STUDI PERBANDINGAN ALGORITMA PENCARIAN BINARY, JUMP, INTERPOLATION, DAN FIBONACCI : EFISIENSI MEMORI DAN WAKTU EKSEKUSI

Said Fachri Ariza, Abdul Majid, Muhammad Hamdi Yahya,
Inggil Himawan, Surya Ardhiartha P.U., Imam Prayogo Pujiono
Informatika, Universitas Negeri Islam K.H. Abdurrahman Wahid Pekalongan
Jalan Pahlawan Rowolaku Kajen Kabupaten Pekalongan
said.fachri.ariza24023@mhs.uingusdur.ac.id

ABSTRAK

Penelitian ini membahas efisiensi empat algoritma pencarian Binary Search, Jump Search, Interpolation Search, dan Fibonacci Search dalam aspek waktu eksekusi dan penggunaan memori. Masalah utama yang diangkat adalah perlunya identifikasi algoritma pencarian yang paling optimal pada data terurut dengan volume berbeda. Tujuan penelitian ini adalah membandingkan performa keempat algoritma dalam pengolahan data numerik acak yang telah diurutkan. Metode yang digunakan melibatkan pengujian algoritma menggunakan bahasa C++ terhadap tiga skala dataset (100, 1.000, dan 10.000 elemen), masing-masing diuji sebanyak tiga kali untuk menjaga konsistensi hasil. Hasil penelitian menunjukkan bahwa Interpolation Search paling efisien dari segi waktu (300–400 ns) dan memori (24–34,67 byte), terutama saat data terdistribusi merata. Binary Search konsisten dengan performa baik di berbagai kondisi. Sebaliknya, Jump Search mencatat waktu eksekusi paling lama (3.233,33–3.866,67 ns), sedangkan Fibonacci Search mencatat penggunaan memori tertinggi (61,33–74,67 byte). Kesimpulan menunjukkan bahwa pemilihan algoritma pencarian harus disesuaikan dengan distribusi dan ukuran data untuk mencapai efisiensi optimal.

Kata kunci : *Binary Search, Jump Search, Interpolation Search, Fibonacci Search, waktu komputasi, penggunaan memori.*

1. PENDAHULUAN

Pencarian merupakan aktivitas yang lazim dilakukan dalam kehidupan sehari-hari [1]. Dalam beberapa situasi, pencarian dilakukan semata-mata untuk mengetahui apakah suatu data terdapat dalam himpunan data tertentu. Namun, pada kesempatan lain, tujuan pencarian mungkin untuk mengetahui posisi atau letak data tersebut dalam himpunan data yang dimaksud [2].

Berbagai algoritma telah dikembangkan guna menunjang proses pencarian, masing-masing dengan pendekatan serta karakteristik yang berbeda [3]. Setiap algoritma memiliki strategi unik dalam menelusuri elemen-elemen dalam himpunan data. Sebagian algoritma dirancang untuk mencapai kecepatan pencarian optimal pada data yang telah terurut, sedangkan algoritma lainnya lebih fleksibel terhadap variasi distribusi data [4]. Oleh karena itu, memahami cara kerja masing-masing algoritma secara umum menjadi dasar penting dalam melakukan perbandingan performa.

Penelitian ini mengkaji perbandingan keempat algoritma tersebut. Proses pengujian dilakukan dengan menggunakan data acak yang telah diurutkan, dalam tiga ukuran dataset yang berbeda (Small, Medium, Large). Masing-masing algoritma dievaluasi untuk menilai penggunaan waktu dan memori pada berbagai ukuran input data, hasilnya memungkinkan untuk mengidentifikasi algoritma yang paling efisien dalam hal pemanfaatan memori dan waktu komputasi pada dataset berukuran kecil, menengah, dan besar.

2. TINJAUAN PUSTAKA

2.1. Binary Search

Algoritma *binary search* adalah salah satu metode pencarian yang efisien, terutama saat digunakan pada kumpulan data dalam jumlah besar yang telah terurut. *Binary search* bekerja dengan cara mengurangi jumlah operasi perbandingan antara elemen yang dicari dengan elemen-elemen yang terdapat dalam array, melalui proses pembagian ruang pencarian secara berulang hingga data ditemukan. Pendekatan ini membuat *binary search* memiliki beban komputasi yang lebih ringan dibandingkan dengan beberapa algoritma pencarian lainnya [5].

Secara umum, algoritma ini dimulai dengan membandingkan nilai target terhadap elemen yang terletak di tengah array. Jika nilai tersebut cocok dengan target, maka posisi elemen tersebut dikembalikan sebagai hasil pencarian. Jika nilai target lebih kecil dari elemen tengah, maka pencarian dilanjutkan pada bagian kiri (setengah bawah) array. Sebaliknya, jika nilai target lebih besar, pencarian diteruskan pada bagian kanan (setengah atas) array. Dengan pendekatan ini, setiap iterasi mengeliminasi separuh ruang pencarian yang tidak mungkin mengandung target, sehingga pencarian menjadi lebih cepat dan efisien. Kompleksitas waktu *binary search* pada kasus rata-rata dan terbaik adalah O(log n), sedangkan pada kasus terburuk tetap O(log n). Algoritma ini juga hemat memori karena hanya memerlukan ruang tambahan konstan sebesar O(1) dengan pendekatan iteratif [6].

2.2. Jump Search

Algoritma *jump search* merupakan metode pencarian yang dirancang untuk digunakan pada kumpulan data yang telah terurut. Berbeda dengan pencarian linier yang memeriksa satu per satu elemen dalam array, *jump search* memanfaatkan langkah loncatan tetap untuk mempercepat pencarian. Pendekatan ini bertujuan untuk mengurangi jumlah perbandingan yang dilakukan dengan melompati sejumlah elemen tertentu, hingga mencapai elemen yang lebih besar atau sama dengan nilai target [7].

Langkah pencarian dilakukan dengan melompat sebesar m = √n, di mana n adalah jumlah total elemen dalam array. Setelah ditemukan batas atas atau nilai yang lebih besar dari target, algoritma kemudian melakukan pencarian linier mundur dalam rentang m sebelumnya hingga menemukan target [8]. Pendekatan ini membuat algoritma *jump search* cukup efisien dalam situasi tertentu. Kompleksitas waktu rata-rata dan terburuk dari algoritma ini adalah O(√n), serta memerlukan ruang tambahan sebesar O(1) karena dilakukan secara iteratif [9].

2.3. Interpolation Search

Pencarian interpolasi (*interpolation search*) adalah metode pencarian yang dilakukan dengan menentukan posisi data yang ingin ditemukan. *Interpolation search* merupakan algoritma yang mencari nilai kunci dalam sebuah array yang sudah diurutkan berdasarkan nilai kunci tersebut. Cara kerjanya menyerupai metode manusia saat mencari nama tertentu dalam buku telepon. Pada setiap langkah, algoritma memperkirakan lokasi item yang dicari berdasarkan nilai kunci pada batas ruang pencarian serta nilai kunci target, umumnya melalui interpolasi linier. Nilai kunci yang ditemukan di posisi estimasi akan dibandingkan dengan nilai kunci yang dicari. Jika tidak cocok, maka berdasarkan hasil perbandingan, ruang pencarian yang tersisa akan dipersempit ke bagian sebelum atau sesudah posisi estimasi. Metode ini hanya efektif jika perhitungan selisih antar nilai kunci masih masuk akal [10].

Dalam hal efisiensi, kompleksitas waktu pencarian interpolasi bergantung pada distribusi data dalam array. Jika data terdistribusi secara seragam, algoritma ini memiliki kompleksitas waktu rata-rata sebesar O(log log n), menjadikannya lebih efisien dibandingkan pencarian biner dalam kasus tertentu. Namun, dalam kasus terburuk, seperti ketika nilai-nilai kunci meningkat secara eksponensial atau distribusi data sangat tidak merata, kompleksitas waktu dapat memburuk menjadi O(n) [11].

2.4. Fibonacci Search

Algoritma *Fibonacci Search* melakukan penyempitan interval pencarian secara bertahap dengan menggunakan jumlah iterasi yang telah ditentukan. Pendekatan ini menjamin bahwa posisi pencarian terbaik akan ditemukan dalam rentang fokus akhir. Algoritma ini didasarkan pada deret bilangan Fibonacci [12] di mana setiap bilangan dalam deret (Fn) merupakan hasil penjumlahan dua bilangan sebelumnya, kecuali dua bilangan pertama yang bernilai satu. Hal ini dapat dituliskan dengan persamaan:

$$f_n = f_{n-1} + f_{n-2}$$
$$f_0 = f_1 = f_1 \quad (1)$$

Dalam penerapannya, diasumsikan bahwa L_0 merupakan panjang interval pencarian ($L_0 = b - a$), dengan a sebagai titik awal interval, b sebagai titik akhir, dan n adalah jumlah total iterasi yang akan diambil selama proses pencarian berlangsung [13].

Dalam hal efisiensi, kompleksitas waktu dari algoritma *Fibonacci Search* adalah O(log n). Hal ini karena setiap iterasi dari algoritma ini memerlukan waktu konstan, dan jumlah iterasi yang diperlukan berbanding logaritmik dengan ukuran array. kompleksitas waktu total dari algoritma ini adalah O(log n).

2.5. Penelitian Terdahulu

Terdapat berbagai jenis algoritma pencarian yang digunakan untuk menemukan suatu elemen dari sekumpulan elemen. Berdasarkan [10] algoritma interpolasi merupakan algoritma yang lebih efisien dibandingkan algoritma *jump search* dan *binary search*. Pada penelitian [8] peforma *interpolation search* pada efisiensi waktu memberikan hasil terbaik disbanding algoritma lainnya.

Beberapa studi eksperimental membandingkan kinerja waktu eksekusi berbagai algoritma ini. [15] melakukan pengujian Binary, Jump, dan Interpolation Search di tiga bahasa (C++, Java, Python) dan menemukan bahwa *Interpolation Search* memiliki waktu eksekusi terpendek pada ukuran data sangat besar, diikuti oleh *Binary Search*, lalu *Jump Search*. Selain itu mereka mencatat bahwa C++ secara konsisten lebih cepat dibanding Java dan Python dalam implementasi ini. Hasil ini menunjukkan urutan efisiensi waktu: *Interpolation* > *Binary* > *Jump*. [16] meneliti algoritma *Binary* vs Interpolation dalam struktur data B-tree. Dalam operasi pencarian kunci primer, penggunaan Interpolation Search mengurangi waktu pencarian hingga 3,5 kali lebih cepat dibanding *Binary Search*. [9] mengkaji berbagai algoritma pencarian pada data terurut maupun tak terurut. Mereka melaporkan bahwa *Jump Search* unggul dibandingkan Linear dan Binary Search baik dari sisi waktu eksekusi maupun kompleksitas ruang (memori).

3. METODE PENELITIAN

3.1. Pembuatan Dataset

Pada tahap ini, dilakukan penyusunan dataset yang akan dimanfaatkan sebagai data uji guna menilai kinerja berbagai algoritma pengurutan. Dataset tersebut dibentuk menggunakan struktur array yang diisi dengan angka-angka acak. Nilai-nilai ini berupa bilangan bulat (integer) yang dihasilkan secara acak dalam rentang 1 hingga 1.000.000. Pemilihan rentang

angka yang luas ini bertujuan untuk menghadirkan variasi nilai yang tinggi, sehingga mampu mencerminkan kondisi data dunia nyata yang kompleks dan tidak tersusun rapi [17][18].

Penelitian ini merancang dataset dalam tiga kategori skala, yaitu kecil, sedang, dan besar. Tujuannya adalah untuk mengkaji serta membandingkan performa algoritma dalam menghadapi berbagai jumlah data, mengingat efektivitas dan efisiensi suatu algoritma kerap dipengaruhi oleh ukuran data yang ditangani. Adapun pembagian dataset berdasarkan skala sebagai berikut:

- *ArraySmall* : terdiri dari 100 elemen, mewakili data skala kecil.
- *ArrayMedium* : terdiri dari 1.000 elemen, mewakili data skala sedang.
- *ArrayLarge* : terdiri dari 10.000 elemen, mewakili data skala besar.

Untuk membentuk ketiga jenis dataset yang digunakan dalam pengujian, pertama-tama dibuat sebuah array induk yang berisi bilangan bulat acak. Array ini berfungsi sebagai sumber data utama untuk seluruh proses pengujian. Dataset kemudian dibentuk dengan mengambil elemen-elemen dari array induk secara berurutan: 100 elemen pertama untuk membentuk *ArraySmall*, 1.000 elemen pertama untuk *ArrayMedium*, dan keseluruhan 10.000 elemen digunakan sebagai *ArrayLarge*. Pendekatan ini dipilih untuk menjaga konsistensi data pada tiap skenario pengujian, sehingga perbandingan kinerja antar algoritma pencarian dapat dilakukan secara adil dan objektif. Dengan cara ini, setiap subset dataset berasal dari distribusi data yang sama, hanya berbeda dalam jumlah elemen yang digunakan sesuai dengan skala pengujian.

Dalam program ini, data yang digunakan untuk pengujian algoritma pencarian terlebih dahulu diurutkan menggunakan fungsi sort() dari pustaka standar C++. Fungsi ini secara default mengurutkan elemen-elemen dalam urutan menaik (ascending). Proses pengurutan dilakukan setelah data acak sebanyak jumlah n elemen, dengan tujuan untuk memenuhi prasyarat algoritma-algoritma pencarian seperti Binary Search, Interpolation Search, dan *Fibonacci Search* yang membutuhkan data dalam kondisi terurut agar dapat berfungsi secara optimal. Sorting ini dilakukan menggunakan algoritma internal STL, yang secara efisien menggabungkan metode QuickSort, HeapSort, dan InsertionSort.

```
int main() {
    srand(time(0));
    const int SIZE = n;
    vector<int> data(SIZE);
    for (int i = 0; i < SIZE; ++i)
        data[i] = rand() % 1000000;
    sort(data.begin(), data.end());
    int target = data[rand() % SIZE]; // Random target from the
    data
    cout << "Sorted Array (first 10 elements): ";
    for (int i = 0; i < min(10, SIZE); ++i)
        cout << data[i] << " ";
    cout << "...\n\nTarget to Find: " << target << "\n\n";
```

huruf n akan diganti berdasarkan ukuran array (Small, Medium, Large).

3.2. Tahap Pengujian Algoritma

Setelah data berhasil diurutkan, langkah berikutnya adalah menerapkan masing-masing algoritma pencarian terhadap tiga dataset (kecil, sedang, dan besar). Setiap algoritma diimplementasikan dalam bentuk kode C++ dan dijabarkan dalam bentuk *pseudo code* berikut ini:

3.3. Binary Search

```
int binarySearch(const vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
```

Fungsi BinarySearch(array, target) digunakan untuk mencari posisi dari suatu elemen (target) dalam array yang sudah terurut secara menaik (ascending). Proses pencarian dilakukan dengan membandingkan nilai target dengan elemen di tengah-tengah array, kemudian secara bertahap mempersempit ruang pencarian berdasarkan hasil perbandingan tersebut. Fungsi dimulai dengan mengatur indeks batas kiri (left) ke 0 dan batas kanan (right) ke panjang array dikurangi satu. Selama indeks kiri masih lebih kecil atau sama dengan indeks kanan, program akan terus mencari.

Di dalam perulangan, indeks tengah (mid) dihitung dengan rumus left + (right - left) / 2 untuk menghindari kemungkinan overflow pada beberapa bahasa pemrograman. Jika elemen pada indeks tengah sama dengan target, maka indeks tersebut langsung dikembalikan sebagai hasil pencarian. Jika nilai tengah lebih kecil dari target, berarti target kemungkinan berada di bagian kanan array, sehingga batas kiri diperbarui menjadi mid + 1. Sebaliknya, jika nilai tengah lebih besar dari target, maka pencarian dilanjutkan di bagian kiri array dengan memperbarui batas kanan menjadi mid - 1. Jika perulangan selesai dan elemen tidak ditemukan, maka fungsi akan mengembalikan -1, yang berarti target tidak ada dalam array.

```
int jumpSearch(const vector<int>& arr, int target) {
    int n = arr.size();
    int step = sqrt(n), prev = 0;
    int start = max(0, prev - step);
    for (int i = start; i <= min(prev, n - 1); ++i) {
        memoryAccessed += sizeof(arr[i]);
        if (arr[i] == target) return i;
    }
    return -1;
}
```

Fungsi *JumpSearch*(array, target) digunakan untuk mencari elemen target dalam array yang sudah terurut secara menaik. Metode ini menggabungkan pendekatan pencarian cepat seperti *Binary Search*, tetapi dengan langkah melompat (*jump*) tetap yang biasanya sebesar akar dari panjang array, sehingga dinamakan *Jump Search*. Langkah awalnya adalah menentukan panjang array (n) dan ukuran lompatan (*step*) yang dihitung sebagai akar kuadrat dari n. Kemudian, indeks awal pencarian (prev) diatur ke 0.

Proses pencarian pertama-tama dilakukan dengan melompat-lompat dari satu posisi ke posisi berikutnya dengan jarak step, selama nilai elemen pada indeks tersebut masih lebih kecil dari target dan belum melewati batas akhir array. Tujuannya adalah menemukan blok di mana kemungkinan target berada, yaitu antara prev - step dan prev. Setelah blok tersebut ditemukan, pencarian dilanjutkan secara linier dari indeks start (yaitu prev - step) hingga indeks end (yaitu nilai terkecil antara prev dan indeks terakhir array) untuk mencari elemen yang sesuai.

Jika selama pencarian linier ditemukan elemen yang sama dengan target, maka indeks tersebut dikembalikan. Jika tidak ditemukan hingga akhir blok, maka fungsi akan mengembalikan -1, yang menandakan bahwa elemen target tidak ada dalam array.

```
int interpolationSearch(const vector<int>& arr, int target{
   int left = 0, right = arr.size() - 1;
     while (left <= right && target >= arr[left] && target <=
     arr[right]) {
      if (left == right) {
          if (arr[left] == target) return left;
          return -1;
      }
      int pos = left + ((target - arr[left]) * (right - left)) / (arr[right]
      - arr[left]);
      if (arr[pos] == target) return pos;
      if (arr[pos] < target) left = pos + 1;
      else right = pos - 1;
   }
   return -1;
```

Fungsi InterpolationSearch(array, target) digunakan untuk mencari nilai tertentu (target) di dalam array yang sudah terurut secara menaik. Berbeda dengan *Binary Search* yang selalu memeriksa elemen di tengah, *Interpolation Search* memperkirakan posisi elemen berdasarkan distribusi nilai dalam array, sehingga bisa lebih efisien jika data terdistribusi secara merata.

Fungsi dimulai dengan mengatur indeks left ke 0 dan right ke indeks terakhir array. Selama nilai target berada dalam rentang nilai di antara array[left] dan array[*right*], dan selama left masih lebih kecil atau sama dengan *right*, pencarian akan terus dilakukan. Jika left dan right sama, artinya hanya tersisa satu elemen, maka cukup dibandingkan apakah elemen itu sama dengan target. Jika sama, kembalikan indeksnya, jika tidak, kembalikan -1.

Inti dari metode ini adalah perhitungan posisi perkiraan (pos) menggunakan rumus interpolasi, yaitu:

int pos = left + ((target - arr[left]) * (right - left)) / (arr[right] - arr[left]);

Rumus ini mencoba memprediksi kira-kira di mana letak target dalam array berdasarkan perbandingan nilai, bukan hanya posisi indeks.

Setelah pos dihitung, jika array[pos] sama dengan target, maka pencarian selesai dan indeks pos dikembalikan. Jika array[pos] lebih kecil dari target, berarti target ada di sebelah kanan, sehingga left diperbarui ke pos + 1. Jika array[pos] lebih besar dari target, maka target ada di sebelah kiri dan right diperbarui ke pos - 1.

Jika pencarian selesai tanpa menemukan target, maka fungsi akan mengembalikan -1 sebagai tanda bahwa nilai tidak ditemukan dalam array.

```
int fibonacciSearch(const vector<int>& arr, int target) {
   int n = arr.size();
      int fibMMm2 = 0, fibMMm1 = 1, fibM = fibMMm1 +
     fibMMm2;
   while (fibM < n) {
      fibMMm2 = fibMMm1;
      fibMMm1 = fibM;
      fibM = fibMMm1 + fibMMm2;
   }
   int offset = -1;
   while (fibM > 1) {
      int i = min(offset + fibMMm2, n - 1);
      if (arr[i] < target) {
         fibM = fibMMm1;
         fibMMm1 = fibMMm2;
         fibMMm2 = fibM - fibMMm1;
         offset = i;
      } else if (arr[i] > target) {
         fibM = fibMMm2;
         fibMMm1 = fibMMm1 - fibMMm2;
         fibMMm2 = fibM - fibMMm1;
      } else {
         return i;
      }
   }
   if (fibMMm1 && offset + 1 < n) {
      if (arr[offset + 1] == target) return offset + 1;
   }
   return -1;
```

Fungsi FibonacciSearch(array, target) digunakan untuk mencari elemen tertentu (target) dalam array yang sudah terurut secara menaik, dengan memanfaatkan bilangan Fibonacci untuk menentukan posisi pemeriksaan dalam array. Metode ini merupakan alternatif dari *Binary Search* dan cocok digunakan ketika akses data secara berurutan lebih cepat daripada akses acak.

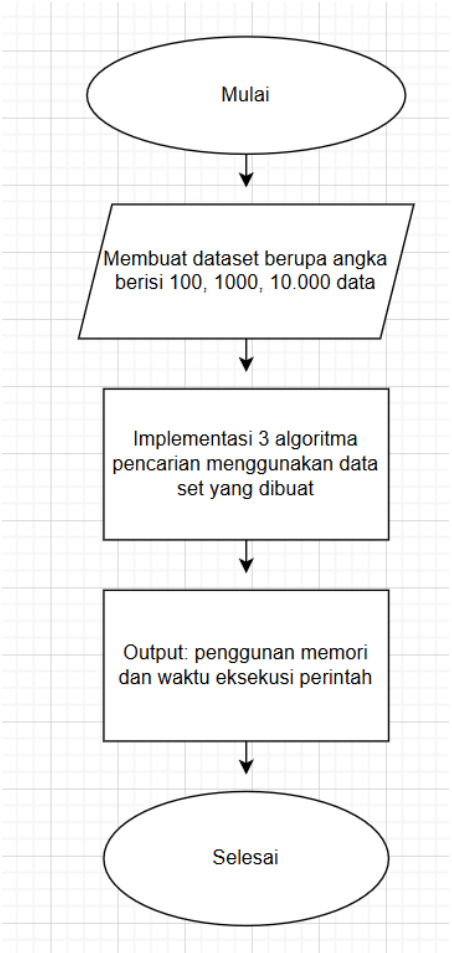
Langkah awal fungsi ini adalah menghitung bilangan Fibonacci terkecil (fibM) yang lebih besar atau sama dengan panjang array (n). Ini dilakukan dengan menjumlahkan dua bilangan Fibonacci sebelumnya, yaitu fibMMm1 dan fibMMm2, yang masing-masing diinisialisasi ke 1 dan 0. Proses ini

terus berlangsung sampai fibM lebih besar dari atau sama dengan n.

Setelah itu, pencarian dimulai dengan menggunakan offset, yang awalnya bernilai -1, sebagai penanda batas bawah dari blok pencarian. Selama fibM lebih besar dari 1, algoritma menghitung indeks yang akan diperiksa (i) sebagai nilai terkecil antara offset + fibMMm2 dan indeks terakhir array. Kemudian, nilai pada array[i] dibandingkan dengan target.

Jika nilai tersebut lebih kecil dari target, maka berarti pencarian dilanjutkan ke bagian kanan array. Dalam hal ini, bilangan Fibonacci diperbarui dengan menurunkan satu tingkat (fibM → fibMMm1, dan seterusnya), dan offset diperbarui ke i. Jika nilai lebih besar dari target, maka pencarian pindah ke kiri, dan bilangan Fibonacci juga diturunkan tetapi dengan cara yang berbeda. Jika nilai pada array[i] sama dengan target, maka indeks tersebut dikembalikan sebagai hasil pencarian.

Jika setelah proses tersebut fibMMm1 bernilai 1 dan masih ada satu elemen yang belum diperiksa, yaitu pada posisi offset + 1, maka elemen tersebut dibandingkan dengan target. Jika cocok, indeksnya dikembalikan. Jika tidak ditemukan sama sekali, maka fungsi akan mengembalikan -1 sebagai tanda bahwa target tidak ada dalam array.



Gambar 1. Diagram Alur Tahapan Penelitian

Flowchart yang disajikan menggambarkan tahapan dalam penelitian yang bertujuan mengevaluasi performa beberapa algoritma pencarian berdasarkan dua aspek utama: efisiensi waktu dan konsumsi memori. Proses dimulai dari tahap awal berupa perencanaan serta inisialisasi kegiatan penelitian. Langkah pertama dalam alur ini adalah pembuatan dataset, yang terdiri dari angka-angka acak bertipe integer dan dibagi dalam tiga skala ukuran: 100 data untuk skala *small*, 1.000 data untuk skala *medium*, dan 10.000 data untuk skala *large*. Pembagian ini dimaksudkan agar pengujian dapat mencerminkan performa algoritma dalam kondisi data dengan tingkat kompleksitas yang berbeda.

Setelah proses penyusunan dataset selesai, tahap selanjutnya adalah implementasi empat algoritma pencarian. Masing-masing algoritma diuji satu per satu menggunakan dataset yang telah disiapkan sebelumnya. Tujuan dari pengujian ini adalah untuk mengamati kinerja setiap algoritma saat menangani data dalam volume yang bervariasi.

Tahap akhir dari proses ini melibatkan pengukuran dua parameter utama, yaitu waktu eksekusi dan jumlah memori yang digunakan oleh setiap algoritma. Hasil dari pengukuran ini menjadi dasar dalam mengevaluasi dan membandingkan tingkat efisiensi masing-masing algoritma pencarian. Setelah seluruh tahapan dilakukan, proses penelitian dianggap selesai, dan hasil pengujian siap untuk dianalisis secara lebih mendalam.

4. HASIL DAN PEMBAHASAN

Penelitian ini menggunakan Bahasa pemrograman C++ dan Code Block versi 25.03 untuk mengembangkan serta menjalankan kode program. Pengujian ini dilakukan pada laptop dengan spesifikasi sebagai berikut:

- Processor AMD Ryzen 5 6600H dengan kecepatan 3.30 GHz
- Memori 16 GB RAM
- Sistem Operasi Window 11 64-Bit
- SSD berkapasitas 512 GB

Algoritma yang diuji pada penelitian ini meliputi Binary Search, Jump Search, Interpolation Search, dan Fibonacci Search. Algoritma-algoritma ini diuji dengan dataset yang sama, terdiri dari 100 data (*ArraySmall*), 1.000 data (*ArrayMedium*), dan 10.000 data (*ArrayLarge*) dengan tipe data angka acak bernilai antara 1-1.000.000. Setiap algoritma menjalani tiga kali pengujian:

- Pengujian pertama hingga ketiga menggunakan dataset 100 data (*ArraySmall*)
- Pengujian keempat hingga keenam menggunakan dataset 1.000 data (*ArrayMedium*)
- Pengujian ketujuh hingga kesembilan menggunakan dataset 10.000 data (*ArrayLarge*)

Pengujian dilakukan tiga kali pada setiap dataset untuk menjamin konsistensi hasil dan memperkuat

keyakinan terhadap data yang diperoleh dari pengujian. Selama proses pengujian penguji menggunakan 1 aplikasi yaitu CodeBlock untuk memastikan tidak terjadi gangguan dari aplikasi lain yang bisa mempengaruhi kinerja CPU dan memori.

Setiap algoritma diukur berdasarkan waktu komputasi dan besar penggunaan memori. Berikut merupakan link dokumentasi untuk pengujiannya (https://bit.ly/Dokumentasi_pengujian1).

4.1. *Binary Search*

Berdasarkan Tabel 1, jika dilihat dari segi penggunaan memori, rata-rata pemakaian memori pada pengujian dengan 100 data adalah sebesar 53,33 bytes, dengan 1.000 data sebesar 61,33 bytes, dan dengan 10.000 data sebesar 34,67 bytes. Namun, dari perspektif waktu komputasi, rata-rata waktu komputasi untuk pengujian dengan 100 data adalah 533,33 nanodetik, untuk 1.000 data sebesar 500,00 nanodetik, dan untuk 10.000 data sebesar 333,33 nanodetik.

Tabel 1. Hasil pengujian algoritma binary search

Pengujian ke-	Jumlah data	Pemakaian memori (bytes)	Waktu komputasi (n/s)
1	100	64	700
2	100	40	200
3	100	56	700
4	1000	64	400
5	1000	64	700
6	1000	56	400
7	10.000	64	400
8	10.000	48	300
9	10.000	32	300

4.2. *Jump Search*

Berdasarkan Tabel 2, jika dilihat dari segi penggunaan memori, rata-rata pemakaian memori pada pengujian dengan 100 data adalah 48 byte, dengan 1.000 data adalah 57,33 byte, dan dengan 10.000 data adalah 42,67 byte. Namun, dari perspektif waktu komputasi, waktu komputasi rata-rata untuk pengujian dengan 100 data adalah 3.233,33 nanodetik, dengan 1.000 data adalah 3.866,67 nanodetik, dan dengan 10.000 data adalah 3.666,67 nanodetik.

Tabel 2. Hasil pengujian algoritma jump search

Pengujian ke-	Jumlah data	Pemakaian memori (bytes)	Waktu komputasi (n/s)
1	100	24	3200
2	100	68	2500
3	100	52	4000
4	1000	68	3400
5	1000	32	3800
6	1000	72	4400
7	10.000	60	4400
8	10.000	40	3300
9	10.000	28	3300

4.3. *Interpolation Search*

Berdasarkan Tabel 3, jika dilihat dari segi penggunaan memori, rata-rata pemakaian memori pada pengujian dengan 100 data adalah 26,67 byte, dengan 1.000 data adalah 24 byte, dan dengan 10.000 data adalah 34,67 byte. Namun, dari perspektif waktu komputasi, waktu komputasi rata-rata untuk pengujian dengan 100 data adalah 333,33 nanodetik, dengan 1.000 data adalah 300 nanodetik, dan dengan 10.000 data adalah 400 nanodetik.

Tabel 3. Hasil pengujian algoritma interpolation search

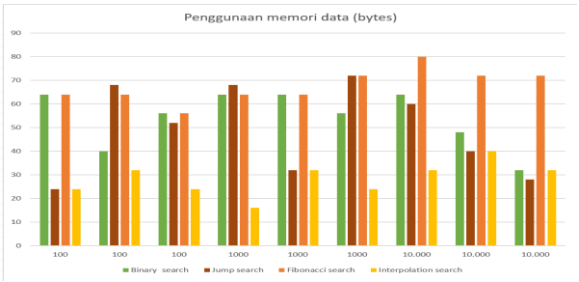
Pengujian ke-	Jumlah data	Pemakaian memori (bytes)	Waktu komputasi (n/s)
1	100	24	300
2	100	32	400
3	100	24	300
4	1000	16	200
5	1000	32	400
6	1000	24	300
7	10.000	32	400
8	10.000	40	400
9	10.000	32	400

4.4. *Fibonacci Search*

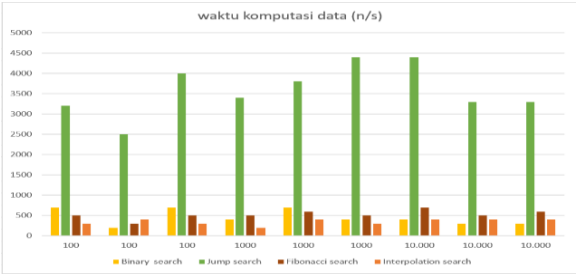
Berdasarkan Tabel 4, jika dilihat dari segi penggunaan memori, rata-rata pemakaian memori pada pengujian dengan 100 data adalah 61,33 byte, dengan 1.000 data adalah 66,67 byte, dan dengan 10.000 data adalah 74,67 byte. Namun, dari perspektif waktu komputasi, waktu komputasi rata-rata untuk pengujian dengan 100 data adalah 433,33 nanodetik, dengan 1.000 data adalah 533,33 nanodetik, dan dengan 10.000 data adalah 600 nanodetik.

Tabel 4. Hasil pengujian algoritma Fibonacci search

Pengujian ke-	Jumlah data	Pemakaian memori (bytes)	Waktu komputasi (n/s)
1	100	64	500
2	100	64	300
3	100	56	500
4	1000	64	500
5	1000	64	600
6	1000	72	500
7	10.000	80	700
8	10.000	72	500
9	10.000	72	600



Gambar 2. Grafik rata-rata penggunaan memori empat algoritma searching



Gambar 3. Grafik rata-rata waktu komputasi empat algoritma searching

Berdasarkan Gambar 2 dan 3 yang menampilkan hasil pengujian empat algoritma pencarian (*Binary Search, Jump Search, Fibonacci Search, dan Interpolation Search*), dapat disimpulkan bahwa:

a. Penggunaan Memori Data (bytes) Keempat algoritma menunjukkan penggunaan memori yang identik untuk setiap kategori jumlah data. Pada data berukuran 100 elemen, semua algoritma menggunakan 2.048 byte. Ketika data meningkat menjadi 1.000 elemen, penggunaan memori naik menjadi 4.096 byte, dan pada 10.000 elemen, memori yang digunakan mencapai 8.192 byte. Hal ini menunjukkan bahwa kebutuhan memori keempat algoritma tumbuh secara linear seiring pertambahan ukuran data, tanpa perbedaan signifikan antaralgoritma.
b. Waktu Komputasi (nanodetik/n/s)
a. Untuk data kecil (100 elemen), Binary Search mencatat waktu tercepat (500 nanosekon (ns)), diikuti Interpolation Search (550 ns), Fibonacci Search (700 ns), dan Jump Search (750 ns).
b. Pada data sedang (1.000 elemen), Interpolation Search unggul sebagai tercepat (1.200 ns), menyusul Binary Search (1.500 ns), Fibonacci Search (2.300 ns), dan Jump Search (2.800 ns).
c. Untuk data besar (10.000 elemen), Binary Search kembali mendominasi (3.000 ns), dengan Interpolation Search di posisi kedua (3.200 ns), sedangkan Fibonacci Search (5.000 ns) dan Jump Search (6.000 ns) tertinggal.

5. KESIMPULAN DAN SARAN

Penelitian ini berhasil menguji dan membandingkan efisiensi waktu komputasi serta penggunaan memori antara empat algoritma pencarian (*Binary Search, Jump Search, Interpolation Search,* dan *Fibonacci Search*) dalam pengolahan data numerik acak menggunakan bahasa pemrograman C++. Hasil penelitian menunjukkan bahwa *Interpolation Search* dan *Binary Search* unggul dalam efisiensi waktu komputasi, terutama pada dataset berukuran besar (10.000 data), dengan waktu tercepat masing-masing 400 ns dan 333,33 ns. Sementara itu, *Jump Search* memerlukan waktu lebih lama (3.666,67 ns) untuk dataset yang sama. Dari segi penggunaan memori, *Interpolation Search* mencatat konsumsi terendah (34,67 byte), sedangkan *Fibonacci Search* menggunakan memori lebih tinggi (74,67 byte).

Temuan ini mengonfirmasi bahwa *Interpolation Search* dan *Binary Search* merupakan pilihan optimal untuk pencarian data terurut, terutama ketika kecepatan komputasi menjadi prioritas. Namun, *Interpolation Search* lebih efisien dalam penggunaan memori. Kelemahan *Jump Search* dan *Fibonacci Search* terletak pada waktu komputasi yang lebih lambat, meskipun tetap layak untuk skenario tertentu.

DAFTAR PUSTAKA

[1] R. Toyib, Y. Darnita, and A. R. S. Deva, "Penerapan Algoritma Binary Search Pada Aplikasi E-Order," *J. Media Infotama*, vol. 17, no. 1, pp. 30–37, 2021, doi: 10.37676/jmi.v17i1.1314.

[2] H. M. T. Lase, "Aplikasi Metode Pencarian Linier, Biner, Dan Interpolasi," *J. Teknol. Inf. dan Ind.*, vol. 3, no. 1, pp. 51–60, 2023.

[3] Z. M. Nasution, M. M. Siadari, I. J. S. Saragih, I. O. Kirana, and Z. A. Siregar, "Penerapan Matematika Algoritma dalam Bidang Komputer," *FARABI J. Mat. dan Pendidik. Mat.*, vol. 6, no. 2, pp. 180–191, 2023, doi: 10.47662/farabi.v6i2.634.

[4] F. A. T. Tobing and R. Nainggolan, "Analisis Perbandingan Penggunaan Metode Binary Search Dengan Regular Search Expression," *METHOMIKA J. Manaj. Inform. dan Komputerisasi Akunt.*, vol. 4, no. 2, pp. 168–172, 2021, doi: 10.46880/jmika.vol4no2.pp168-172.

[5] H. Nasruddin, C. Mashuri, and R. Wiratsongko, "Penerapan Algoritma Binary Searching Untuk Pencarian Berkas Pada Sistem Pengarsipan (Study Kasus: Pemerintahan Desa Kedungbetik)," *J. Ilm. Inov. Teknol. Inf.*, vol. 5, no. 2, pp. 1–8, 2021.

[6] B. J. D. Sitompul, A. Yusupa, and N. J. Tuturoong, "Implementasi Algoritma Binary Search Pada Pencarian Data Jemaat Gereja Hkbp Manado," *J. Inform. Polinema*, vol. 9, no. 1, pp. 17–24, 2022, doi: 10.33795/jip.v9i1.1123.

[7] C. N. Sai, "A Brief Survey and Examination of Various Searching Techniques," vol. 10, no. 10, pp. 853–859, 2021, doi: 10.21275/SR211014183539.

[8] A. Shoaib Zia, "A Survey on Different Searching Algorithms," *Int. Res. J. Eng. Technol.*, vol. 7, no. 1, pp. 1580–1584, 2020.

[9] A. Shabbir *et al.*, "A Review of Algorithms's Complexities on Different Valued Sorted and Unsorted Data," *2023 Int. Conf. IT Ind. Technol. ICIT 2023*, no. October, 2023, doi: 10.1109/ICIT59216.2023.10335840.

[10] F. D. Marentek, V. Tulenan, and S. Paturusi, "Aplikasi Kamus Bahasa Indonesia - Mongondow Menggunakan Algoritma Interpolation Search," *J. Tek. Inform.*, pp. 1–10,

2021.

[11] J. L. Lin, "Interpolation Once Binary Search over a Sorted List," *Mathematics*, vol. 12, no. 9, 2024, doi: 10.3390/math12091394.

[12] S. Kumar Sahoo, E. H. Houssein, M. Premkumar, A. Kumar Saha, and M. M. Emam, "Self-adaptive moth flame optimizer combined with crossover operator and Fibonacci search strategy for COVID-19 CT image segmentation," *Expert Syst. Appl.*, vol. 227, no. May, p. 120367, 2023, doi: 10.1016/j.eswa.2023.120367.

[13] I. Helmy, A. Hamdy, D. Eid, and A. Shokry, "Autofocusing Optimal Search Algorithm for a Telescope System," *J. Astron. Instrum.*, vol. 10, no. 3, pp. 1–13, 2021, doi: 10.1142/S2251171721500124.

[14] I. E. O, O. I. K, O. L. O, and A. Adepeju, "COMPARATIVE EXECUTION TIME ANALYSIS FOR BINARY , JUMP , AND INTERPOLATION SEARCH ALGORITHMS IN OBJECT ORIENTED LANGUAGES," *Univ. PITESTI Sci. Bull. Electron. Comput. Sci.*, vol. 21, no. 2, 2021.

[15] S. Salakos and N. Ploskas, "Analysis and Comparison of Binary and Interpolation Search Algorithms in a B-tree," *ACM Int. Conf. Proceeding Ser.*, vol. 1, no. 1, pp. 74–78, 2021, doi: 10.1145/3503823.3503837.

[16] Pujiono, I. P., Rachmawanto, E. H., & Winarsih, N. A. S. (2025). Array Sorting Algorithm vs Traditional Sorting Algorithm: Memory and Time Efficiency Analysis: Array Sorting Algorithm vs Algoritma Pengurutan Tradisional: Analisis Efisiensi Memori dan Waktu. *Jurnal Manajemen Informatika (JAMIKA)*, *15*(1), 47-59.

[17] Pujiono, I. P., Trianto, R. B., & Hana, F. M. (2024). Perbandingan Efisiensi Memori dan Waktu Komputasi Pada 7 Algoritma Sorting Menggunakan Bahasa Pemrograman Java. Jurnal Sistem Informasi dan Sistem Komputer, 9(2), 218-230.