Tugas Besar Analisis Kompleksitas Algoritma

Anggota kelompok :
• Cakra Budiman Putra (1301213273)
• Fajar Maulana Kadir (1301210494)

Deskripsi :
Pada tugas besar ini kami akan melakukan analisis kompleksitas algoritma pada algoritma Quick Sort dengan proses atau prosesedur secara rekursif dan iterative. Untuk mengetahui tingkat efesiensi suatu algoritma menyangkut efesiensi kecepatan memori, digunakan suatu besaran waktu dan ruang, yaitu kompleksitas waktu ruang dari algoritma tersebut. Kompleksitas suatu algoritma sangat berguna untuk membanding-bandingkan beragam algoritma termasuk algoritma Quick Sort yang kami pilih. Sehingga dapat ditentukan algoritma yang paling efisien dari algoritma tersebut, serta algoritma yang tepat untuk digunakan untuk suatu kondisi.

Basic Operation Iterative Dan Recursive

Efisiensi suatu algoritma diukur dari berapa jumlah waktu dan ruang (space) memori yang dibutuhkan untuk menjalankannya,Kebutuhan waktu dan ruang suatu algoritma bergantung pada ukuran masukan (n), yang menyatakan jumlah data yang diproses, sehingga untuk mengetahui efisiensi waktu dari algoritma yang kita gunakan kita memilih basic operation sebagai berikut

1. Membandingkan Array S[i] dengan *pivotitem* pada *partition*
2. Input size = n

Program Quicksort Recursive

Deklarasi :

S = array[]

low, high, i = intejer

Algoritma :

```
function quick_sort_recursion(S):
    function partition(S, low, high):
        i = (low - 1)
        pivot = S[high]
        while j>=low && j<=high do
            if S[ j ] <= pivot then
                i = i + 1
                S[i], s[j] = S[j], S[i]
            {end if}
            j++
        {end while}
        S[i+1], S[high] = S[high], S[i+1]
        return (i+1)

    function quickSort(S, low, high):
        if low < high then
            pi = partition(S, low, high)
            quicksort(S, low, pi-1)
            quicksort(S, pi+1, high)
        {end if}
    n = len(S)
    quickSort(S, 0, n-1)
```

Program Quicksort Iterative

Deklarasi :

S = array[]

l, h, l, size = intejer

Algoritma :

```
function quick_sort_iterative(S):
    function partition(S, l, h):
        i = (l - 1)
        x = S[h]
        while j>=l && j<=h do
            if S[ j ] <= x then
                i = i + 1
                S[ i ], S[ j ] = S[ j ], S[ i ]
            {end if}
            j++
        {end while}
        S[i+1], S[high] = S[high],  S[i+1]
        return (i+1)

    function quickSortIterative(S, l, h):
        size = h – l + 1
        stack = [0] * (size)
        top = -1
        top = top + 1
        stack[top] = 1
        top = top + 1
        stack[top] = h
```

```
    while top >= 0 do
        h = stack[top]
        top = top – 1
        l = stack[top]
        top = top – 1

        p = partition(S, l, h)
        if (p – 1) > l then
            top = top + 1
            stack[top] = l
            top = top + 1
            stack[top] = p – 1
        {end if}

        if (p + 1) < h then
            top = top + 1
            stack[top] = l
            top = top + 1
            stack[top] = h
        {end if}
    {end while}

N = len(S)
quickSortIterative(S, 0, n-1)
```

Perhitungan Kompleksitas Waktu Iterative Dan Recursive

Untuk Perhitungan Kompleksita waktu Iterative dan Recursive kami menggunakan worst case

Partisi yang paling tidak seimbang terjadi ketika salah satu sublist yang dikembalikan oleh rutin partisi berukuran n - 1 . Hal Ini dapat terjadi jika pivot menjadi elemen terkecil atau terbesar dalam daftar, atau dalam beberapa implementasi (misalnya, skema partisi Lomuto seperti dijelaskan di atas) ketika semua elemen sama.

Jika ini terjadi berulang kali di setiap partisi, maka setiap panggilan rekursif memproses daftar yang ukurannya lebih kecil dari daftar sebelumnya. Akibatnya, kita bisa membuat n - 1 panggilan bersarang sebelum kita mencapai daftar ukuran 1. Ini berarti bahwa pohon panggilan adalah rantai linier dari n - 1 panggilan bersarang. panggilan ke-i melakukan O (n - i) pekerjaan yang harus dilakukan partisi, jadi dalam hal ini Quicksort memerlukan waktu O (n^2) .

$$
\begin{aligned}
&(1)\\
T(0) &= 0 &(2)\\
T(n) &= T(0) + T(n-1) + n - 1 &(3)\\
T(n) &= T(n-1) + n - 1 &(4)\\
T(n) - T(n-1) &= n - 1 &(5)\\
t_n - t_{n-1} &= n - 1 &(6)\\
t_n - t_{n-1} &= 1^n(n^1 - 1) &(7)\\
&&(8)\\
(r-1)(r-1)^2 &= 0 &(9)\\
(r-1)^3 &= 0 &(10)\\
&&(11)\\
t_n &= C_1 1^n + C_2 n 1^n + C_3 n^2 1^n &(12)\\
&= C_1 + C_2 n + C_3 n^2 &(13)\\
&&(14)\\
t_1 &= t_0 + 1 - 1 = 0 &(15)\\
t_2 &= t_1 + 2 - 1 = 1 &(16)\\
&&(17)\\
t_1 &= C_1 + 1C_2 + 1^2 C_3 &(18)\\
0 &= C_1 + C_2 + C_3 &(19)\\
t_2 &= C_1 + 2C_2 + 2^2 C_3 &(20)\\
1 &= C_1 + 2C_2 + 4C_3 &(21)\\
&&(22)\\
C_1 &= 0 &(23)\\
&&(24)\\
0 &= C_2 + C_3(x2) &(25)\\
1 &= 2C_2 + 4C_3(x1) &(26)\\
&&(27)\\
0 &= 2C_2 + 2C_3 &(28)\\
1 &= 2C_2 + 4C_3 &(29)\\
&&(30)\\
-1 &= 2C_3 &(31)\\
C_3 &= -1/2 &(32)\\
&&(33)\\
0 &= 2C_2 + 2(-1/2) &(34)\\
1 &= 2C_2 &(35)\\
C_2 &= 1/2 &(36)
\end{aligned}
$$

$$(37)$$

$$T(n) = \frac{n(n-1)}{2} \tag{38}$$

$$T(n) = \frac{n^2 - n}{2} \in O(n^2) \tag{39}$$

Kasus terburuk terjadi ketika array sudah diurutkan karena kita selalu memilih item pertama untuk item pivot. Oleh karena itu, jika kita memiliki alasan untuk percaya bahwa array hampir diurutkan, ini bukanlah pilihan yang baik untuk item pivot. Jika kita menggunakan metode ini, kasus terburuk tidak terjadi saat array sudah diurutkan. Tetapi kompleksitas waktu kasus terburuk masih n (n - 1) / 2.

Notasi Asimtotik Iterative Dan Recursive

Notasi Asimtotik Worst Case Scenario Iterative Dan Recursive

$$T(n) = \frac{n^2 - n}{2} \in O(n^2)$$

Class Efficiency Iterative Dan Recursive

Class Efficiency Worst case : Quadratic

Modelling Analisis Kompleksitas Waktu Algoritma

In [ ]:
```
# Mengimport Library Yang Dibutuhkan
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import timeit
import sys
import os

# Set Jumlah Limit Maksimal Proses Rekursif
sys.setrecursionlimit(4000)
os.makedirs('data', exist_ok=True)

# Mendefinisikan Function Calculate Untuk Perhitungan Waktu Algoritma
def calculate(fname, start):
    stop = timeit.default_timer()
    execution_time = stop - start
    return execution_time
```

In [ ]:
```
# Algoritma Quick Sort Rekursif
def quick_sort_recursion(S):
  def partition(S, low, high):
      i = (low - 1)
      pivot = S[high]

      for j in range(low, high):

          if S[j] <= pivot:

              i += 1
              S[i], S[j] = S[j], S[i]

      S[i + 1], S[high] = S[high], S[i + 1]
      return (i + 1)

  def quickSort(S, low, high):
      if low < high:

          pi = partition(S, low, high)

          quickSort(S, low, pi-1)
          quickSort(S, pi + 1, high)

  n = len(S)
  quickSort(S, 0, n - 1)
```

In [ ]:

```
# Algoritma Quick Sort Iteratif
def quick_sort_iterative(S):
    def partition(S, l, h):
        i = (l - 1)
        x = S[h]

        for j in range(l, h):
            if S[j] <= x:

                i = i+1
                S[i], S[j] = S[j], S[i]

        S[i+1], S[h] = S[h], S[i+1]
        return (i+1)

    def quickSortIterative(S, l, h):
        size = h - l + 1
        stack = [0] * (size)

        top = -1
        top = top + 1
        stack[top] = l
        top = top + 1
        stack[top] = h

        while top >= 0:
            h = stack[top]
            top = top - 1
            l = stack[top]
            top = top - 1

            p = partition(S, l, h)

            if p-1 > l:
                top = top + 1
                stack[top] = l
                top = top + 1
                stack[top] = p - 1

            if p+1 < h:
                top = top + 1
                stack[top] = p + 1
                top = top + 1
                stack[top] = h

    n = len(S)
    quickSortIterative(S, 0, n-1)
```

```
# Menggenerate Angka Untuk Test Kompleksitas
size = [1000, 1500, 2000, 2500, 3000, 3500]

for n in size:
    array = list(np.random.randint(1, 5000,  n))
    name = 'rand' + str(n) + '.dat'
    np.savetxt('data/'+name, array, fmt=['%i'])
```

```
# Melakukan Proses Perhitungan Waktu
qs = []

# Perulangan Untuk Quicksort Recursive Dengan Data Random
for n in size:
    name = 'rand' + str(n) + '.dat'

    for _ in range(3):
      data = np.loadtxt('data/' + name)
      start = timeit.default_timer()
      quick_sort_recursion(data)
      qs.append([n, calculate('Quicksort Recursive', start), 'Quicksort Recursive Random'])

# Perulangan Untuk Quicksort Iterative Data Random
for n in size:
    name = 'rand' + str(n) + '.dat'

    for _ in range(3):
      data = np.loadtxt('data/' + name)
      start = timeit.default_timer()
      quick_sort_iterative(data)
      qs.append([n, calculate('Quicksort Iterative', start), 'Quicksort Iterative Random'])
```

```
# Membuat Data Array Menjadi Dataframe Untuk Dapat Diolah Lebih Lanjut
qs_np = np.array(qs)
data = pd.DataFrame({'Jumlah Data': qs_np[:, 0], 'Kompleksitas Waktu': qs_np[:, 1], 'Kategori': qs_np[:, 2]})
# Melakukan Konversi Dan Pembulatan Hasil Perhitungan Kompleksitas Waktu
def float_qs(x):
  return round(float(x), 3)

data['Kompleksitas Waktu'] = data['Kompleksitas Waktu'].apply(float_qs)
```

```
# Memisahkan Data Berdasrkan Jenis Data Test Untuk Ditampilkan Pada Grafik
qs_random = data.loc[(data['Kategori'] == 'Quicksort Recursive Random') | (data['Kategori'] == 'Quicksort Iterative Random')]
# Melakukan Modelling Chart (Grafik)
def line_qs_random():
    plt.figure(figsize=(15, 8))
    sns.set_context("notebook", rc={"lines.linewidth": 2.5})
    sns.set_style("whitegrid")
    sns.lineplot(data=qs_random, x="Jumlah Data", y="Kompleksitas Waktu", hue='Kategori', marker="o", palette="deep", ci=None)

    plt.yticks(np.arange(0.003, 0.025, 0.002))
    plt.show()
# Memisahkan Data Untuk Ditampilkan Pada Tabel
qs_random = qs_random.sort_values(by=['Jumlah Data', 'Kategori'])
# Mengubah Jenis Perhitungan Menjadi Array Untuk Diolah Perhitungan Kompleksitasnya
data_qs_random = [list(x) for x in qs_random.to_records(index=False)]
```

Kesimpulan

Grafik Quicksort Keseluruhan Data

Pada grafik garis dibawah ini kami mendapatkan kesimpulan pada Algoritma Quick Sort memiliki :

Dari grafik dibawah dapat ditarik kesimpulan bahwa algoritma Quick Sort Iterative dengan data yang random cenderung sedikit lebih lambat di awal dibandingkan dengan algoritma Quick Sort Recursive dengan data random yang cenderung stabil namun keduanya berakhir dengan efisiensi kecepatan yang sama dan menjadikan test case ini sebagai best case pada algoritma Quick Sort.

```
table_qs_sum(data_qs_random)
```

	Jumlah Data	Rata-Rata Kompleksitas Waktu (s)	Kategori
0	1000	0.005	Quicksort Iterative Random
1	1000	0.005	Quicksort Recursive Random
2	1500	0.007	Quicksort Iterative Random
3	1500	0.007	Quicksort Recursive Random
4	2000	0.01	Quicksort Iterative Random
5	2000	0.01	Quicksort Recursive Random
6	2500	0.014	Quicksort Iterative Random
7	2500	0.013	Quicksort Recursive Random
8	3000	0.015	Quicksort Iterative Random
9	3000	0.016	Quicksort Recursive Random
10	3500	0.019	Quicksort Iterative Random
11	3500	0.02	Quicksort Recursive Random

```
line_qs_random()
```