

## **Laporan Tugas Desain dan Analisis Algoritma**

Untuk Memenuhi Tugas Desain dan Analisis Algoritma



Oleh:

Jimly Asidiq Anwar - 4522210018

Akhmad Rizaldy - 4522210050

Michael Danu Ekklesiya - 4522210056

Muhamad Farhan - 4522210057

Dosen:

**Dyah Sulistyowati Rahayu**

**S1-Teknik Informatika**

**Fakultas Teknik Universitas Pancasila**

**2022/2023**

## Merge Sort

Merge Sort adalah salah satu algoritma pengurutan (sorting) yang digunakan untuk mengurutkan sebuah himpunan data, seperti array atau daftar, secara efisien. Algoritma ini termasuk dalam kelompok algoritma pengurutan yang menggunakan pendekatan "divide and conquer," yang berarti membagi masalah besar menjadi masalah yang lebih kecil, mengurutkan masing-masing bagian, dan kemudian menggabungkannya menjadi satu himpunan yang terurut.

## Source Code

```
mergesort.cpp
1  #include <iostream>
2  using namespace std;
3
4  void merge(int arr[], int left, int mid, int right) {
5      int n1 = mid - left + 1;
6      int n2 = right - mid;
7
8      int L[n1], R[n2];
9
10     for (int i = 0; i < n1; i++) {
11         L[i] = arr[left + i];
12     }
13     for (int j = 0; j < n2; j++) {
14         R[j] = arr[mid + 1 + j];
15     }
16
17     int i = 0;
18     int j = 0;
19     int k = left;
20
21     while (i < n1 && j < n2) {
22         if (L[i] <= R[j]) {
23             arr[k] = L[i];
24             i++;
25         }
26         else {
27             arr[k] = R[j];
28             j++;
29         }
30         k++;
31     }
32
33     while (i < n1) {
34         arr[k] = L[i];
35         i++;
36         k++;
37     }
38
39     while (j < n2) {
40         arr[k] = R[j];
41         j++;
42         k++;
43     }
44 }
```

C++ source file

```
39     while (j < n2) {
40         arr[k] = R[j];
41         j++;
42         k++;
43     }
44 }
45
46 void mergeSort(int arr[], int left, int right) {
47     if (left < right) {
48         int mid = left + (right - left) / 2;
49         mergeSort(arr, left, mid);
50         mergeSort(arr, mid + 1, right);
51         merge(arr, left, mid, right);
52     }
53 }
54
55 void printArray(int arr[], int size) {
56     for (int i = 0; i < size; i++) {
57         cout << arr[i] << " ";
58     }
59     cout << endl;
60 }
61
62 int main() {
63     int arr[] = { 11, 6, 3, 24, 46, 22, 7 };
64     int arrSize = sizeof(arr) / sizeof(arr[0]);
65
66     cout << "Array sebelum diurutkan:" << endl;
67     printArray(arr, arrSize);
68
69     mergeSort(arr, 0, arrSize - 1);
70
71     cout << "Array setelah diurutkan:" << endl;
72     printArray(arr, arrSize);
73
74     return 0;
75 }
```

C++ source file

## Pseudocode

```
void merge
FOR (i = 0; i < n1; i++)
    L[i] = arr[left + i]

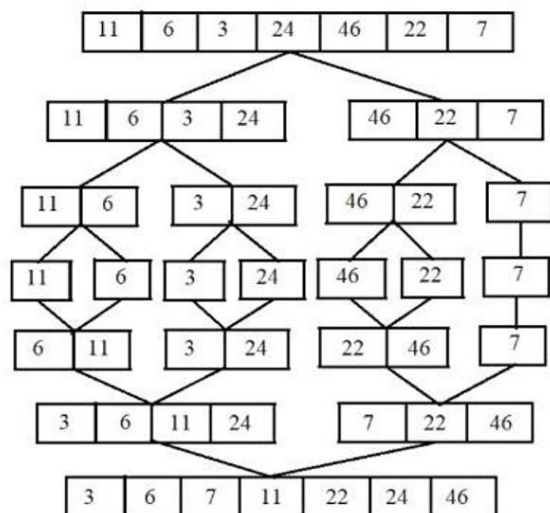
FOR (j = 0; j < n2; j++)
    R[j] = arr[mid + 1 + j]

WHILE (i < n1 && j < n2)
    IF (L[i] <= R[j])
        arr[k] = L[i]
        i++
    ELSE
        arr[k] = R[j]
        j++
    k++

WHILE (i < n1)
    arr[k] = L[i]
    i++
    k++

WHILE (j < n2)
    arr[k] = R[j]
    j++
    k++
```

## Ilustrasi



```
Administrator: C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.22621.2283]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Akhmad Rizaldy\Documents\kampus\semester3\DAA>g++ mergesort.cpp

C:\Users\Akhmad Rizaldy\Documents\kampus\semester3\DAA>a
Array sebelum diurutkan:
11 6 3 24 46 22 7
Array setelah diurutkan:
3 6 7 11 22 24 46

C:\Users\Akhmad Rizaldy\Documents\kampus\semester3\DAA>
```

## Kompleksitasnya

Kompleksitas waktu Merge Sort dapat bervariasi tergantung pada kondisi masukan (input data). Di bawah ini adalah penjelasan mengenai kompleksitas waktu Merge Sort untuk worst-case, average-case, dan best-case:

- **Worst-case Complexity (Kompleksitas Kasus Terburuk):** Worst-case terjadi ketika algoritma Merge Sort harus membagi array berulang kali hingga elemen-elemen tunggal terbentuk dan kemudian menggabungkannya kembali. Dalam kasus terburuk, Merge Sort selalu membagi array menjadi dua bagian yang sama ukurannya hingga mencapai elemen-elemen tunggal. Sebagai contoh, jika ada  $n$  elemen dalam array, algoritma ini akan melakukan  $\log_2(n)$  tahap pembagian. Di setiap tahap pembagian, semua  $n$  elemen perlu digabungkan. Oleh karena itu, kompleksitas waktu worst-case Merge Sort adalah  $O(n \log n)$ .
- **Average-case Complexity (Kompleksitas Kasus Rata-rata):** Kompleksitas rata-rata Merge Sort juga  $O(n \log n)$ . Meskipun kompleksitas ini sama dengan kompleksitas worst-case, Merge Sort sering dianggap lebih konsisten dan dapat diandalkan dalam berbagai kasus. Ini karena algoritma ini selalu membagi array menjadi dua bagian yang sama ukurannya, menghasilkan kinerja yang konsisten di semua kondisi masukan.
- **Best-case Complexity (Kompleksitas Kasus Terbaik):** Kompleksitas best-case Merge Sort juga  $O(n \log n)$ . Hal ini mungkin mengejutkan, karena sebagian besar algoritma pengurutan memiliki kompleksitas waktu terbaik yang lebih baik dalam kasus terbaik. Namun, Merge Sort selalu membagi dan menggabungkan array dengan cara yang sama, tanpa memperhatikan apakah array tersebut sudah terurut atau tidak. Oleh karena itu, kompleksitas best-case Merge Sort adalah  $O(n \log n)$ , meskipun dalam prakteknya tidak memerlukan banyak pekerjaan untuk mengurutkan array yang sudah terurut.

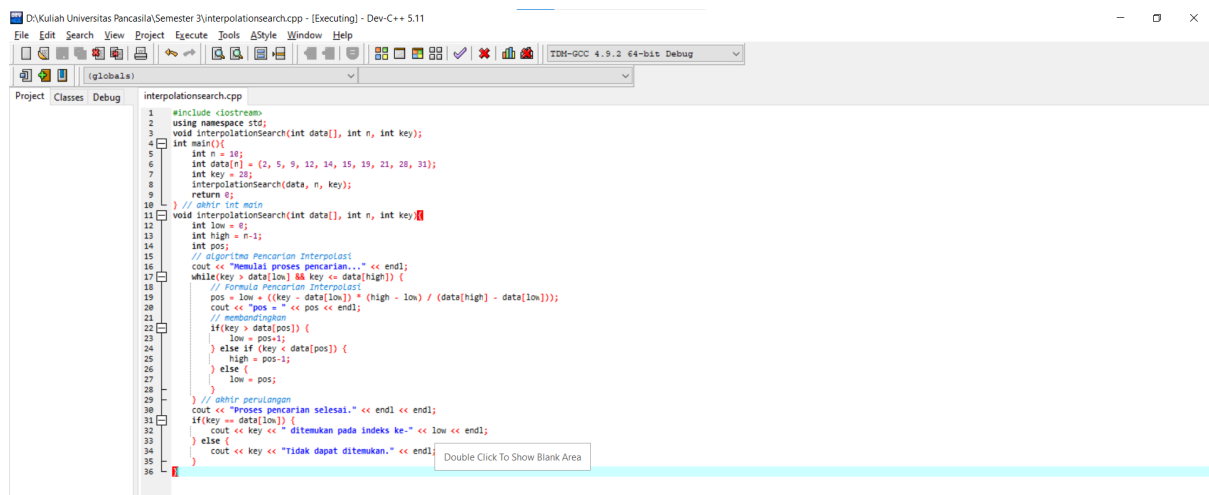
Jadi, Merge Sort memiliki kompleksitas waktu yang cukup baik dan konsisten, sehingga sering digunakan dalam aplikasi di mana kinerja yang dapat diandalkan penting, terutama jika Anda tidak dapat memprediksi dengan pasti kondisi masukan.

## Interpolation Search

Interpolation search adalah algoritma pencarian yang digunakan untuk mencari elemen tertentu dalam himpunan data yang telah diurutkan. **Interpolation Search** adalah pengembangan dari binary search, persamaan keduanya yaitu sama sama digunakan untuk mencari suatu nilai pada data yang telah terurut, sedangkan perbedaan keduanya terletak pada cara kerjanya. Binary search mencari data dengan membagi array secara terus menerus, sedangkan pada binary search interpolation ini mencari data dengan menggunakan Formula, yaitu sebagai berikut:

$$\text{pos} = \text{low} + \frac{(\text{key} - \text{data}[\text{low}]) * (\text{high} - \text{low})}{(\text{data}[\text{high}] - \text{data}[\text{low}])}$$

## Source Code



```
1 #include <iostream>
2 using namespace std;
3 void interpolationSearch(int data[], int n, int key);
4 int main()
5 {
6     int n = 10;
7     int data[n] = {2, 5, 9, 12, 14, 15, 19, 21, 28, 31};
8     int key = 28;
9     interpolationSearch(data, n, key);
10    return 0;
11 }
12 // algoritma pencarian Interpolasi
13 void interpolationSearch(int data[], int n, int key)
14 {
15     int low = 0;
16     int high = n-1;
17     int pos;
18     cout << "Memulai proses pencarian..." << endl;
19     while(key > data[low] && key <= data[high]) {
20         pos = low + ((key - data[low]) * (high - low) / (data[high] - data[low]));
21         cout << "pos = " << pos << endl;
22         // membandingkan
23         if(key > data[pos]) {
24             low = pos+1;
25         } else if (key < data[pos]) {
26             high = pos-1;
27         } else {
28             low = pos;
29         }
30     }
31     // akhir perulangan
32     cout << "Proses pencarian selesai." << endl << endl;
33     if(key == data[low]) {
34         cout << key << " ditemukan pada indeks ke-" << low << endl;
35     } else {
36         cout << key << "Tidak dapat ditemukan." << endl;
37     }
```

## Pseudocode

While (key > data[low]) && (key <= data[high]):

pos = low + ((key - data[low]) \* (high - low) / (data[high] - data[low]))

IF key > data[pos]

then pos + 1

ELSE IF key < data[pos]

then pos - 1

ELSE low = pos

## Ilustrasi

# Interpolation Search

Cari 28

12	14	15	19	21	28	31
----	----	----	----	----	----	----

$$\text{pos} = 0 + \frac{(28 - 12) * (6 - 0)}{(31 - 12)}$$

pos = 5,05

28

indeks ke-5

```
D:\Kuliah Universitas Pancasila\Semester 3\interpolationsearch.exe
Memulai proses pencarian...
pos = 8
Proses pencarian selesai.
28 ditemukan pada indeks ke-8
-----
Process exited after 0.05884 seconds with return value 0
Press any key to continue . . .
```

## Kompleksitasnya

Kompleksitas waktu dari algoritma Interpolation Search biasanya dinyatakan dalam notasi Big O (O). Untuk memahami kompleksitas waktu Interpolation Search, pertimbangkan beberapa kasus yang berbeda:

- **Kasus Terburuk** : Pada kasus terburuk, kompleksitas Interpolation Search adalah  $O(n)$ , di mana  $n$  adalah jumlah elemen dalam himpunan data. Ini terjadi ketika elemen yang dicari terletak di salah satu ujung himpunan data dan algoritma harus mengunjungi atau memeriksa setiap elemen sebelum menemukan elemen yang dicari
- **Kasus Rata-rata** : Jika data terdistribusi secara merata (artinya, setiap elemen dalam himpunan data mendekati nilai yang sama), Interpolation Search dapat memiliki kompleksitas waktu yang lebih cepat daripada  $O(n)$ . Namun, kompleksitas rata-ratanya masih lebih tinggi daripada logaritmik. Dalam beberapa kasus, Interpolation Search dapat memiliki kinerja yang serupa dengan binary search ( $O(\log n)$ ), tetapi dalam kasus terbaiknya, dapat lebih cepat daripada binary search.

- **Kasus Terbaik** : Dalam kasus terbaik, ketika elemen yang dicari berada tepat di tengah-tengah himpunan data, Interpolation Search memiliki kompleksitas waktu  $O(1)$ , yang berarti pencarian dapat diselesaikan dengan satu langkah.

Interpolation Search memiliki kinerja terbaik ketika data terdistribusi secara merata. Jika data memiliki pola distribusi yang tidak merata, seperti banyak elemen yang mendekati ujung himpunan data, Interpolation Search mungkin tidak memberikan kinerja yang lebih baik daripada binary search.

Jadi, secara umum, kompleksitas waktu Interpolation Search bisa menjadi  $O(n)$  dalam kasus terburuk, tetapi jika data terdistribusi secara merata, maka kinerjanya dapat mendekati  $O(\log n)$  atau bahkan lebih baik dalam kasus terbaiknya.