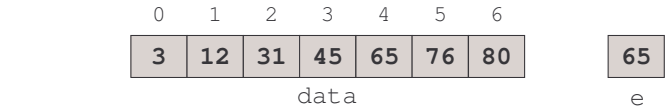
2110211 โครงสร้างข้อมูลเบื้องต้น

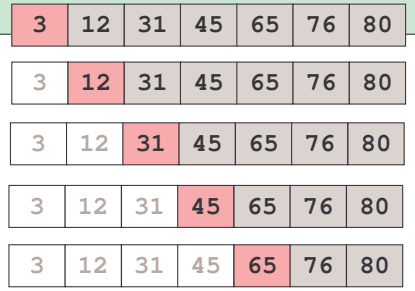
Big O

สมชาย ประสิทธิ์จูตระกูล

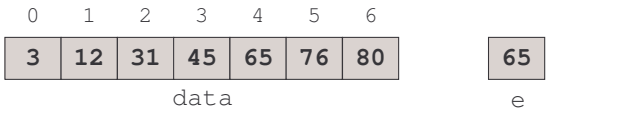
การค้นข้อมูลแบบลำดับในอาเรย์




```
private static int indexOf(int[] data, int e) {
  int n = data.length;
  for(int i=0; i<n; i++)
    if (data[i] == e) return i;
  return -1;
}
```



การค้นข้อมูลแบบทวิภาค (binary search)



65

e

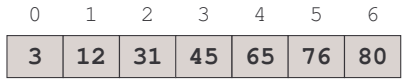
65 > 45

65 < 76

65 == 65

ข้อมูลในอาเรย์ต้องเรียงลำดับ

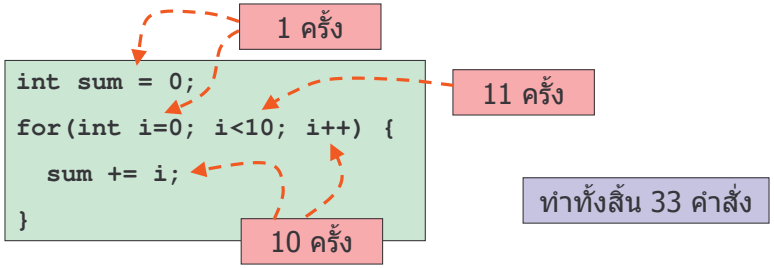
การค้นข้อมูลแบบทวิภาค (binary search)




```
private static int indexOf(int[] data, int e) {
  int left = 0, right = data.length - 1;
  while( left <= right ) {
    int mid = (left + right) / 2;
    if (e == data[mid]) return mid;
    if (e < data[mid])
      left = mid + 1;
    else
      right = mid - 1;
  }
  return -1;
}
```

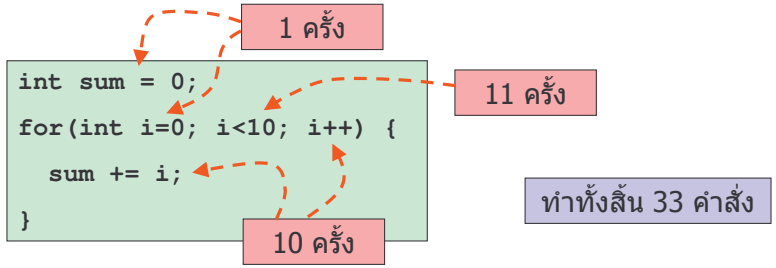
Sequential Search vs. Binary Search

- ขอเปรียบเทียบเวลาการทำงาน
- เวลาการทำงานขึ้นกับข้อมูลที่นำมาค้น
  - โชคดี (best case) : ค้นพบตัวแรกที่นำมาเปรียบเทียบ
  - โชคร้าย (worst case) : กรณีค้นไม่พบ
- แล้วเราจะวัดเวลาการทำงานกันอย่างไร ?
  - ประมาณด้วยจำนวนคำสั่ง (แบบพื้นฐาน) ที่ได้ทำงาน



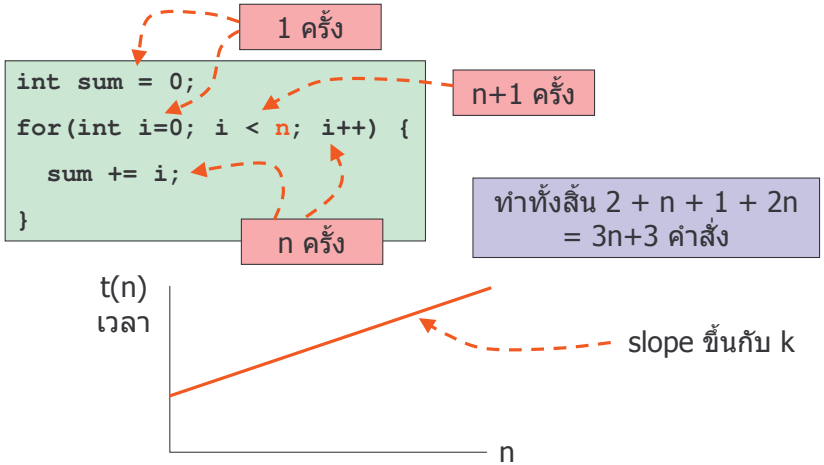
ประมาณเวลาการทำงานด้วยปริมาณคำสั่ง

- `sum = 0, i = 0, i<10, i++, sum += i` ล้วนเป็นคำสั่งพื้นฐาน
  - รู้แน่ว่าคำสั่งเหล่านี้ทำงานไม่เกิน k วินาที
  - k เป็นค่าคงตัว เป็นเวลามากสุดของคำสั่งพื้นฐานต่าง ๆ
  - ถ้าทำ 33 คำสั่ง แสดงว่าทำงานไม่เกิน 33k วินาทีแน่ ๆ



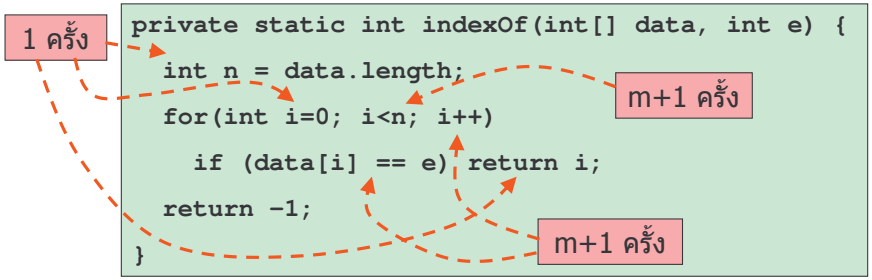
ประมาณเวลาการทำงานด้วยปริมาณคำสั่ง

- ให้ k คือเวลามากสุดของคำสั่งพื้นฐานต่าง ๆ
  - ถ้าทำ 3n+3 คำสั่ง แสดงว่าทำงานไม่เกิน (3n+3)k วินาที



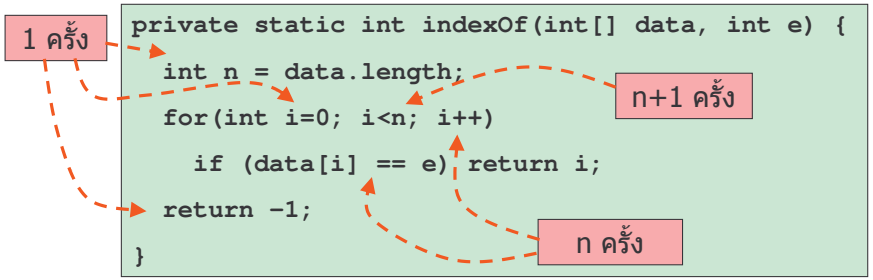
Sequential Search : กรณีค้นพบ

- ให้ n คือปริมาณข้อมูลในอาเรย์
- ค้นพบที่ index ที่ m โดยที่ $0 \le m \le n-1$
- ทำทั้งสิ้น $3 + 3(m+1) = 3m+4 \le (3(n-1)+4)$ $= 3n+1$ คำสั่ง
- ให้ k คือเวลามากสุดของคำสั่งพื้นฐานต่าง ๆ
- เวลาการทำงานไม่เกิน (3n+1)k วินาที

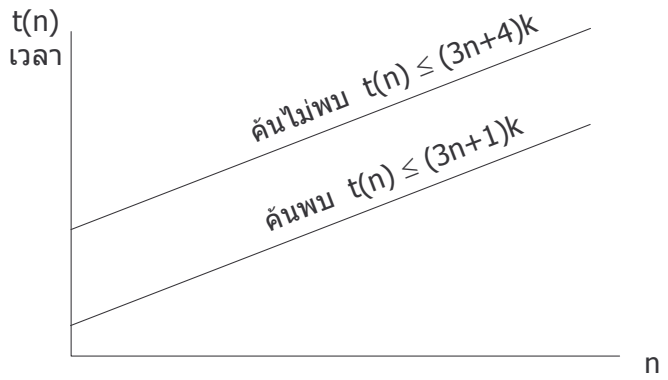


Sequential Search : กรณีค้นไม่พบ

- ให้ n คือปริมาณข้อมูลในอาเรย์
- ทำทั้งสิ้น 3 + n+1 + 2n = 3n+4 คำสั่ง
- ให้ k คือเวลามากสุดของคำสั่งพื้นฐานต่าง ๆ
- เวลาการทำงานไม่เกิน (3n+4)k วินาที



Sequential Search



Big Omicron

- นิยามให้
  - $O(g(n)) = \{f(n) \mid$ มี c และ n_0 ที่ $f(n) \le cg(n), n \ge n_0 \}$
- จะได้ว่า
  - $t(n) \le (3n+4)k = 3kn + 4k \le (7k)n$ เมื่อ $n \ge 1$
    สรุปได้ว่า $t(n) \le (3n+4)k = O(n)$
  - $t(n) \le (3n+1)k = 3kn + k \le (4k)n$ เมื่อ $n \ge 1$
    สรุปได้ว่า $t(n) \le (3n+1)k = O(n)$
- เวลาการทำงานของ sequential search = O(n)

อ่านแล้วรู้ว่าเวลาการทำงานเป็นฟังก์ชันที่โตแบบ linear ของ n

การเขียนฟังก์ชันในรูปของ Big O แบบง่าย ๆ

- ถ้า t(n) เป็นผลรวมของพจน์หลาย ๆ พจน์ ให้เลือกพจน์ที่โตเร็วสุด
- g(n) โตเร็วกว่า f(n) เมื่อ $\lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$
- ข้อสังเกต
  - cg(n) = O(g(n)) เมื่อ c เป็นค่าคงตัว
  - $\log_2 n = O(\log_{10} n)$ เพราะ $\log_2 n = c \log_{10} n$ โดยที่ $c = (1/\log_{10} 2)$
- เช่น
  - $t(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_0 = O(n^k)$
  - $0.001n^3 + 7000n^2 - 11 = O(n^3)$
  - $\log n^{10} = 10(\log n) = O(\log n)$
  - $n^{0.1} + (\log n)^{10} = O(n^{0.1})$

วิเคราะห์เวลาการทำงานแบบ Big O

- จากนี้ไปเราจะเขียนเวลาทำงานในรูปของ Big O
- เพราะว่าง่าย
- เปรียบเทียบ $O(f_1(n))$ กับ $O(f_2(n))$ ได้เมื่อ n มีค่ามาก
- วิธีวิเคราะห์เวลาการทำงานแบบ Big O
  - นับจำนวนคำสั่งพื้นฐาน เฉพาะคำสั่งสำคัญ ๆ ที่สามารถใช้เป็นตัวแทนของเวลาโดยรวมได้
  - แล้วค่อยนำมาเขียนในรูป Big O

```
int sum = 0;
for(int i=0; i<n; i++) {
  for(int j=i; j<n; j++) {
    sum += i;
  }
}
```

นับเฉพาะคำสั่งนี้ก็พอ

$$\sum_{i=0}^{n-1}\sum_{j=i}^{n-1} 1 = \sum_{i=0}^{n-1} n-i$$

$$= \sum_{k=1}^{n} k = \frac{n(n+1)}{2}$$

$$= \frac{n^2}{2} + \frac{n}{2} = O\left(n^2\right)$$

กลับมาวิเคราะห์ Binary Search

```
private static int indexOf(int[] data, int e) {
  int left = 0, right = data.length - 1;
  while( left <= right ) {
    int mid = (left + right) / 2;
    if (e == data[mid]) return mid;
    if (e < data[mid])
      left = mid + 1;
    else
      right = mid - 1;
  }
  return -1;
}
```

loop#	right-left+1
1	n
2	n/2
3	n/4
...	
k	$n/2^{k-1}$
...	
$1+\log_2 n$	1
$2+\log_2 n$	0

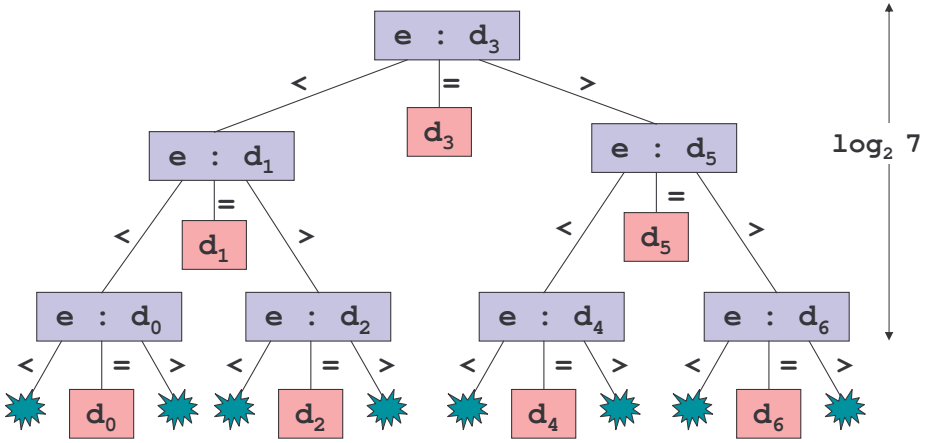
กรณีหาไม่พบ

t(n) = O(#loops) = O(log n)

วิเคราะห์ด้วย decision tree

- ค้น e ในอาเรย์ที่มีข้อมูล 7 ตัว :

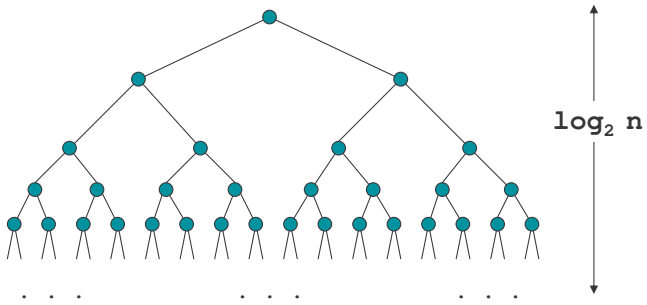
$d_0,\ d_1,\ d_2,\ d_3,\ d_4,\ d_5,\ d_6$



วิเคราะห์ด้วย decision tree

- มีข้อมูล n ตัว (เพื่อความง่ายให้ $n = 2^0, 2^1, 2^2, \ldots$)
- 1 ปมในต้นไม้แทนการเปรียบเทียบกับข้อมูล 1 ตัว
- balanced tree : ความสูงของต้นไม้เป็น $\log_2 n$
- ดังนั้นเกิดการเปรียบเทียบกับข้อมูล

$1 + \log_2 n = O(\log n)$ ตัว



เขียน recursive วิเคราะห์ recurrence

```
private static int indexOf(int[] data, int e) {
  return indexOf(data, e, 0, data.length-1);
}

private static int indexOf(int [] data, int e,
                           int left, int right) {
  if (left > right) return -1;
  int mid = (left + right) / 2;
  if (e == data[mid]) return mid;
  if (e < data[mid])
    return indexOf(data, e, left, mid-1);
  else
    return indexOf(data, e, mid+1, right);
}
```

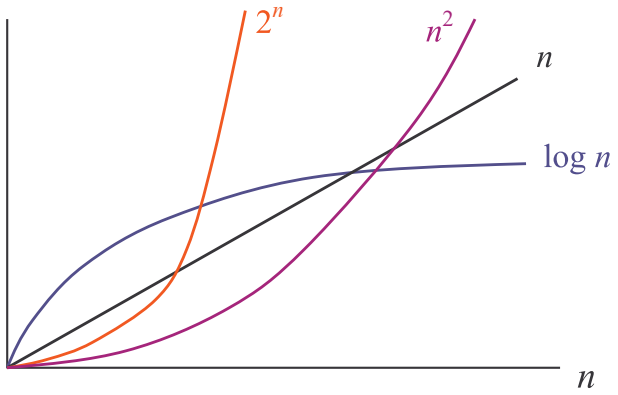
$$
\begin{aligned}
t(n) &= t(n/2) + 1,\ n > 0,\ t(0) = 1 \\
&= t(n/4) + 1 + 1 \\
&= t(n/8) + 1 + 1 + 1 \\
&= t(0) + 1 + \log_2 n \\
&= O(\log n)
\end{aligned}
$$

กรณีหาไม่พบ

Sequential Search vs. Binary Search

- sequential search : O(n)
  - ข้อมูลเพิ่ม 1000 เท่า ใช้เวลานานเป็น 1000 เท่า
- binary search : O(log n)
  - ข้อมูลเพิ่ม 1000 เท่า ใช้เวลานานเป็น 10 เท่า
- log n โตช้ากว่า n (log n $\prec$ n)
  - binary search ทำงานเร็วกว่า sequential search เมื่อมีปริมาณข้อมูลมาก (อย่าลืมโตเร็ว โตช้า เราใช้ limit)
- อย่าลืม
  - ใช้ binary search ได้เฉพาะเมื่อข้อมูลเรียงลำดับแล้ว
  - sequential search ใช้ได้เสมอ ไม่ว่าจะเรียงหรือไม่

Growth Rates



$$\log n \prec n \prec n^2 \prec 2^n$$

มาวิเคราะห์ ArrayCollection กัน

- constructor
  - เป็นการจองอาเรย์จำนวนช่องที่เป็นค่าคงตัว จึงใช้เวลาคงตัว (ไม่แปรตามขนาดข้อมูล)
  - ใช้เวลาคงตัว = O(1)
- isEmpty
  - ทำแค่ 1 คำสั่งเพื่อเปรียบเทียบ : ใช้เวลา O(1)

```
public class ArrayCollection implements Collection {
  private Object[] elementData;
  private int size;
  public ArrayCollection() {
    elementData = new Object[1];
  }
  public boolean isEmpty() { return size == 0; }
  ...
}
```

การบ้าน : วิเคราะห์ ArrayCollection

- วิเคราะห์เวลาการทำงานของเมท็อดที่เหลือทั้งหมด

```
public class ArrayCollection implements Collection {
  ...
  public int size() { return size; }

  public void add(Object e) {
    ensureCapacity(size+1);
    elementData[size++] = e;
  }

  private void ensureCapacity(int capacity) {
    if (capacity > elementData.length) {
      Object[] newA = new Object[2*elementData.length];
      for(int i=0; i<elementData.length; i++) {
        newA[i] = elementData[i];
      }
      elementData = newA;
    }
  }
```


การบ้าน : วิเคราะห์ ArrayCollection

```
  ...
  public boolean contains(Object e) {
    return indexOf(e) != -1;
  }
  public void remove(Object e) {
    int i = indexOf(e);
    if (i != -1) {
      elementData[i] = elementData[--size];
      elementData[size] = null;
    }
  }
  private int indexOf(Object e) {
    for(int i=0; i<size; i++)
      if (elementData[i].equals(e)) return i;
    return -1;
  }
}
```


การบ้าน : add

- ข้อสังเกต : การขยายอาเรย์ใน add นั้น นาน ๆ ทำที
- ถ้าตอน constructor เราจองอาเรย์ 0 ช่อง แล้วลอง add ติด ๆ กันสัก 32 ครั้ง จะมีการขยายอาเรย์ในการ add ครั้งที่ 1, 2, 5, 9, 17 (จริงไหม?)
- ถาม :
  - ถ้าเรา add ติด ๆ กัน n ครั้ง การทำงานที่ add แบบสะสมรวมหมดทั้งการ add n ครั้ง จะเสียเวลาเท่าใด ?
  - ถ้านำเวลารวมทั้งหมดที่หาได้ มาหารด้วย n จะเป็นเวลาถัวเฉลี่ยต่อการ add หนึ่งครั้งจะเสียเวลาถัวเฉลี่ยเท่าใด ?