Laporan Struktur Data dan Algoritma

Perbandingan Waktu dan Penggunaan Memory untuk Algoritma Sorting Berbeda

disusun untuk memenuhi
tugas mata kuliah Struktur Data dan Algoritma

Oleh:

REZZA RAMADHANA
2308107010019

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

A. Deskripsi Algoritma

Algoritma yang digunakan pada percobaan ini ada 6 algoritma sorting, yaitu bubble sort, selection sort, insertion sort, merge sort, quick sort, dan shell sort. Masing-masing algoritma tersebut memiliki kompleksitas waktu dan memory yang berbeda-beda. Berikut adalah deskripsi dan pengimplementasian dari masing-masing algoritma.

1. Bubble Sort

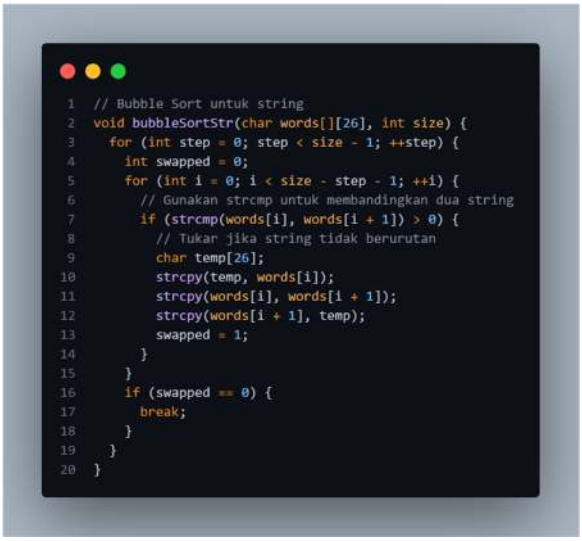
Bubble Sort adalah algoritma pengurutan yang membandingkan dua elemen yang bersebelahan dan menukar posisinya jika mereka tidak berada dalam urutan yang benar. Proses ini diulang terus menerus hingga seluruh elemen dalam array terurut. Nama "bubble" berasal dari proses seperti gelembung udara yang naik ke permukaan air—elemen terbesar akan "mengambang" ke posisi akhirnya di setiap iterasi.

Cara kerja :

- Mulai dari indeks pertama array.
- Bandingkan elemen saat ini dengan elemen setelahnya.
- Jika elemen saat ini lebih besar, tukar posisinya.
- Lanjutkan proses ini hingga ke elemen terakhir.
- Setelah satu iterasi, elemen terbesar akan berada di posisi akhir array.
- Ulangi proses untuk sisa array (tanpa menyertakan elemen yang sudah terurut di akhir).
- Hentikan iterasi saat tidak ada lagi pertukaran yang terjadi.

Implementasi dalam C :

```c
// perform the bubble sort
void bubbleSort(int array[], int size) {

  // loop to access each array element
  for (int step = 0; step < size - 1; ++step) {

    // check if swapping occurs
    int swapped = 0;

    // loop to compare array elements
    for (int i = 0; i < size - step - 1; ++i) {

      // compare two array elements
      // change > to < to sort in descending order
      if (array[i] > array[i + 1]) {

        // swapping occurs if elements
        // are not in the intended order
        int temp = array[i];
        array[i] = array[i + 1];
        array[i + 1] = temp;

        swapped = 1;
      }
    }

    // no swapping means the array is already sorted
    // so no need for further comparison
    if (swapped == 0) {
      break;
    }

  }
}
```



```c
// Bubble Sort untuk string
void bubbleSortStr(char words[][26], int size) {
  for (int step = 0; step < size - 1; ++step) {
    int swapped = 0;
    for (int i = 0; i < size - step - 1; ++i) {
      // Gunakan strcmp untuk membandingkan dua string
      if (strcmp(words[i], words[i + 1]) > 0) {
        // Tukar jika string tidak berurutan
        char temp[26];
        strcpy(temp, words[i]);
        strcpy(words[i], words[i + 1]);
        strcpy(words[i + 1], temp);
        swapped = 1;
      }
    }
    if (swapped == 0) {
      break;
    }
  }
}
```

2. Selection Sort

Selection Sort adalah algoritma pengurutan yang bekerja dengan cara memilih elemen terkecil dari array yang belum terurut, kemudian menempatkannya pada posisi awal dari bagian array yang belum terurut. Algoritma ini membagi array menjadi dua bagian: subarray yang sudah terurut dan subarray yang belum terurut.

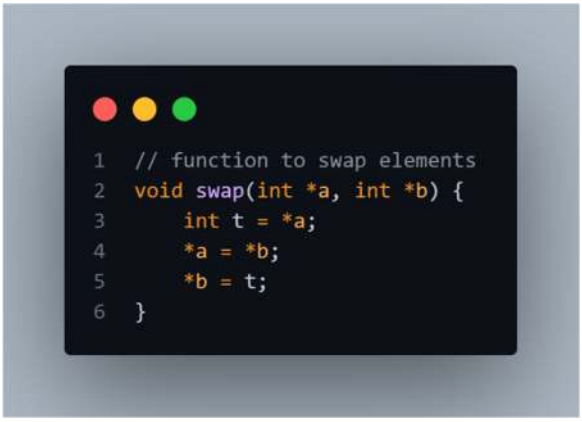
Cara kerja :

- Anggap elemen pertama sebagai nilai minimum.
- Bandingkan nilai minimum dengan seluruh elemen setelahnya.
- Jika ditemukan elemen yang lebih kecil, perbarui nilai minimum.
- Setelah menemukan nilai minimum dalam satu iterasi, tukar dengan posisi awal dari subarray yang belum terurut.
- Ulangi proses untuk elemen berikutnya hingga seluruh array terurut.

Implementasi dalam C :

```c
// Function to swap strings
void swapStr(char *a, char *b) {
  char *temp = (char *)malloc((strlen(a) + 1) * sizeof(char)); // Alokasi buffer dinamis
  if (temp == NULL) {
    perror("Gagal mengalokasikan memori");
    return;
  }
  strcpy(temp, a); // Salin string `a` ke buffer
  strcpy(a, b);    // Salin string `b` ke string `a`
  strcpy(b, temp); // Salin string dari buffer ke string `b`
  free(temp);      // Bebaskan memori setelah digunakan
}
```

```c
// Selection Sort untuk string
void selectionSortStr(char words[][26], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < size; i++) {
      // Gunakan strcmp untuk menemukan string terkecil
      if (strcmp(words[i], words[min_idx]) < 0) {
        min_idx = i;
      }
    }
    // Tukar string
    char temp[26];
    strcpy(temp, words[min_idx]);
    strcpy(words[min_idx], words[step]);
    strcpy(words[step], temp);
  }
}
```



```c
// function to swap elements
void swap(int *a, int *b) {
    int t = *a;
    *a = *b;
    *b = t;
}
```

```c
// Selection Sort in C
void selectionSort(int array[], int size) {
  for (int step = 0; step < size - 1; step++) {
    int min_idx = step;
    for (int i = step + 1; i < size; i++) {

      // To sort in descending order, change > to < in this line.
      // Select the minimum element in each loop.
      if (array[i] < array[min_idx])
        min_idx = i;
    }

    // put min at the correct position
    swap(&array[min_idx], &array[step]);
  }
}
```

3. Insertion Sort

Insertion Sort adalah algoritma pengurutan yang menempatkan setiap elemen dari array ke posisi yang sesuai dengan cara menyisipkannya ke bagian array yang sudah terurut. Proses ini mirip dengan cara seseorang menyusun kartu di tangan dalam permainan kartu.

Cara kerja :

- Anggap elemen pertama dalam array sudah terurut.
- Ambil elemen kedua sebagai key.
- Bandingkan key dengan elemen sebelumnya.
- Jika elemen sebelumnya lebih besar, geser ke kanan.
- Tempatkan key di posisi yang sesuai.
- Ulangi proses untuk elemen ketiga dan seterusnya sampai akhir array.

Implementasi dalam C :

```c
// Insertion sort in C
void insertionSort(int array[], int size) {
  for (int step = 1; step < size; step++) {
    int key = array[step];
    int j = step - 1;

    // Compare key with each element on the left of it until an element smaller than
    // it is found.
    // For descending order, change key<array[j] to key>array[j].
    while (j >=0 && key < array[j]) {
      array[j + 1] = array[j];
      --j;
    }
    array[j + 1] = key;
  }
}
```

```c
// Insertion Sort untuk string
void insertionSortStr(char words[][26], int size) {
  for (int step = 1; step < size; step++) {
    char key[26];
    strcpy(key, words[step]);
    int j = step - 1;
    // Pindahkan elemen yang lebih besar dari key ke satu posisi ke depan
    while (j >= 0 && strcmp(key, words[j]) < 0) {
      strcpy(words[j + 1], words[j]);
      j--;
    }
    strcpy(words[j + 1], key);
  }
}
```

4. Merge Sort

Merge Sort adalah algoritma pengurutan yang menggunakan pendekatan *divide and conquer*. Algoritma ini membagi data menjadi dua bagian, mengurutkan masing-masing bagian secara rekursif, lalu menggabungkannya kembali menjadi satu daftar yang terurut.

Cara Kerja:

- Divide: Bagi array menjadi dua bagian hingga setiap subarray hanya memiliki satu elemen.
- Conquer: Urutkan setiap subarray secara rekursif menggunakan Merge Sort.
- Combine: Gabungkan dua subarray yang sudah terurut menjadi satu array terurut.

Implementasi dalam C :

```c
// Fungsi Merge untuk string
void mergeStr(char words[][26], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    // Alokasi memori untuk array L dan R
    char **L = (char **)malloc(n1 * sizeof(char *));
    char **R = (char **)malloc(n2 * sizeof(char *));
    for (int i = 0; i < n1; i++) {
        L[i] = (char *)malloc(26 * sizeof(char));
        strcpy(L[i], words[l + i]);
    }
    for (int j = 0; j < n2; j++) {
        R[j] = (char *)malloc(26 * sizeof(char));
        strcpy(R[j], words[m + 1 + j]);
    }

    int i = 0, j = 0, k = l;

    while (i < n1 && j < n2) {
        if (strcmp(L[i], R[j]) <= 0) {
            strcpy(words[k], L[i]);
            i++;
        } else {
            strcpy(words[k], R[j]);
            j++;
        }
        k++;
    }

    while (i < n1) {
        strcpy(words[k], L[i]);
        i++;
        k++;
    }

    while (j < n2) {
        strcpy(words[k], R[j]);
        j++;
        k++;
    }

    // Bebaskan memori yang dialokasikan
    for (int i = 0; i < n1; i++) {
        free(L[i]);
    }
    free(L);
    for (int j = 0; j < n2; j++) {
        free(R[j]);
    }
    free(R);
}

// Merge Sort untuk string
void mergeSortStr(char words[][26], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSortStr(words, l, m);
        mergeSortStr(words, m + 1, r);
        mergeStr(words, l, m, r);
    }
}
```

```c
// Merge sort in C
// Merge two subarrays L and M into arr
void merge(int arr[], int p, int q, int r) {

  // Create L ← A[p..q] and M ← A[q+1..r]
  int n1 = q - p + 1;
  int n2 = r - q;

  int *L = (int *)malloc(n1 * sizeof(int));
  int *M = (int *)malloc(n2 * sizeof(int));

  for (int i = 0; i < n1; i++)
    L[i] = arr[p + i];
  for (int j = 0; j < n2; j++)
    M[j] = arr[q + 1 + j];

  // Maintain current index of sub-arrays and main array
  int i, j, k;
  i = 0;
  j = 0;
  k = p;

  // Until we reach either end of either L or M, pick larger among
  // elements L and M and place them in the correct position at A[p..r]
  while (i < n1 && j < n2) {
    if (L[i] <= M[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = M[j];
      j++;
    }
    k++;
  }

  // When we run out of elements in either L or M,
  // pick up the remaining elements and put in A[p..r]
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }

  while (j < n2) {
    arr[k] = M[j];
    j++;
    k++;
  }
}

// Divide the array into two subarrays, sort them and merge them
void mergeSort(int arr[], int l, int r) {
  if (l < r) {

    // m is the point where the array is divided into two subarrays
    int m = l + (r - l) / 2;

    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);

    // Merge the sorted subarrays
    merge(arr, l, m, r);
  }
}
```

5. Quick Sort

Quick Sort adalah algoritma pengurutan yang menggunakan pendekatan *divide and conquer*, di mana array dibagi menjadi subarray dengan memilih elemen

pivot. Subarray kemudian diurutkan secara rekursif untuk mendapatkan array yang terurut.

Cara Kerja:

1. Pilih Pivot: Pilih satu elemen dari array sebagai pivot.
2. Partisi: Susun elemen sehingga semua elemen yang lebih kecil dari pivot berada di sebelah kiri, dan yang lebih besar di sebelah kanan.
3. Rekursi: Terapkan langkah 1 dan 2 secara rekursif pada subarray kiri dan kanan hingga seluruh array terurut.

Implementasi dalam C :

```c
// Quick sort in C
// function to find the partition position
int partition(int array[], int low, int high) {

  // select the rightmost element as pivot
  int pivot = array[high];

  // pointer for greater element
  int i = (low - 1);

  // traverse each element of the array
  // compare them with the pivot
  for (int j = low; j < high; j++) {
    if (array[j] <= pivot) {

      // if element smaller than pivot is found
      // swap it with the greater element pointed by i
      i++;

      // swap element at i with element at j
      swap(&array[i], &array[j]);
    }
  }

  // swap the pivot element with the greater element at i
  swap(&array[i + 1], &array[high]);

  // return the partition point
  return (i + 1);
}

void quickSort(int array[], int low, int high) {
  if (low < high) {

    // find the pivot element such that
    // elements smaller than pivot are on left of pivot
    // elements greater than pivot are on right of pivot

```

```c
// Fungsi Partition untuk string
int partitionStr(char words[][26], int low, int high) {
  char pivot[26];
  strcpy(pivot, words[high]);
  int i = low - 1;

  for (int j = low; j < high; j++) {
    if (strcmp(words[j], pivot) <= 0) {
      i++;
      char temp[26];
      strcpy(temp, words[i]);
      strcpy(words[i], words[j]);
      strcpy(words[j], temp);
    }
  }

  char temp[26];
  strcpy(temp, words[i + 1]);
  strcpy(words[i + 1], words[high]);
  strcpy(words[high], temp);

  return (i + 1);
}

// Quick Sort untuk string
void quickSortStr(char words[][26], int low, int high) {
  if (low < high) {
    int pi = partitionStr(words, low, high);
    quickSortStr(words, low, pi - 1);
    quickSortStr(words, pi + 1, high);
  }
}
```

6. Shell Sort

Shell Sort adalah algoritma pengurutan yang merupakan generalisasi dari Insertion Sort. Algoritma ini pertama-tama mengurutkan elemen-elemen yang berjauhan satu sama lain, kemudian secara bertahap mengurangi jarak antar elemen yang dibandingkan. Tujuannya adalah untuk memindahkan elemen-elemen ke posisi yang lebih dekat dengan posisi akhirnya, sehingga ketika dilakukan pengurutan akhir dengan jarak 1 (seperti Insertion Sort), array sudah hampir terurut dan prosesnya menjadi lebih efisien.

Cara Kerja:

4. Tentukan jarak awal (gap), biasanya dimulai dari n/2.
5. Urutkan elemen-elemen yang terpisah sejauh gap menggunakan Insertion Sort.
6. Kurangi nilai gap secara bertahap (misalnya dibagi dua).

7. Ulangi proses hingga gap menjadi 1, dan array terurut sepenuhnya.

Implementasi dalam C :

```c
// Shell Sort untuk string
void shellSortStr(char words[][26], int size) {
  for (int interval = size / 2; interval > 0; interval /= 2) {
    for (int i = interval; i < size; i++) {
      char temp[26];
      strcpy(temp, words[i]);
      int j;
      for (j = i; j >= interval && strcmp(words[j - interval], temp) > 0; j -= interval) {
        strcpy(words[j], words[j - interval]);
      }
      strcpy(words[j], temp);
    }
  }
}
```

```c
// Shell Sort in C programming
// Shell sort
void shellSort(int array[], int n) {
  // Rearrange elements at each n/2, n/4, n/8, ... intervals
  for (int interval = n / 2; interval > 0; interval /= 2) {
    for (int i = interval; i < n; i += 1) {
      int temp = array[i];
      int j;
      for (j = i; j >= interval && array[j - interval] > temp; j -= interval) {
        array[j] = array[j - interval];
      }
      array[j] = temp;
    }
  }
}
```

B. Tabel Hasil Eksperimen

Parameter dari pengujian ini adalah waktu dan penggunaan memory selama algoritma sorting dijalankan. Pengujian dilakukan pada ukuran data yang berbeda-beda, mulai dari 10.000, 50.000, 100.000, 250.000, 500.000, 1.000.000, 1.500.000, dan 2.000.000. Pengujian juga dilakukan untuk 2 jenis data berbeda yaitu angka dan huruf.

1. Data Angka

a. Ukuran data 10.000

Angka			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	10000	0.190 second	4524 KB
Selection sort	10000	0.082 second	4536 KB
Insertion sort	10000	0.077 second	4536 KB
Merge sort	10000	0.006 second	4568 KB
Quick sort	10000	0.000 second	4568 KB
Shell sort	10000	0.000 second	4568 KB

b. Ukuran data 50.000

Angka			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	50000	7.362 second	4820 KB
Selection sort	50000	1.835 second	4832 KB
Insertion sort	50000	1.143 second	4832 KB
Merge sort	50000	0.017 second	5020 KB
Quick sort	50000	0.007 second	5020 KB
Shell sort	50000	0.017 second	5020 KB

c. Ukuran data 100.000

Angka			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	100000	37.861 second	5168 KB
Selection sort	100000	11.983 second	5180 KB
Insertion sort	100000	4.672 second	5180 KB
Merge sort	100000	0.016 second	5564 KB
Quick sort	100000	0.016 second	5564 KB
Shell sort	100000	0.044 second	5564 KB

d. Ukuran data 250.000

Angka			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	250000	214.064 second	6340 KB
Selection sort	250000	45.488 second	6352 KB
Insertion sort	250000	29.338 second	6352 KB
Merge sort	250000	0.078 second	7324 KB
Quick sort	250000	0.050 second	7324 KB
Shell sort	250000	0.103 second	7324 KB

e. Ukuran data 500.000

Angka			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	500000	1021.831 second	8288 KB
Selection sort	500000	283.457 second	8272 KB
Insertion sort	500000	116.710 second	8272 KB
Merge sort	500000	0.142 second	10220 KB
Quick sort	500000	0.109 second	10220 KB
Shell sort	500000	0.212 second	10220 KB

f. Ukuran data 1.000.000

Angka			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	1000000	2670.180 second	12188 KB
Selection sort	1000000	949.893 second	12124 KB
Insertion sort	1000000	677.565 second	12168 KB
Merge sort	1000000	0.628 second	112268 KB
Quick sort	1000000	0.360 second	112268 KB
Shell sort	1000000	0.617 second	112268 KB

g. Ukuran data 1.500.000

Angka			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	1500000	5927.827 second	14224 KB
Selection sort	1500000	2052.205 second	14436 KB
Insertion sort	1500000	1548.113 second	16056 KB
Merge sort	1500000	0.940 second	170480 KB
Quick sort	1500000	0.628 second	170480 KB
Shell sort	1500000	0.923 second	170480 KB

h. Ukuran data 2.000.000

Angka			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	2000000	8726.031 second	16744KB
Selection sort	2000000	4172.726 second	17012KB
Insertion sort	2000000	2670.264 second	17108 KB
Merge sort	2000000	1.292 second	227400 KB
Quick sort	2000000	1.053 second	227412 KB
Shell sort	2000000	1.402 second	227412 KB

2. Data Kata

a. Ukuran data 10.000

Kata			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	10000	1.551 second	4856 KB
Selection sort	10000	0.133 second	4868 KB
Insertion sort	10000	0.600 second	4868 KB
Merge sort	10000	0.020 second	5092 KB
Quick sort	10000	0.014 second	5092 KB
Shell sort	10000	0.003 second	5092 KB

b. Ukuran data 50.000

Kata			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	50000	42.731 second	6856 KB
Selection sort	50000	3.607 second	6868 KB
Insertion sort	50000	17.262 second	6868 KB
Merge sort	50000	0.131 second	7176 KB
Quick sort	50000	0.053 second	7176 KB
Shell sort	50000	0.087 second	7176 KB

c. Ukuran data 100.000

Kata			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	100000	259.080 second	9392 KB
Selection sort	100000	14.562 second	9404 KB
Insertion sort	100000	75.080 second	9432 KB
Merge sort	100000	0.659 second	10752 KB
Quick sort	100000	0.225 second	10752 KB
Shell sort	100000	0.390 second	10752 KB

d. Ukuran data 250.000

Kata			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	250000	1479.604 second	17000 KB
Selection sort	250000	105.404 second	17012 KB
Insertion sort	250000	465.085 second	17012 KB
Merge sort	250000	0.896 second	17680 KB
Quick sort	250000	0.310 second	17680 KB
Shell sort	250000	0.591 second	17680 KB

e. Ukuran data 500.000

Kata			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	500000	10484.885 second	22400 KB
Selection sort	500000	945.916 second	22452 KB
Insertion sort	500000	2572.632 second	22452 KB
Merge sort	500000	2.932 second	30028 KB
Quick sort	500000	1.25 second	30028 KB
Shell sort	500000	2.183 second	30028 KB

f. Ukuran data 1.000.000

Kata			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	1000000	26932.209 second	32240 KB
Selection sort	1000000	7226.729 second	32252 KB
Insertion sort	1000000	10231.567 second	34480 KB
Merge sort	1000000	3.123 second	61532 KB
Quick sort	1000000	1.466 second	61544 KB
Shell sort	1000000	2.875 second	61544 KB

g. Ukuran data 1.500.000

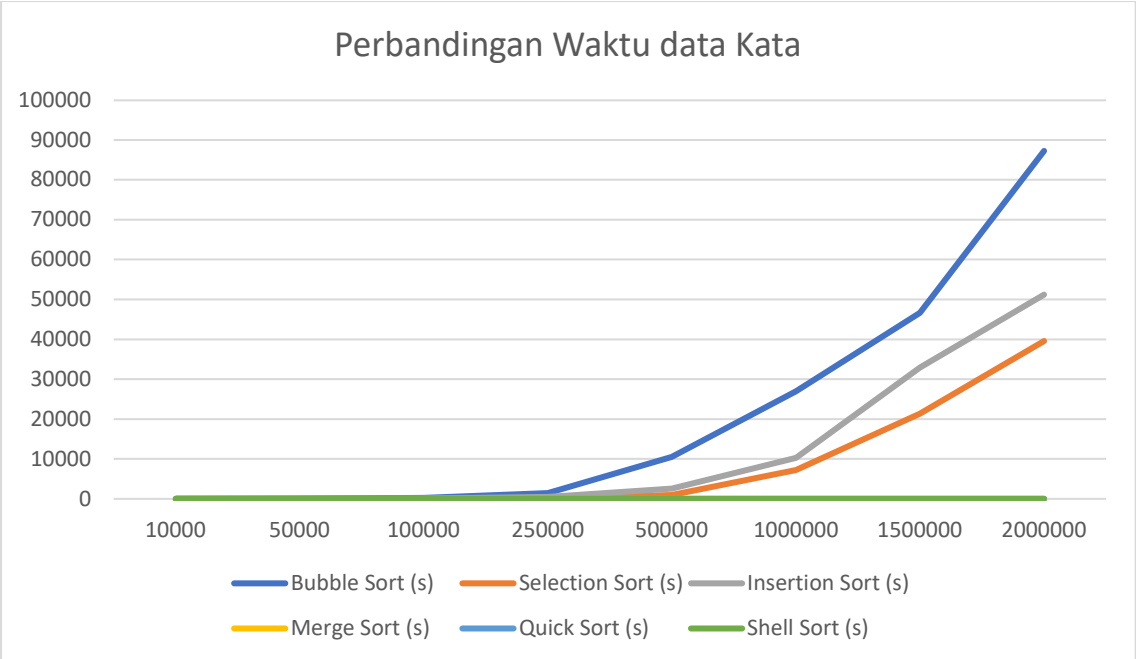
Kata			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	1500000	46615.652 second	52400 KB
Selection sort	1500000	21364.122 second	54400 KB
Insertion sort	1500000	32903.598 second	58840 KB
Merge sort	1500000	4.874 second	81140 KB
Quick sort	1500000	2.219 second	81152 KB
Shell sort	1500000	4.373 second	81152 KB

h. Ukuran data 2.000.000

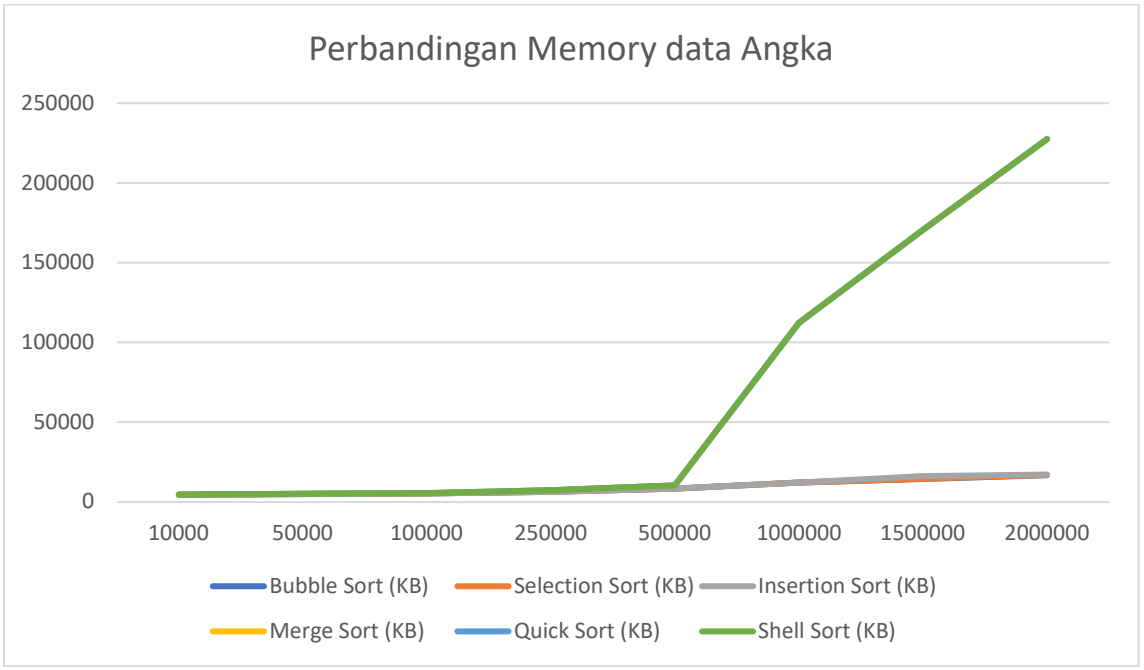
Kata			
Sorting	**Ukuran**	**Waktu**	**Memory**
Bubble sort	2000000	87306.447 second	80124 KB
Selection sort	2000000	39592.975 second	88240 KB
Insertion sort	2000000	51217.283 second	100124 KB
Merge sort	2000000	6.507 second	117844 KB
Quick sort	2000000	3.036 second	117856 KB
Shell sort	2000000	6.642 second	117856 KB

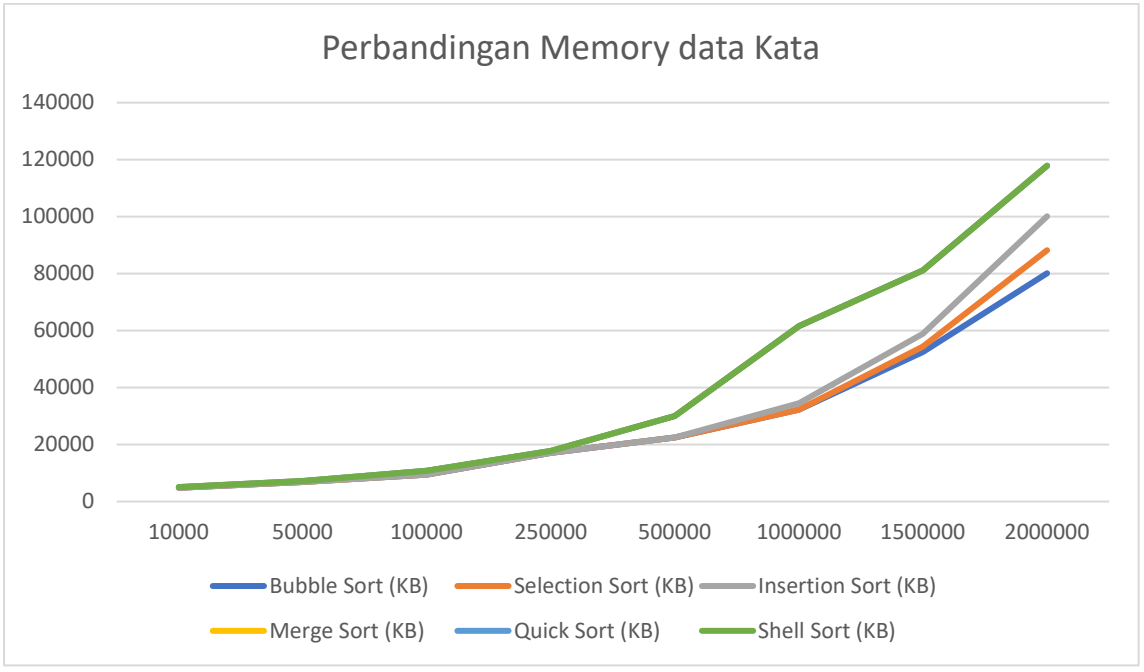
C. Grafik Perbandingan Waktu dan Memory

1. Grafik Perbandingan Waktu



2. Grafik Perbandingan Memory





Analisis

Dari tabel dan grafik yang ditampilkan dalam dokumen tersebut, kita dapat menarik beberapa analisis terkait performa algoritma sorting. Pengujian dilakukan untuk dua jenis data, yaitu angka dan kata, dengan berbagai ukuran data mulai dari 10.000 hingga 2.000.000. Berikut adalah poin-poin utama yang dapat dianalisis:

1. **Waktu Eksekusi**:
   - Algoritma seperti Bubble Sort menunjukkan waktu eksekusi yang jauh lebih lama dibandingkan algoritma lain, terutama pada ukuran data besar. Hal ini sesuai dengan kompleksitasnya, $$O(n^2)$$.
   - Merge Sort dan Quick Sort secara konsisten memberikan waktu eksekusi yang lebih cepat, menunjukkan efisiensi dari pendekatan divide and conquer yang mereka gunakan.
   - Algoritma Shell Sort memberikan hasil yang cukup baik pada ukuran data besar, dengan waktu eksekusi lebih rendah dibandingkan algoritma berbasis iteratif seperti Bubble Sort, Selection Sort, dan Insertion Sort.
2. **Penggunaan Memori**:
   - Penggunaan memori meningkat seiring dengan ukuran data, namun algoritma seperti Merge Sort dan Quick Sort memerlukan lebih banyak memori dibandingkan algoritma lainnya. Hal ini disebabkan oleh kebutuhan tambahan untuk penyimpanan saat melakukan rekursi atau penggabungan data.
   - Algoritma iteratif seperti Bubble Sort dan Selection Sort memiliki penggunaan memori yang lebih stabil, karena tidak memerlukan struktur tambahan untuk rekursi atau partisi.
3. **Perbandingan antara Data Angka dan Kata**:
   - Waktu eksekusi untuk data berupa kata cenderung lebih tinggi daripada data berupa angka, terutama untuk ukuran data besar. Hal ini disebabkan oleh proses penanganan string yang lebih kompleks dibandingkan angka.
   - Penggunaan memori untuk data berupa kata juga lebih besar, karena string membutuhkan alokasi memori yang lebih besar dibandingkan angka.

Kesimpulan

Berdasarkan tabel dan grafik, kita dapat menyimpulkan bahwa:

1. **Efisiensi waktu**:
   - Untuk aplikasi yang memerlukan sorting dengan ukuran data besar, Merge Sort dan Quick Sort merupakan pilihan terbaik karena efisiensinya yang tinggi.
   - Algoritma seperti Bubble Sort, Selection Sort, dan Insertion Sort lebih sesuai untuk data kecil atau kasus sederhana.
2. **Penggunaan memori**:
   - Algoritma iteratif lebih hemat memori dibandingkan algoritma berbasis rekursi, menjadikannya pilihan yang baik untuk perangkat dengan keterbatasan memori.

3. **Data yang digunakan**:
    - Pemrosesan data berupa kata lebih kompleks dibandingkan angka, sehingga algoritma yang efisien sangat diperlukan untuk mengurangi waktu eksekusi dan penggunaan memori.