアルゴリズム手順

１．隣接するデータ同士を比較する。
２．順序が逆（左が大きく、右が小さい）なら入れ替える。
３．この手順を配列の最後まで繰り返す
　　（これで最大値が右端に配置される）。
４．右端の確定した要素を除き、再度１～３を繰り返す。
５．入れ替えが発生しなくなったら終了。

＜プログラム例①＞

```
##　バブルソート

my_array_no_func = [64, 34, 25, 12, 22, 11, 90]          # ソートしたい数値のリストを定義します
print(f"初期配列: {my_array_no_func}")                    # 初期配列の状態を表示します

n = len(my_array_no_func)                                  # 配列の要素数を取得します
tmp = 0                                                    # 一時保存用の変数 tmp の初期値を設定
                          # n-1 回パスを繰り返す（要素が n 個の場合、最大 n-1 回のパスでソートが完了する）
for i in range(n - 1):              # 外側のループ: 各パス（スキャン）を制御します。n-1 回繰り返します。
                                    # 各パスで、未ソート部分の末尾まで比較と交換を行う
                                    # 最後の i 個の要素は既にソート済みなので、範囲から除外する
  for j in range(n - 1 - i):      # 内側のループ: 未ソート部分の要素を比較し、必要に応じて交換します。
                                    # 隣り合う要素を比較し、順序が逆であれば交換する
    if my_array_no_func[j] > my_array_no_func[j + 1]:
      # ↑隣り合う 2 つの要素を比較します。左が右より大きい場合（昇順ソート）
      tmp = my_array_no_func[j]                            # 一時保存用の変数(tmp)に避難させる
      my_array_no_func[j] = my_array_no_func[j + 1]    # 空いた箱(j)に、隣の値(j+1)を入れる
      my_array_no_func[j + 1] = tmp                        # 避難させていた値を、隣の箱(j+1)に戻す

print(f"実行結果: {my_array_no_func}")                    # 最終的なソート結果を表示します
```

確認①.　上記のコードにおいて、『各パスの並び替えの状況』がわかるようにプログラムを更新した。
　　このプログラムを実行して、配列がどのように変化するのかを確認しよう。

確認②.　上記のコードにおいて、『総並び替えの回数』が表示するように更新した。
　　このプログラムを実行して、回数を確認しよう。また、各パスの回数を数え上げよう。
　　例：パス１は５回、パス２は４回、・・・・など

発問①.　ソート前　[7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1]として、実行しよう。
　　『総並び替えの回数』と『各パスの回数』を読み取ろう。

エキスパート活動　課題Ｃ　「選択ソート vs バブルソート」

選択ソートとバブルソートの比較

アルゴリズムの学習において、選択ソートとバブルソートは「初歩の二大巨頭」です。どちらも直感的で分かりやすい反面、効率が良いとは言えません。

まず、アルゴリズムの仕組みについてです。

選択ソート (Selection Sort)は、「未整列のグループの中から最小値を探し出し、左端の要素と交換する」という操作を繰り返す手法です。１番小さいものを探して１番目へ、次に小さいものを探して２番目へ…と、「場所」を固定してそこに入れるべき値を選んでいきます。データの交換回数が少ないのがメリットです。

バブルソート (Bubble Sort)は、「隣り合う要素を比較し、大小が逆なら入れ替える」という操作を繰り返す手法です。重いものが下に沈み、軽いものが上に浮いていく（泡のように動く）様子からバブルソートと呼ばれます。実装が非常にシンプルですが、データの交換回数が多くなりがちです。

計算量は、どちらも $O(n^2)$ となります。どちらもデータ量が２倍になれば、かかる時間は４倍になります。大量のデータを扱うには不向きです。安定性の違い、ここが実用上の大きな差です。例えば「点数順に並べ替えるが、同点の場合は出席番号順を維持したい」という場合、バブルソートなら維持（安定）できますが、選択ソートでは順序がバラバラになる（不安定）可能性があります。

発問.　以下のプログラムの実行結果からどのようなことが分かりますか。それぞれの特徴を踏まえ考察しよう。
（発生させる乱数の範囲やデータサイズを変更しながら特徴と傾向を読み取ろう。）

＜プログラム例＞

```
##　選択ソートとバブルソートのステップ回数を表示するプログラム


import random　# random モジュールをインポートします。

# データサイズを設定します。
data_size = 10　# 比較するデータ数を 10 に設定します。

# ランダムなデータを作成します。
data = [random.randint(1, 100) for _ in range(data_size)]　# 1 から 100 までのランダムな整数 10 個のリストを作成します。

# 選択ソートの準備
selection_data = data.copy()　# 元のデータを変更しないようにコピーを作成します。
selection_count = 0　# 選択ソートのステップ回数を初期化します。
temp = 0 # 選択ソートの一時的に保存する変数 temp の初期値を設定

# 選択ソートの実装
n = len(selection_data)　# リストの長さを取得します。
for i in range(n):　# リストの最初から最後までループします。
  min_index = i　# 現在の要素を最小値のインデックスとして設定します。
  for j in range(i + 1, n):　# 未ソート部分を探索します。
    selection_count += 1　# 比較ごとにステップ回数を加算します。
```

```
    if selection_data[j] < selection_data[min_index]:  # 現在の最小値よりも小さい要素が見つかった場合
        min_index = j  # 最小値のインデックスを更新します。
  temp = selection_data[i]                  # arr[i] の値を一時避難
  selection_data[i] = selection_data[min_index]   # arr[i] に arr[min_index] を上書き
  selection_data[min_index] = temp     # 避難させていた値を arr[min_index] に入れる

print("選択ソート:")  # 選択ソートの結果であることを表示します。
print("ステップ回数:", selection_count)  # 選択ソートのステップ回数を表示します。
# print("ソート済みデータ:", selection_data) # ソートされたデータを表示する場合のコメントアウトです。

print("¥n")  # 空行を出力して見やすくします。

# バブルソートの準備
bubble_data = data.copy()  # 元のデータを変更しないようにコピーを作成します。
bubble_count = 0  # バブルソートのステップ回数を初期化します。
tmp = 0 # バブルソートで使用する一時保存用の変数 tmp の初期値を設定

# バブルソートの実装
n = len(bubble_data)  # リストの長さを取得します。
for i in range(n - 1):  # n-1 回パスを繰り返します。
  for j in range(n - 1 - i):  # 各パスで、未ソート部分を比較します。
    bubble_count += 1  # 比較ごとにステップ回数を加算します。
    if bubble_data[j] > bubble_data[j + 1]:  # 隣接する要素を比較し、順序が逆ならば
      tmp = bubble_data[j]                          # 一時保存用の変数(tmp)に避難させる
      bubble_data[j] = bubble_data[j + 1]    # 空いた箱(j)に、隣の値(j+1)を入れる
      bubble_data[j + 1] = tmp                      # 避難させていた値を、隣の箱(j+1)に戻す

print("バブルソート:")  # バブルソートの結果であることを表示します。
print("ステップ回数:", bubble_count)  # バブルソートのステップ回数を表示します。
# print("ソート済みデータ:", bubble_data) # ソートされたデータを表示する場合のコメントアウトです。

print("¥n 選択ソートとバブルソートのステップ回数比較:")  # 比較結果の見出しを表示します。
print("選択ソート:", selection_count)  # 選択ソートのステップ回数を表示します。
print("バブルソート:", bubble_count)  # バブルソートのステップ回数を表示します。
```

４．ジグソー課題

(1)挙動のシミュレーション

以下の数値が並んだリストを、昇順（小さい順）に並べ替える場面を想像してください。

対象データ： [5, 3, 8, 2]

①選択ソートを使用した場合、「1 回目の最小値探索と交換」が終わった直後のリストの状態を答えてください。

②バブルソートを使用した場合、「隣り合う要素の比較・交換」が 1 周（1 パス）終わった直後のリストの状態を答えてください。

① 選択ソート（Selection Sort）

選択ソートは、「未整列の部分から最小値を探し、左端の要素と交換する」という動作を繰り返します。

1 回目の最小値探索： 全体 [5, 3, 8, 2] の中から最小値を探すと 2 が見つかります。

交換： 見つかった最小値 2 と、現在の左端にある 5 を入れ替えます。

【結果】

[2, 3, 8, 5]

② バブルソート（Bubble Sort）

バブルソートは、「隣り合う要素を比較し、順序が逆なら入れ替える」という操作を端から端まで行います。1 周（1 パス）終わると、最大値が一番右側に押し出されます。

5 と 3 を比較： 5 の方が大きいので入れ替え → [3, 5, 8, 2]

5 と 8 を比較： 5 の方が小さいのでそのまま → [3, 5, 8, 2]

8 と 2 を比較： 8 の方が大きいので入れ替え → [3, 5, 2, 8]

【結果】

[3, 5, 2, 8]

(2)特性の理解

次のような特性を持つデータセットに対して、選択ソートとバブルソートのどちらが「より効率的」か、理由とともに答えてください。

ケース A： すでにほとんど昇順に並んでいるが、一部だけ順序が入れ替わっているデータ。

ケース B： データの入れ替え（スワップ）コストが非常に高く、できるだけ交換回数を抑えたいシステム。

ケース A：すでにほとんど昇順に並んでいる場合

【効率的な方】バブルソート （※ただし「フラグ」付きの改良版）

理由：バブルソートには、「1 周（1 パス）の間に一度も交換が発生しなければ、その時点でソートを打ち切る」という最適化が可能です。すでにほぼ整列しているデータなら、ごくわずかなスキャンで完了するため、実効速度が非常に速くなります。

選択ソートが不向きな理由：選択ソートは、データの並び順に関係なく、常に最後まで最小値を探し続ける（比較をサボれない）性質があるため、ほぼ整列していても手間が減りません。

ケース B：交換コスト（スワップ）を抑えたい場合

【効率的な方】選択ソート

理由：選択ソートの最大の特徴は、「1 回の探索（1 パス）につき、交換は最大 1 回しか行わない」という点です。要素数 n に対して、交換回数は最大でも n−1 回で済みます。これは他の多くのソートアルゴリズムと比較しても非常に少ない回数です。

バブルソートが不向きな理由：バブルソートは、隣り合う要素が逆順であるたびに細かく交換を繰り返します。最悪の場合、膨大な回数の書き換えが発生するため、交換コストが高い環境（例：フラッシュメモリへの書き込み寿命を延ばしたい、巨大な構造体を動かしている等）では不利になります。

(3)安定性の考察

「安定なソート（Stable Sort)」についての問題です。

データ： [(A, 80), (B, 50), (C, 80)]　※(名前，スコア）の形式

このデータを「スコアの低い順」に並べ替えます。バブルソートの結果として正しいものはどちらですか？また、選択ソートで並べ替えたとき、意図せず「A と C の順序」が入れ替わってしまう可能性があるのはなぜですか？

候補①： [(B, 50), (A, 80), (C, 80)]

候補②： [(B, 50), (C, 80), (A, 80)]

1. バブルソートの結果として正しいもの

正しいのは 候補①： [(B, 50), (A, 80), (C, 80)] です。

【解説】

バブルソートは「安定なソート」に分類されます。

隣り合う要素を比較する際、同じスコア（80 点）であれば入れ替えを行わないため、もともと左側にいた A が、右側にいた C よりも前に残り続けます。

2. 選択ソートで「A と C の順序」が入れ替わる理由

選択ソートは「不安定なソート（Unstable Sort)」と呼ばれ、同じ値を持つデータの前後関係が壊れてしまうことがあります。

その理由は、「遠く離れた場所にある要素と、強制的に位置を交換（スワップ）する」という仕組みにあります。

今回のデータ [(A, 80), (B, 50), (C, 80)] でシミュレーションしてみましょう。

最小値を探す： 全体の中で最小値は（B, 50）です。

交換を実行： 最小値（B, 50）と、現在の左端にある（A, 80）を交換します。

結果： [(B, 50), (C, 80), (A, 80)] （※候補②の状態）

【なぜ入れ替わったのか？】

最小値 B を左端に持っていく際、左端にいた A が右側へ大きく飛ばされてしまったため、もともと自分の後ろにいた C よりもさらに後ろへ配置されてしまいました。このように「最小値との交換」という一見効率的な動きが、同順位のデータの順番を飛び越えさせてしまう原因になります。