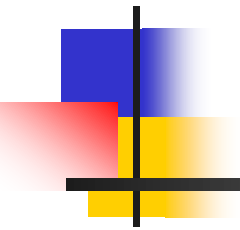


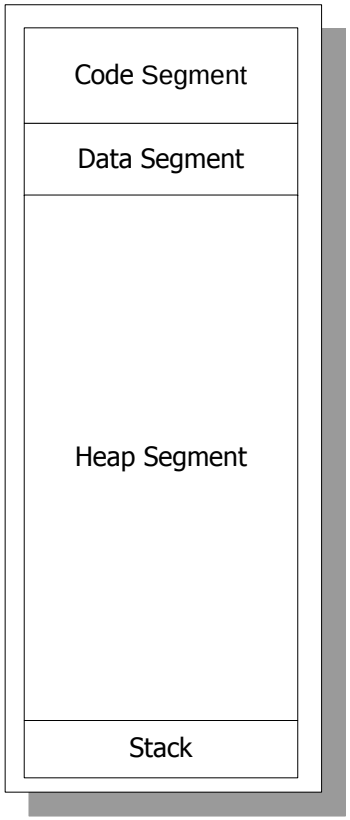
Chapter 01

Dynamic memory Allocation
Array Searching & Sorting

Memory Allocation

- การรันโปรแกรมภาษาซี ระบบปฏิบัติการจะจัดสรรพื้นที่หน่วยความจำไว้สำหรับเก็บโปรแกรม และหน่วยความจำต่างๆ ให้ ดังนี้
  - **Code Segment** ใช้เก็บส่วนของคำสั่ง ฟังก์ชัน ต่างๆ
  - **Data Segment** ใช้เก็บตัวแปรที่ต้องเตรียมไว้ก่อนรันโปรแกรม
    - ตัวแปรประเภท **global variable**
    - ตัวแปรที่กำหนดเป็น **static**
    - ขนาดขึ้นอยู่กับคอมไพล์เลอร์ และระบบปฏิบัติการ
  - **Stack** เป็นส่วนที่ใช้เก็บตัวแปรที่เกี่ยวข้องกับฟังก์ชัน
    - ค่าตัวแปรต่างๆ ในฟังก์ชัน และพารามิเตอร์
    - สถานะของการรันโปรแกรม
    - ถ้าตัวแปรมากเกินกว่าขนาดของ **Stack** โปรแกรมจะหยุดทำงาน
    - สามารถปรับเพิ่ม/ลดได้ ขึ้นอยู่กับคอมไพล์เลอร์
    - **Code::Blocks -> Settings -> Compiler & Debugger..**
      - **Linker settings -> other linker options:**
        พิมพ์เพิ่ม "**-Wl,--stack,123456789**"
  - **Heap Segment** หน่วยความจำว่างที่เหลือ
    - ระบบจะจัดสรร ตามการเรียกใช้ของโปรแกรม
    - ในบางภาษาเช่นจาวา ระบบจะจัดการพื้นที่ให้ทั้งหมด
    - ในภาษาซี ผู้เขียนโปรแกรมจะต้องจัดการพื้นที่เอง
      - การจองเพิ่ม (malloc) ตัวแปร dynamic
      - ต้องมีการลบทิ้ง (free) เพื่อไม่ให้เกิดขยะ(leak)

Memory Map



Pointers & Variables

- **Pointer คือตัวชี้ตำแหน่งของข้อมูลที่เก็บอยู่ในหน่วยความจำ (Memory Address)**
- **ภาษาซี สามารถสร้างตัวแปรที่ใช้เก็บ address ของตัวแปรตัวอื่นได้ เรียกว่า pointer โดยต้องให้ใส่เครื่องหมาย *นำหน้าชื่อตัวแปรที่ต้องการให้เป็น pointer เช่น**

```
int  *p;  // กำหนดให้ตัวแปร p เป็นพอยต์เตอร์ของ int
```

  - **ใช้งานจริง จะต้องมีตัวแปรที่เก็บข้อมูลอยู่ก่อนแล้ว แล้วนำพอยต์เตอร์ชี้ไปยังตำแหน่งที่เก็บข้อมูลตัวนั้น**

```
int   x = 10 ;  // ตัวแปร x เก็บใช้ข้อมูลได้โดยตรงมีค่าเป็น 10;
      p = &x ;  //  กำหนดให้ p ชี้ไปยังตำแหน่งของตัวแปร x
```

  - **สามารถจองตัวแปร pointer สำหรับเก็บตำแหน่งของตัวแปร pointer ซ้อนกันได้**

```
int  **q ;        // ตัวแปร q  เป็น pointer to pointer (double pointer)
     q  =  &p ;  // ให้ q เก็บตำแหน่งของตัวแปร p
```

- **การอ้างถึงค่าข้อมูล ที่ตัวแปรพอยต์เตอร์ชี้อยู่ ให้ใช้ *นำหน้าชื่อพอยต์เตอร์**
  - **มีตัวแปร 3 ตัว คือ x, p, q**
  - **การอ้างถึงค่าตัวแปรที่มีค่าเป็น 10 คือ x, *p และ **q**
- **การเพิ่ม/ลดค่า pointer (Pointer Arithmetic) 1 ตำแหน่ง จะเป็นการเลื่อนตำแหน่งไปยังข้อมูลตัวถัดไปของพอยต์เตอร์ ซึ่งจะมีค่าเท่ากับขนาดตัวแปร (sizeof(type)) ตามชนิดของ pointer เช่น**

  สมมุติตัวแปร int ใช้เนื้อที่ในการเก็บข้อมูล 4 bytes

```
int  *p;  // ถ้าสมมุติให้ p มีค่า address เป็น  2000(ฐาน10)
```

  - **p++ จะมีค่า เป็น 2004 และ p-- จะมีค่าเป็น 1996 (ฐาน10) ฯลฯ**
  - **p+10 จะมีค่า เป็น 2040 และ p-10 จะมีค่าเป็น 1960 (ฐาน10) ฯลฯ**
- **ต้องใช้พอยต์เตอร์ในการส่งค่าพารามิเตอร์แบบ reference**

Pointer & Array

- **ARRAY**
  - อาร์เรย์คือ pointer ที่ชี้ไปยังข้อมูลตัวแรกของกลุ่มข้อมูลแถวลำดับ มีค่าเท่ากับ address แรกของกลุ่มข้อมูลแถวลำดับ
  - อาร์เรย์ไม่สามารถเปลี่ยนค่า address เริ่มต้นของข้อมูลได้
  - ตัวแปรพอยต์เตอร์ที่ชี้ไปยังอาร์เรย์ สามารถใช้เป็นอาร์เรย์ได้
  - สามารถใช้ตัวแปรพอยต์เตอร์ชี้ไปยังตำแหน่งใดๆ(index) ของอาร์เรย์ได้

```
int *x , data[100] ;
        x = data;        // x[0] = data[0], x[1]=data[1],...
        x = &data[5]; // x[0] = data[5], x[1]=data[6],...
```

- การเข้าถึงข้อมูลในอาร์เรย์ โดยใช้ pointer arithmetic (++, --, + , -) โปรแกรมจะทำงานได้เร็วกว่าการอ้าง index [] กับอาร์เรย์โดยตรง
  - int data[100]; การอ้างตัวแปรด้วย *(data+10) จะทำงานเร็วกว่า data[10]

```
for(x=data,i=0; i<100;i++,x++)
    printf("data[%d] = %d\n",i,*x);
```

- ระวังเรื่อง precedence ของ pointer ในการทำงานด้วย
  - int *x , data[100] ; x = data;
    - *x+10 มีค่าเท่ากับ data[0]+10
    - *(x+10) มีค่าเท่ากับ data[10]
- การส่งตัวแปรอาร์เรย์ไปใช้ในฟังก์ชันอื่น เนื่องจากอาร์เรย์เป็น address หรือ pointer อยู่แล้ว จึงส่งได้เฉพาะ pass by reference เท่านั้น

Memory Allocation

- **Static Variables** ตัวแปรที่ประกาศไว้ล่วงหน้าทั้งหมดในภาษาซี
  - ตัวแปรจะถูกสร้างขึ้นใน Stack หรือในส่วนของ Data Segment
    - `int i, A[1000] ;` // ตัวแปร int ตัวแปรอาร์เรย์
    - `int *p , **q ;` // p ใช้เก็บ address ของ i หรือ A, q ใช้เก็บ addr. ของ p
    - `gradetype x, st[100] ;` //ตัวแปรสตรัคเจอร์
- **Dynamic Variables** ตัวแปรที่ถูกใช้คำสั่งให้สร้างขึ้นระหว่างการรันโปรแกรม
  - สร้างอยู่ใน Heap Segment
  - สร้างขึ้นด้วยคำสั่ง malloc()

    `void * malloc(unsigned size);` //สร้างเนื้อที่ = size (bytes)

    `void * calloc(unsigned num, unsigned size);` //สร้างเนื้อที่ = num*size และ clear
  - ปรับขนาดของเนื้อที่ได้ด้วยคำสั่ง realloc()

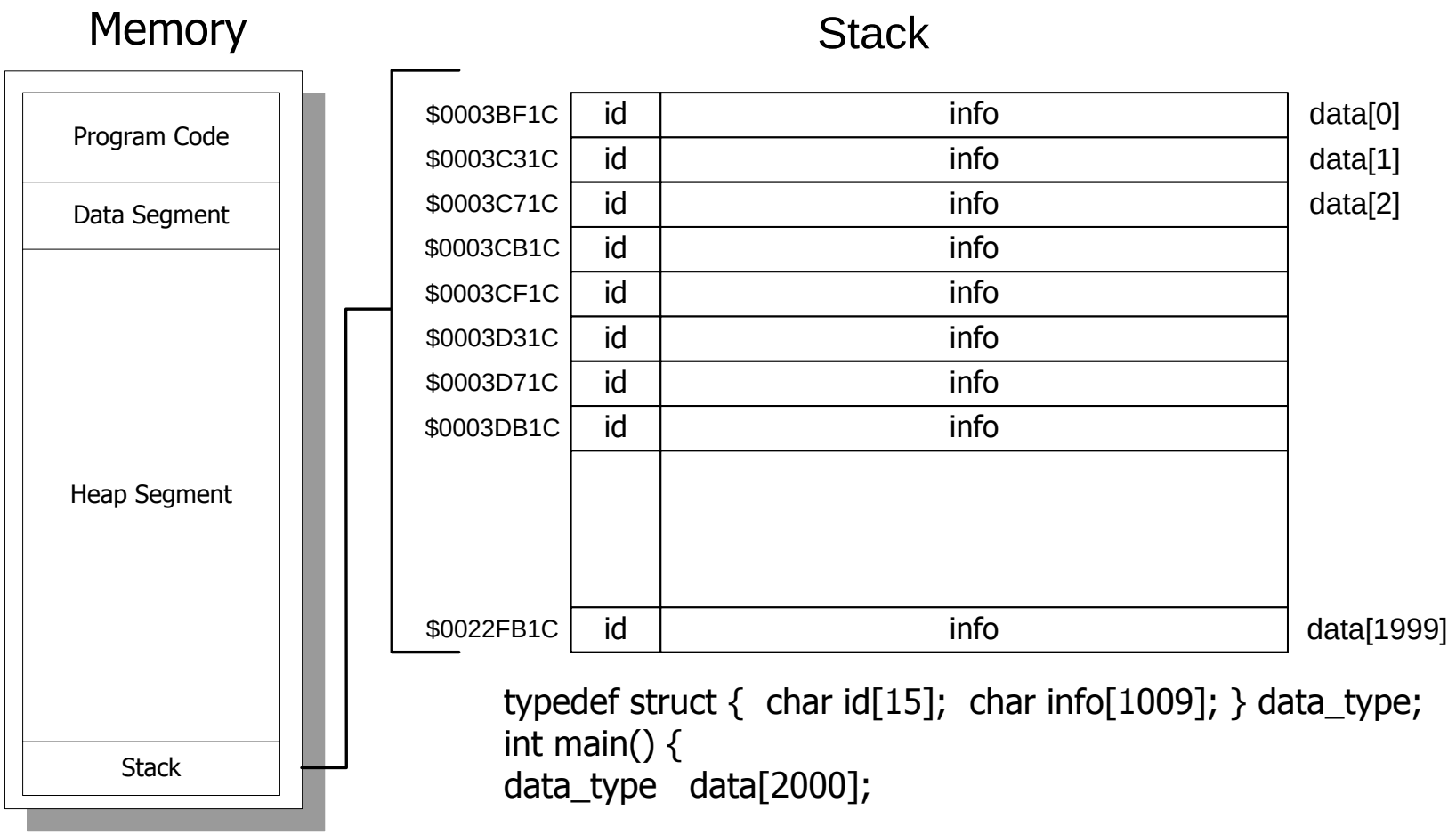
    `void * realloc(void *ptr, unsigned newsize);` //ถ้าลดลงอาจทำให้ข้อมูลบางส่วนหาย
  - `void *` คือ void pointer ของภาษาซีซึ่งยังไม่กำหนดชนิดข้อมูล จะต้องทำ Type conversion (Casting) ภายหลังเพื่อเปลี่ยนให้เป็นชนิดข้อมูลที่ต้องการด้วย
  - ต้องมีตัวแปรประเภท pointer (static ,สร้างไว้ก่อน) เพื่อชี้มายังข้อมูลส่วนนี้
    - `int *m;` // m คือ pointer ที่สร้างไว้(static)

      `m = (int *)malloc(50000*sizeof(int));` // cast m เตรียมไว้ 50,000 ตัว

      `m = (int *)realloc(m,1000000*sizeof(int));` //เปลี่ยน m เป็น 1,000,000 ตัว
    - จะได้ตัวอาร์เรย์ของ int ไว้ใช้งาน 1000000 ตัว ตั้งแต่ m[0] .. m[999999]
  - ใช้งานเสร็จต้องคืนหน่วยความจำให้ระบบ(ลบทิ้ง) ด้วยคำสั่ง free(m) ;

Stack & Arrays

- การจองตัวแปรอาร์เรย์ในฟังก์ชันของภาษาซี ปกติจะจองอยู่ใน **stack**
  - ถูกจำกัดจำนวนข้อมูลสูงสุด ด้วยขนาดของ **Stack** (สามารถกำหนดเพิ่มได้)
  - วิธีการเพิ่ม **Stack** ขึ้นอยู่กับ **IDE** ที่เลือกใช้

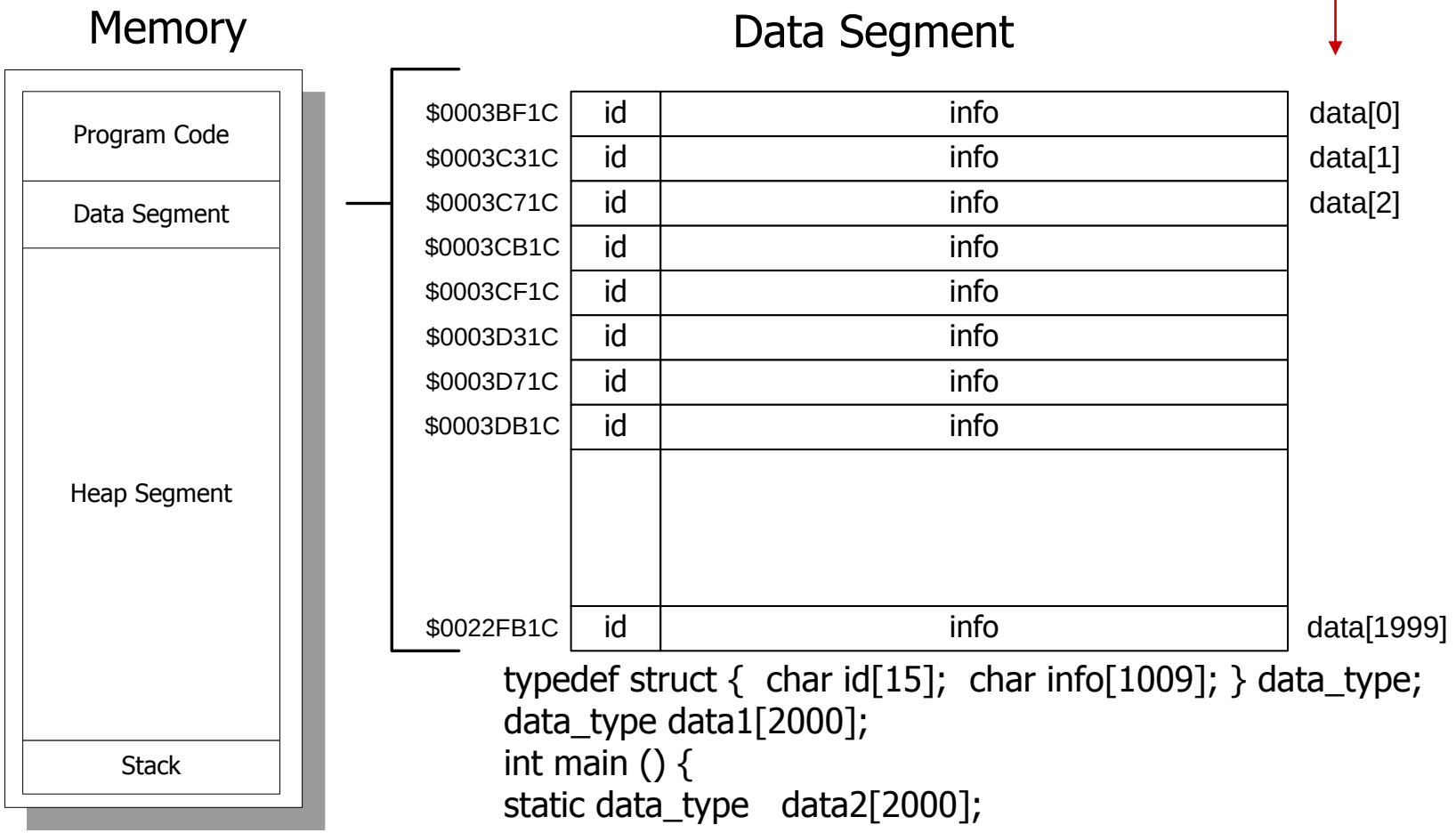



```
typedef struct {  char id[15];  char info[1009]; } data_type;
int main() {
data_type   data[2000];
```

Data Segment, Static Arrays

- การจองตัวแปรอาร์เรย์ในภาษาซี สามารถจองเป็น Global หรือ static เพื่อสร้างใน Data Segment ได้ โดยการประกาศไว้ในส่วน Global หรือประกาศให้เป็น static

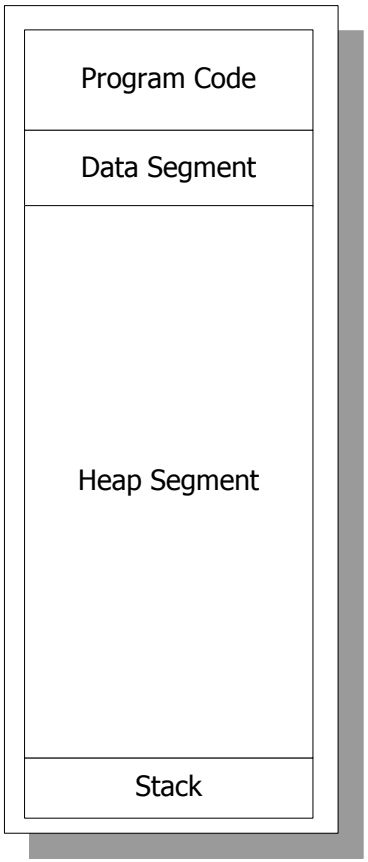
ตัวแปร data1[] และ data2 อยู่ใน Data Segment

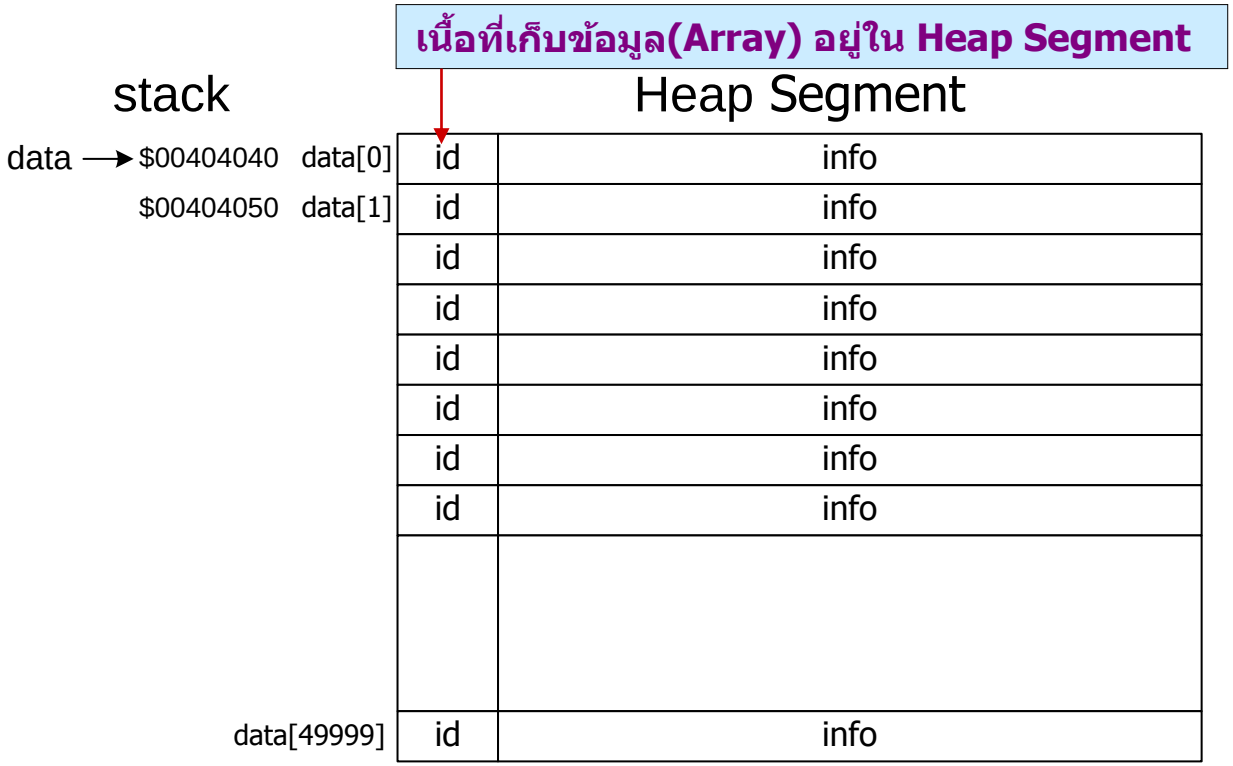



```
typedef struct {  char id[15];  char info[1009]; } data_type;
data_type data1[2000];
int main () {
static data_type   data2[2000];
```

Pointer & Dynamic Arrays

- สามารถจองตัวแปรอาร์เรย์ในภาษาซี ไว้ในส่วนของ **dynamic memory** ได้
  - จองตัวแปรพอยต์เตอร์ไว้ก่อน (ใน **Stack** หรือ **Data Segment**)
  - สร้างเนื้อที่ใน **Heap (Dynamic)** แล้วใช้พอยต์เตอร์ชี้มาที่หน่วยความจำส่วนนี้

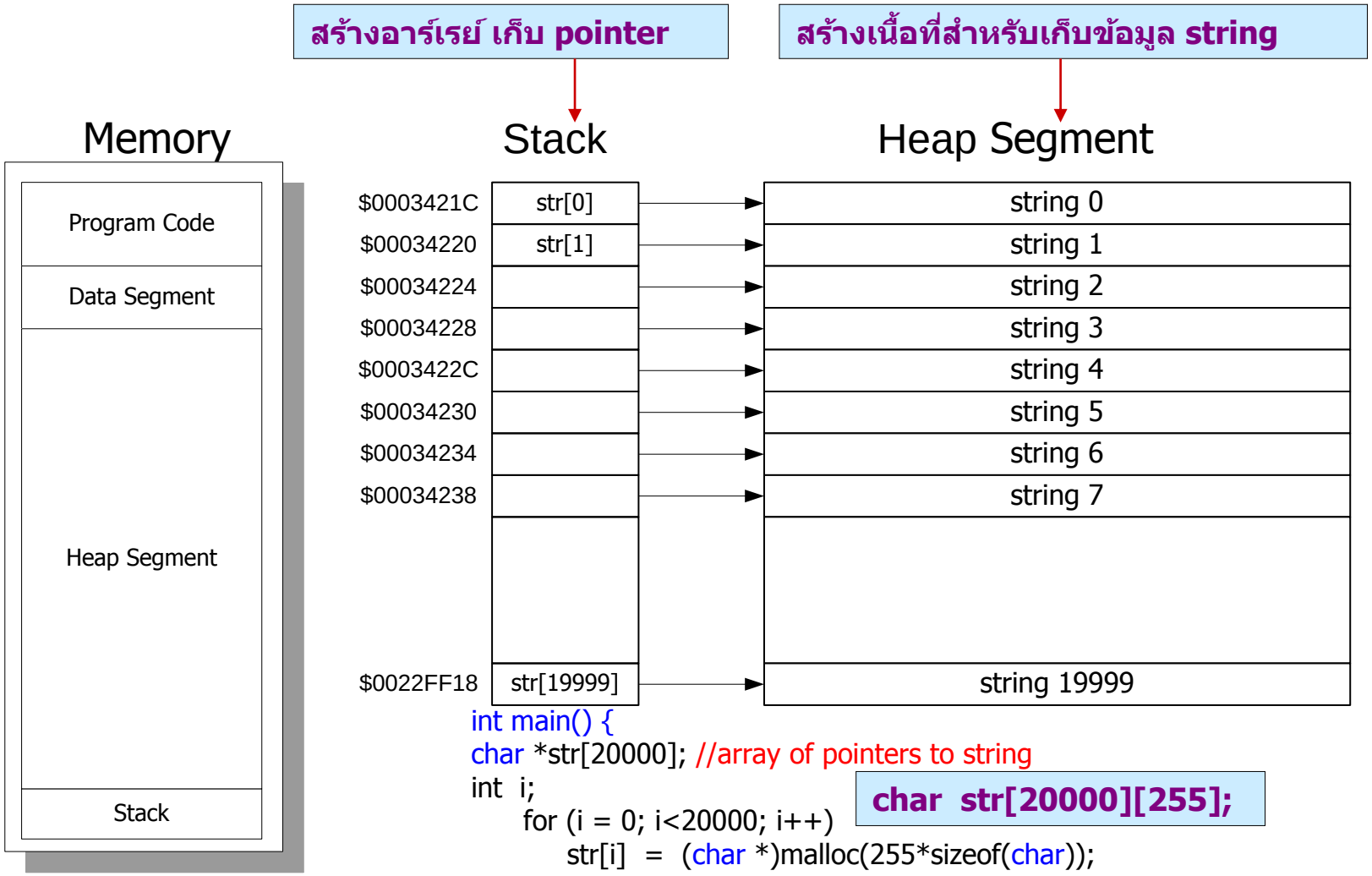





```
typedef struct {  char id[15];  char info[1009]; } data_type;
int main()
{ data_type *data= (data_type *)malloc(50000*sizeof(data_type));
```

Array of Pointers (Array of string)

- จองอาร์เรย์ของพอยต์เตอร์ที่จะชี้ไปยังสตริงไว้ก่อน
- วนรอบ **malloc** เนื้อที่ทีละตัว แล้วใช้อาร์เรย์พอยต์เตอร์ ชี้มาที่หน่วยความจำส่วนนี้

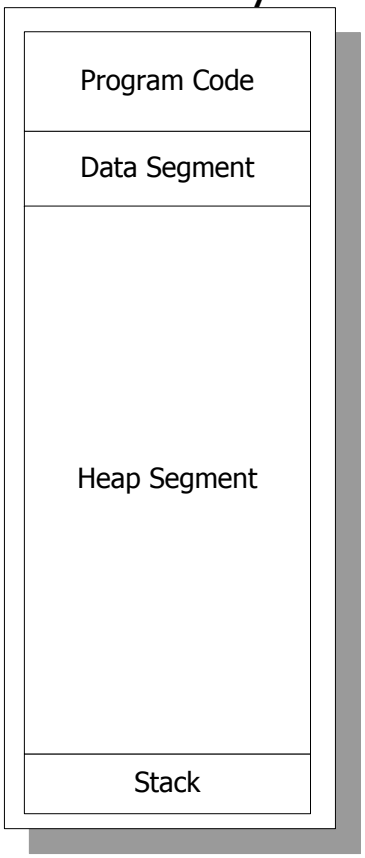


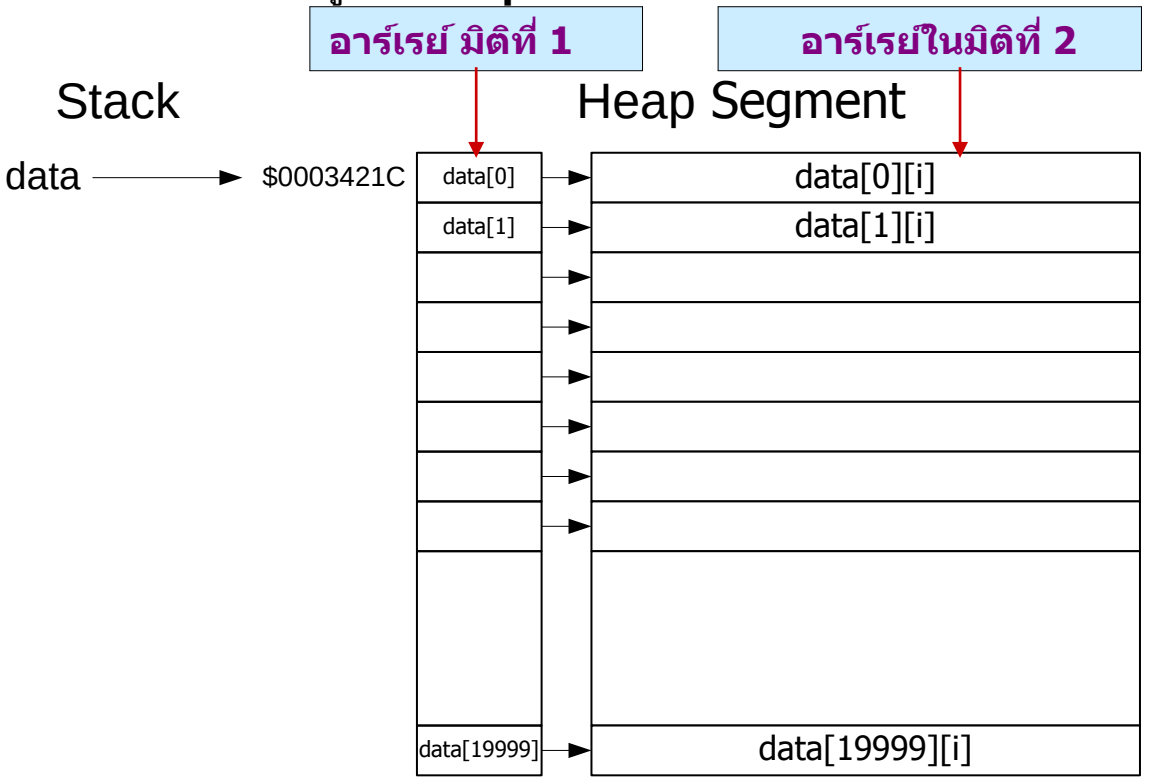

```
int main() {
char *str[20000]; //array of pointers to string
int  i;
     for (i = 0; i<20000; i++)
          str[i]  =  (char *)malloc(255*sizeof(char));
```

char str[20000][255];

Pointer to Pointer (2D Array)

- จอง pointer to pointer ที่จะชี้ไปยังสตริงไว้ก่อน
- malloc pointer to pointer เพื่อเป็นอาร์เรย์สำหรับ pointerของชนิดข้อมูลที่ต้องการ เพื่อใช้เป็นมิติแรก
- วนรอบ สร้างเนื้อที่ของอาร์เรย์มิติที่ 2 ของข้อมูล แล้วใช้ pointer ชี้มาที่หน่วยความจำส่วนนี้

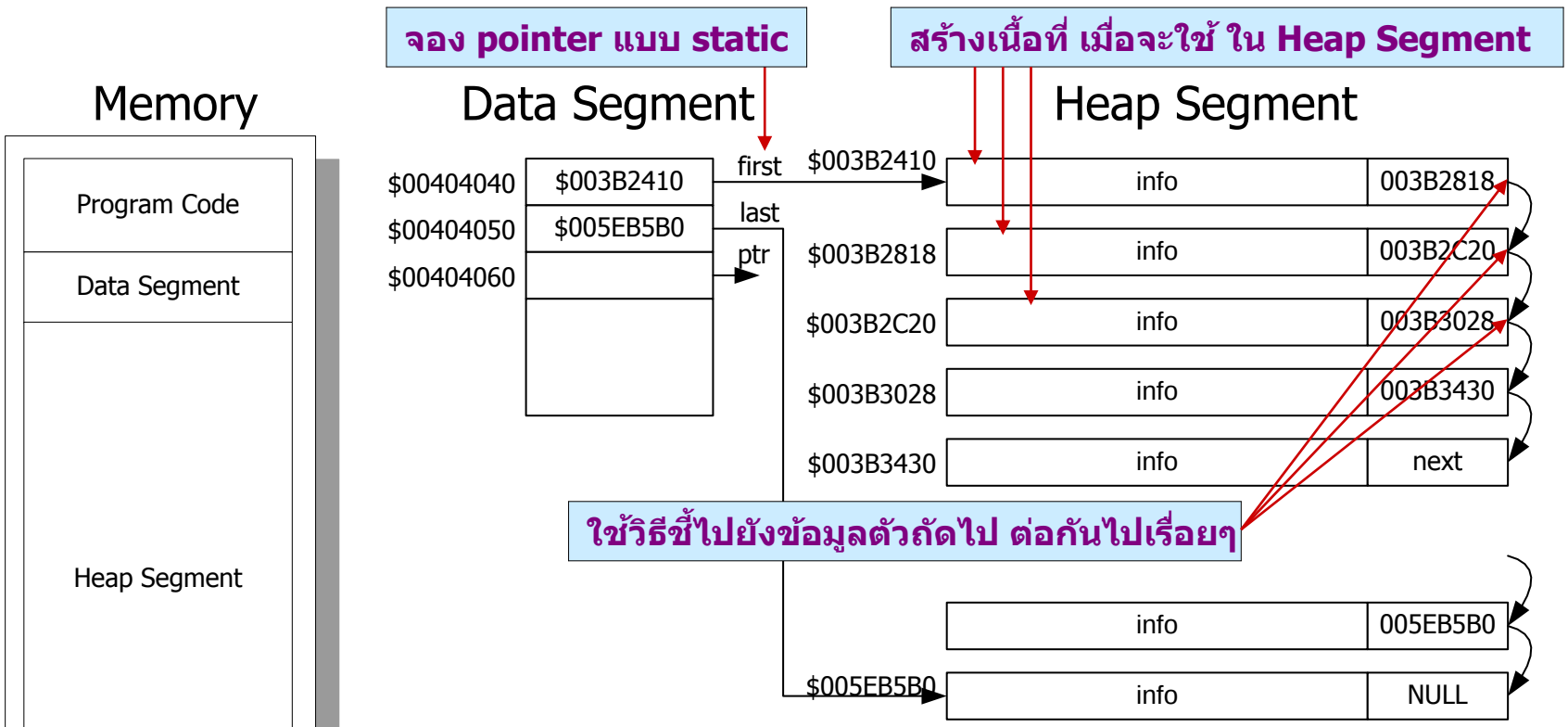





```
char **str; // pointer to pointer
        str = (char**)malloc(20000*sizeof(char*));//1st dimension
int  i;
        for (i = 0; i<20000; i++)
           str[i]  =  (char *)malloc(255*sizeof(char));//2nd dimension
```

Pointer & Linked List

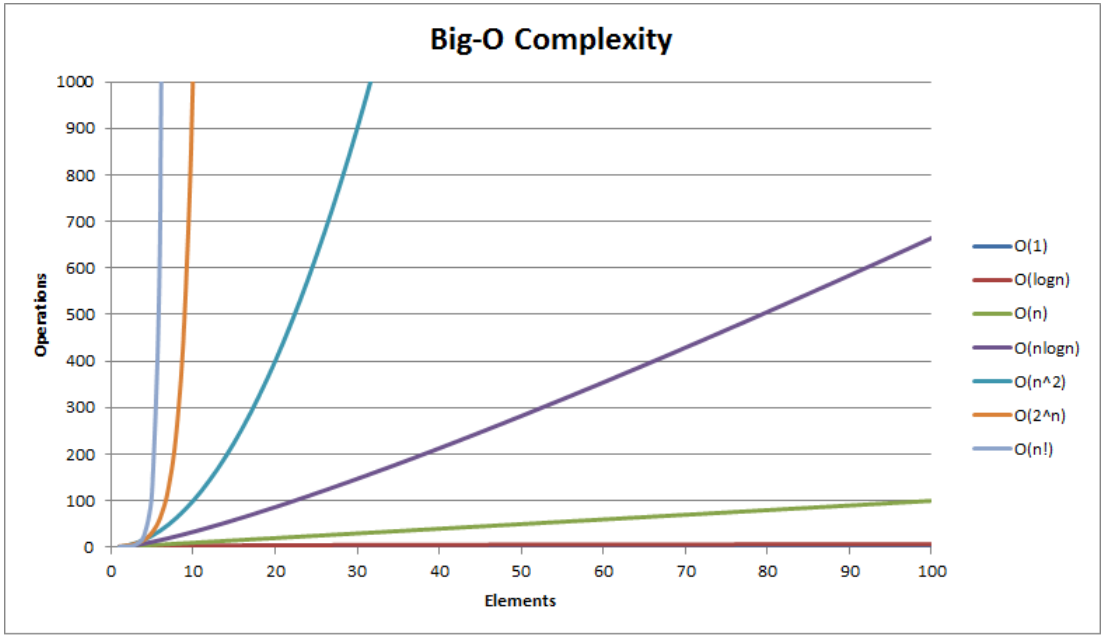
- โครงสร้างข้อมูลแบบ **linked list** จะต้องมี **pointer** ไว้ชี้ไปยังข้อมูลตัวที่อยู่ถัดไป
  - จองตัวแปรพอยต์เตอร์ไว้ สำหรับเก็บข้อมูลตัวแรก และ ตัวสุดท้าย
  - สร้างข้อมูลใน **Heap (Dynamic)** ทีละตัว แล้วใช้พอยต์เตอร์ชี้ต่อๆ กัน




```
typedef struct  { char id[15];    char info[1009]; } data_type;
typedef struct  node_type  { data_type info;
                             struct node_type *next;} linkedlist ;
linkedlist *first, *last, *ptr;
```

Complexity Analysis

- การวัดประสิทธิภาพ (ความซับซ้อน) ของอัลกอริธึม
  - วัดจากเวลาที่ใช้ประมวลผลจริง
  - วัดจากปริมาณหน่วยความจำที่ใช้
  - วัดจากจำนวนรอบของการประมวลผล
- Big-Oh O(g(n))
  - ขบวนการทางคณิตศาสตร์ ที่ให้คำตอบเป็นรูปฟังก์ชันพื้นฐาน
  - ใช้บอกความซับซ้อนของอัลกอริธึมที่ใช้ในการเขียนโปรแกรม
  - ใช้บอกความสัมพันธ์ระหว่าง ความเร็วของอัลกอริธึม กับจำนวนข้อมูล ว่ามีแนวโน้มเป็นอย่างไร เช่น O(1) , O(log n) , O(n) , O(n log n) , $O(n^2)$
  - อัลกอริธึมที่มี Big-Oh เท่ากัน ยังไม่สามารถบอกได้ว่าตัวใดดีกว่ากัน



Example

- ตัวอย่างการประเมินประสิทธิภาพอัลกอริธึม จากจำนวนรอบของคำสั่ง

`z = x;  x = y;  y = z; ...`	// จำนวนรอบ = 1	O(1)
`for (i=1, i<= n, i++) {.....}`	// จำนวนรอบ = c*n	O(n)
`for (i=1, i<= n, i=i+2) {.....}`	//จำนวนรอบ = c*(n/2)	O(n)
`for (i=1, i<= n, i=i*2) {.....}`	//จำนวนรอบ = c*$\log_2 n$	O($\log_2 n$)
`for (i=1; i<= n; i++) for (j=1; j<=n; j=j*2) {.....}`	//จำนวนรอบ = c*(n$\log_2 n$)	O(n $\log_2 n$)
`for (i=1; i<= n; i++) for (j=1; j<=n; j++) {.....}`	//จำนวนรอบ = c*(n*n)	O(n^2)
`for (i=1; i<= n; i++) for (j=i+1; j<=n; j++) {.....}`	//จำนวนรอบ = c*n*(n-1)/2	O(n^2)

ตัวนับลดขนาดลงครึ่งหนึ่ง

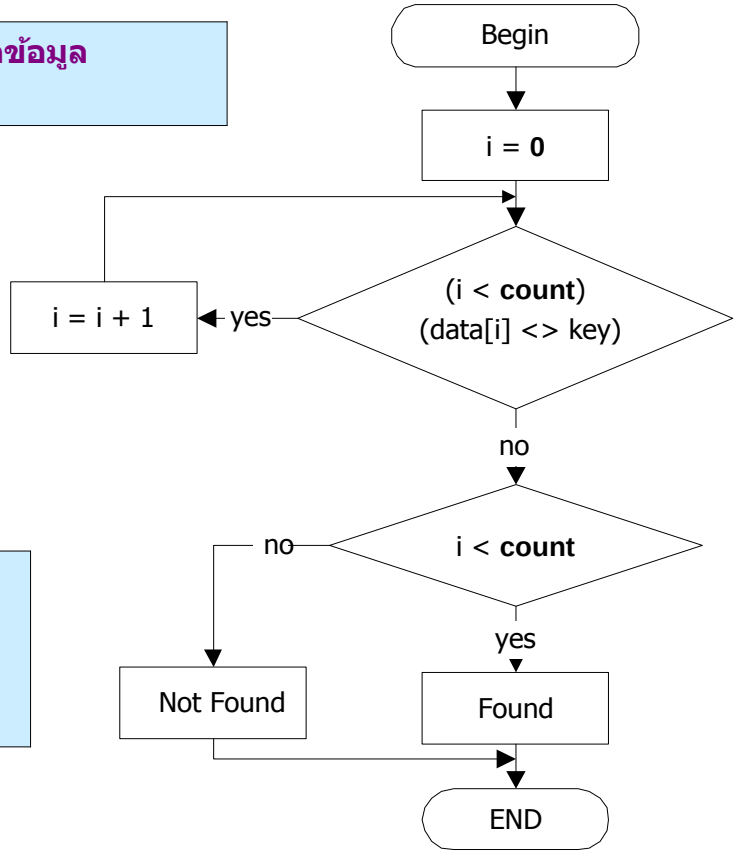
1. Linear Search(Unsorted)

- การค้นหาข้อมูลในอาร์เรย์ของตัวเลขจำนวนเต็ม ที่ไม่ได้เรียงลำดับ
  - สมมุติเก็บข้อมูลที่ตำแหน่ง 0 .. count -1
  - ให้ return ตำแหน่งที่เจอ และ return -1 ถ้าค้นไม่เจอ

```
int linear_search (int data[], int count, int key)
{ int i;
  i = 0;
  while ((i<count)&&(data[i] != key))
          i++;
   if (i < count)
       return i;
   else return -1;
}
```

เปรียบเทียบจนกว่าจะเจอ หรือจนกว่าจะหมดข้อมูล
for(i=0;i<count&&data[i]!=key;i++);

การหลุดออกจากวนรอบจะมีได้ 2 กรณีคือ
- เจอข้อมูลในตำแหน่งที่ i
  (เมื่อ data[i] == key หรือ i < count)
- ไม่เจอ(เมื่อ i >= count)



1.1 Improve Algorithm

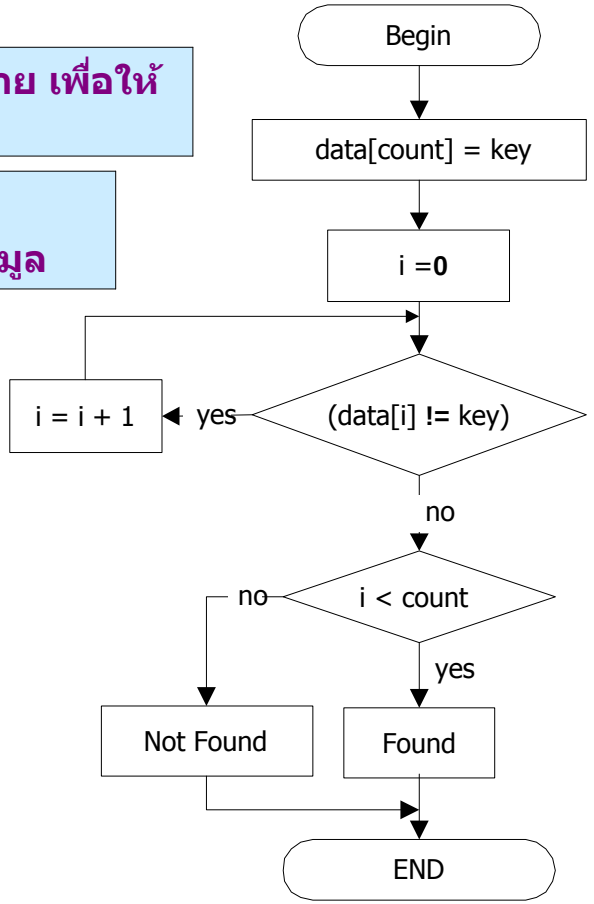
- ปรับปรุงเทคนิค เพื่อลดจำนวนเงื่อนไขที่ต้องเปรียบเทียบค้นหา

```
int dummy_search (int data[], int count, int key)
{int i;
  data[count] = key;
  i = 0;
  while (data[i] != key)
          i++;
  if (i < count)
     return i;
  else return -1;
}
```

เพิ่มข้อมูลตัวหลอกเข้าต่อท้าย เพื่อให้มั่นใจว่าจะต้องค้นเจอ

เมื่อมั่นใจว่าต้องค้นเจอ จึงไม่ต้องตรวจสอบจำนวนข้อมูล

ตรวจสอบว่าที่ค้นเจอเป็นตัวจริง หรือตัวหลอก



- ความซับซ้อนของอัลกอริธึม

ค่าเฉลี่ยเมื่อค้นเจอข้อมูล = (1+2+3+4+ ... + N)/N
= (N+1)/2
= *O(n)*

ค่าเฉลี่ยเมื่อค้นไม่เจอ = (N+1)
= *O(n)*

1.2 Linear Search (Sorted)

- ค้นหาข้อมูลในอาร์เรย์ของตัวเลขจำนวนเต็ม ที่เรียงลำดับแล้ว

```
int linear_search (int key)
{ int i;
  i = 0;
  while ( (i<count) && (data[i] < key) )
            i++;
  if (data[i]== key)
     return i ;
   else return -1;
}
```

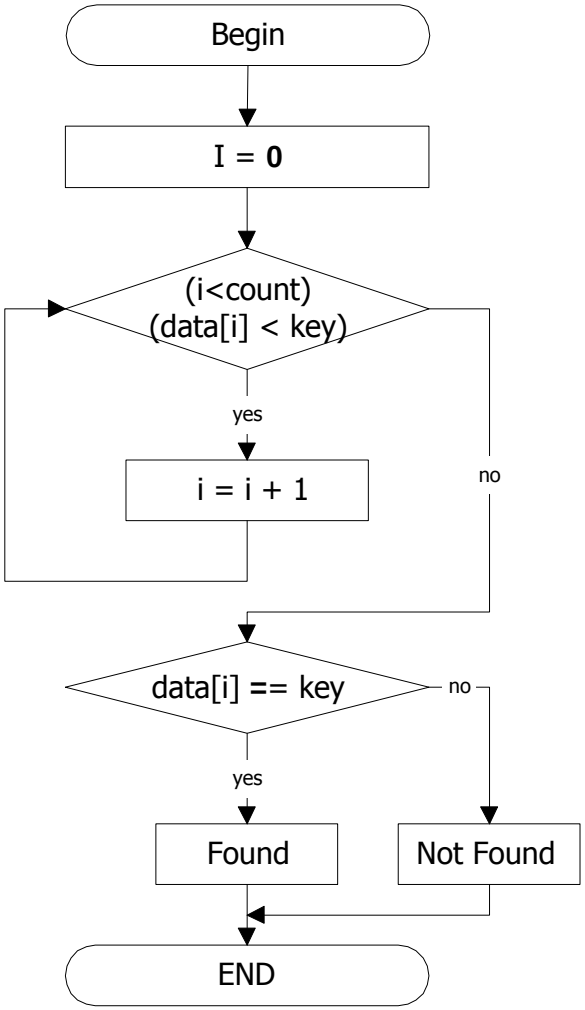
ถ้าค้นไม่เจอจะรู้ตัวเร็วขึ้น

ถ้าค้นเจอ

- ความซับซ้อนของอัลกอริธึม

ค่าเฉลี่ยเมื่อค้นเจอข้อมูล = (1+2+3+4+ ... + N)/N
= (N+1)/2
= *O(n)*

ค่าเฉลี่ยเมื่อค้นไม่เจอ = (1+2+3+4+ ... + N)/N
= (N+1)/2
= *O(n)*



Analysis of algorithm

- การวิเคราะห์หาความซับซ้อนของอัลกอริธึม
  - กรณีที่ค้นเจอข้อมูล

    ถ้าข้อมูลที่ต้องการค้น มีโอกาสเจออยู่ในทุกตำแหน่งเท่ากัน

    (ค้นครั้งที่ 1 เจอในตำแหน่งที่ 1, ค้นครั้งที่ 2 เจอในตำแหน่งที่ 2, ...)

    ค่าเฉลี่ยเมื่อค้นเจอข้อมูล n ครั้ง = (1+2+3+4+ ... + n)/n

    = (n+1)/2

    จะเห็นว่าเวลา(ความซับซ้อน)ที่ใช้ในการค้นหาข้อมูลแปรเปลี่ยนตามค่า n ดังนั้น

    ความซับซ้อนของอัลกอริธึม = *O(n)* *สมการเส้นตรง*

  - กรณีที่ค้นไม่เจอข้อมูล
    - กรณีไม่เรียงลำดับ โปรแกรมจะต้องค้นหาจนถึงตัวที่ n จึงจะรู้ว่าไม่เจอ

    เวลาเมื่อค้นไม่เจอข้อมูล = n

    ความซับซ้อนของอัลกอริธึม = *O(n)*
    - กรณีเรียงลำดับ โปรแกรมจะต้องค้นหาจนถึงตัวที่มากกว่า key จะรู้ว่าไม่เจอ

    เวลาเฉลี่ยเมื่อค้นไม่เจอข้อมูล = (1+2+3+4+ ... + n)/n

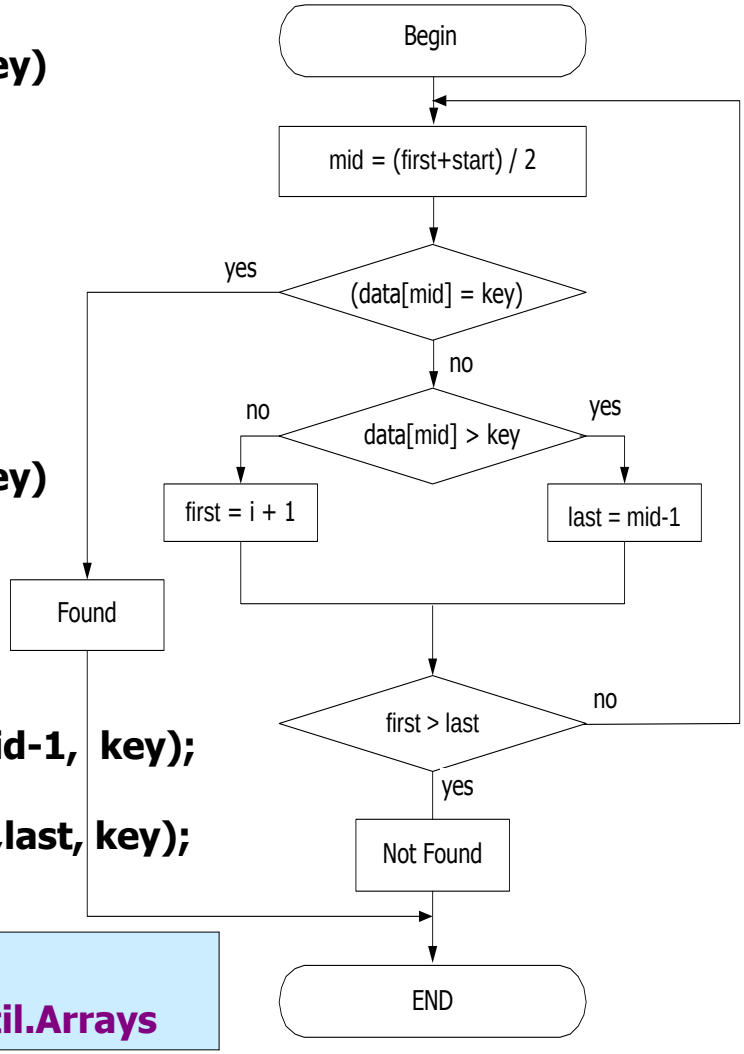
    = (n+1)/2

    ความซับซ้อนของอัลกอริธึม = *O(n)* *สมการเส้นตรง*

2. Binary Search

- ใช้เทคนิคแบ่งครึ่งข้อมูล แล้วเลือกค้นเฉพาะในกลุ่มที่คิดว่ามีข้อมูล
- การค้นข้อมูลแบบไบนารี ข้อมูลต้องเรียงลำดับเสมอ

```
int binary_search (int data[], int first, int last, int key)
{ int mid ;
  do {  mid = (first+last)/2;
        if  (data[mid]==key) return mid;
        else if (data[mid]>key)  last = mid-1;
             else  first = mid+1;
     } while (first<=last);
  return -1;
}
int binary_search (int data[], int first, int last, int key)
{ int mid ;
  if (first > last)  return -1;
     else { mid  = (first + last) / 2 ;
            if (data[mid] == key) return mid;
            else  if (data[mid] > key)
                     return binary_search(data, first, mid-1,  key);
                  else
                     return binary_search (data, mid+1,last, key);
          }
}
```



C: มีฟังก์ชัน bsearch() อยู่ใน <stdlib.h>
Java : มีเมธอด Arrays.binarySearch() ใน Java.util.Arrays

Analysis of algorithm

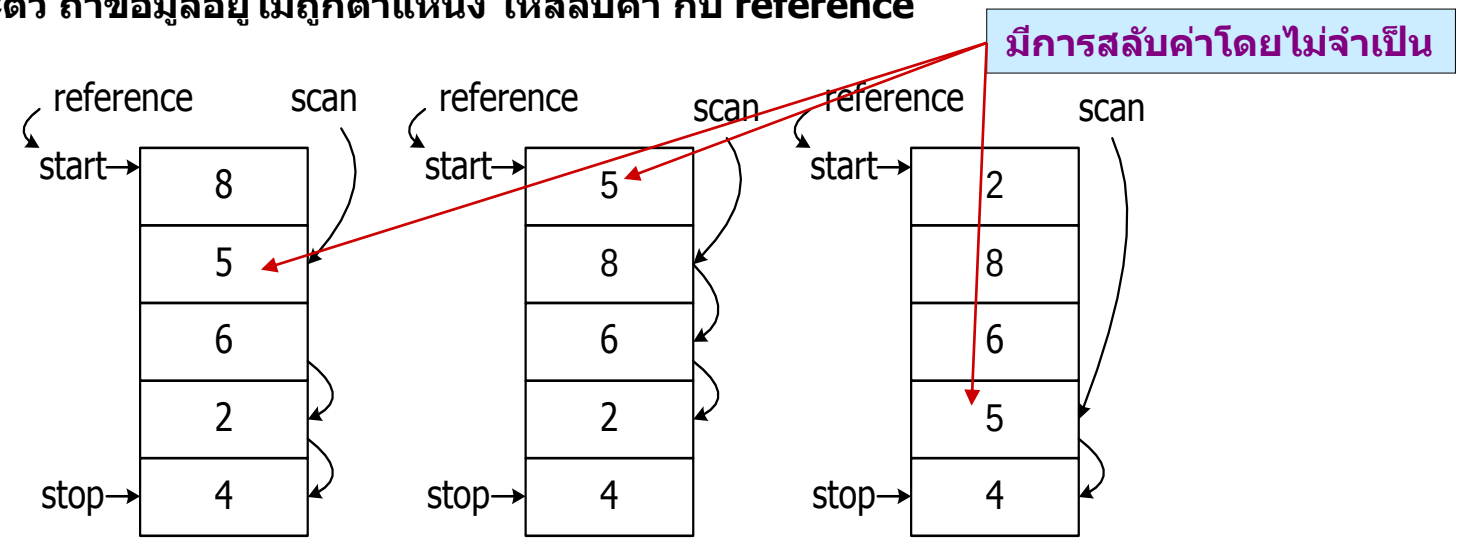
- การวิเคราะห์หาความซับซ้อนของอัลกอริธึม
  - ถ้าขนาดของข้อมูล n ลดลงครึ่งหนึ่งในทุกๆ รอบ
    - ขอบเขตในการค้นหาข้อมูล ครั้งที่ 1 = n
    - ขอบเขตในการค้นหาข้อมูล ครั้งที่ 2 = n/2
    - ขอบเขตในการค้นหาข้อมูล ครั้งที่ 3 = n/4
    - .....
    - ขอบเขตในการค้นหาข้อมูล ครั้งสุดท้าย = 1
    - จำนวนครั้งในการค้นหา = (1 + 1 + 1 + 1 + ... + 1) = $\log_2 n + 1$
  - ความซับซ้อนของอัลกอริธึม

    $T(n) = T(n/2) + O(1) = O(\log_2 n) + O(1)$

    $= O(\log_2 n)$

Sorting Data in Linear Arrays

- เรียงลำดับตัวเลขที่เก็บอยู่ในอาร์เรย์
  - เลือกข้อมูลตัวแรก(start) มาเป็น reference
    - เอาข้อมูลที่เหลือ (ตั้งแต่ reference+1 จนถึง stop) มาเปรียบเทียบกับ reference ทีละตัว ถ้าข้อมูลอยู่ไม่ถูกตำแหน่ง ให้สลับค่า กับ reference



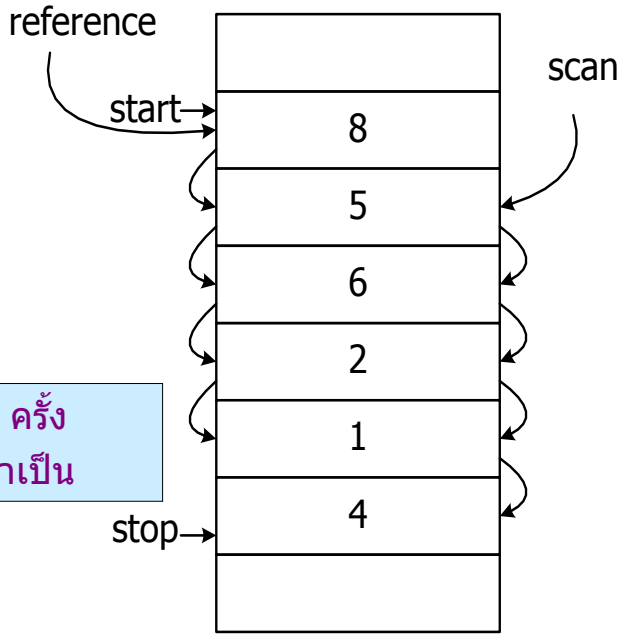
  - เลือกข้อมูลตัวถัดมา(ตัวที่สอง) มาเป็น reference แทนแล้วเปรียบเทียบกับข้อมูลตัวที่เหลือเช่นเดียวกับข้อมูลตัวแรก
- ทำเช่นเดิมจนกระทั่งครบทุกตัว ก็จะได้ข้อมูลที่เรียงลำดับ
- ความเร็วของการเรียงลำดับข้อมูลขึ้นอยู่กับ
  - จำนวนครั้งของการเปรียบเทียบข้อมูล
  - จำนวนครั้งของการย้ายข้อมูล (ข้อมูลที่มีจำนวนไบท์มากจะใช้เวลานานมากขึ้น)

Scan Sort

```
void scan_sort (int data[], int start, int stop)
{ int  i, j, x;
  for (i=start; i<=stop-1; i++) /* i = reference index */
    for (j=i+1; j<=stop; j++)   /* j = scan index */
      if (data[j] < data[i])          /* compare */
        swap(&data[i], &data[j]);     /* exchange */
}
```

```
void  swap(int *a, int *b)
{  int c;
      c = *a ; *a = *b ; *b = c ;
 }
```

มีการเปรียบเทียบ ∑(n-1) ครั้ง
และมีการสลับค่าโดยไม่จำเป็น



- การวิเคราะห์หาความซับซ้อนของอัลกอริธึม
  พิจารณาการวนรอบ ที่เกี่ยวกับจำนวนข้อมูล n
  มีรอบการวนรอบซ้อนกัน 2 ชั้น คือ
  วนรอบของ i ที่มีค่าตั้งแต่ 1 จนถึง n-1 มีจำนวน n - 1 ครั้ง
  ในแต่ละครั้งของ i จะมีวนรอบของ j จำนวน n-i รอบ
  จำนวนรอบที่ใช้เปรียบเทียบในการเรียงลำดับ
  = (n-1)+(n-2)+ … + 2 + 1 ที่ i=1, 2, 3, … , n-1
  = (n-1)*n/2
  = $O(n^2)$

Example 18 16 14 23 26 13 11 21 12 19

- ข้อมูล 10 ตัว ต้องเปรียบเทียบข้อมูล 45 ครั้ง สลับข้อมูลทั้งที่ไม่จำเป็น

loop 1 (Change=3)

16 18 14 23 26 13 11 21 12 19
14 18 16 23 26 13 11 12 21 19
14 18 16 23 26 13 11 12 21 19
14 18 16 23 26 11 13 12 21 19
11 18 16 23 26 14 13 12 21 19
11 18 16 23 26 14 13 12 21 19
11 18 16 23 26 14 13 12 21 19
11 18 16 23 26 14 13 12 21 19
11 18 16 14 23 26 13 12 21 19

loop 2(Change=4)

11 16 18 14 23 26 13 12 21 19
11 14 18 18 23 26 13 12 21 19
11 14 18 16 23 26 13 12 21 19
11 14 18 16 23 26 13 12 21 19
11 13 18 16 23 26 14 12 21 19
11 12 18 16 23 26 14 13 21 19
11 12 18 16 23 26 14 13 21 19
11 12 18 16 23 26 14 13 21 19

loop 3 (Change=3)

11 12 16 18 23 26 14 13 21 19
11 12 16 18 23 26 14 13 21 19
11 12 16 18 23 26 14 13 21 19
11 12 14 18 23 26 16 13 21 19
11 12 13 18 23 26 16 14 21 19
11 12 13 18 23 26 16 14 21 19
11 12 13 18 23 26 16 14 21 19

loop 4 (Change=2)

11 12 13 16 23 26 18 14 21 19
11 12 13 14 23 26 18 16 21 19
11 12 13 14 23 26 18 16 21 19

loop 6 (Change=2)

11 12 13 14 16 23 26 18 21 19
11 12 13 14 16 18 26 23 21 19
11 12 13 14 16 18 26 23 21 19

loop 8(Change=1)

11 12 13 14 16 18 19 21 23 26

loop 9(Change=0)

11 12 13 14 16 18 19 21 23 26

loop 5(Change=2)

11 12 13 14 23 26 16 18 21 19
11 12 13 14 16 26 23 18 21 19
11 12 13 14 16 26 23 18 21 19

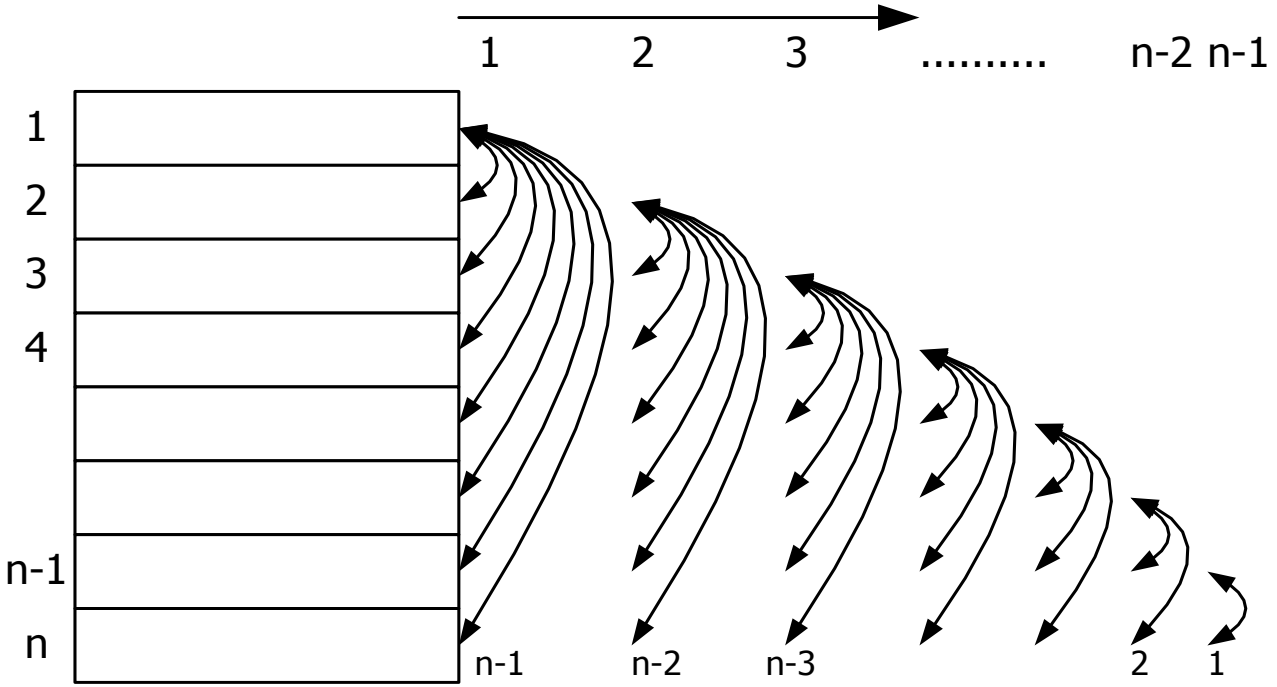
loop 7(Change=1)

11 12 13 14 16 18 19 23 21 26

เปรียบเทียบข้อมูล 45 ครั้ง
ย้ายข้อมูล 18*3=54 ครั้ง

1. Selection Sort

- ค้นหาข้อมูลตัวที่มีค่าน้อยที่สุดที่มีอยู่(กรณีต้องการเรียงลำดับจากน้อยไปมาก) ในแต่ละรอบออกมาเพื่อรอสลับค่า
- จำนวนครั้งในการเปรียบเทียบในแต่ละรอบเท่าเดิม แต่จำนวนครั้งในการสลับค่าเหลือรอบละครั้งเดียว จะได้ข้อมูลที่เรียงแล้วรอบละ 1 ตัว
- ทำซ้ำจนกว่าข้อมูลจะหมด



Selection Sort Method

- เรียงลำดับข้อมูลตัวเลขจำนวนเต็ม จากน้อยไปมากในอาร์เรย์ของ data[]

```
void  selection_sort (int data[], int start, int stop)
{ int i, j, min;
  for (i = start; i < stop; i++)          /* i = Reference */
      { min = i;                          /* k = smallest data position */
        for ( j = i+1; j <= stop; j++)
          { if (data[j] < data[min])
                min = j;  }     /* Keep smallest position */
        swap(&data[min], &data[i)] ; /* exchange data */
      }
}
```

จำตำแหน่งที่มีค่าน้อยที่สุดในแต่ละรอบไว้

มีการวนรอบเปรียบเทียบ ∑(n-1) รอบ
สลับค่า n-1 ครั้ง (รอบละ 1 ครั้ง)

สลับข้อมูลตัวอ้างอิงกับตัวที่มีค่าน้อยที่สุด
สลับค่ารอบละ 1 ครั้งเท่านั้น

- การวิเคราะห์หาความซับซ้อนของอัลกอริธึม
  - วนรอบของ i ที่มีค่าตั้งแต่ 1 จนถึง n-1 มีจำนวน n - 1 ครั้ง
    ในแต่ละครั้งของ i จะมีเปรียบเทียบข้อมูลโดยใช้ j ไม่เกิน n-i รอบ
  - จำนวนรอบเปรียบเทียบสูงสุด ที่ใช้ในการเรียงลำดับ
    = (n-1)+(n-2)+ … + 2 + 1 ที่ i=1, 2, 3, … , n-1
    = (n-1)*n/2
    = $O(n^2)$

Example 18 16 14 23 26 13 11 21 12 19

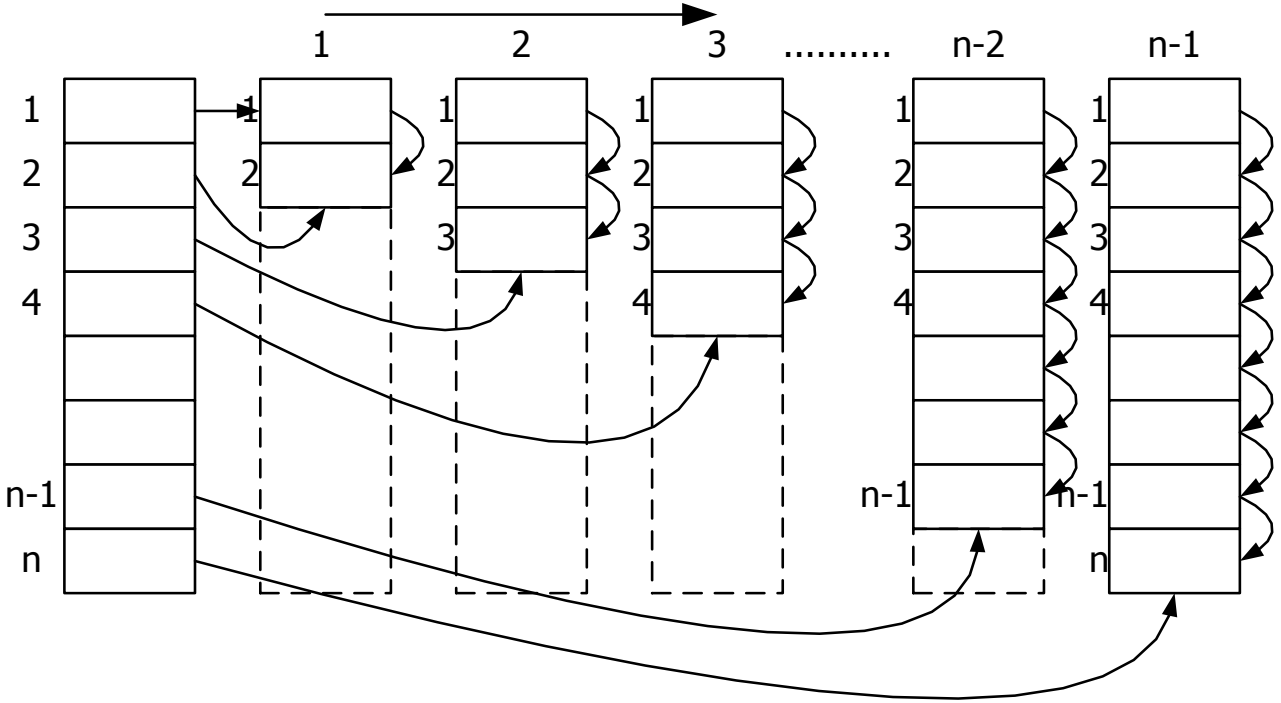
- เปรียบเทียบข้อมูล 45 ครั้ง แต่สลับข้อมูลเพียงรอบละ 1 ครั้ง

loop	start	Min	End
1	18 16 14 23 26 13 11 21 12 19	7	11 16 14 23 26 13 18 21 12 19
2	11 16 14 23 26 13 18 21 12 19	9	11 12 14 23 26 13 18 21 16 19
3	11 12 14 23 26 13 18 21 16 19	6	11 12 13 23 26 14 18 21 16 19
4	11 12 13 23 26 14 18 21 16 19	6	11 12 13 14 26 23 18 21 16 19
5	11 12 13 14 26 23 18 21 16 19	9	11 12 13 14 16 23 18 21 26 19
6	11 12 13 14 16 23 18 21 26 19	7	11 12 13 14 16 18 23 21 26 19
7	11 12 13 14 16 18 23 21 26 19	10	11 12 13 14 16 18 19 21 26 23
8	11 12 13 14 16 18 19 21 26 23	8	11 12 13 14 16 18 19 21 26 23
9	11 12 13 14 16 18 19 21 26 23	10	11 12 13 14 16 18 19 21 23 26

เปรียบเทียบข้อมูล 45 ครั้ง
ย้ายข้อมูล 9*3 =27 ครั้ง

2. Insertion Sort

- ดึงข้อมูลออกมาเพื่อเรียงลำดับทีละตัว โดยการค้นหาตำแหน่งที่เหมาะสม(ให้เกิดการเรียงลำดับ) สำหรับเก็บข้อมูลตัวนั้น แล้วนำข้อมูลตัวนั้นแทรกลงในตำแหน่งที่หาได้
- ใช้วิธีเลื่อนหาตำแหน่งข้อมูลที่เหมาะสมแทนการสลับค่า
- ทำซ้ำจนกว่าข้อมูลจะหมด



Insertion Sort Method

- เรียงลำดับข้อมูลตัวเลขจำนวนเต็ม จากน้อยไปมากในอาร์เรย์ของ data[]

```
void  insertion_sort (int data[], int start, int stop)
{ int i, j ;
  for (i = start+1; i <= stop; i++)    /* Reference */
    { x = data[i];              /* Selected data */
      for (j = i;((j>start)&&(x < data[j-1])); j--)
          data[j] = data[j-1]; /* move down data */
      data[j] = x;             /* Insert data */
    }
}
```

ตัวแรกเป็นตัวอ้างอิง เริ่มแทรกตั้งแต่ตัวที่ 2

เลื่อนข้อมูลในอาร์เรย์ลงมา 1 ตำแหน่งจนกว่าจะเจอตำแหน่งที่เหมาะสม แล้วแทรกข้อมูลลงไป

- การวิเคราะห์หาความซับซ้อนของอัลกอริธึม
  - วนรอบของ i ที่มีค่าตั้งแต่ 2 จนถึง n มีจำนวน n - 1 ครั้ง
    ในแต่ละครั้งของ i จะมีเปรียบเทียบข้อมูลโดยใช้ j ไม่เกิน i-1 รอบ
  - จำนวนรอบเปรียบเทียบสูงสุด ที่ใช้ในการเรียงลำดับ
    = 1 + 2 + ... + (n-2)+(n-1) ที่ i = 2, 3, 4, ... , n
    = (n-1)*n/2
    = $O(n^2)$

มีการวนรอบเปรียบเทียบ ∑(n-1) รอบ
มีการย้ายค่า(Insert) ไม่เกิน ∑(n-1) ครั้ง

Example 18 16 14 23 26 13 11 21 12 19

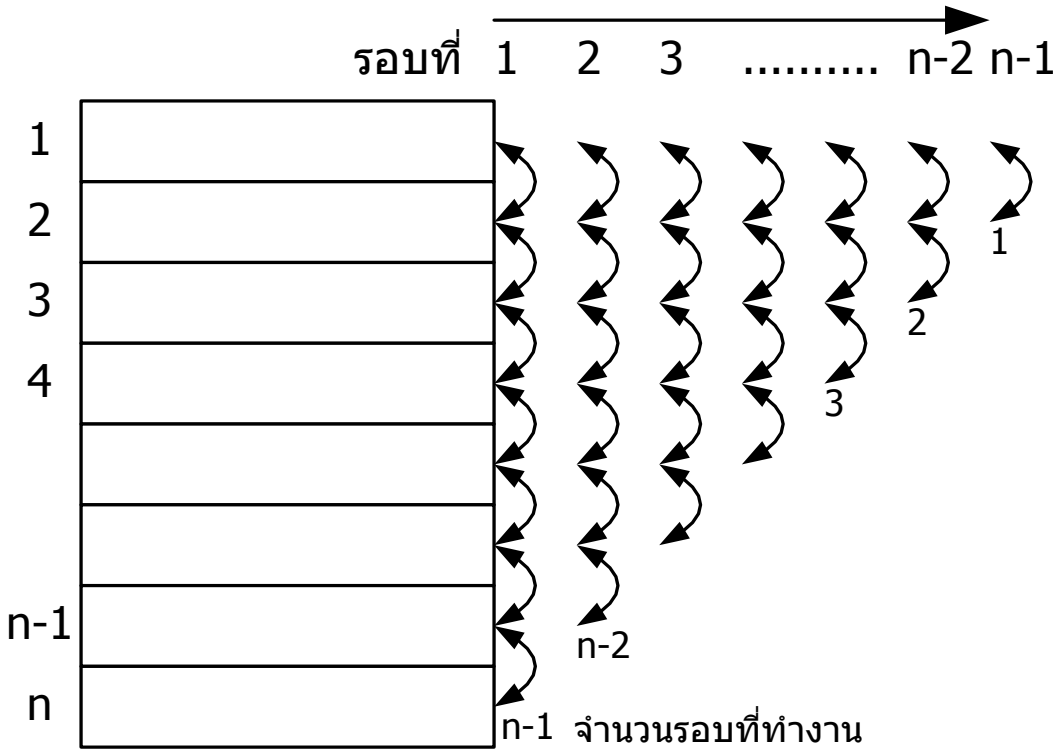
- ย้ายข้อมูลเท่าที่จำเป็น แต่ขยับทีละ 1 ตำแหน่ง

	Select data[i]	Sorted	
2	18 16 14 23 26 13 11 21 12 19	16 18 14 23 26 13 11 21 12 19	1+2
3	16 18 14 23 26 13 11 21 12 19	14 16 18 23 26 13 11 21 12 19	2+2
4	14 16 18 23 26 13 11 21 12 19	14 16 18 23 26 13 11 21 12 19	0+2
5	14 16 18 23 26 13 11 21 12 19	14 16 18 23 26 13 11 21 12 19	0+2
6	14 16 18 23 26 13 11 21 12 19	13 14 16 18 23 26 11 21 12 19	5+2
7	13 14 16 18 23 26 11 21 12 19	11 13 14 16 18 23 26 21 12 19	6+2
8	11 13 14 16 18 23 26 21 12 19	11 13 14 16 18 21 23 26 12 19	2+2
9	11 13 14 16 18 21 23 26 12 19	11 12 13 14 16 18 21 23 26 19	7+2
10	11 12 13 14 16 18 21 23 26 19	11 12 13 14 16 18 19 21 23 26	3+2

ย้ายข้อมูล 26+18=44 ครั้ง

3. Bubble Sort

- เปรียบเทียบข้อมูลคู่ที่อยู่ติดกันไปเรื่อยๆ ถ้าพบว่าข้อมูลคู่ใดมีลำดับที่อยู่ไม่ถูกต้อง ให้สลับตำแหน่งกัน ทำซ้ำจนกว่าจะเรียงลำดับเสร็จ
- การสลับข้อมูลไม่เสียเปล่า
- อาจเรียงลำดับเสร็จก่อนถึงการวนรอบสุดท้าย



Bubble Sort Method

- เรียงลำดับข้อมูลตัวเลขจำนวนเต็ม จากน้อยไปมากในอาร์เรย์ของ data[]

เปรียบเทียบเพื่อสลับตำแหน่งข้อมูลตัวที่อยู่ติดกัน

```
void  bubbleSort (int data[], int start, int stop)
{ int i, j;
  for (i = start; i <= stop-1; i++)
    for (j = stop; j > i; j--)
       if (data[j] < data[j-1])
          swap(&data[j], &dat[j-1]);
}
```

```
void  swap(int *a, int *b)
{  int c;
       c = *a; *a = *b; *b = c; }
```

ถ้าต้องการให้หลุดออกเมื่อไม่มีการสลับค่า

```
int doMore = 1;
  for (i = start; i <= stop-1 && doMore; i++) {
       doMore = 0;
       for (j = stop; j > i; j--)
            if (data[j] < data[j-1]) {
                swap(&data[j] ,&data[j-1]);
                doMore = 1;  } //end if
  }  // end for
}
```

- การวิเคราะห์หาความซับซ้อนของอัลกอริธึม

  พิจารณาการวนรอบ ที่เกี่ยวกับจำนวนข้อมูล n

  มีรอบการวนรอบซ้อนกัน 2 ชั้น คือ

  วนรอบของ i ที่มีค่าตั้งแต่ 1 จนถึง n-1 มีจำนวน n - 1 ครั้ง

  ในแต่ละครั้งของ i จะมีวนรอบของ j จำนวน n-i รอบ

  จำนวนรอบที่ใช้เปรียบเทียบในการเรียงลำดับ

  = (n-1)+(n-2)+ … + 2 + 1 ที่ i=1, 2, 3, … , n-1

  = (n-1)*n/2

  = $O(n^2)$

Example 18 16 14 23 26 13 11 21 12 19

- ขยับข้อมูลทีละ 1 ตำแหน่ง ขยับโดยไม่จำเป็น ทำเสร็จก่อนกำหนดได้

loop 1 (Change=7)

18 16 14 23 26 13 11 21 12 19
18 16 14 23 26 13 11 12 21 19
18 16 14 23 26 13 11 12 21 19
18 16 14 23 26 11 13 12 21 19
18 16 14 23 11 26 13 12 21 19
18 16 14 11 23 26 13 12 21 19
18 16 11 14 23 26 13 12 21 19
18 11 16 14 23 26 13 12 21 19
11 18 16 14 23 26 13 12 21 19

loop 2(Change=7)

11 18 16 14 23 26 13 12 19 21
11 18 16 14 23 26 13 12 19 21
11 18 16 14 23 26 12 13 19 21
11 18 16 14 23 12 26 13 19 21
11 18 16 14 12 23 26 13 19 21
11 18 16 12 14 23 26 13 19 21
11 18 12 16 14 23 26 13 19 21
11 12 18 16 14 23 26 13 19 21

loop 3 (Change=5)

11 12 18 16 14 23 26 13 19 21
11 12 18 16 14 23 26 13 19 21
11 12 18 16 14 23 13 26 19 21
11 12 18 16 14 13 23 26 19 21
11 12 18 16 13 14 23 26 19 21
11 12 18 13 16 14 23 26 19 21
11 12 13 18 16 14 23 26 19 21

loop 4 (Change=4)

11 12 13 18 16 14 23 26 19 21
11 12 13 18 16 14 23 19 26 21
11 12 13 18 16 14 19 23 26 21
11 12 13 18 16 14 19 23 26 21
11 12 13 18 14 16 19 23 26 21
11 12 13 14 18 16 19 23 26 21

loop 5(Change=3)

11 12 13 14 18 16 19 23 21 26
11 12 13 14 18 16 19 21 23 26
11 12 13 14 18 16 19 21 23 26
11 12 13 14 18 16 19 21 23 26
11 12 13 14 16 18 19 21 23 26

loop 6 (Unchange)

11 12 13 14 16 18 19 21 23 26

loop 7 (Unchange)

11 12 13 14 16 18 19 21 23 26

loop 8(Unchange)

11 12 13 14 16 18 19 21 23 26

loop 9 (Unchange)

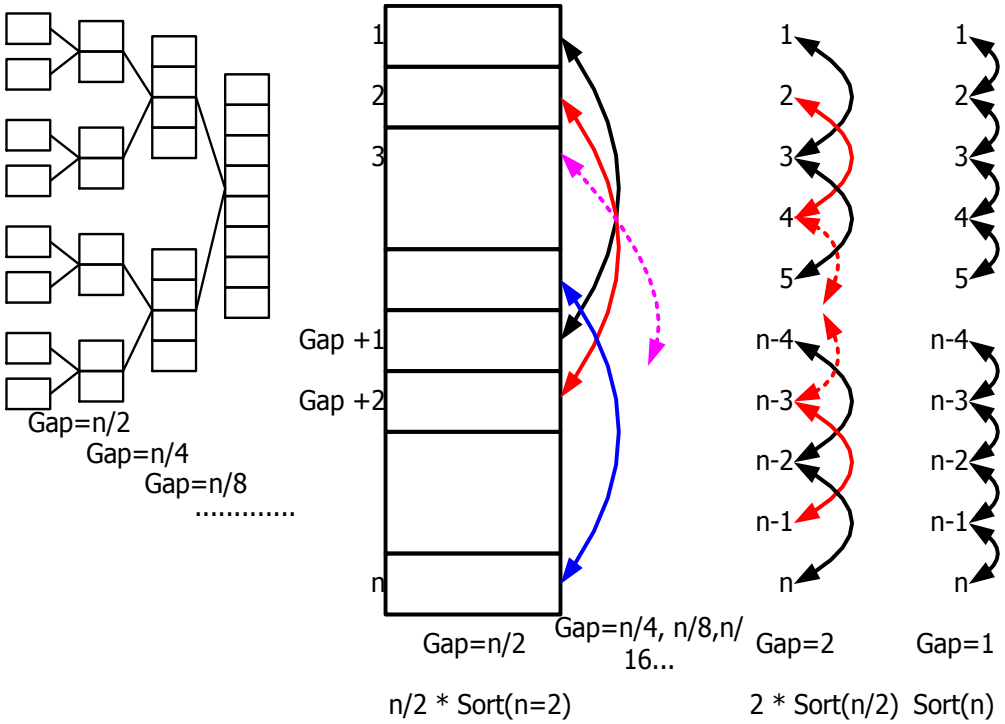
11 12 13 14 16 18 19 21 23 26

มีการย้ายข้อมูล 26*3 = 78 ครั้ง

- ข้อสังเกต โปรแกรมเริ่มไม่มีการสลับค่าตั้งแต่รอบที่ 6*(Unchange)*

4. Shell Sort

- เปรียบเทียบข้อมูลคู่ที่อยู่ห่างกันช่วงหนึ่งไปเรื่อยๆ แล้วสลับค่า เพื่อให้ข้อมูลกระโดดไปยังตำแหน่งที่เหมาะสมได้เร็วกว่า **Bubble Sort**
- ลดช่วงห่างการกระโดดลงครึ่งหนึ่งไปเรื่อยๆ จนกว่าจะเรียงลำดับเสร็จ
- รอบแรกจะมีข้อมูล **n/2** กลุ่ม(เท่ากับ **Gap**) กลุ่มละ **2** ตัว รอบถัดๆไป จำนวนกลุ่มจะลดลงครึ่งหนึ่ง ขณะที่จำนวนข้อมูลในกลุ่มเพิ่มขึ้นเป็น **2** เท่า
- รอบสุดท้ายจะเหลือกลุ่มเดียว (เหมือน **bubble Sort**) แต่จะเรียงลำดับเสร็จก่อนที่จะวนรอบเสร็จ



Shell Sort Method

```
void shellSort ()
{ int gap, changed, i;
  gap = count;
  do { gap = gap/2;
       do { changed = 0;
            for (i =0; i < count-gap; i++)
              if (data[i] > data[i+gap])
                { swap (&data[i], &data[i+gap])
                  changed = 1; }
       } while(changed == 1);
  } while(gap >1);
}
```

กำหนดระยะห่างของข้อมูล (จำนวนกลุ่ม)

ใช้ตัวแปร changed เพื่อตรวจสอบการสลับค่า

ถ้ามีการสลับค่า แสดงว่ายังเรียงไม่เสร็จ ให้วนรอบทำจนกว่าจะไม่มีการสลับ

วนรอบทำจนกว่าจะเหลือกลุ่มเดียว

Example 18 16 14 23 26 13 11 21 12 19

- รอบที่ 1 เปรียบเทียบข้อมูล 10 ครั้ง

Set Gap = h = n/2 = 5, changed=0										
18	16	14	23	26	13	11	21	12	19	
1	2	3	4	5	h+1	h+2	h+3	h+4	h+5	
13	16	14	23	26	18	11	21	12	19	changed=1
13	11	14	23	26	18	16	21	12	19	changed=1
13	11	14	23	26	18	16	21	12	19	
13	11	14	12	26	18	16	21	23	19	changed=1
13	11	14	12	19	18	16	21	23	26	changed=1
Resume, Gap = 5, changed =0										
13	11	14	12	19	18	16	21	23	26	changed=0, Done

Gap = 2

- รอบที่ 2 เปรียบเทียบข้อมูล 16 ครั้ง

Set Gap = 5/2 = 2, changed =0										
13	11	14	12	19	18	16	21	23	26	
13	11	14	12	19	18	16	21	23	26	
13	11	14	12	19	18	16	21	23	26	
13	11	14	12	19	18	16	21	23	26	
13	11	14	12	16	18	19	21	23	26	changed=1
13	11	14	12	16	18	19	21	23	26	
13	11	14	12	16	18	19	21	23	26	
13	11	14	12	16	18	19	21	23	26	
Resume, Gap = 2, changed =0										
13	11	14	12	16	18	19	21	23	26	changed=0, Done

Gap = 1

- รอบที่ 3 เปรียบเทียบข้อมูล 18 ครั้ง

Gap =2/2 = 1 , changed =0										
11	13	14	12	16	18	19	21	23	26	
11	13	14	12	16	18	19	21	23	26	
11	13	12	14	16	18	19	21	23	26	changed=1
11	13	12	14	16	18	19	21	23	26	
11	13	12	14	16	18	19	21	23	26	
11	13	12	14	16	18	19	21	23	26	
11	13	12	14	16	18	19	21	23	26	
11	13	12	14	16	18	19	21	23	26	
11	13	12	14	16	18	19	21	23	26	

Changed = 0

- รอบที่ 4 เปรียบเทียบข้อมูล 18 ครั้ง

Resume, Gap = 1, changed =0										
11	**13**	12	14	16	18	19	21	23	26	
11	**12**	**13**	14	16	18	19	21	23	26	changed=1
11	12	**13**	**14**	16	18	19	21	23	26	
11	12	13	**14**	**16**	18	19	21	23	26	
11	12	13	14	**16**	**18**	19	21	23	26	
11	12	13	14	16	**18**	**19**	21	23	26	
11	12	13	14	16	18	**19**	**21**	23	26	
11	12	13	14	16	18	19	**21**	**23**	26	
11	12	13	14	16	18	19	21	**23**	**26**	
Resume, Gap = 1, changed =0										
11	12	13	14	16	18	19	21	23	26	changed=0, Done

มีการย้ายข้อมูล 7*3=21 ครั้ง