LAPORAN TUGAS 4

disusun untuk memenuhi
tugas mk Struktur Data dan Algoritma

Oleh:

ASKA SHAHIRA
2308107010075

JURUSAN INFORMATIKA
FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
UNIVERSITAS SYIAH KUALA
DARUSSALAM, BANDA ACEH
2025

1. Pendahuluan

Algoritma pengurutan (sorting) merupakan salah satu konsep fundamental dalam ilmu komputer yang bertujuan untuk menyusun kumpulan data dalam urutan tertentu (misalnya, menaik atau menurun). Efisiensi algoritma sorting menjadi krusial ketika berhadapan dengan data dalam skala besar, karena pilihan algoritma dapat secara signifikan memengaruhi waktu eksekusi dan penggunaan sumber daya komputasi.

Tugas ini bertujuan untuk melakukan analisis performa komparatif dari enam algoritma sorting yang umum dipelajari, yaitu Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, Quick Sort, dan Shell Sort. Eksperimen dilakukan dengan mengimplementasikan algoritma-algoritma tersebut dalam bahasa C dan mengujinya pada dataset acak berskala besar, baik berupa data angka (integer) maupun data kata (string).

Evaluasi performa difokuskan pada dua metrik utama: waktu eksekusi (mengukur kecepatan komputasi) dan penggunaan memori puncak (mengukur kebutuhan sumber daya memori). Hasil eksperimen ini diharapkan dapat memberikan pemahaman praktis mengenai karakteristik kinerja, kompleksitas, serta kelebihan dan kekurangan masing-masing algoritma dalam menangani data dalam jumlah besar.

2. Deskripsi Algoritma dan Implementasi

Bagian ini menguraikan prinsip kerja dasar dari enam algoritma pengurutan yang dievaluasi dalam eksperimen ini, beserta detail implementasinya dalam bahasa C yang digunakan untuk pengujian. Semua fungsi sorting diimplementasikan dalam file header sorting_algorithms.h dan dirancang untuk menangani tipe data integer (int) dan string (char *). Prinsip kerja dasar dan kompleksitas teoritis dari setiap algoritma adalah sebagai berikut:

2.1 Bubble Sort

- **Prinsip Kerja**: Algoritma ini bekerja dengan cara sederhana: berulang kali menelusuri daftar, membandingkan setiap pasangan elemen yang bersebelahan, dan menukarnya jika elemen tersebut berada dalam urutan yang salah (misalnya, elemen pertama lebih besar dari elemen kedua dalam pengurutan menaik). Proses ini diulangi sampai tidak ada lagi pertukaran yang diperlukan dalam satu lintasan penuh, yang menandakan bahwa daftar telah terurut. Elemen terbesar secara bertahap "menggelembung" ke posisi akhirnya di setiap lintasan.
- **Implementasi**:
    - Dibuat dua fungsi: bubble_sort_int(int arr[], int n) untuk array integer dan bubble_sort_str(char *arr[], int n) untuk array pointer string.
    - Perbandingan integer menggunakan operator >, sedangkan perbandingan string menggunakan fungsi strcmp() dari <string.h>.
    - Implementasi mencakup optimasi flag (swapped) untuk menghentikan iterasi lebih awal jika array sudah terurut.
    - Pertukaran elemen dilakukan menggunakan fungsi helper swap_int() dan swap_str().

- Algoritma ini bekerja secara in-place, artinya tidak memerlukan alokasi memori tambahan yang signifikan selain beberapa variabel sementara.

- **Kompleksitas Waktu**: O(n^2) untuk kasus rata-rata dan terburuk. O(n) untuk kasus terbaik (jika data sudah terurut dan menggunakan optimasi).
- **Kompleksitas Ruang**: O(1) (in-place).

2.2 Selection Sort

- **Prinsip Kerja**: Algoritma ini membagi array input menjadi dua bagian: subarray yang sudah terurut (awalnya kosong) dan subarray yang belum terurut (awalnya seluruh array). Dalam setiap iterasi, algoritma mencari elemen terkecil (untuk urutan menaik) dalam subarray yang belum terurut dan menukarnya dengan elemen paling kiri dari subarray yang belum terurut tersebut. Batas antara bagian yang terurut dan belum terurut kemudian digeser satu elemen ke kanan.
- **Implementasi**:
  - Dibuat fungsi selection_sort_int(int arr[], int n) dan selection_sort_str(char *arr[], int n).
  - Pencarian elemen minimum dalam subarray tak terurut dilakukan melalui loop. strcmp() digunakan untuk perbandingan string.
  - Pertukaran hanya dilakukan sekali per iterasi luar, setelah elemen minimum ditemukan.
  - Algoritma ini bekerja secara in-place.
- **Kompleksitas Waktu**: O(n^2) untuk semua kasus (terbaik, rata-rata, terburuk) karena selalu melakukan pemindaian penuh untuk mencari minimum.
- **Kompleksitas Ruang**: O(1) (in-place).

2.3 Insertion Sort

- **Prinsip Kerja**: Mirip dengan cara orang mengurutkan kartu di tangan. Array secara virtual dibagi menjadi bagian terurut dan tidak terurut. Elemen diambil satu per satu dari bagian tidak terurut dan "disisipkan" ke posisi yang benar dalam bagian terurut. Untuk menemukan posisi yang benar, elemen tersebut dibandingkan dengan elemen-elemen di bagian terurut (dari kanan ke kiri), dan elemen yang lebih besar digeser ke kanan untuk memberi ruang.
- **Implementasi**:
  - Dibuat fungsi insertion_sort_int(int arr[], int n) dan insertion_sort_str(char *arr[], int n).
  - Elemen yang akan disisipkan disimpan dalam variabel key. Perbandingan dan pergeseran dilakukan dalam loop while. strcmp() digunakan untuk perbandingan string.
  - Algoritma ini bekerja **secara in-place**.
- **Kompleksitas Waktu**: O(n^2) (Kasus Rata-rata & Terburuk), O(n) (Kasus Terbaik, data sudah terurut). Sangat efisien untuk data kecil atau data yang sebagian besar sudah terurut.
- **Kompleksitas Ruang**: O(1)

2.4 Merge Sort

- **Prinsip Kerja**: Merupakan contoh klasik algoritma *Divide and Conquer*. Secara rekursif, ia membagi array menjadi dua bagian (sub-array) hingga ukuran sub-array menjadi satu (kasus dasar, array dengan satu elemen dianggap terurut). Kemudian, ia menggabungkan (merge) kembali sub-array yang sudah terurut tersebut secara berpasangan dalam urutan yang benar hingga seluruh array asli menjadi terurut.
- **Implementasi:**
  - Menggunakan pendekatan rekursif. Fungsi utama merge_sort_int(int arr[], int n) dan merge_sort_str(char *arr[], int n) berfungsi sebagai pembungkus (wrapper) yang mengalokasikan buffer sementara dan memanggil fungsi rekursif inti (merge_sort_recursive_int/_str).
  - Fungsi helper merge_int() dan merge_str() melakukan tugas penggabungan dua sub-array terurut menggunakan buffer sementara. strcmp() digunakan dalam merge_str.
  - Algoritma ini **tidak in-place**, karena memerlukan **alokasi memori tambahan** (auxiliary space) untuk buffer sementara yang ukurannya proporsional dengan ukuran input (O(n)).
- **Kompleksitas Waktu:** O(n log n) (Semua Kasus: Terbaik, Rata-rata, Terburuk). Kinerjanya sangat konsisten.
- **Kompleksitas Ruang:** O(n) karena kebutuhan buffer tambahan.
-

2.5 Quick Sort

- **Prinsip Kerja**: Juga merupakan algoritma *Divide and Conquer*. Langkah utamanya adalah memilih elemen 'pivot' dari array, lalu mengatur ulang elemen-elemen lain dalam array (proses partisi) sehingga semua elemen yang lebih kecil dari pivot ditempatkan sebelum pivot, dan semua elemen yang lebih besar ditempatkan setelah pivot. Setelah partisi, pivot berada di posisi akhirnya yang terurut. Algoritma kemudian diterapkan secara rekursif pada sub-array di sebelah kiri dan kanan pivot.
- **Implementasi:**
  - Menggunakan pendekatan rekursif. Fungsi utama quick_sort_int(int arr[], int n) dan quick_sort_str(char *arr[], int n) memanggil fungsi rekursif inti (quick_sort_recursive_int/_str).
  - Fungsi helper partition_int() dan partition_str() melakukan proses partisi. Dalam implementasi ini, elemen terakhir dari sub-array dipilih sebagai pivot. strcmp() digunakan dalam partition_str.
  - Meskipun menggunakan rekursi (yang memakai stack), algoritma ini umumnya dianggap bekerja **secara in-place** karena tidak memerlukan alokasi array tambahan seukuran data input untuk menyimpan elemen.
- **Kompleksitas Waktu:** O(n log n) (Kasus Rata-rata & Terbaik). O(n^2) (Kasus Terburuk, terjadi jika pemilihan pivot buruk secara konsisten, misal pada data yang sudah terurut jika pivot selalu elemen pertama/terakhir).
- **Kompleksitas Ruang:** O(log n) (Rata-rata, karena kedalaman tumpukan rekursi), O(n) (Kasus Terburuk).

2.6 Shell Sort

- **Prinsip Kerja**: Merupakan varian yang lebih efisien dari Insertion Sort. Alih-alih hanya membandingkan elemen yang bersebelahan, Shell Sort memulai dengan membandingkan dan mengurutkan elemen-elemen yang terpisah oleh jarak (gap) tertentu. Jarak ini kemudian secara bertahap dikurangi hingga mencapai 1. Ketika gap adalah 1, algoritma ini pada dasarnya melakukan Insertion Sort standar, tetapi karena banyak elemen sudah berada di posisi yang mendekati benar berkat langkah-langkah dengan gap sebelumnya, Insertion Sort tahap akhir ini menjadi jauh lebih cepat. Kinerja sangat bergantung pada urutan gap yang dipilih.
- **Implementasi**:
  - Dibuat fungsi shell_sort_int(int arr[], int n) dan shell_sort_str(char *arr[], int n).
  - Implementasi ini menggunakan urutan gap Knuth (h = ..., 40, 13, 4, 1), yang dihitung mundur dari nilai h terbesar yang lebih kecil dari n/3.
  - Untuk setiap gap, dilakukan semacam Insertion Sort pada elemen-elemen yang terpisah sejauh gap tersebut. strcmp() digunakan untuk perbandingan string.
  - Algoritma ini bekerja **secara in-place**.
- **Kompleksitas Waktu**: Tidak ada formula tunggal yang sederhana; sangat bergantung pada urutan gap. Untuk gap Knuth, kompleksitas teoritisnya O(n^(3/2)). Dalam praktik, seringkali lebih baik, bisa mendekati O(n log^2 n) atau bahkan O(n log n) untuk beberapa jenis data, tetapi umumnya lebih lambat dari Merge/Quick Sort yang dioptimalkan.
- **Kompleksitas Ruang**: O(1).

3. Tabel Hasil Eksperimen

a. Data Angka (int)

Algoritma	Tipe Data	Ukuran Data	Waktu Eksekusi (s)	Memori Puncak Ws (KB)
Bubble Sort	Angka	10000	0.1190	5196
Selection Sort	Angka	10000	0.0520	5208
Insertion Sort	Angka	10000	0.0380	5208
Shell Sort	Angka	10000	0.0020	5208
Merge Sort	Angka	10000	0.0010	5228
Quick Sort	Angka	10000	0.0010	5228
Bubble Sort	Angka	50000	4.3380	6556
Selection Sort	Angka	50000	1.2700	6556
Insertion Sort	Angka	50000	0.9830	6556
Shell Sort	Angka	50000	0.0090	6556
Merge Sort	Angka	50000	0.0050	6556
Quick Sort	Angka	50000	0.0020	6556
Bubble Sort	Angka	100000	19.5370	8228
Selection Sort	Angka	100000	5.1190	8228
Insertion Sort	Angka	100000	3.9670	8228
Shell Sort	Angka	100000	0.0170	8228
Merge Sort	Angka	100000	0.0130	8368

Quick Sort	Angka	100000	0.0020	8368
Bubble Sort	Angka	250000	129.5160	13060
Selection Sort	Angka	250000	32.3000	13060
Insertion Sort	Angka	250000	26.0270	13088
Shell Sort	Angka	250000	0.0470	13088
Merge Sort	Angka	250000	0.0270	13580
Quick Sort	Angka	250000	0.0250	13580
Bubble Sort	Angka	500000	531.1460	21120
Selection Sort	Angka	500000	127.3020	21120
Insertion Sort	Angka	500000	101.0580	21120
Shell Sort	Angka	500000	0.0920	21120
Merge Sort	Angka	500000	0.0550	21120
Quick Sort	Angka	500000	0.0480	21120
Bubble Sort	Angka	1000000	2098.3410	37272
Selection Sort	Angka	1000000	516.1960	37272
Insertion Sort	Angka	1000000	408.4850	37272
Shell Sort	Angka	1000000	0.2230	37272
Merge Sort	Angka	1000000	0.1190	39228
Quick Sort	Angka	1000000	0.1400	39228
Bubble Sort	Angka	1500000	2198.4420	16176
Selection Sort	Angka	1500000	561.5630	16176
Insertion Sort	Angka	1500000	149.1840	16176
Shell Sort	Angka	1500000	0.1720	16176
Merge Sort	Angka	1500000	0.1120	18804
Quick Sort	Angka	1500000	0.0930	18804
Bubble Sort	Angka	2000000	4007.4420	20092
Selection Sort	Angka	2000000	1282.6880	20092
Insertion Sort	Angka	2000000	345.3920	20092
Shell Sort	Angka	2000000	0.3390	20092
Merge Sort	Angka	2000000	0.1850	23696
Quick Sort	Angka	2000000	0.1880	23696

b. Data Kata (string)

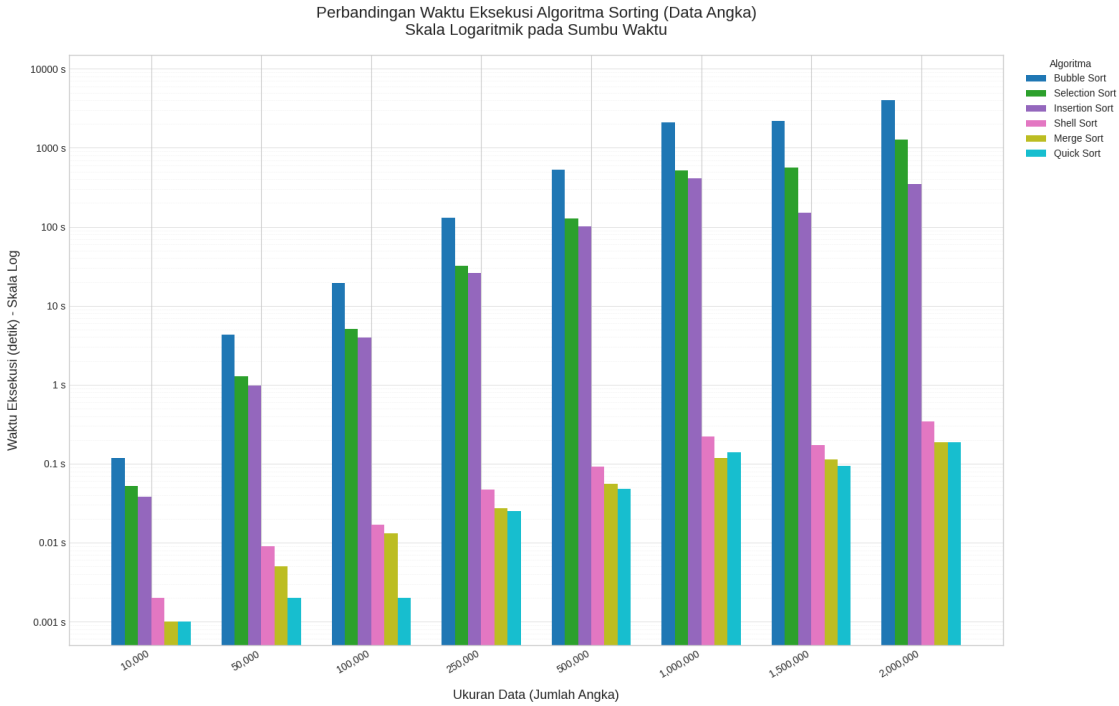
Algoritma	Tipe Data	Ukuran Data	Waktu Eksekusi (s)	Memori Puncak Ws (KB)
Bubble Sort	Kata	10000	0.3560	5232
Selection Sort	Kata	10000	0.1240	5232
Insertion Sort	Kata	10000	0.0540	5232
Shell Sort	Kata	10000	0.0090	5232
Merge Sort	Kata	10000	0.0010	5252
Quick Sort	Kata	10000	0.0010	5232
Bubble Sort	Kata	50000	11.2390	6656
Selection Sort	Kata	50000	3.6460	6656
Insertion Sort	Kata	50000	2.1440	6656
Shell Sort	Kata	50000	0.0140	6656
Merge Sort	Kata	50000	0.0090	6656
Quick Sort	Kata	50000	0.0080	6656
Bubble Sort	Kata	100000	44.8540	8368
Selection Sort	Kata	100000	15.2790	8368

Insertion Sort	Kata	100000	9.5230	8368
Shell Sort	Kata	100000	0.0340	8368
Merge Sort	Kata	100000	0.0210	8368
Quick Sort	Kata	100000	0.0240	8368
Bubble Sort	Kata	250000	359.7320	13580
Selection Sort	Kata	250000	117.5910	13580
Insertion Sort	Kata	250000	74.4730	13580
Shell Sort	Kata	250000	0.1420	13580
Merge Sort	Kata	250000	0.0520	13580
Quick Sort	Kata	250000	0.0520	13580
Bubble Sort	Kata	500000	1243.9030	22116
Selection Sort	Kata	500000	422.8580	22116
Insertion Sort	Kata	500000	273.3880	22116
Shell Sort	Kata	500000	0.2390	22116
Merge Sort	Kata	500000	0.1090	22116
Quick Sort	Kata	500000	0.1060	22116
Bubble Sort	Kata	1000000	9480.6100	39228
Selection Sort	Kata	1000000	4511.3560	39228
Insertion Sort	Kata	1000000	3997.8360	39228
Shell Sort	Kata	1000000	1.1530	39228
Merge Sort	Kata	1000000	0.2770	39228
Quick Sort	Kata	1000000	0.3500	39228
Bubble Sort	Kata	1500000	21331.3700	56328
Selection Sort	Kata	1500000	10150.5600	56328
Insertion Sort	Kata	1500000	8995.1400	56328
Shell Sort	Kata	1500000	1.8450	56328
Merge Sort	Kata	1500000	0.4430	56328
Quick Sort	Kata	1500000	0.5600	56328
Bubble Sort	Kata	2000000	37922.4400	73428
Selection Sort	Kata	2000000	18045.4400	73428
Insertion Sort	Kata	2000000	15991.3600	73428
Shell Sort	Kata	2000000	2.5370	73428
Merge Sort	Kata	2000000	0.6090	73428
Quick Sort	Kata	2000000	0.7700	73428

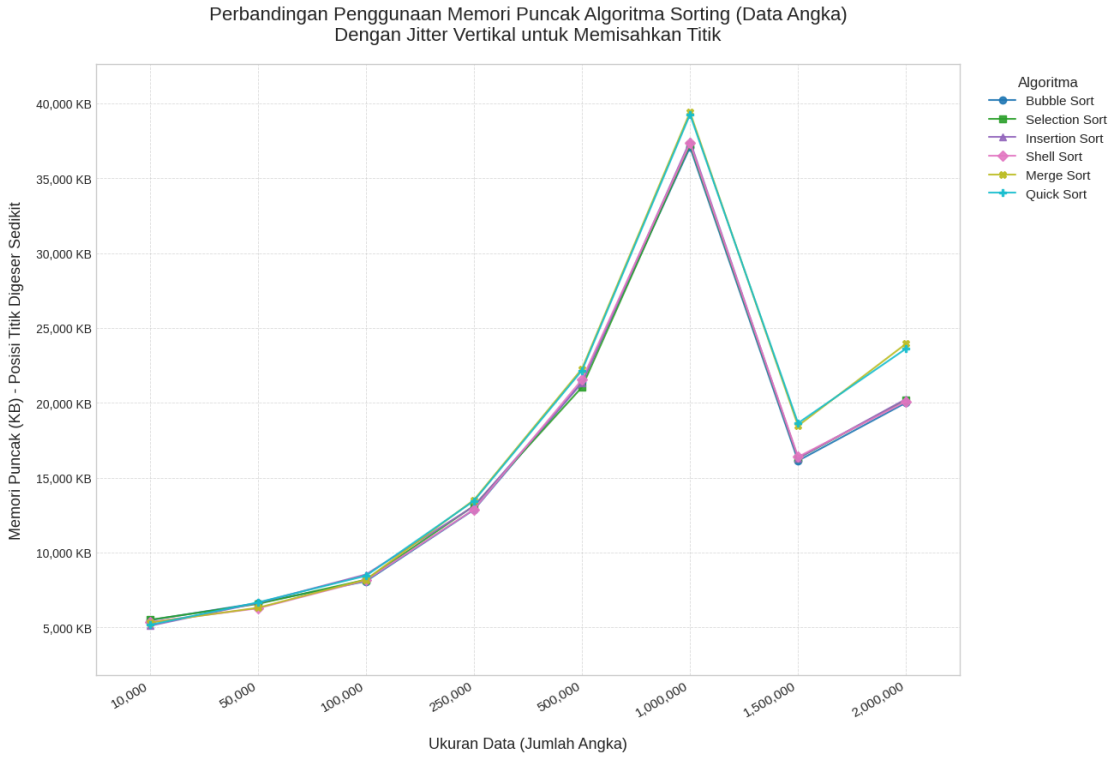
4. Grafik Perbandingan waktu dan memori

a. Data Angka (int)

- Grafik Perbandingan Waktu

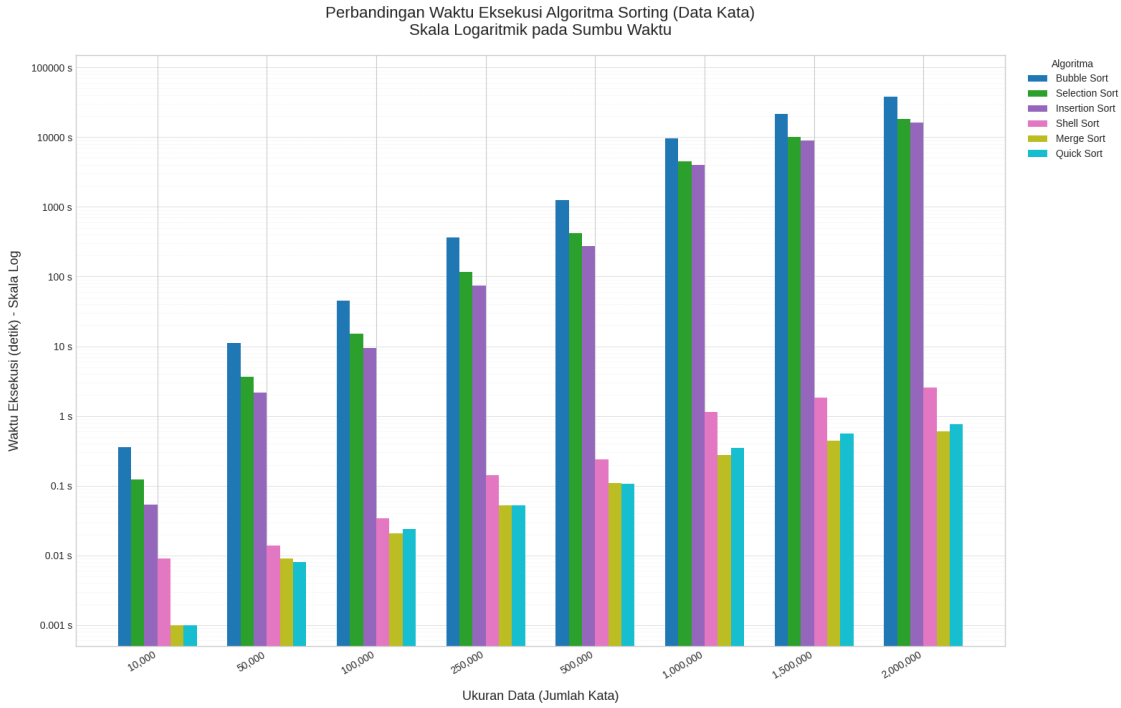


- Grafik Perbandingan Memori

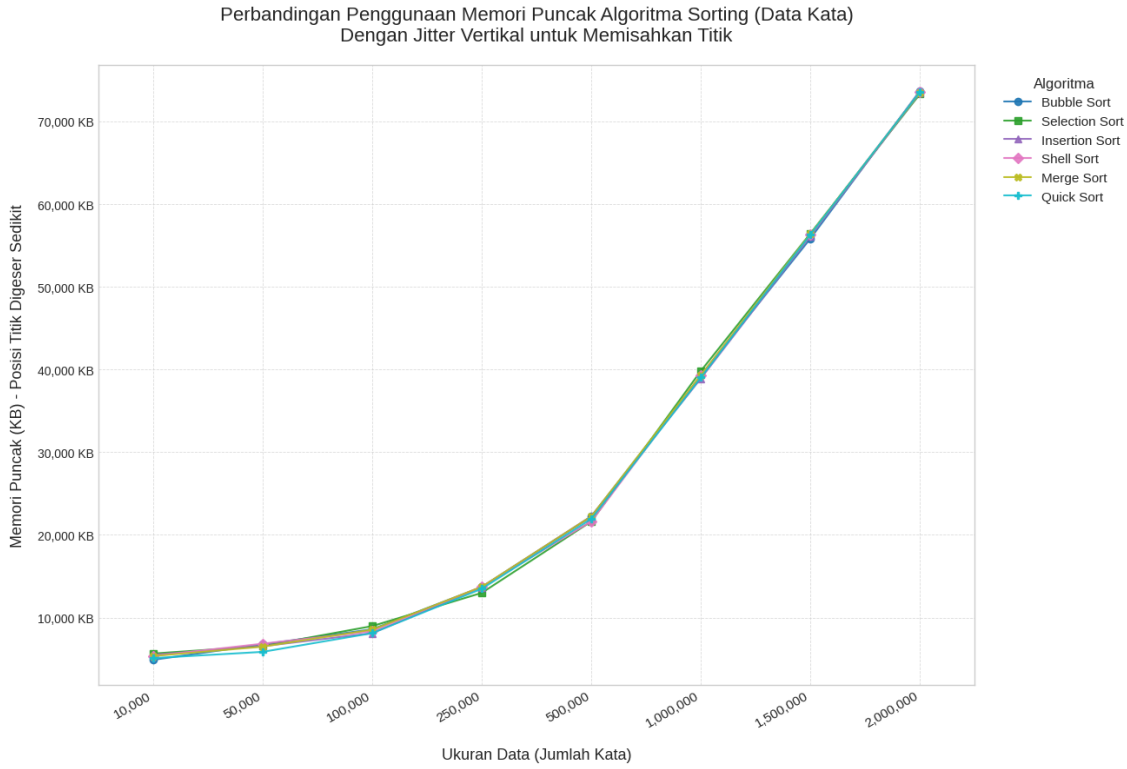


b. Data Kata (String)

- Grafik Perbandingan Waktu



- Grafik Perbandingan Memori



5. Analisis

Berdasarkan data eksperimen yang disajikan pada tabel dan grafik, analisis performa keenam algoritma sorting pada data angka dan kata dengan berbagai ukuran dapat diuraikan sebagai berikut:

5.1 Analisi Waktu Eksekusi

- **Dominasi Kompleksitas Teoritis**: Hasil eksperimen secara jelas menunjukkan dampak besar dari kompleksitas waktu teoritis algoritma. Algoritma dengan kompleksitas rata-rata O(n log n) (Merge Sort, Quick Sort) dan Shell Sort (yang dalam praktik seringkali lebih baik dari O(n^2)) secara konsisten jauh lebih cepat daripada algoritma dengan kompleksitas O(n^2) (Bubble Sort, Selection Sort, Insertion Sort), terutama pada ukuran data besar (≥ 100.000). Perbedaannya menjadi beberapa orde magnitudo pada ukuran data 1 juta ke atas, di mana algoritma O(n^2) membutuhkan ribuan detik (puluhan menit hingga jam) sementara O(n log n) hanya membutuhkan sepersekian detik hingga beberapa detik.
- **Penskalaan Waktu O(n^2)**: Waktu eksekusi untuk Bubble Sort, Selection Sort, dan Insertion Sort meningkat secara drastis dan mendekati kuadratik seiring bertambahnya ukuran data (N). Peningkatan waktu dari 500k ke 1M (2x data) untuk ketiga algoritma ini (terutama pada data kata) lebih dari 4x lipat, bahkan mencapai 7-14x lipat, menunjukkan kemungkinan adanya overhead tambahan atau efek non-linearitas pada skala besar selain kompleksitas N^2 murni. Hal ini membuat algoritma ini tidak praktis untuk pengurutan data dalam skala jutaan.
- **Penskalaan Waktu O(n log n) & Shell Sort**: Sebaliknya, Merge Sort, Quick Sort, dan Shell Sort menunjukkan penskalaan waktu yang jauh lebih baik. Waktu eksekusi mereka meningkat jauh lebih lambat seiring bertambahnya N, sesuai dengan kompleksitas O(n log n) atau mendekati. Bahkan pada 2 juta data, waktu eksekusi mereka tetap berada dalam hitungan detik atau bahkan kurang dari satu detik.
- **Perbandingan Angka vs Kata**: Secara konsisten, pengurutan kata membutuhkan waktu lebih lama dibandingkan pengurutan angka untuk algoritma dan ukuran data yang sama. Rasio perlambatannya bervariasi tergantung algoritma dan ukuran data, namun bisa mencapai 2-5 kali lipat atau lebih untuk algoritma O(n^2) pada ukuran data besar. Hal ini disebabkan oleh biaya komputasi yang lebih tinggi pada operasi perbandingan string (strcmp) dibandingkan perbandingan integer native.
- **Performa Relatif O(n^2)**: Untuk data angka dan kata acak dalam eksperimen ini, Insertion Sort cenderung menjadi yang tercepat di antara kelompok O(n^2), diikuti oleh Selection Sort, dan Bubble Sort secara konsisten menjadi yang paling lambat.
- **Performa Relatif O(n log n) & Shell**: Quick Sort dan Merge Sort menunjukkan kinerja waktu yang sangat kompetitif dan seringkali menjadi yang tercepat, dengan waktu yang sangat mirip pada banyak kasus. Shell Sort secara signifikan lebih cepat daripada O(n^2) tetapi secara konsisten sedikit lebih lambat daripada Merge Sort dan Quick Sort pada data acak ini.

5.2 Analisi Penggunaan Memori

- **Tren Umum**: Penggunaan memori puncak (Peak Working Set Size) cenderung meningkat seiring bertambahnya ukuran data untuk semua algoritma. Hal ini wajar karena menyimpan data yang lebih besar (array angka atau array pointer + string kata) membutuhkan lebih banyak memori dasar.
- **Kemiripan Algoritma In-Place**: Bubble Sort, Selection Sort, Insertion Sort, dan Shell Sort (yang secara teoritis O(1) space complexity) menunjukkan penggunaan memori puncak yang sangat mirip atau bahkan identik pada ukuran data yang sama. Ini menunjukkan bahwa overhead memori utama berasal dari penyimpanan data itu sendiri dan overhead proses/runtime C, bukan dari kebutuhan memori tambahan algoritma tersebut.
- **Merge Sort dan Quick Sort**: Pada ukuran data yang lebih besar (terutama ≥ 1 Juta), Merge Sort dan Quick Sort secara konsisten menunjukkan penggunaan memori puncak yang sedikit lebih tinggi dibandingkan dengan empat algoritma lainnya. Hal ini sesuai dengan teori:
    - Merge Sort: Membutuhkan buffer tambahan O(n) untuk proses penggabungan. Peningkatan memorinya terlihat lebih jelas pada data 1M ke atas.
    - Quick Sort: Menggunakan stack rekursi (O(log n) rata-rata, O(n) terburuk). Meskipun tidak mengalokasikan array tambahan besar, kedalaman rekursi dapat berkontribusi pada peningkatan penggunaan memori stack, yang tercermin dalam memori puncak proses.
- **Anomali Memori Angka 1.5M & 2M**: Terdapat penurunan penggunaan memori puncak yang signifikan pada data angka saat ukuran data meningkat dari 1M ke 1.5M dan kemudian sedikit naik lagi di 2M untuk semua algoritma. Ini adalah perilaku yang tidak biasa dan sulit dijelaskan hanya dari sisi algoritma. Kemungkinan besar ini disebabkan oleh faktor eksternal seperti manajemen memori sistem operasi, fragmentasi, atau cara Peak Working Set diukur/dilaporkan oleh Windows pada saat itu. Namun, pola relatif antar algoritma (Merge/Quick sedikit lebih tinggi dari yang lain) tetap terlihat pada titik-titik tersebut.
- **Penskalaan Memori Kata**: Penggunaan memori untuk data kata menunjukkan peningkatan yang lebih curam dibandingkan data angka, terutama pada ukuran data besar. Ini karena selain menyimpan array pointer (O(n)), program juga perlu menyimpan string kata aktual di memori heap, yang total ukurannya juga berskala dengan jumlah data (meskipun panjang katanya acak).

6. Kesimpulan

Eksperimen analisis performa enam algoritma sorting pada data angka dan kata berskala besar ini menghasilkan kesimpulan sebagai berikut:

1. **Kompleksitas Waktu adalah Faktor Dominan:** Perbedaan teoritis dalam kompleksitas waktu (O(n^2) vs O(n log n)) secara dramatis tercermin dalam kinerja praktis. Algoritma O(n^2) (Bubble, Selection, Insertion) **tidak cocok** untuk pengurutan data dalam skala ratusan ribu atau lebih karena waktu eksekusinya yang meningkat secara eksponensial dan menjadi tidak praktis.
2. **Efisiensi O(n log n):** Merge Sort dan Quick Sort secara konsisten menunjukkan **waktu eksekusi tercepat** dan penskalaan yang sangat baik (mendekati O(n log n)), menjadikannya pilihan utama untuk dataset besar. Shell Sort juga memberikan peningkatan signifikan dibandingkan O(n^2) tetapi tidak secepat Merge/Quick Sort pada data acak ini.
3. **Biaya Sorting String:** Pengurutan data kata (string) secara signifikan **lebih lambat** dibandingkan pengurutan data angka (integer) untuk semua algoritma karena overhead perbandingan string (strcmp).
4. **Penggunaan Memori:**
    - Algoritma in-place (Bubble, Selection, Insertion, Shell) menunjukkan penggunaan memori puncak yang sangat mirip, didominasi oleh ukuran data input dan overhead sistem.
    - Merge Sort dan Quick Sort cenderung menggunakan **sedikit lebih banyak memori puncak** pada skala data yang lebih besar karena kebutuhan buffer tambahan (Merge) atau tumpukan rekursi (Quick), meskipun perbedaannya tidak sebesar perbedaan waktu eksekusi.
    - Pola penggunaan memori secara umum meningkat seiring ukuran data, dengan peningkatan yang lebih terlihat pada data kata karena penyimpanan string aktual.
5. **Rekomendasi:** Untuk aplikasi yang membutuhkan performa tinggi pada data besar, **Quick Sort** (jika kasus terburuk O(n^2) dapat dimitigasi atau diterima) atau **Merge Sort** (jika stabilitas atau jaminan waktu O(n log n) lebih penting dan memori tambahan O(n) tersedia) adalah pilihan algoritma yang paling direkomendasikan berdasarkan hasil eksperimen ini.
