データ構造とアルゴリズム  
レポート課題5

提出日: 2024/11/08

HI4 45号山口惺司

1. 課題内容

乱択法を用いてクイックソートを改良しなさい.

## アルゴリズム

# クイックソートについて

クイックソートは効率的な比較ソートアルゴリズムの1つで, 大規模なデータセットを効率的に並べ替えるために用いられる.

また, 「分割統治法」に基づいており, 配列を再帰的に分割し, それぞれをソートしていくことで全体をソートさせる.

# クイックソートの手順

1. 配列の先頭をピボットとして選び, 配列を「ピボットより小さい要素」と「ピボットより大きい要素」に分ける.
2. ピボットを基に①のように分割し新たな配列を作成する.
3. ②で分割された配列それぞれに, 再び①②のようにクイックソートを適用する. これを再帰的に繰り返すことで, 配列全体がソートされる.

# フローチャート

クイックソートのフローチャートを図1に示す.

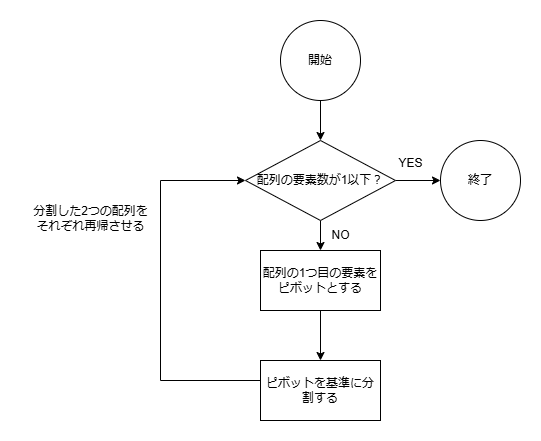


図1 クイックソートのフローチャート

1. **クイックソートの改良法**

教科書に書いてあるquick\_sort.pyを改良するには, ピボットを最初の要素ではなく, ランダムに決めるという方法がある.

ピボットをランダムな要素にすることによって, 平均的に高速にすることができる.

このようにランダム化によって平均的によい性能を期待できるアルゴリズムを**乱択アルゴリズム**という.

1. スクリプト

# クイックソート

改良前のクイックソートのスクリプトを以下に示す.

def quicksort(A):

    if len(A) < 2:

        return A

    p = A[0]

    X, Y = divide(p, A[1:])

    return np.concatenate([quicksort(X), [p], quicksort(Y)])

def divide(p, A):

    A = np.array(A)

    X = A[A < p]

    Y = A[A >= p]

    return X, Y

# 改良版クイックソート

改良後クイックソートのスクリプトを以下に示す.

def randomized\_quicksort(A):

    if len(A) < 2:

        return A

    pivot\_index = random.randint(0, len(A) - 1)

    p = A[pivot\_index]

    A[0], A[pivot\_index] = A[pivot\_index], A[0]

    X, Y = divide(p, A[1:])

    return np.concatenate([randomized\_quicksort(X), [p], randomized\_quicksort(Y)])

def divide(p, A):

    A = np.array(A)

    X = A[A < p]

    Y = A[A >= p]

    return X, Y

# 元のクイックソートと改良版クイックソートを比較

今回は元のクイックソートと改良版クイックソートを比較するため, 以下の手順でソートを行う.

1. 0～99までの整数をランダムに100個生成しdataリストに入れる
2. dataをクイックソートし, 実行時間を計測する.
3. ①②を100回繰り返す.
4. 実行時間の分散を調べる
5. ③④を1000回繰り返し, 実行時間の分散をグラフにする.
6. それぞれのソートの平均実行時間と実行時間の分散平均の値を表示する.

以上の手順でソートを行うためのスクリプトを以下に示す.

import random

import time

import matplotlib.pyplot as plt

import numpy as np

# 従来のクイックソート

def quicksort(A):

    if len(A) < 2:

        return A

    p = A[0]

    X, Y = divide(p, A[1:])

    return np.concatenate([quicksort(X), [p], quicksort(Y)])

# 乱択法を用いたクイックソート

def randomized\_quicksort(A):

    if len(A) < 2:

        return A

    pivot\_index = random.randint(0, len(A) - 1)

    p = A[pivot\_index]

    A[0], A[pivot\_index] = A[pivot\_index], A[0]

    X, Y = divide(p, A[1:])

    return np.concatenate([randomized\_quicksort(X), [p], randomized\_quicksort(Y)])

# 分割関数

def divide(p, A):

    A = np.array(A)

    X = A[A < p]

    Y = A[A >= p]

    return X, Y

# main関数

def main():

    #実行時間と分散を入れるリスト

    data\_num = [[], []] #ave var

    data\_num\_rand = [[], []] #ave var

    for j in range(1000):

        data\_time = []

        data\_time\_rand = []

        for i in range(100):

            random.seed()

            data = np.random.randint(0, 100, 100)  #配列の作成

            start\_time = time.time()

            quicksort(data.copy())

            end\_time = time.time()

            data\_time.append(end\_time - start\_time)

            start\_time = time.time()

            randomized\_quicksort(data.copy())

            end\_time = time.time()

            data\_time\_rand.append(end\_time - start\_time)

        data\_num[0].append(np.average(data\_time))

        data\_num\_rand[0].append(np.average(data\_time\_rand))

        data\_num[1].append(np.var(data\_time))

        data\_num\_rand[1].append(np.var(data\_time\_rand))

    print(data\_num[0])

    print(data\_num\_rand[0])

    print(data\_num[1])

    print(data\_num\_rand[1])

    print("改良前平均実行時間: ", np.average(data\_num[0]))

    print("改良後平均実行時間: ",np.average(data\_num\_rand[0]))

    print("改良前実行時間の分散平均: ",np.average(data\_num[1]))

    print("改良後実行時間の分散平均: ",np.average(data\_num\_rand[1]))

    l = len(data\_num[0])

    #グラフ

    plt.scatter(range(0, l), data\_num[1], color="orange", label="Before", s = 10)

    plt.scatter(range(0, l), data\_num\_rand[1], color="blue", label="After", s = 10)

    plt.xlabel("Run Count")

    plt.ylabel("Variance of Runtime")

    y\_max = np.average(data\_num[1])\*3

    plt.ylim(0, y\_max)

    plt.legend()

    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

1. 実行結果

まず、元のクイックソートと改良版のクイックソートが正しくソートされているかを確認した実行結果を図2~4に示す.

また、実行したコードは以下のとおりで100個のランダムな要素を持った配列をソート関数に渡している。

    data = np.random.randint(0, 100, 100)

    print("改良前クイックソート: ", quicksort(data.copy()), sep = "\n")

    print("改良後クイックソート: ", randomized\_quicksort(data.copy()), sep = "\n")

コンピューターのキーボード

自動的に生成された説明

図2 実行結果1：正しくソートが行われているかの確認

コンピューターのキーボード

自動的に生成された説明

図3 実行結果2：正しくソートが行われているかの確認

コンピューターのキーボード

自動的に生成された説明

図4 実行結果3：正しくソートが行われているかの確認

次に、改良版クイックソートが正しく改良されているかを確認した実行結果を図5~10に示す。

横軸が実行回数, 縦軸が実行時間の分散である.

グラフ, 散布図

自動的に生成された説明

図5 実行結果1：実行時間の分散のグラフ

テキスト

自動的に生成された説明

図6 実行結果1：それぞれのソートの平均実行時間と実行時間の分散平均

グラフ, 散布図

自動的に生成された説明

図7 実行結果2：実行時間の分散のグラフ

グラフィカル ユーザー インターフェイス, テキスト

自動的に生成された説明

図8 実行結果2：それぞれのソートの平均実行時間と実行時間の分散平均

グラフ, 散布図

自動的に生成された説明

図9 実行結果3：実行時間の分散のグラフ

テキスト

自動的に生成された説明

図10 実行結果3：それぞれのソートの平均実行時間と実行時間の分散平均

1. 考察

図2~4より、どちらのクイックソートも正しくソートができていると言える。

図5~10より、改良後のクイックソートは改良前より平均実行時間が大きくなっているが、実行時間の分散平均が改良前より小さくなっているため、改良することによって安定化できていると言える。

また、改良後のクイックソートの平均実行時間が遅くなっている理由としてはピボットを選択するときに、ランダム関数を使っていて、その処理に時間がかかっているのではないかと推測する。