TUGAS BESAR

STRATEGI ALGORITMA

PROSES PENGURUTAN KOMBINASI ANGKA DAN HURUF

(Quick Sort, Bubble Sort, dan Merge Sort)

SIGOOD

S1-IF-09-05

WIFQI WIFAKUL AZMI

21102277

PROGRAM STUDI S1 TEKNIK INFORMATIKA

FAKULTAS INFORMATIKA

UNIVERSITAS TELKOM PURWOKERTO

2024

DAFTAR ISI

BAB I
DASAR TEORI

1.1. Algoritma Sorting

Algoritma sorting adalah kumpulan langkah-langkah penyelesaian dalam suatu masalah dengan metode tertentu, sedangkan sorting didefinisikan sebagai pengurutan sejumlah data berdasarkan nilai kunci tertentu untuk mengurutkan nilai dari yang terkecil (*ascending*) atau sebaliknya (*descending*). tujuan algoritma sorting dilakukan untuk:

1. Meningkatkan efisiensi pencarian: Data yang terurut lebih mudah dicari, misalnya dengan binary search.
2. Penyajian data yang lebih baik: Menyusun data agar lebih mudah dipahami dan digunakan.
3. Mempermudah proses lanjutan: Banyak algoritma lain yang memerlukan data terurut untuk bekerja secara optimal.

1.2. Kompleksitas Algoritma

Kompleksitas algoritma adalah konsep yang digunakan untuk mengukur tingkat efisiensi suatu algoritma berdasarkan kebutuhan komputasi yang diperlukan guna menyelesaikan suatu masalah. pengukuran ini mencakup aspek waktu eksekusi dan penggunaan ruang memori. kompleksitas menjadi indikator penting dalam mengevaluasi performa algoritma, karena secara langsung memengaruhi seberapa cepat dan efisien algoritma dapat menyelesaikan tugas tertentu. Beberapa faktor yang memengaruhi kompleksitas algoritma meliputi:

1. Jumlah data masukan (sering disebut dengan n), yang memengaruhi waktu dan sumber daya yang dibutuhkan.
2. Waktu eksekusi, yaitu durasi yang diperlukan untuk menyelesaikan algoritma dari awal hingga akhir.

3. Penggunaan ruang memori, yang berkaitan dengan kapasitas memori yang harus disediakan untuk menyimpan data, variabel, atau struktur yang digunakan dalam algoritma.

Kompleksitas algoritma dapat diklasifikasikan menjadi tiga kategori utama, yaitu:

- ***Worst case*** (kasus terburuk): Menggambarkan waktu eksekusi terlama yang mungkin terjadi.
- ***Best case*** (kasus terbaik): Menunjukkan waktu eksekusi tercepat yang bisa dicapai.
- ***Average case*** (kasus rata-rata): Merepresentasikan waktu eksekusi rata-rata untuk berbagai kemungkinan input.

Setiap kategori ini memberikan gambaran berbeda mengenai efisiensi algoritma dalam kondisi tertentu. Analisis kompleksitas sangat penting, terutama untuk memastikan algoritma dapat berfungsi secara optimal meskipun dihadapkan pada jumlah data besar atau kondisi ekstrem.

1.3. Algoritma Buble Sort

Algoritma *Bubble Sort* adalah salah satu metode pengurutan (*sorting*) yang sederhana dan paling mudah dipahami, digunakan untuk mengatur elemen-elemen dalam suatu array atau daftar secara berurutan, baik dalam urutan menaik (*ascending*) maupun menurun (*descending*). metode ini bekerja dengan cara membandingkan pasangan elemen yang berdekatan, kemudian menukar posisinya jika elemen yang berada di posisi pertama lebih besar (atau lebih kecil, tergantung urutan yang diinginkan) daripada elemen di posisi berikutnya. Proses ini diulangi secara terus-menerus hingga seluruh elemen dalam daftar berada pada urutan yang diinginkan.

Berikut langkah-langkah yang dilakukan dalam pengurutan menggunakan algoritma *Bubble Sort* dapat dijelaskan melalui contoh berikut: misalkan sebuah array dengan elemen-elemen "4 2 5 3 9". proses pengurutan menggunakan algortima *Bubble Sort*

Pass Pertama:

Pada pass pertama, elemen-elemen dalam array akan dibandingkan secara berpasangan, dimulai dari elemen pertama hingga elemen terakhir. setiap pasangan yang ditemukan tidak sesuai dengan urutan yang diinginkan akan ditukar. Berikut adalah prosesnya:

- Array awal: **(4 2 5 3 9)**
- Bandingkan 4 dan 2: karena 4 > 2, tukar posisi → **(2 4 5 3 9)**
- Bandingkan 4 dan 5: tidak ada perubahan, karena 4 < 5 → **(2 4 5 3 9)**
- Bandingkan 5 dan 3: karena 5 > 3, tukar posisi → **(2 4 3 5 9)**
- Bandingkan 5 dan 9: tidak ada perubahan, karena 5 < 9 → **(2 4 3 5 9)**

Hasil setelah pass pertama: **(2 4 3 5 9)**. Elemen terbesar, yaitu 9, sudah berada pada posisi akhir.

Pass Kedua:

Proses dilanjutkan dengan membandingkan elemen-elemen lain, tetapi elemen terakhir (9) tidak lagi diikutsertakan, karena sudah berada di posisi yang benar.

- Array awal: **(2 4 3 5 9)**
- Bandingkan 2 dan 4: tidak ada perubahan, karena 2 < 4 → **(2 4 3 5 9)**
- Bandingkan 4 dan 3: karena 4 > 3, tukar posisi → **(2 3 4 5 9)**
- Bandingkan 4 dan 5: tidak ada perubahan, karena 4 < 5 → **(2 3 4 5 9)**

Hasil setelah pass kedua: **(2 3 4 5 9)**. Elemen kedua terbesar, yaitu 5, sudah berada pada posisi yang benar.

Pass Ketiga:

Pada pass ini, elemen-elemen yang tersisa akan dibandingkan untuk memastikan bahwa array benar-benar terurut.

- Array awal: **(2 3 4 5 9)**
- Bandingkan 2 dan 3: tidak ada perubahan, karena 2 < 3 → **(2 3 4 5 9)**
- Bandingkan 3 dan 4: tidak ada perubahan, karena 3 < 4 → **(2 3 4 5 9)**

Tidak ada penukaran pada pass ketiga. hal ini menunjukkan bahwa array telah terurut sepenuhnya. meskipun setelah pass kedua array sudah terlihat terurut, *Bubble Sort* tetap melanjutkan proses hingga tidak ada lagi penukaran yang terjadi dalam satu pass. karena algoritma *Bubble Sort* mengidentifikasi bahwa array sudah terurut hanya ketika seluruh pass selesai tanpa adanya pertukaran. Dalam kasus ini, pass ketiga diperlukan untuk memastikan bahwa semua elemen telah berada pada posisi yang benar.

1.4. Algoritma Merge Sort

Merge Sort adalah algoritma pengurutan yang sangat efisien dan memiliki kompleksitas waktu O (N log N) yang menjadikannya lebih unggul dibandingkan algoritma pengurutan sederhana seperti *Bubble Sort*, terutama untuk dataset yang besar. algoritma ini bekerja dengan prinsip *Divide and Conquer*, yaitu membagi masalah besar menjadi masalah yang lebih kecil, menyelesaikan masalah kecil, dan kemudian menggabungkannya kembali untuk mendapatkan solusi akhir yang terurut. pada dasarnya, *Merge Sort* melakukan pembagian array atau daftar yang akan diurutkan menjadi bagian-bagian yang lebih kecil, hingga akhirnya setiap bagian hanya memiliki satu elemen. selanjutnya, bagian-bagian ini digabungkan kembali dalam urutan yang benar.

1.4.1 Divide (*Pembagian*)

Proses pertama adalah membagi array yang akan diurutkan menjadi dua bagian yang hampir sama besar. Jika jumlah elemen array ganjil, salah satu bagian akan memiliki satu elemen lebih banyak daripada yang lain. Pembagian ini dilakukan secara rekursif hingga setiap bagian hanya mengandung satu elemen, karena sebuah array dengan satu elemen sudah otomatis terurut.

1.4.2 Conquer (*Penguasaan*)

Array dibagi menjadi subarray yang lebih kecil, setiap subarray yang hanya memiliki satu elemen sudah dianggap terurut. Tahap ini tidak memerlukan pengurutan lebih lanjut, karena bagian-bagian tersebut sudah memenuhi kriteria array terurut.

1.4.3 Combine (*Penggabungan*)

Setelah subarray-subarray kecil terurut, langkah berikutnya adalah menggabungkan subarray-subarray tersebut menjadi satu array yang lebih besar dan terurut. Penggabungan dilakukan dengan cara membandingkan elemen-elemen terdepan dari dua subarray yang terurut, kemudian elemen yang lebih kecil dari keduanya dipindahkan ke array hasil gabungan. Proses ini terus berlangsung sampai seluruh elemen dalam kedua subarray digabungkan dengan urutan yang benar.

1.5. Algoritma Quick Sort

Quick Sort adalah salah satu algoritma pengurutan (*sorting*) yang sangat efisien dan banyak digunakan dalam berbagai aplikasi untuk mengurutkan data dalam jumlah besar. algoritma ini dikembangkan oleh Tony Hoare pada tahun 1960 dan merupakan algoritma pengurutan yang berbasis pada prinsip *Divide and Conquer* (bagi dan kuasai), yang berarti bahwa algoritma ini bekerja dengan cara membagi masalah besar menjadi sub-masalah yang lebih kecil, mengurutkan sub-masalah tersebut, dan kemudian menggabungkannya kembali untuk mendapatkan solusi akhir.

Pada masalah pengurutan data bilangan bulat (integer) secara terindeks pada suatu list atau array dari bilangan yang paling besar sampai ke bilangan yang paling kecil atau sebaliknya. Tidak hanya dapat diterapkan pada pengindeksan bilangan saja, namun juga untuk pengindeksan huruf (abjad) dari A ke Z atau sebaliknya. Algoritma ini sangat baik diterapkan pada kasus pengindeksan kumpulan kata (library sort utility) atau kumpulan bilangan atau kombinasinya.

1.5.1 Cara Kerja *Quick Sort*

Quick Sort bekerja dengan memilih satu elemen dalam array sebagai pivot, dan kemudian membagi array tersebut menjadi dua bagian:

1. Bagian kiri: Berisi elemen-elemen yang lebih kecil dari pivot.
2. Bagian kanan: Berisi elemen-elemen yang lebih besar dari pivot.

Langkah-langkah ini diulang secara rekursif untuk bagian kiri dan kanan dari pivot, sehingga pada akhirnya seluruh elemen dalam array akan terurut. Berikut adalah gambaran langkah-langkah lebih detail dari algoritma Quick Sort:

1. **Pilih Pivot**:
   Pilih salah satu elemen dalam array sebagai pivot. Pemilihan pivot dapat dilakukan dengan berbagai cara, misalnya memilih elemen pertama, elemen terakhir, elemen tengah, atau memilih pivot secara acak.
2. **Partisi (Partitioning)**:
   Bagi array berdasarkan pivot, sehingga elemen-elemen yang lebih kecil dari pivot diletakkan di sebelah kiri pivot, dan elemen-elemen yang lebih besar diletakkan di sebelah kanan pivot. Proses partisi ini dilakukan dengan membandingkan elemen-elemen array satu per satu dengan pivot dan menukar posisi elemen jika diperlukan.
3. **Rekursi**:
   Setelah array dibagi, algoritma kemudian dipanggil secara rekursif untuk sub-array di sebelah kiri dan kanan pivot. Proses ini berlanjut hingga sub-array yang tersisa hanya berisi satu elemen atau kosong, yang berarti sub-array tersebut sudah terurut.
4. **Penggabungan**:
   Setelah rekursi selesai, seluruh array akan terurut karena setiap elemen telah diposisikan pada tempat yang benar relatif terhadap pivotnya masing-masing.

1.5.2 Kelebihan dan Kekurangan *Quick Sort*

a. Kelebihan:

- Sangat efisien dengan kompleksitas rata-rata O (n log n) O (n log n) O (n log n), lebih cepat dibandingkan algoritma pengurutan lain seperti *Bubble Sort* dalam kebanyakan kasus.

- Dapat diimplementasikan dalam versi *in-place sorting*, yang berarti algoritma ini tidak membutuhkan ruang tambahan yang besar, hanya ruang untuk rekursi.

b. Kekurangan:

- Pada kasus terburuk, yaitu jika pivot selalu dipilih dengan buruk (misalnya pivot selalu elemen terkecil atau terbesar), kompleksitasnya dapat menjadi O (*n*2) O (*n*^2) O (*n*2), meskipun hal ini dapat diminimalkan dengan pemilihan pivot yang lebih cerdas, seperti menggunakan teknik *median-of-three*.
- Algoritma ini bekerja dengan cara rekursif, yang dapat menyebabkan masalah pada stack overflow jika ukuran array sangat besar atau rekursi terlalu dalam.
- Tidak stabil (seperti *Merge Sort*), yang berarti urutan elemen yang sama dalam array tidak dijamin tetap setelah pengurutan.

BAB II
IMPLEMENTASI

2.1 Spesifikasi Hardware dan Software

1. Perangkat Keras *(Hardware)*
   a. Device : Acer Swift SF314-71
   b. Processor : 12th Gen Intel® Core™ i7-12700H 2.30 GHz
   c. RAM : 16,0 GB (15,7 GB usable)

2. Perangkat Lunak *(Software)*
   a. Sistem Operasi : *Windows 11 64-bit*
   b. Bahasa Pemrograman : *Python*
   c. Aplikasi : *Visual studio code*
   d. Compiler/Interpreter *Python Interpreter*

2.2 Penjelasan Pseudocode

Pseudocode ini menjelaskan cara kerja dari program yang membandingkan tiga algoritma pengurutan (*Bubble Sort, Quick Sort,* dan *Merge Sort*) berdasarkan waktu eksekusi dengan data acak. Berikut adalah ringkasan *pseudocode* untuk menjelaskan cara kerja program secara keseluruhan:

1. **Impor Library**:

   Program dimulai dengan mengimpor pustaka yang diperlukan untuk pembuatan data acak, pengurutan data, pengukuran waktu, dan pembuatan antarmuka pengguna.

```
import random # Untuk menghasilkan data acak.
import string # Untuk memilih karakter acak.
import time #  Untuk mengukur waktu eksekusi algoritma.
import tkinter as tk # Untuk membangun GUI aplikasi.
from tkinter import ttk, messagebox
from time import perf_counter
import matplotlib.pyplot as plt  # Untuk menggambar grafik hasil perbandingan.
```

2. **Fungsi generate_random_data**:

Fungsi ini menghasilkan sejumlah data acak dalam format string yang terdiri dari satu huruf (karakter acak) dan dua digit angka. Data ini nantinya akan digunakan sebagai input untuk algoritma pengurutan.

```
# Fungsi untuk menghasilkan data acak
# Data berupa string dengan format: satu huruf dan dua digit angka
def generate_random_data(size):
    return [f"{random.choice(string.ascii_letters)}{random.randint(0, 99)}" for _ in range(size)]
```

3. **Define algoritma sorting:**

a. *Bubble Sort*

Algoritma *Bubble Sort* bekerja dengan membandingkan pasangan elemen berurutan dalam array. Jika elemen pada indeks ke-i lebih besar dari elemen pada indeks ke-(i+1), maka kedua elemen tersebut akan ditukar posisinya. Proses ini diulang hingga seluruh elemen dalam array terurut.

```
# Bubble Sort
# Algoritma sorting sederhana yang membandingkan pasangan elemen berurutan dan menukarnya jika salah urutan
# Kompleksitas waktu: O(n^2) dalam kasus terburuk
def bubble_sort(arr):
    data = arr.copy()  # Salin array asli untuk menjaga integritas data input
    n = len(data)
    for i in range(n):
        for j in range(0, n - i - 1):
            if data[j] > data[j + 1]:
                data[j], data[j + 1] = data[j + 1], data[j]
    return data
```

b. *Quick Sort*

Algoritma *Quick Sort* memilih elemen yang lebih kecil dari pivot dan elemen yang lebih besar dari pivot. selanjutnya, *Quick Sort* akan dipanggil secara rekursif untuk kedua bagian tersebut. Proses ini terus berlanjut hingga bagian-bagian array menjadi sangat kecil.

```
# Quick Sort
# Algoritma sorting berbasis divide-and-conquer
# Menggunakan pivot untuk membagi array menjadi bagian kiri (elemen lebih kecil) dan kanan (elemen lebih besar)
# Kompleksitas waktu: O(n log n) dalam kasus rata-rata
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)
```

c. *Merge Sort*

Algoritma *Merge Sort* membagi array menjadi dua bagian lebih kecil, menyortir masing-masing bagian secara rekursif, dan akhirnya

menggabungkan kedua bagian tersebut menjadi satu array yang terurut.

```python
# Merge Sort
# Algoritma sorting berbasis divide-and-conquer
# Membagi array menjadi bagian kecil, menyortirnya, lalu menggabungkan kembali secara berurutan
# Kompleksitas waktu: O(n Log n)
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = merge_sort(arr[:mid])
    right = merge_sort(arr[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:])
    result.extend(right[j:])
    return result
```

4. **Fungsi start_sorting():**

Fungsi ini seluruh proses pengurutan dan pengukuran waktu. Untuk memasukkan ukuran masalah (problem_size) dan jumlah iterasi (iterations). program akan menghasilkan data acak sesuai dengan ukuran yang diminta, kemudian mengukur waktu eksekusi setiap algoritma pengurutan (*Bubble Sort, Quick Sort, Merge Sort*) menggunakan perf_counter() untuk menghitung waktu mulai dan selesai setiap iterasi.

```python
# Fungsi untuk memulai pengurutan dan menampilkan hasilnya di GUI
problem_sizes = []
bubble_times, quick_times, merge_times = [], [], []

def start_sorting():
    try:
        # Membaca ukuran masalah dan jumlah iterasi dari input pengguna
        problem_size = int(problem_size_var.get())
        if problem_size < 10:
            raise ValueError("Problem size harus >= 10.")
        iterations = int(iterations_var.get())

        # Menghasilkan data acak
        original_data = generate_random_data(problem_size)

        # Mengukur waktu Bubble Sort
        start_time = perf_counter()
        for _ in range(iterations):
            bubble_sort(original_data)
        bubble_time = ((perf_counter() - start_time) / iterations) * 1000  # Waktu rata-rata dalam ms

        # Mengukur waktu Quick Sort
        start_time = perf_counter()
        for _ in range(iterations):
            quick_sort(original_data)
        quick_time = ((perf_counter() - start_time) / iterations) * 1000  # Waktu rata-rata dalam ms

        # Mengukur waktu Merge Sort
        start_time = perf_counter()
        for _ in range(iterations):
            merge_sort(original_data)
        merge_time = ((perf_counter() - start_time) / iterations) * 1000  # Waktu rata-rata dalam ms
```

Kemudian hasil waktu eksekusi rata-rata dari setiap algoritma kemudian ditampilkan dalam bentuk tabel di antarmuka pengguna (GUI). Data sebelum dan sesudah pengurutan juga akan ditampilkan dalam listbox untuk setiap algoritma, sehingga dapat melihat bagaimana data berubah setelah pengurutan.

```python
        # Menampilkan data sebelum dan sesudah pengurutan di listbox GUI
        bubble_listbox.delete(0, tk.END)
        quick_listbox.delete(0, tk.END)
        merge_listbox.delete(0, tk.END)
        before_listbox.delete(0, tk.END)

        for item in original_data:
            before_listbox.insert(tk.END, item)
        for item in bubble_sort(original_data):
            bubble_listbox.insert(tk.END, item)
        for item in quick_sort(original_data):
            quick_listbox.insert(tk.END, item)
        for item in merge_sort(original_data):
            merge_listbox.insert(tk.END, item)

        # Memasukkan hasil waktu eksekusi ke tabel
        result_table.insert("", tk.END, values=(problem_size, f"{bubble_time:.3f}", f"{merge_time:.3f}", f"{quick_time:.3f}"))

        # Menyimpan hasil untuk grafik
        problem_sizes.append(problem_size)
        bubble_times.append(bubble_time)
        quick_times.append(quick_time)
        merge_times.append(merge_time)

        # Jika sudah 10 ukuran data, tampilkan grafik
        if len(problem_sizes) >= 10:
            plot_graph()

    except ValueError as e:
        # Menampilkan pesan kesalahan jika input tidak valid
        messagebox.showerror("Error", str(e))
```

5. **Fungsi plot_graph():**

Fungsi ini menggambar grafik yang menunjukkan waktu eksekusi rata-rata dari setiap algoritma terhadap ukuran input (problem size). grafik ini akan menunjukkan tren waktu eksekusi masing-masing algoritma seiring dengan meningkatnya ukuran data yang diuji.

```
# Fungsi untuk membuat grafik laju kenaikan waktu eksekusi
def plot_graph():
    plt.figure(figsize=(10, 6))
    plt.plot(problem_sizes, bubble_times, label="Bubble Sort", marker="o")
    plt.plot(problem_sizes, quick_times, label="Quick Sort", marker="s")
    plt.plot(problem_sizes, merge_times, label="Merge Sort", marker="^")
    plt.title("Laju Kenaikan Waktu Eksekusi Sorting Algorithms")
    plt.xlabel("Ukuran Input (n)")
    plt.ylabel("Waktu Eksekusi (ms)")
    plt.legend()
    plt.grid(True)
    plt.show()
```

6. **Antarmuka Pengguna (GUI):**

Fungsi ini untuk membuat antarmuka pengguna yang interaktif di dalam GUI, dapat memasukkan ukuran data dan jumlah iterasi melalui field input yang disediakan. setelah itu, dapat menekan tombol "Start" untuk memulai proses pengurutan.

```
# Membuat GUI menggunakan tkinter
root = tk.Tk()
root.title("Sorting Algorithm Comparison")

# Variabel untuk input ukuran masalah dan iterasi
problem_size_var = tk.StringVar(value="10")
iterations_var = tk.StringVar(value="100")

# Frame untuk input parameter
frame_input = tk.Frame(root)
frame_input.pack(pady=10)

tk.Label(frame_input, text="Problem size:").grid(row=0, column=0, padx=5, pady=5)
tk.Entry(frame_input, textvariable=problem_size_var).grid(row=0, column=1, padx=5, pady=5)

tk.Label(frame_input, text="Iteration:").grid(row=1, column=0, padx=5, pady=5)
tk.Entry(frame_input, textvariable=iterations_var).grid(row=1, column=1, padx=5, pady=5)

tk.Button(frame_input, text="Start", command=start_sorting).grid(row=2, column=0, columnspan=2, pady=10)
```

Kemudian program akan menampilkan hasil pengurutan dalam listbox yang menunjukkan data sebelum dan setelah pengurutan oleh masing-masing algoritma.

```
# Frame untuk menampilkan list hasil sorting
frame_lists = tk.Frame(root)
frame_lists.pack()

before_listbox = tk.Listbox(frame_lists, height=15, width=15)
bubble_listbox = tk.Listbox(frame_lists, height=15, width=15)
quick_listbox = tk.Listbox(frame_lists, height=15, width=15)
merge_listbox = tk.Listbox(frame_lists, height=15, width=15)

before_listbox.grid(row=0, column=0, padx=5)
bubble_listbox.grid(row=0, column=1, padx=5)
quick_listbox.grid(row=0, column=2, padx=5)
merge_listbox.grid(row=0, column=3, padx=5)

tk.Label(frame_lists, text="Before").grid(row=1, column=0)
tk.Label(frame_lists, text="Bubble").grid(row=1, column=1)
tk.Label(frame_lists, text="Quick").grid(row=1, column=2)
tk.Label(frame_lists, text="Merge").grid(row=1, column=3)
```

selain itu, tabel di bagian bawah GUI akan menampilkan waktu eksekusi dari setiap algoritma untuk setiap ukuran masalah yang diuji.

```
# Frame untuk tabel hasil waktu eksekusi
frame_table = tk.Frame(root)
frame_table.pack(pady=10)

columns = ("n", "Bubble (ms)", "Merge (ms)", "Quick (ms)")
result_table = ttk.Treeview(frame_table, columns=columns, show="headings", height=8)

for col in columns:
    result_table.heading(col, text=col)
    result_table.column(col, anchor="center")

result_table.pack()

# Menjalankan aplikasi GUI
root.mainloop()
```

BAB III
PENGUJIAN

3.1 Pengujian Program Sorting Algorithm

1. Data yang digunakan

   Data yang digunakan untuk pengujian adalah data acak berupa string yang terdiri dari satu huruf (karakter acak dari string.ascii_letters) dan dua digit angka (angka acak antara 0 hingga 99). Setiap data yang dihasilkan memiliki format seperti A23, B45, atau Z67. Misalnya, jika ukuran inputnya adalah 100, maka program akan menghasilkan 100 data acak.

2. Pengujian jumlah input yang berbeda

   Pengujian dilakukan dengan ukuran input yang semakin besar, seperti:

   - 100 Iteration
   - 200 Iteration
   - 300 Iteration
   - 400 Iteration
   - 500 Iteration dan seterusnya

   Masing-masing pengujian dilakukan beberapa kali untuk mendapatkan rata-rata waktu eksekusi yang lebih stabil.

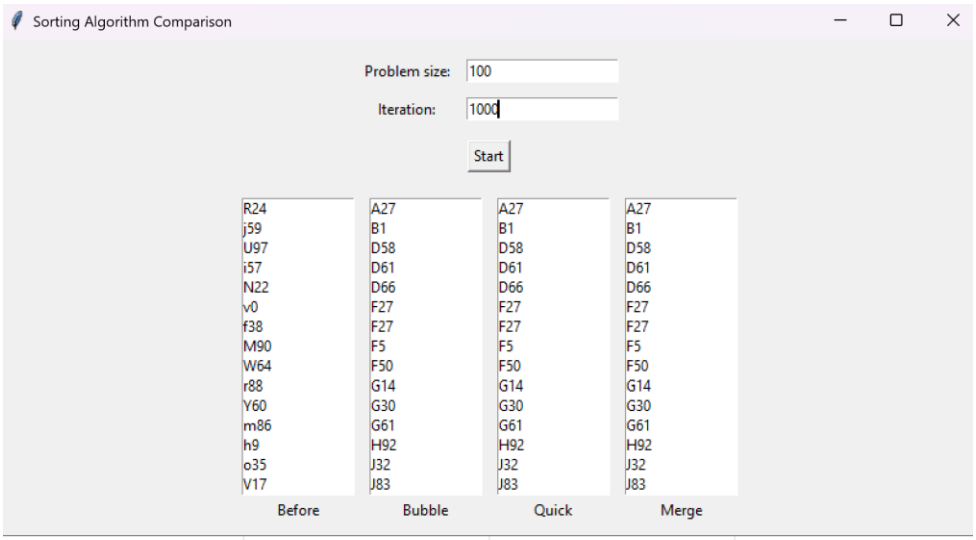
3. Pengujian dengan data yang sama untuk semua algoritma

   Untuk memastikan bahwa data yang digunakan untuk pembandingan algoritma adalah data yang sama, dengan menyimpan data acak yang dihasilkan dalam array yang terpisah dan menggunakan data yang sama sebagai input untuk ketiga algoritma. Langkah-langkahnya adalah sebagai berikut:

   - Hasilkan data acak dengan fungsi generate_random_data(size) untuk setiap ukuran input yang diuji.
   - Simpan data acak tersebut dalam array baru.
   - Gunakan data yang sama untuk *Bubble Sort, Quick Sort, dan Merge Sort.*

4. Catat waktu eksekusi untuk masing-masing Algoritma:

   Setelah data dihasilkan, waktu eksekusi untuk masing-masing algoritma (*Bubble Sort, Quick Sort, Merge Sort*) dihitung dengan menggunakan fungsi perf_counter(). Program kemudian mencatat waktu rata-rata untuk setiap algoritma berdasarkan jumlah iterasi yang diberikan. Berikut adalah hasil pengujian proses pengukuran data dan waktu eksekusi untuk setiap algoritma:



n	Bubble (ms)	Merge (ms)	Quick (ms)
30	0.040	0.017	0.019
40	0.050	0.026	0.025
50	0.075	0.035	0.038
60	0.107	0.043	0.044
70	0.141	0.052	0.051
80	0.181	0.062	0.066
90	0.225	0.071	0.073
100	0.264	0.078	0.080

5. Hasil Pengujian

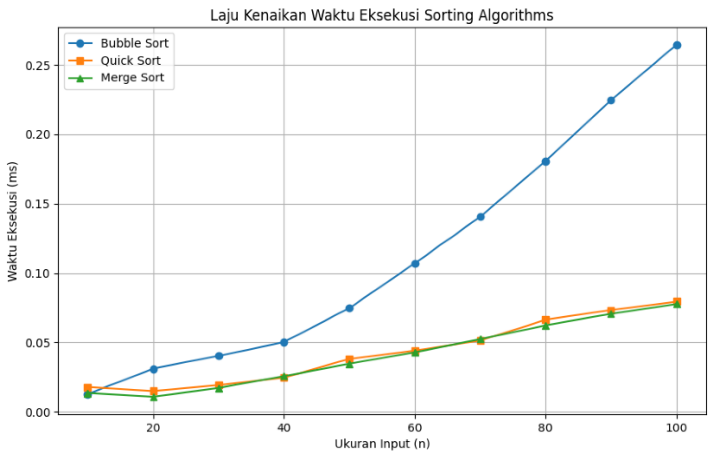
Setelah pengujian dilakukan, waktu eksekusi untuk masing-masing algoritma dapat dicatat dalam tabel dengan kolom sebagai berikut:

Keterangan:

n = jumlah inputan

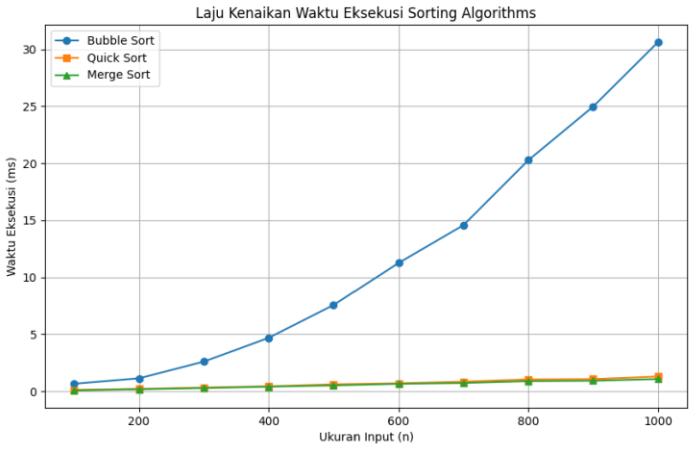
Tabel Hasil Perbandingan Pada Proses Problem Size

No	***Problem Size* (n)**	***Iteration***	***Bubble sort***	***Quick sort***	***Merge sort***
			Waktu Proses (ms)		
1.	10	100	0.013	0.014	0.018
2.	20	200	0.031	0.011	0.015
3.	30	300	0.040	0.017	0.019
4.	40	400	0.050	0.026	0.025
5.	50	500	0.075	0.035	0.038
6.	60	600	0.107	0.043	0.044
7.	70	700	0.141	0.052	0.051
8.	80	800	0.181	0.062	0.066
9.	90	900	0.225	0.071	0.073
10	100	1000	0.264	0.078	0.080



Tabel Hasil Perbandingan Pada Proses Iteration

No	*Problem Size* (n)	*Iteration*	*Bubble sort*	*Quick sort*	*Merge sort*
			Waktu Proses (ms)		
1.	100	10	0.650	0.073	0.104
2.	200	20	1.125	0.171	0.194
3.	300	30	2.591	0.275	0.311
4.	400	40	4.677	0.397	0.423
5.	500	50	7.557	0.506	0.599
6.	600	60	11.256	0.645	0.689
7.	700	70	14.575	0.718	0.833
8.	800	80	20.272	0.881	1.032
9.	900	90	24.979	0.914	1.057
10	1000	100	30.639	1.063	1.290



BAB IV
ANALISIS HASIL PENGUJIAN

Dari hasil pengujian yang dilakukan, waktu eksekusi (running time) untuk masing-masing algoritma sorting, yaitu *Bubble Sort, Quick Sort*, dan *Merge Sort,* pada berbagai ukuran masalah (problem size) dan iterasi yang berbeda. Berikut adalah beberapa analisis perbandingan hasil waktu proses masing-masing algoritma:

1. Perbandingan Waktu Proses untuk Setiap Algoritma
    - **Bubble Sort**: Pada pengujian dengan ukuran data kecil (misalnya 10 hingga 20 elemen), waktu eksekusi untuk *Bubble Sort* relatif sangat kecil (hanya beberapa milidetik). namun, seiring dengan bertambahnya ukuran data, waktu eksekusi *Bubble Sort* meningkat secara signifikan. Misalnya, pada ukuran data 100 elemen, waktu eksekusi *Bubble Sort* adalah 0.264 ms, yang jauh lebih lama dibandingkan dengan *Quick Sort dan Merge Sort.*
    - **Quick Sort**: menunjukkan waktu eksekusi yang relatif lebih cepat dibandingkan dengan *Bubble Sort,* meskipun ada sedikit peningkatan waktu saat ukuran data meningkat. Misalnya, pada ukuran data 100 elemen, waktu eksekusi *Quick Sort* adalah 0.078 ms, lebih rendah dibandingkan *Bubble Sort* (0.264 ms), meskipun sedikit lebih tinggi daripada *Merge Sort.*
    - **Merge Sort**: menunjukkan performa yang sangat baik, dengan waktu eksekusi yang hampir sebanding dengan *Quick Sort.* Misalnya, pada ukuran data 100 elemen, waktu eksekusi *Merge Sort* adalah 0.080 ms, yang sangat mirip dengan *Quick Sort* (0.078 ms).

2. Pengaruh Ukuran Data terhadap Waktu Eksekusi
    - Ketiga algoritma menunjukkan peningkatan waktu eksekusi yang sebanding dengan pertambahan ukuran data. Namun, *Bubble Sort* jelas lebih lambat dibandingkan dengan kedua algoritma lainnya seiring dengan bertambahnya ukuran data.
    - *Quick Sort* dan *Merge Sort* memberikan waktu eksekusi yang lebih konsisten dan lebih cepat pada ukuran data yang lebih besar, berkat kompleksitas waktu yang lebih baik, yaitu **O(n log n)**.
3. Perbandingan Waktu Proses pada Ukuran Data yang Lebih Besar
    - Pada pengujian ukuran data yang lebih besar (seperti 100 elemen), *Quick Sort* dan *Merge Sort* hampir memberikan waktu eksekusi yang sama, dengan perbedaan waktu yang sangat kecil (0.078 ms vs. 0.080 ms).
    - Sementara itu, *Bubble Sort* menunjukkan waktu eksekusi yang jauh lebih tinggi, yang semakin terlihat jelas ketika ukuran data meningkat.

BAB V
KESIMPULAN

1. *Bubble Sort* memiliki waktu yang kurang efisien untuk ukuran data besar. Meskipun untuk data kecil hasilnya cukup cepat, kompleksitas waktu **O(n^2)** membuat algoritma ini tidak cocok untuk ukuran data besar, karena waktu eksekusinya meningkat secara signifikan.
2. *Quick Sort* dan *Merge Sort* memiliki waktu eksekusi yang sangat mirip dan jauh lebih cepat dibandingkan dengan Bubble Sort. Kedua algoritma ini memiliki kompleksitas waktu **O(n log n)**.

Jadi, *Quick Sort* adalah algoritma yang lebih baik jika dibandingkan dengan *Bubble Sort* berdasarkan hasil pengujian ini, terutama untuk dataset yang lebih besar. Meskipun *Merge Sort* sedikit lebih lambat daripada *Quick Sort,* keduanya jauh lebih efisien daripada Bubble Sort.

REFRENSI

Wisudawan, Wahyu Fahmy, 2007. *Kompleksitas Algoritma Sorting yang Populer Dipakai. Bandung:* Teknik Informatika, Institut Teknologi Bandung

Hendra Saptadi, Arief . 2013, *Analisis Algoritma Insertion sort, Merge sort dan Implementasinya dalam Bahasa Pemograman C++*. Akademi Teknik Telekomunikasi Shandy Putra Purwokerto