ALGORITMA PENCARIAN (**SEARCHING**)

Dwi Ratna

Topik

- Linear Search
- Binary Search

Searching pada data

Proses mendapatkan (*retrieve*) informasi berdasarkan kunci (*key*) tertentu dari sejumlah informasi yang telah disimpan.

Proses pencarian data yang ada pada suatu deret data dengan cara menelusuri data-data tersebut.

Kunci (*key*) digunakan untuk melakukan pencarian data yang diinginkan

Tahapan paling penting pada searching: memeriksa jika data yang dicari sama dengan data yang ada pada deret data.

Jenis searching :

Single match : Pencarian yang menghasilkan satu data.

- Contoh : mencari mahasiswa dengan nim “120651234”

Multiple match : Pencarian yang memungkinkan menghasilkan beberapa data.

- Contoh : mencari mahasiswa dengan ipk >= 3.5

LINEAR SEARCH
SEQUENTIAL SEARCH

Linear Search

Metode pencarian beruntun atau linear atau sequential search.

Adalah suatu teknik pencarian data yang akan menelusuri tiap elemen satu per-satu dari awal sampai akhir.

Suatu deret data dapat disimpan dalam bentuk array maupun linked list.

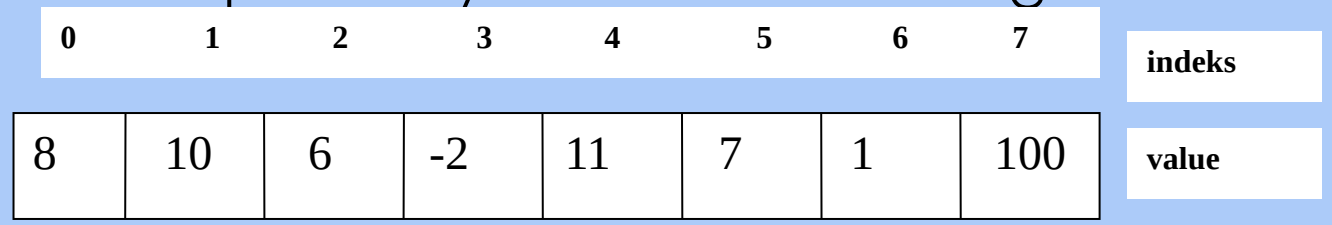
Case

Best case : jika data yang dicari terletak di indeks array terdepan (elemen array pertama) sehingga waktu yang dibutuhkan untuk pencarian data sangat sebentar (minimal).

Worst case : jika data yang dicari terletak di indeks array terakhir (elemen array terakhir) sehingga waktu yang dibutuhkan untuk pencarian data sangat lama (maksimal).

Contoh

- Misalnya terdapat array satu dimensi sebagai berikut:



- Kemudian program akan meminta data yang akan dicari, misalnya **6**.
- Iterasi :
  6 = 8 (tidak!)
  6 = 10 (tidak!)
  6 = 6 (Ya!) => output : 2 (index)

```
procedure PencarianBeruntun(input a1, a2, ..., an : integer, x
                            output idx : integer)
Deklarasi
  k : integer
  ketemu : boolean    { bernilai true jika x ditemukan atau fa
tidak ditemukan }

Algoritma:
  k←1
  ketemu ← false
  while (k ≤ n) and (not ketemu) do
    if ak = x then
      ketemu←true
    else
      k ← k + 1
    endif
  endwhile
  { k > n or ketemu }

  if ketemu then    { x ditemukan }
     idx←k
  else
     idx← 0         { x tidak ditemukan }
  endif
```

ALGORITMA

KOMPLEKSITAS

Jumlah operasi perbandingan elemen tabel:

1. *Kasus terbaik*: ini terjadi bila $a_1 = x$.

$$T_{\min}(n) = 1$$

2. *Kasus terburuk*: bila $a_n = x$ atau x tidak ditemukan.

$$T_{\max}(n) = n$$

3. *Kasus rata-rata*: Jika x ditemukan pada posisi ke-j, maka operasi perbandingan ($a_k = x$) akan dieksekusi sebanyak j kali.

$$T_{\text{avg}}(n) = \frac{(1+2+3+\ldots+n)}{n} = \frac{\frac{1}{2}n(1+n)}{n} = \frac{(n+1)}{2}$$

Cara Lain

Asumsikan bahwa $P(a_j = x) = 1/n$. Jika $a_j = x$ maka T_j yang dibutuhkan adalah $T_j = j$. Jumlah perbandingan elemen larik rata-rata:

$$T_{\text{avg}}(n) = \sum_{j=1}^{n} T_j P(A[j] = X) = \sum_{j=1}^{n} T_j \frac{1}{n} = \frac{1}{n} \sum_{j=1}^{n} T_j$$

$$= \frac{1}{n} \sum_{j=1}^{n} j = \frac{1}{n} \left(\frac{n(n+1)}{2} \right) = \frac{n+1}{2}$$

Q & A

- **Problem:** Apakah cara di atas efisien? Jika datanya ada 10000 dan semua data dipastikan unik?
- **Solution**: Untuk meningkatkan efisiensi, seharusnya jika data yang dicari sudah ditemukan maka perulangan harus dihentikan!
  - **Hint**: Gunakan **break**!
- **Question**: Bagaimana cara menghitung ada berapa data dalam array yang tidak unik, yang nilainya sama dengan data yang dicari oleh user?
  - **Hint**: Gunakan variabel counter yang nilainya akan selalu bertambah jika ada data yang ditemukan!

```
package Searching;

public class LinierSearchingV1 {

    public static void main(String[] args) {
        int [] data = {5, 29, 12, 15, 37, 23, 27, 38, 22, 54, 14, 78, 70};
        int key=22;
        int indeks;
        indeks= pencarianLinier(data, key);

        if (indeks !=0)
            System.out.println("Nomor "+key+" Berada Pada Urutan Ke - "+(indeks));
        else
            System.out.println("Nomor "+key+" Tidak ditemukan ");
    }

    public static int pencarianLinier(int[] data, int key){
          int k = 0;
          int n = data.length;
          boolean ketemu = false;
          while (k < n  && ketemu==false){
            if(key==(data[k])){
                ketemu = true;     }
            else {
                k=k+1;   }
          }
          if (ketemu)
              return k+1;
          else
              return 0;
    }
}
```

PROGRAM V2

```
package Searching;
public class LinearSearching {
    static String [] Data = {"A", "B", "C", "D", "E", "F", "C"};

    public static void main(String[] args) {
        String karakter="K";
        searching(karakter);
    }

    public static void searching(String X){
          int k = 0;
          int n = Data.length;
          boolean ketemu = false;
          while (k < n  && ketemu==false){
            if(X.equals(Data[k])){
                ketemu = true;}
            else {
                k=k+1;
            }
        }

        if(ketemu){
            System.out.println("Data "+ X +" berada pada urutan ke-"+(k+1));
        } else {
            System.out.println("Data "+ X +" Tidak Ditemukan");
        }
      }
}
```

PROGRAM V2

BINARY SEARCH

Binary Search ?

- Pencarian data dimulai dari pertengahan data yang telah terurut.
- Jika kunci pencarian lebih kecil daripada kunci posisi tengah, maka kurangi lingkup pencarian pada separuh data pertama.
- Begitu juga sebaliknya jika kunci pencarian lebih besar daripada kunci tengah, maka pencarian ke separuh data kedua.
- Teknik Binary Search hanya dapat digunakan pada sorted array.

Binary Search

- Menggunakan Binary Search, jika :
  - Nilai-nilai tersebut sudah berurutan (ascending). Disimpan dalam bentuk larik (*array*) atau struktur data sejenis.

Ilustrasi

Contoh Data:

Misalnya data yang dicari **17**

- 0 1 2 3 4 5 6 7 8
- **3 9 11 12 15 17 23 31 35**
- **A B C**
- Karena 17 > 15 (data tengah), maka: awal = tengah + 1

- 0 1 2 3 4 5 6 7 8
- **3 9 11 12 15 17 23 31 35**
- **A B C**
- Karena 17 < 23 (data tengah), maka: akhir = tengah – 1

- 0 1 2 3 4 5 6 7 8
- **3 9 11 12 15 17 23 31 35**
- **A=B=C**
- Karena 17 = 17 (data tengah), maka KETEMU!

Algoritma Binary Search

1. Data diambil dari posisi 1 sampai posisi akhir N
2. Kemudian cari posisi data tengah dengan rumus: **(posisi awal + posisi akhir) / 2**
3. Kemudian data yang dicari dibandingkan dengan data yang di tengah, apakah sama atau lebih kecil, atau lebih besar?
4. Jika lebih besar, maka proses pencarian dicari dengan posisi awal adalah **posisi tengah + 1**
5. Jika lebih kecil, maka proses pencarian dicari dengan posisi akhir adalah **posisi tengah – 1**
6. Jika data sama, berarti ketemu.

```
procedure PencarianBiner(input a1, a2, ..., an : integer, x : integer,
                         output idx : integer)
Deklarasi
   i, j, mid : integer
   ketemu : boolean

Algoritma
   i←1
   j←n
   ketemu←false
   while (not ketemu) and ( i ≤ j) do
      mid ← (i+j) div 2
      if amid = x then
        ketemu ← true
      else
         if amid < x then     { cari di belahan kanan }
          i←mid + 1
         else                    { cari di belahan kiri }
          j←mid - 1;
         endif
      endif
   endwhile
   {ketemu or i > j }

   if ketemu then
      idx←mid
   else
      idx←0
   endif
```

ALGORITMA BINARY SEARCH

KOMPLEKSITAS:

1. *Kasus terbaik*

$$T_{min}(n) = 1$$

2. *Kasus terburuk*:

$$T_{max}(n) = {}^{2}\log n + 1$$

n=1 → pencarian T=1
n=2=2^1 → pencarian T=2
n=4=2^2 → pencarian T=3
n=8=2^3 → pencarian T=4 =3 +1
:
n=2^k → pencarian T = k +1= ^{2}log n + 1

```
package Searching;
public class BinarySearch {

    public static void main(String[] args) {
        int [] data = {5, 9, 12, 15, 17, 23, 27, 38, 42, 54, 64, 78, 90};
        int key=55;
        int indeks;
        indeks= pencarianBinary(data, key);

        if (indeks !=0)
            System.out.println("Nomor "+key+" Berada Pada Urutan Ke - "+(indeks));
        else
            System.out.println("Nomor "+key+" Tidak ditemukan ");
    }

    public static int pencarianBinary(int[] data, int key) {
        int bawah = 0;
        int atas = data.length - 1;

        while (atas >= bawah) {
            int tengah = (bawah + atas) / 2;
            if (key == data[tengah]){
                return (tengah+1);
            }
            else if (key < data[tengah]){
                atas = tengah - 1;  }
            else{
                bawah = tengah + 1; }
        }
        return 0;
    }
}
```

PROGRAM V1

```
package Searching;
public class BinarySearch {
    static int [] data = {5, 9, 12, 15, 17, 23, 27, 38, 42, 54, 64, 78, 90};

    public static void main(String[] args) {
        int key=38;
        System.out.println(pencarianBinary(key));
    }

    public static String pencarianBinary(int key) {
        int bawah = 0;
        int atas = data.length - 1;

        while (atas >= bawah) {
            int tengah = (bawah + atas) / 2;
            if (key < data[tengah]){
                atas = tengah - 1;
            } else if (key == data[tengah]){
                return "Nomor "+key+" Berada Pada Urutan Ke - "+(tengah+1);
            } else{
                bawah = tengah + 1;
            }
        }
        return "Data Tidak Ditemukan";
    }
}
```

PROGRAM V2

SELESAI

we get the following recurrence relation for $C_{worst}(n)$:

$$C_{worst}(n) = C_{worst}(\lfloor n/2 \rfloor) + 1 \quad \text{for } n > 1, \quad C_{worst}(1) = 1. \tag{4.3}$$

(Stop and convince yourself that $n/2$ must be, indeed, rounded down and that the initial condition must be written as specified.)

We already encountered recurrence (4.3), with a different initial condition, in Section 2.4 (see recurrence (2.4) and its solution there for $n = 2^k$). For the initial condition $C_{worst}(1) = 1$, we obtain

$$C_{worst}(2^k) = k + 1 = \log_2 n + 1. \tag{4.4}$$

Further, similarly to the case of recurrence (2.4) (Problem 7 in Exercises 2.4), the solution given by formula (4.4) for $n = 2^k$ can be tweaked to get a solution valid for an arbitrary positive integer n:

$$C_{worst}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil. \tag{4.5}$$

Formula (4.5) deserves attention. First, it implies that the worst-case time efficiency of binary search is in $\Theta(\log n)$. Second, it is the answer we should have

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } k > 0,$$
$$A(2^0) = 0.$$

Now backward substitutions encounter no problems:

$$\begin{aligned}
A(2^k) &= A(2^{k-1}) + 1 && \text{substitute } A(2^{k-1}) = A(2^{k-2}) + 1 \\
&= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2 && \text{substitute } A(2^{k-2}) = A(2^{k-3}) + 1 \\
&= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3 && \dots \\
&\dots \\
&= A(2^{k-i}) + i \\
&\dots \\
&= A(2^{k-k}) + k.
\end{aligned}$$

Thus, we end up with

$$A(2^k) = A(1) + k = k,$$

or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$$A(n) = \log_2 n \in \Theta(\log n).$$