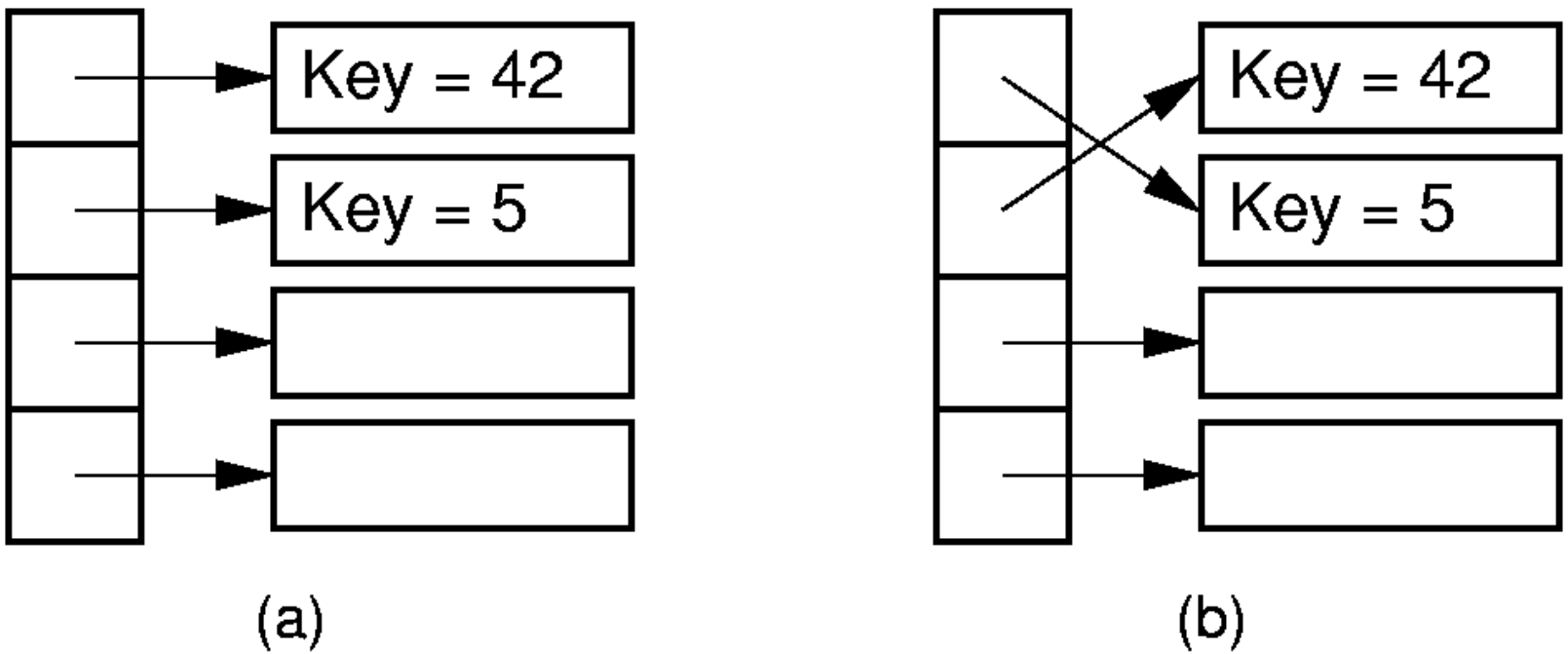
Design and Analysis of Data Structures and Algorithms :: Sorting part 2

อ.ดร.วรินทร์ วัฒนพรพรหม

เปรียบเทียบ Sorting Algorithms

Algorithm	Exact time	Complexity
Bubble Sort	$n * (n/2) * k$	n^2
Selection Sort	$(n-1) * (n/2)$	n^2
Insertion Sort	$n^2/4$	n^2

Selection Sort - Pointer Swapping



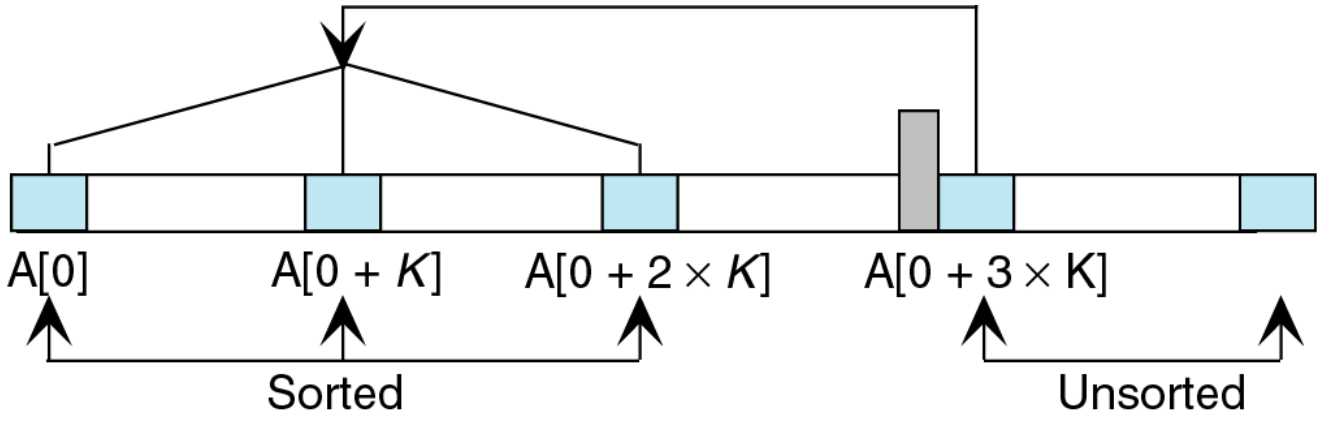
การใช้กระดาษทดที่ใหญ่ขึ้น

Shellsort

- Shellsort ตั้งตามชื่อของผู้คิดค้นการจัดเรียงแบบนี้ คือ Donald Shell และเป็นอัลกอริทึมแรกที่ทำลายขอบเขตเวลาที่เป็น quadratic
- ในขณะทำงานแต่ละ phase นั้น shell sort ใช้การเปรียบเทียบค่าที่อยู่ในตำแหน่งที่ห่างกัน ระยะห่างดังกล่าวนี้จะลดลงลงเรื่อย ๆ จนกระทั่งถึงขั้นตอนสุดท้ายที่เป็นการเปรียบเทียบค่าที่อยู่ติดกัน
  - ด้วยเหตุที่ระยะห่างของค่าที่นำมาเปรียบเทียบกันลดลงในระหว่างการทำงานของอัลกอริทึมนี้เอง จึงเรียก Shellsort อีกอย่างว่า *diminishing increment sort*

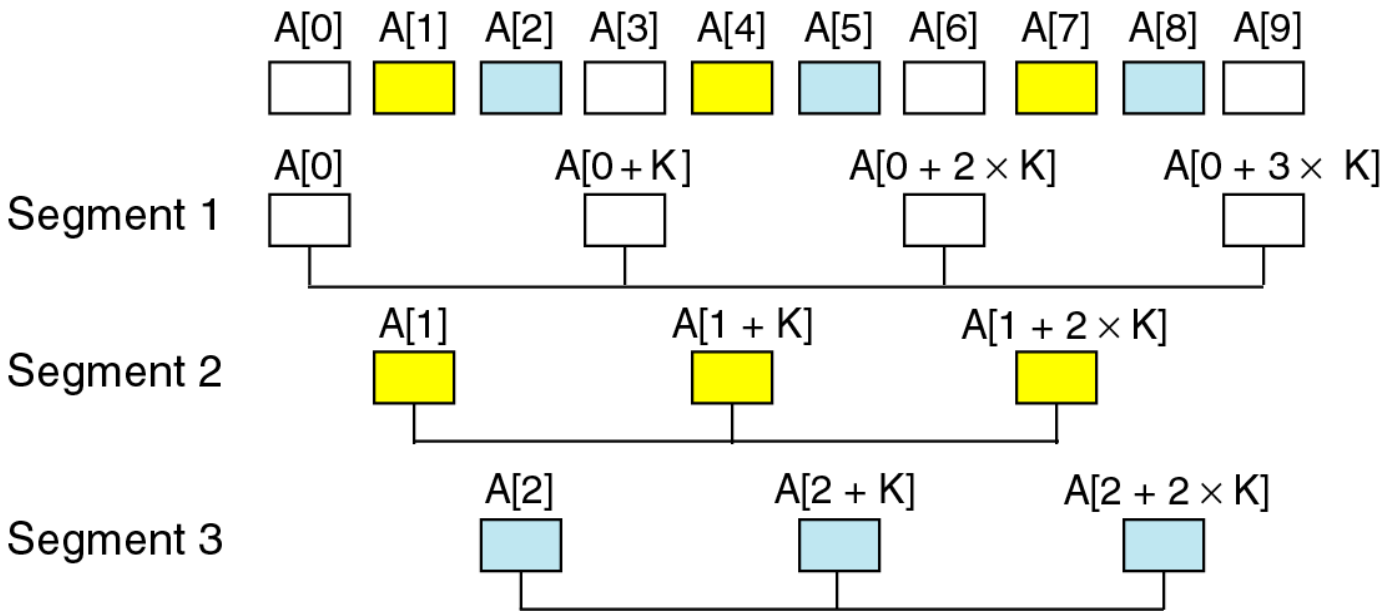
Shell Sort

- การจัดเรียงแบบเชลล์ เป็นการจัดเรียงที่อาศัยเทคนิคการแบ่งข้อมูลออกเป็นกลุ่มย่อยหลายๆ กลุ่ม แล้วจัดเรียงข้อมูลในกลุ่มย่อยๆ นั้น หลังจากนั้นก็ให้รวมกลุ่มย่อยๆ ให้ใหญ่ขึ้นเรื่อยๆ ขั้นสุดท้ายให้จัดเรียงข้อมูลทั้งหมดนั้นอีกครั้ง



Shell Sort

ตัวอย่าง กำหนด Segment เมื่อ K=3

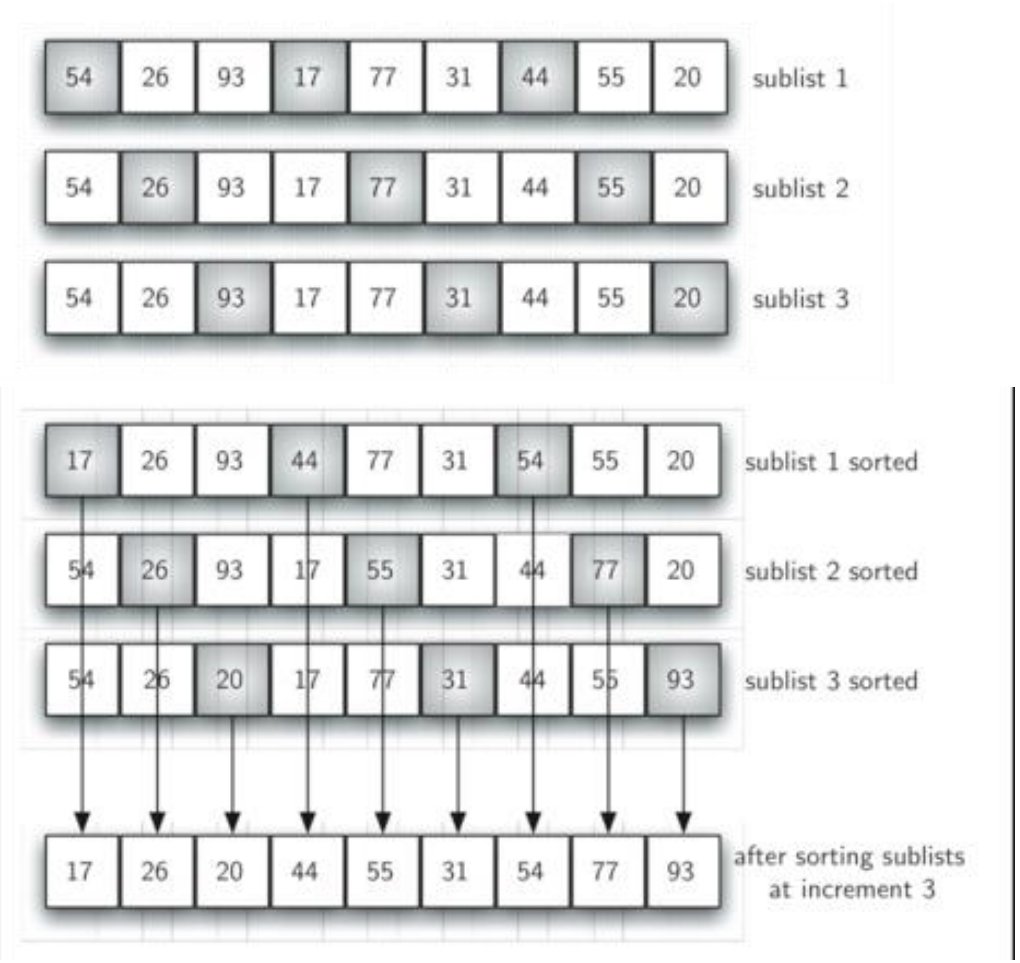


Shell Sort

ขั้นตอน

- โดยทั่วไปการเลือกค่า K ตัวแรกมักจะเลือกใช้ค่าเท่ากับครึ่งหนึ่งของข้อมูล เช่น ข้อมูลมี 10 ตัว K = n/2 = 10/2 = 5
- เรียงข้อมูลทุกตัวให้เสร็จสิ้น แล้วกำหนดค่า K ใหม่ (โดยทั่วไปจะเป็นครึ่งหนึ่งของค่า K ตัวแรก เช่น K1 = 5; K2 = 5/2 = 2)
- ถ้า K > 1 ให้ทำซ้ำ จนกระทั่งเหลือข้อมูลกลุ่มเดียว ถ้า K = 1 ให้เรียงลำดับตามปกติ

Shell Sort


54 26 93 17 77 31 44 55 20
sublist 1
54 26 93 17 77 31 44 55 20
sublist 2
54 26 93 17 77 31 44 55 20
sublist 3
17 26 93 44 77 31 54 55 20
sublist 1 sorted
54 26 93 17 55 31 44 77 20
sublist 2 sorted
54 26 20 17 77 31 44 55 93
sublist 3 sorted
17 26 20 44 55 31 54 77 93
after sorting sublists
at increment 3

Shell Sort

- **คุณสมบัติที่สำคัญของ** Shellsort **คือ การทำ** h_k-sorted **แล้วตามด้วย** h_{k-1}-sorted **นั้นยังคงสภาพของ** h_k-sorted

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95

Figure Shellsort หลังการทำงานแต่ละ pass

Shell Sort

- **คุณสมบัติที่สำคัญของ** Shellsort **คือ การทำ** h_k-sorted **แล้วตามด้วย** h_{k-1}-sorted **นั้นยังคงสภาพของ** h_k-sorted

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95

Figure Shellsort หลังการทำงานแต่ละ pass

Shell Sort

- **คุณสมบัติที่สำคัญของ** Shellsort **คือ การทำ** h_k-sorted **แล้วตามด้วย** h_{k-1}-sorted **นั้นยังคงสภาพของ** h_k-sorted

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

Figure Shellsort **หลังการทำงานแต่ละ** pass

Shell Sort

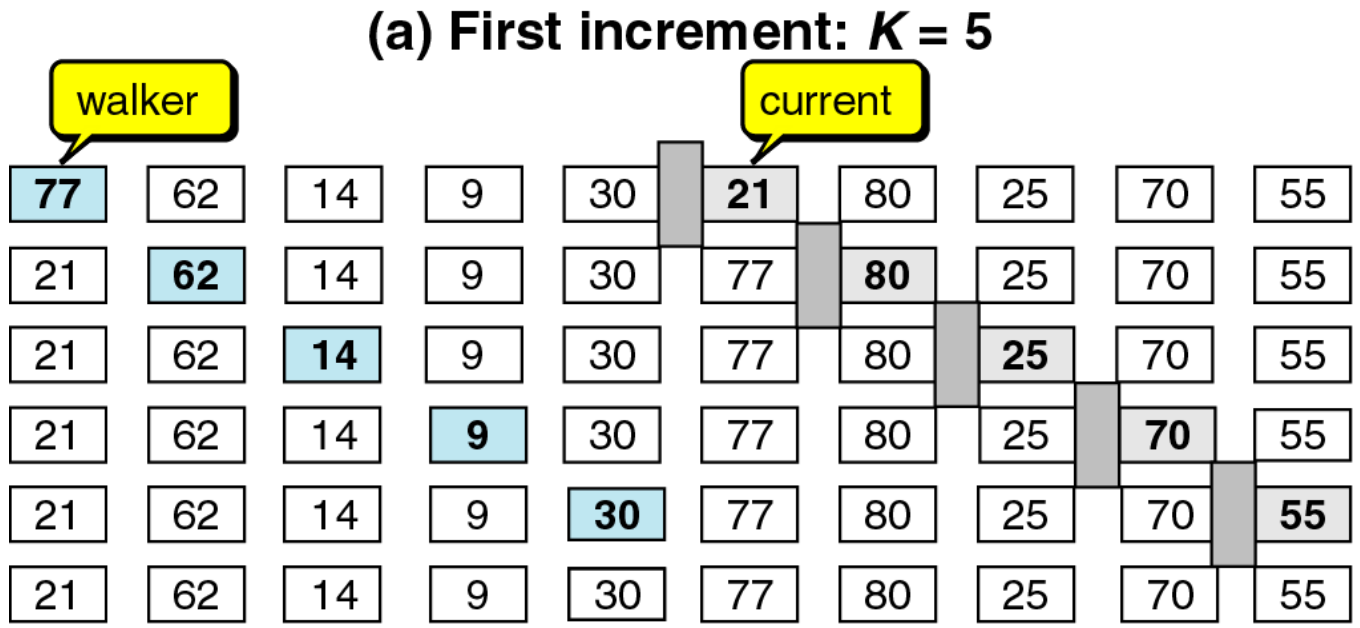
- **คุณสมบัติที่สำคัญของ** Shellsort **คือ การทำ** h_k-sorted **แล้วตามด้วย** h_{k-1}-sorted **นั้นยังคงสภาพของ** h_k-sorted

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

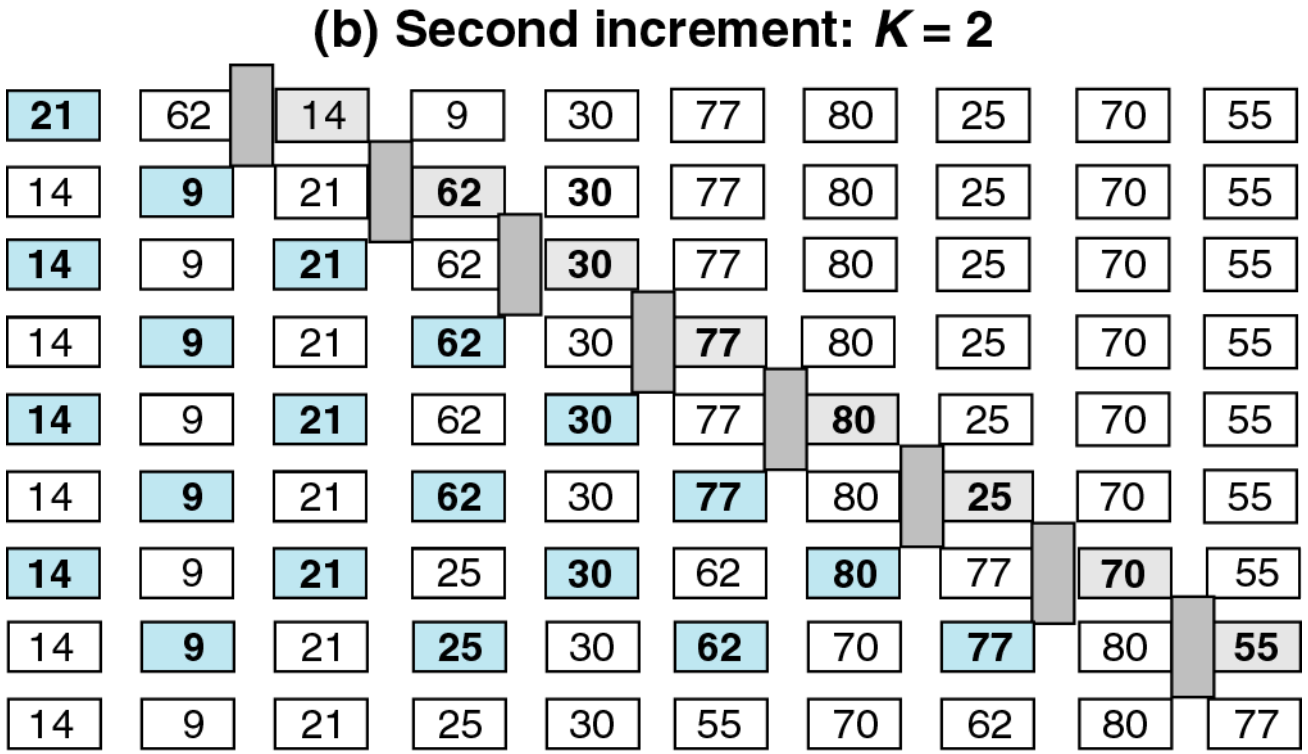
Figure Shellsort **หลังการทำงานแต่ละ** pass

Shell Sort

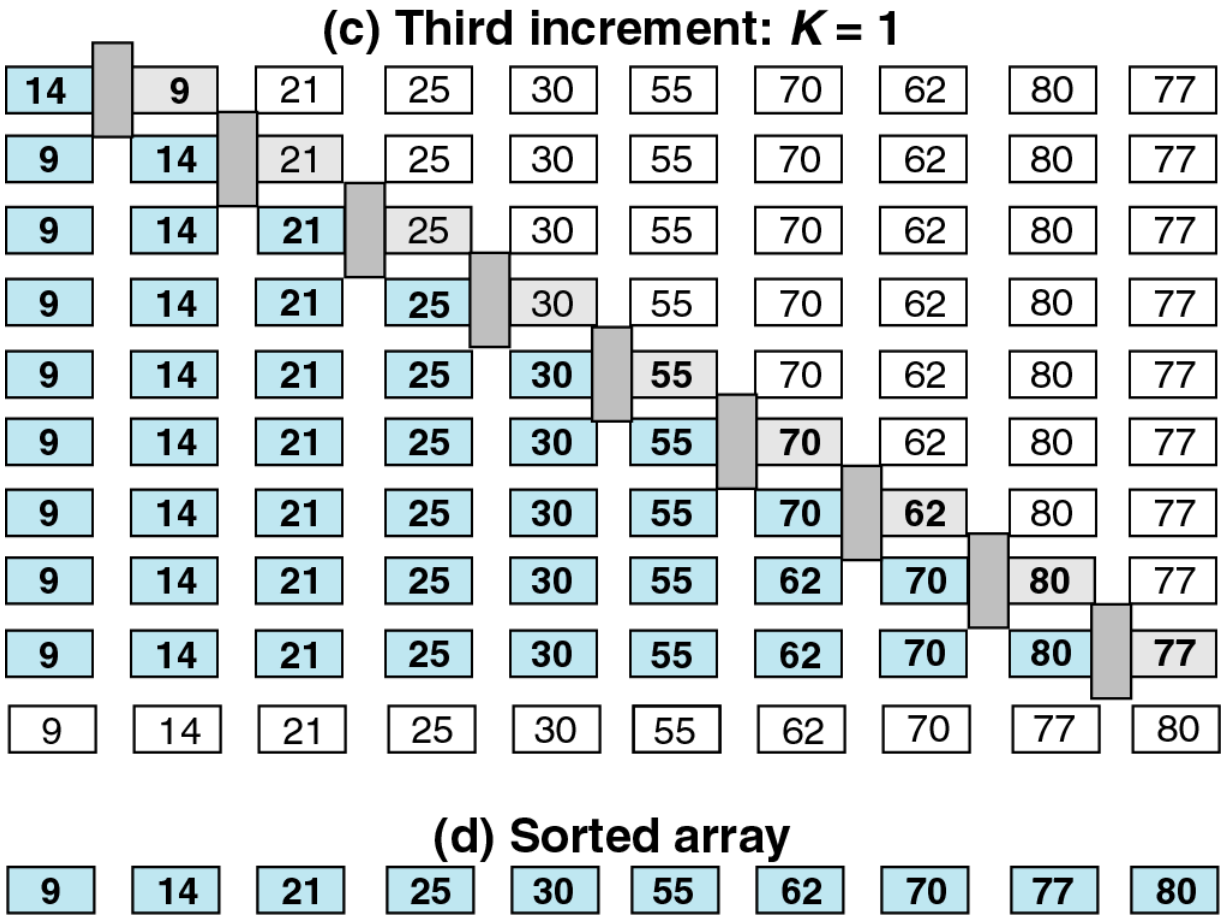
ตัวอย่างการทำงานในแต่ละรอบของ K



Shell Sort


(b) Second increment: K = 2
21 62 14 9 30 77 80 25 70 55
14 9 21 62 30 77 80 25 70 55
14 9 21 62 30 77 80 25 70 55
14 9 21 62 30 77 80 25 70 55
14 9 21 62 30 77 80 25 70 55
14 9 21 62 30 77 80 25 70 55
14 9 21 25 30 62 80 77 70 55
14 9 21 25 30 62 70 77 80 55
14 9 21 25 30 55 70 62 80 77

Shell Sort


(c) Third increment: K = 1
14 9 21 25 30 55 70 62 80 77
9 14 21 25 30 55 70 62 80 77
9 14 21 25 30 55 70 62 80 77
9 14 21 25 30 55 70 62 80 77
9 14 21 25 30 55 70 62 80 77
9 14 21 25 30 55 70 62 80 77
9 14 21 25 30 55 70 62 80 77
9 14 21 25 30 55 62 70 80 77
9 14 21 25 30 55 62 70 80 77
9 14 21 25 30 55 62 70 77 80
(d) Sorted array
9 14 21 25 30 55 62 70 77 80

Shell Sort routine ใช้ลำดับการเพิ่มของ Shell

```
int shellSort(int arr[], int n)
{
    for (int gap = n/2; gap > 0; gap /= 2)
    {
        for (int i = gap; i < n; i += 1)
        {
            int temp = arr[i];
            for (int j = i; j >= gap && arr[j - gap] > temp; j -= gap)
                arr[j] = arr[j - gap];

            arr[j] = temp;
        }
    }
    return 0;
}
```

ประสิทธิภาพ

- ความเร็ว Big O = ?
- Linked หรือว่า Array ต่างกันไหม

เทคนิคการลดขนาดของปัญหา

MergeSort คิดค้นโดย John von Neumann
ผู้บุกเบิกการคำนวณในปี 1945

“If people do not believe that mathematics is simple, it is only because they do not realize how complicated life is.”
— John von Neumann

หลายหัวดีกว่าหัวเดียว

- เทคนิคสำหรับ CPU มากกว่าหนึ่ง Core
- ทำขนาดของปัญหา n ให้เล็กลงทีละครึ่ง

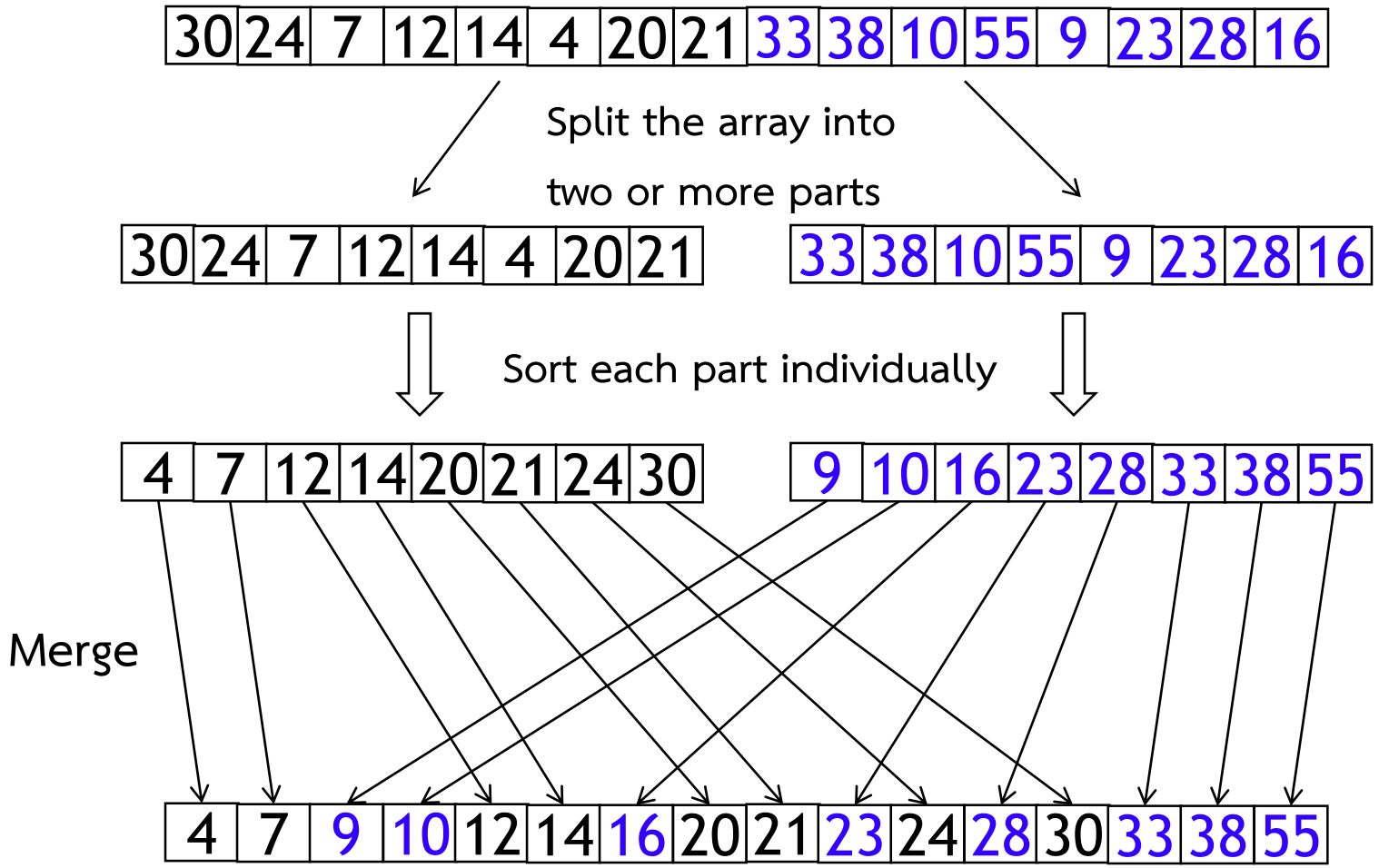
Merge Sort

- การเรียงแบบผสาน (Merge Sort) -- การทำ Merge Sort ใช้หลักการ divide-and-conquer เหมือนกับ Quick Sort มีลักษณะของการแบ่งข้อมูลออกเป็นส่วนๆ แต่กระบวนการเรียงข้อมูลนั้นจะแตกต่างไปจาก Quick sort
- Quick sort กระทำการสลับข้อมูลไปพร้อมกับการแบ่งข้อมูลออกเป็นส่วนๆ แต่ merge sort นี้ กระทำการแบ่งข้อมูลออกเป็นส่วนๆก่อน แล้วค่อยเรียงข้อมูลในส่วนย่อย จากนั้นนำเอาข้อมูลส่วนย่อยที่เรียงไว้แล้ว มารวมกันและเรียงไปในเวลาเดียวกัน อัลกอริทึมจะเรียงพร้อมกับผสานข้อมูล เข้าด้วยกันจนกระทั่งข้อมูลทุกตัวรวมกันกลายเป็นข้อมูลเดียวอีกครั้ง

Merge Sort

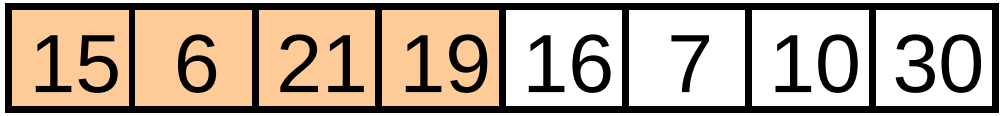
36 20 17 13 28 14 23 15	แบ่งข้อมูลออกเป็นข้อมูลย่อยๆ
20 36 | 13 17 | 14 28 | 15 23	จัดเรียงข้อมูลย่อย
13 17 20 36 | 14 15 23 28	นำข้อมูลย่อยๆ นั้นมารวมกันให้เป็นข้อมูลเดียว
13 14 15 17 20 23 28 36	

Merge Sort – Simple version



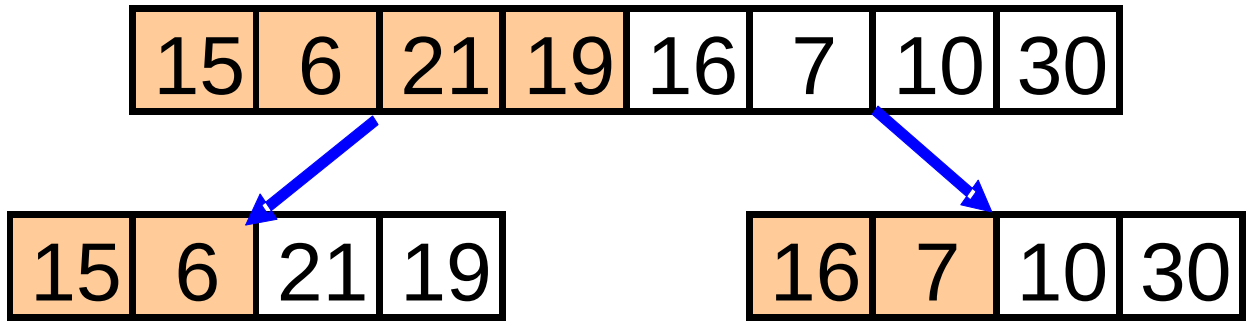
Merge Sort -Recursive

กระบวนการแยกชุดข้อมูล



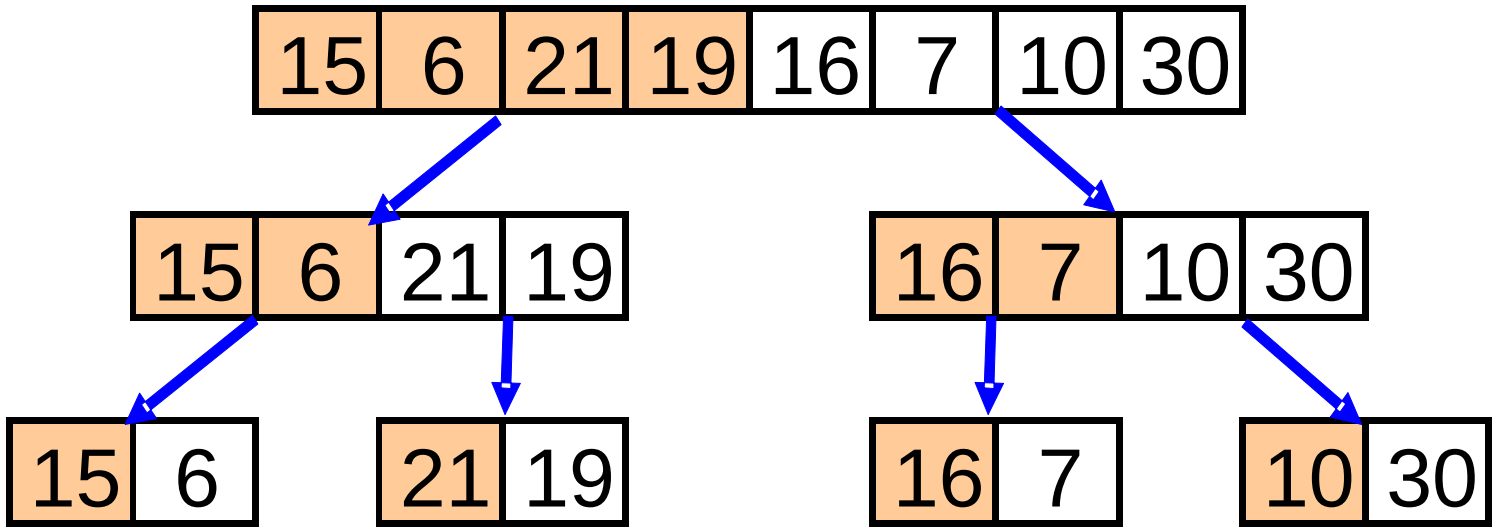
Merge Sort -Recursive

กระบวนการแยกชุดข้อมูล



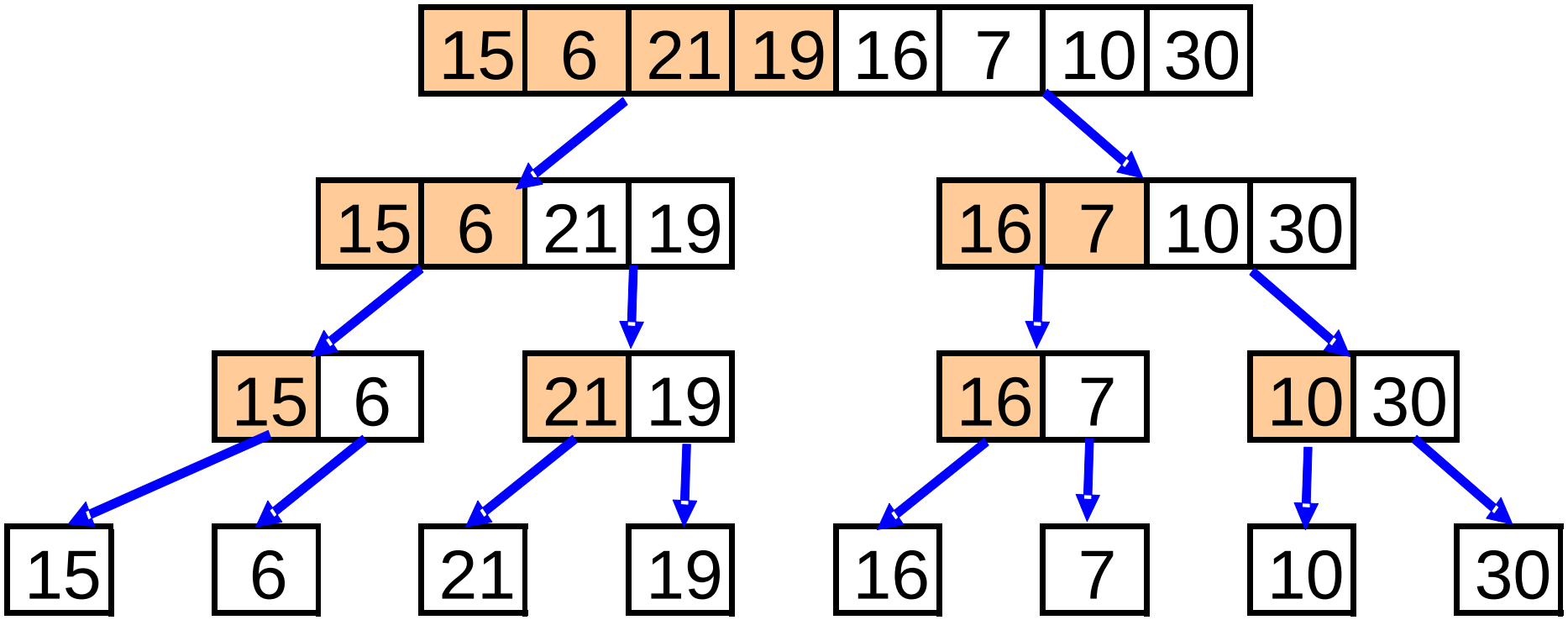
Merge Sort -Recursive

กระบวนการแยกชุดข้อมูล



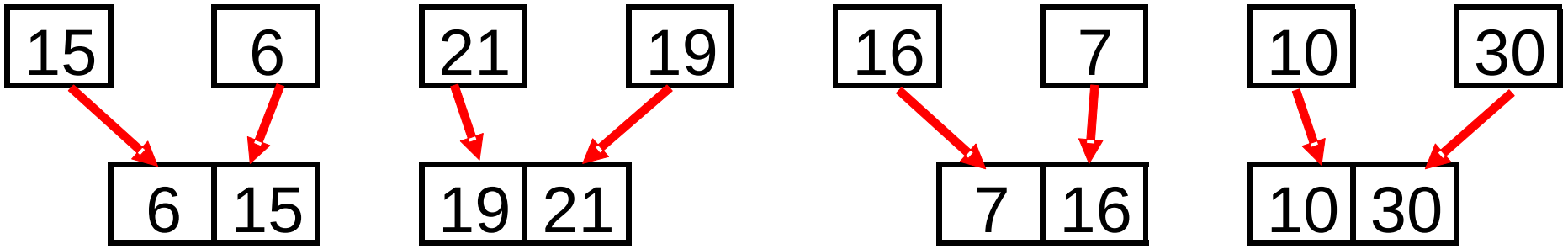
Merge Sort - Recursive

กระบวนการแยกชุดข้อมูล

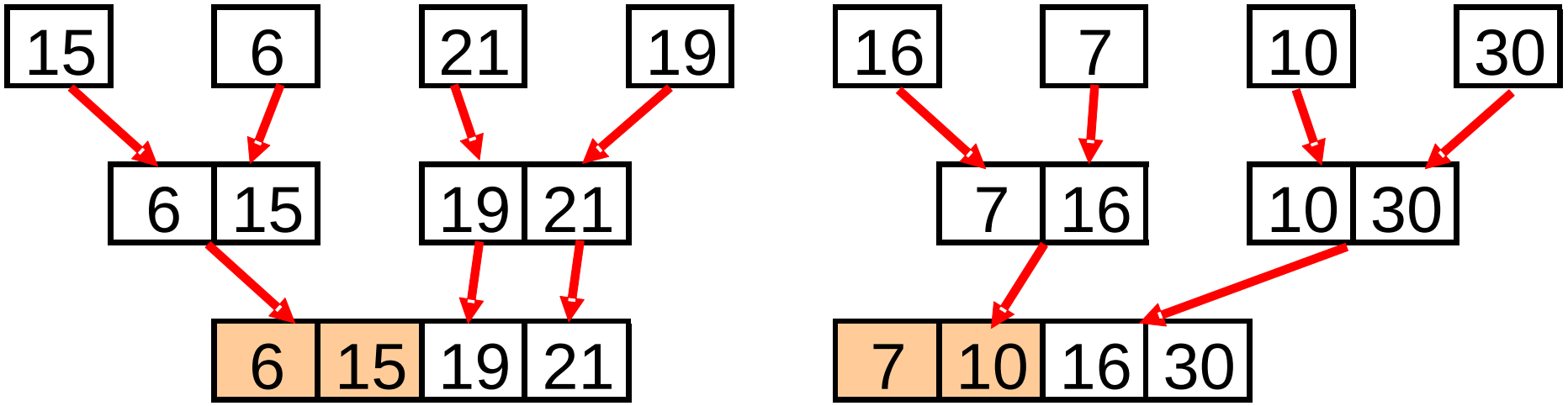


กระบวนการรวมชุดข้อมูล

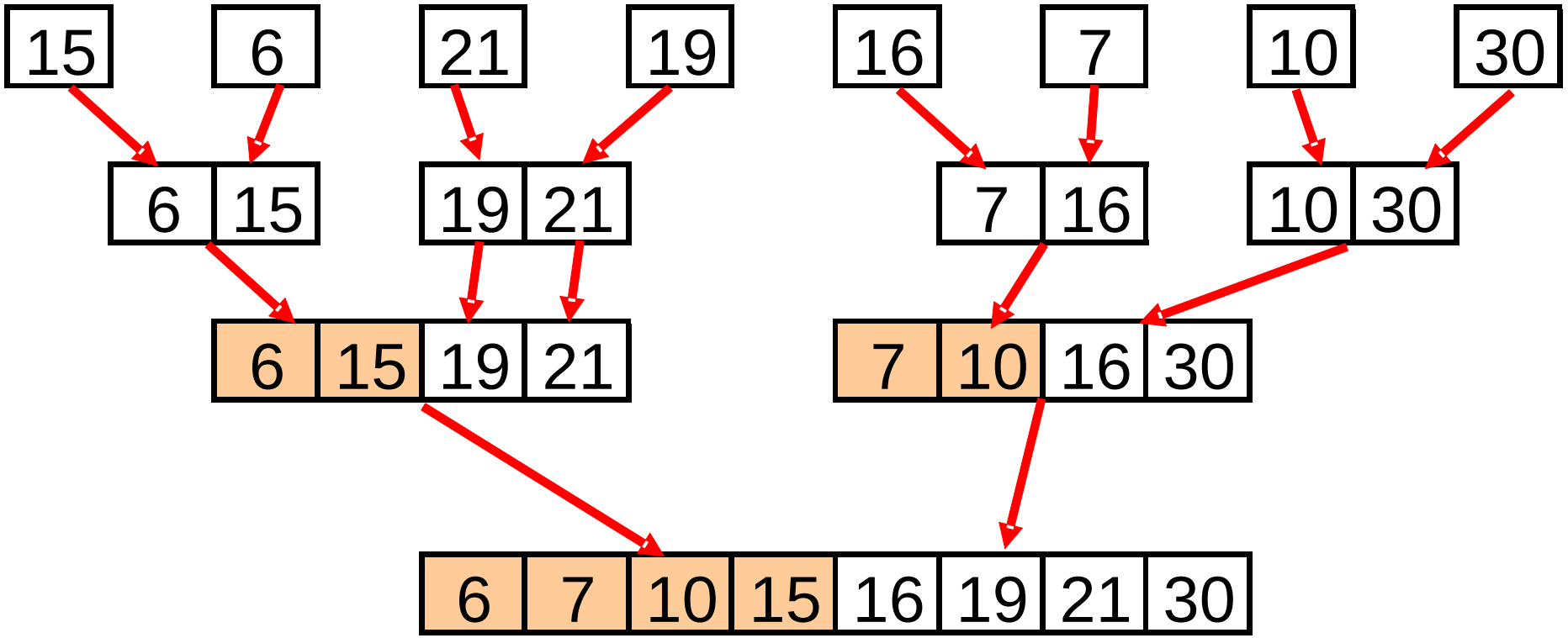
กระบวนการรวมชุดข้อมูล



กระบวนการรวมชุดข้อมูล



กระบวนการรวมชุดข้อมูล



ได้ชุดข้อมูลที่เรียงลำดับเรียบร้อยแล้ว

Merge Sort Analysis

จำนวนครั้งของการเปรียบเทียบทั้งหมด

= ผลบวกทุกระดับของ (จำนวนลิสต์ในแต่ละระดับ * จำนวนครั้งของการเปรียบเทียบแต่ละลิสต์)

$$= (N-1) * 1 + \left(\frac{N}{2} - 1\right) * 2 + \left(\frac{N}{4} - 1\right) * 4 + .. + (2-1) * \frac{N}{2}$$

$$= (N-1) + (N-2) + (N-4) + .. + \left(N - \frac{N}{2}\right)$$

$$= Nlog_2N$$

$$O(Nlog_2N)$$

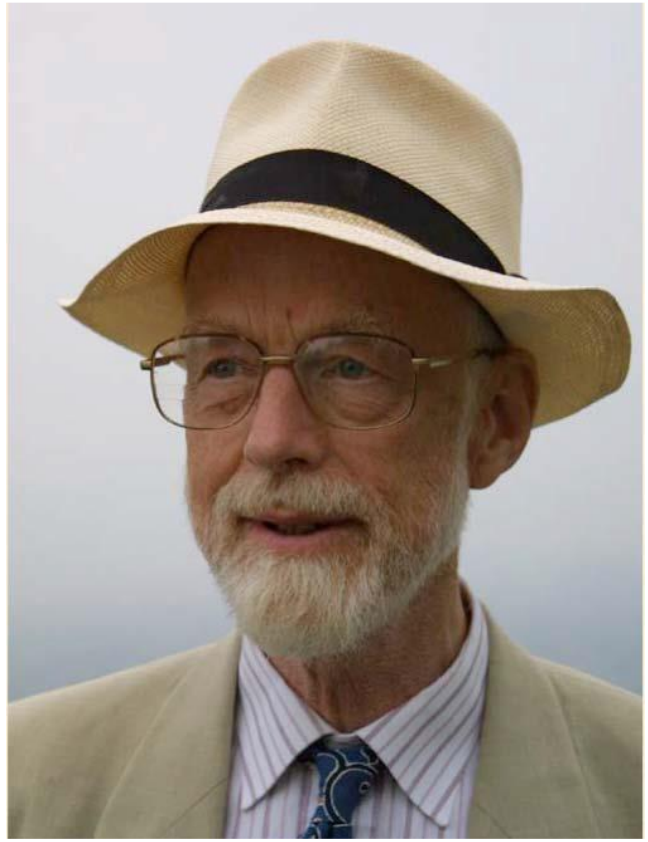
```
void mergesort(int *a, int*b, int low, int high)
{
    int pivot;
    if(low<high)
    {
        pivot=(low+high)/2;
        mergesort(a,b,low,pivot); // O(n/2)
        mergesort(a,b,pivot+1,high); // O(n/2)
        merge(a,b,low,pivot,high); // O(???)
    }
}
```

```
void merge(int *a, int *b, int low, int pivot, int high)
{
    int h,i,j,k;
    h=low;
    i=low;
    j=pivot+1;
    while((h<=pivot)&&(j<=high)){              // O(n)
        if(a[h]<=a[j]){
            b[i]=a[h];
            h++;}
        else{
            b[i]=a[j];
            j++;}
        i++;
    }
    if(h>pivot){                               // O(<n)
        for(k=j; k<=high; k++){
            b[i]=a[k];
            i++;}
    }
    else{
        for(k=h; k<=pivot; k++){               // O(<n)
            b[i]=a[k];
            i++;
        }
    }
    for(k=low; k<=high; k++) a[k]=b[k];        // O(n)
}
```

Quick Sort

- คิดค้นโดย C.A.R. Hoare ในปี 1960

“There are two ways of constructing a software design:
One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.
The first method is far more difficult.”



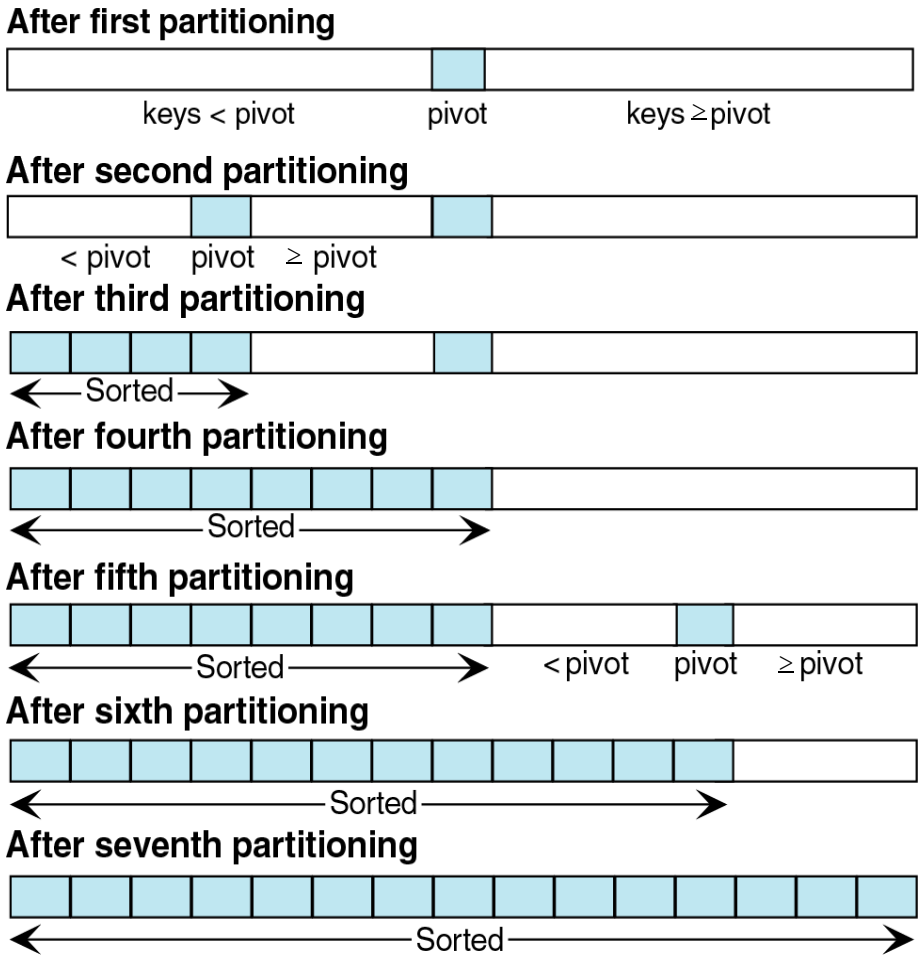
Quick Sort

- การจัดเรียงแบบควิก ใช้หลักการ divide-and-conquer
  จัดแบ่งข้อมูลทั้งหมดออกเป็น 2 กลุ่ม
  - **โดยกลุ่มแรกจะเป็นกลุ่มของข้อมูลที่มีค่าน้อยกว่าค่ากลางที่กำหนด**
  - **และส่วนที่สองเป็นกลุ่มของข้อมูลที่มีค่ามากกว่าค่ากลางที่กำหนด**
- หลังจากนั้นแบ่งข้อมูลแต่ละส่วนออกเป็น 2 ส่วนเช่นเดิม แบ่งไปเรื่อยๆ จนไม่สามารถแบ่งได้ก็จะได้ข้อมูลที่เรียงกัน

Quick Sort

ขั้นตอน

- มีการเลือกข้อมูลตัวหนึ่งเรียกว่า Pivot ที่ใช้เป็นตัวแบ่งแยกชุดข้อมูลที่เรามีออกเป็นส่วน คือ ข้อมูลที่มีค่าน้อยกว่า Pivot และข้อมูลที่มีค่ามากกว่า Pivot
- แบ่งข้อมูลไปเรื่อยๆ
- เรียงข้อมูลแต่ละส่วนย่อยๆ



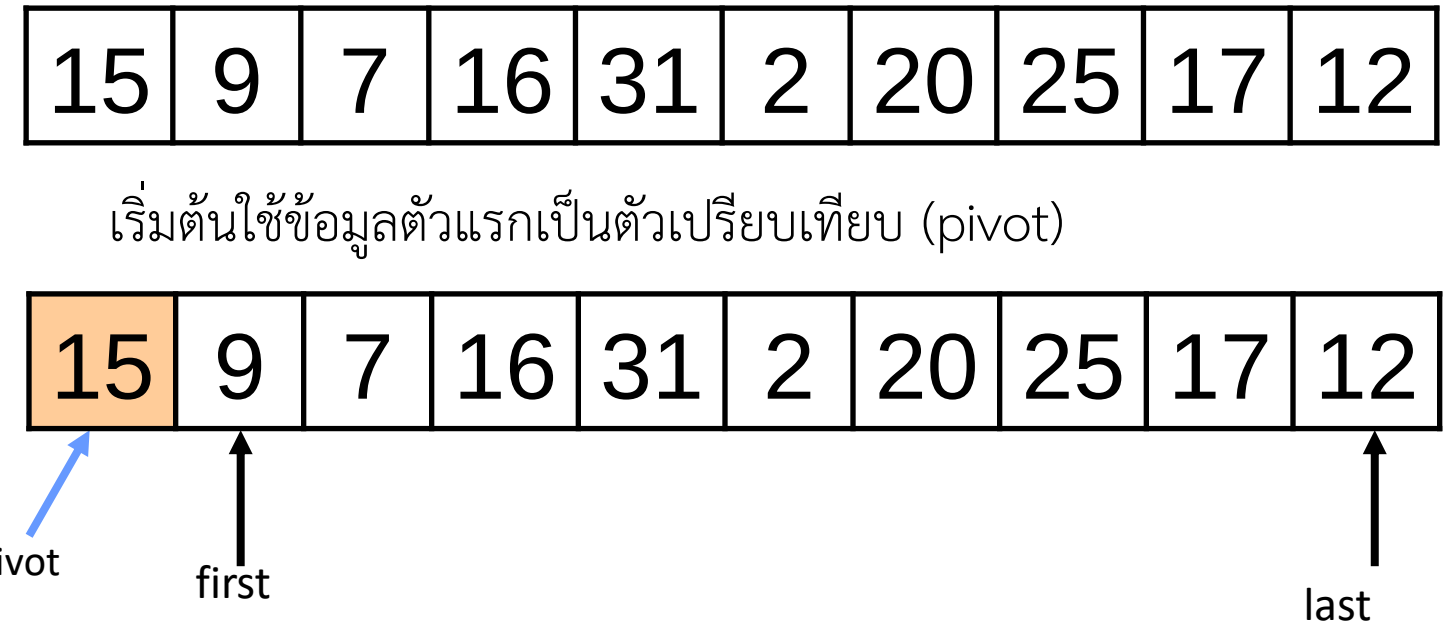
Quick Sort

การแบ่งส่วนข้อมูลใช้หลักการ Picking the pivot คือการกำหนดค่าสมมุติที่อยู่ตรงกลาง โดยจะเลือกข้อมูลที่อยู่ตรงกลางของข้อมูลทั้งหมด มาใช้เป็นค่ากึ่งกลาง ดังนั้น ข้อมูลอื่นๆ ที่มีค่ามากกว่าค่าที่อยู่ตรงกลางจะอยู่กลุ่มทางขวามือ ค่าที่น้อยกว่าจะอยู่กลุ่มทางซ้ายมือ

Quick Sort

<u>หลักการดำเนินงาน</u>

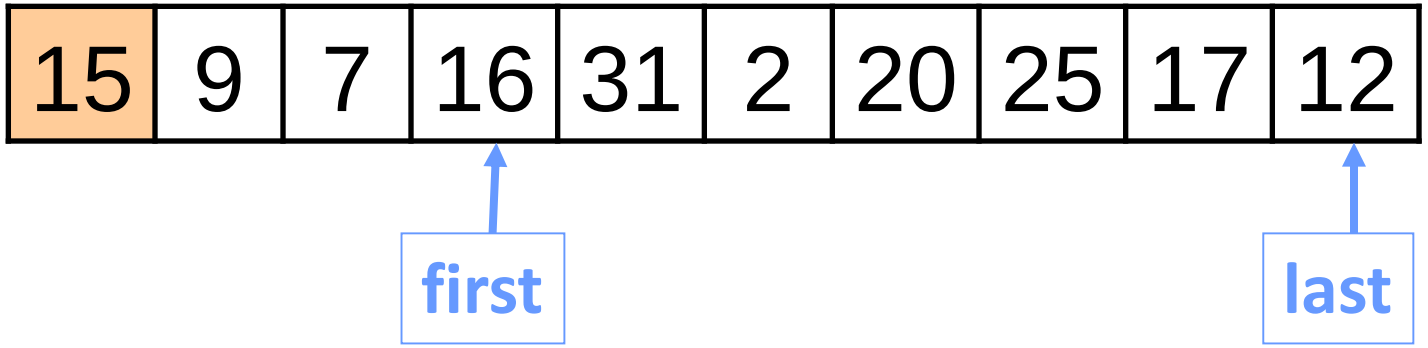
* หาตำแหน่งในการแยกลิสต์
* แบ่งแยกข้อมูลออกเป็นสองส่วน และหาตำแหน่งในการแยกลิสต์
* ทำจนกว่าข้อมูลจะเรียงลำดับเรียบร้อย



ทำการเปรียบเทียบค่าของข้อมูลที่ชี้โดย first กับ Pivot

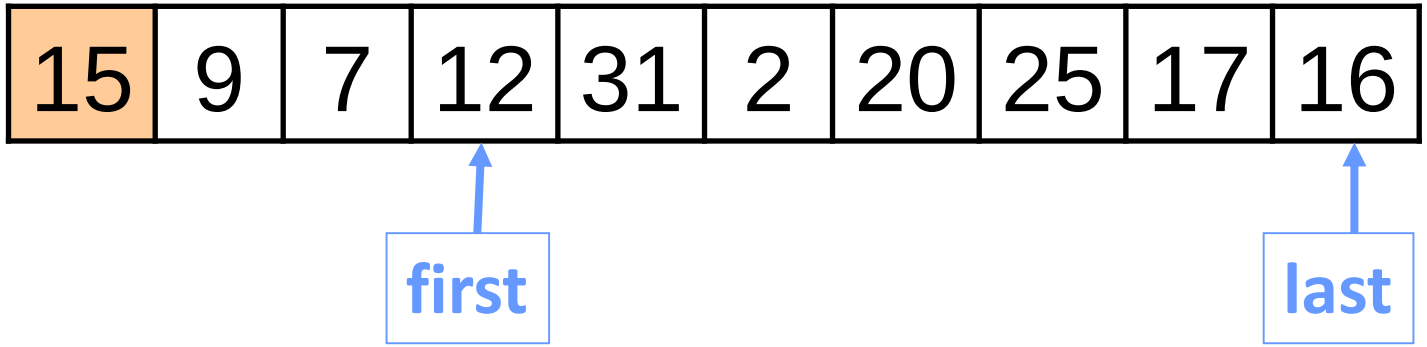
ถ้าน้อยกว่า pivot ให้ทำการเลื่อน first ไปยังข้อมูลต่อไป

และเปรียบเทียบไปจนกว่าจะไม่น้อยกว่าจึงหยุดการเปรียบเทียบ

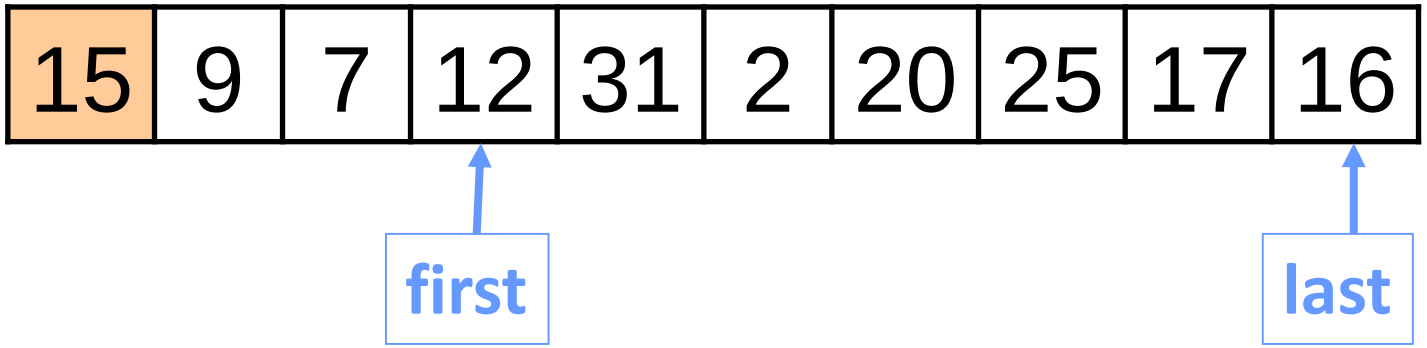


เมื่อพบตำแหน่งแล้ว จะหันมาพิจารณาที่ last ชี้อยู่

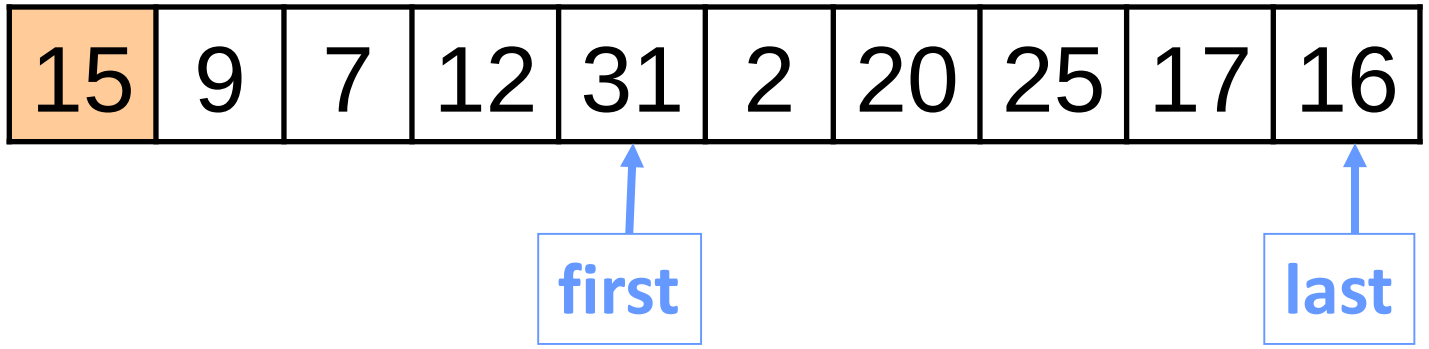
หากค่าที่ last ชี้อยู่มีค่าน้อยกว่าที่ first ชี้อยู่ให้สลับตำแหน่ง



จากชุดข้อมูลหลังจากการสลับค่า

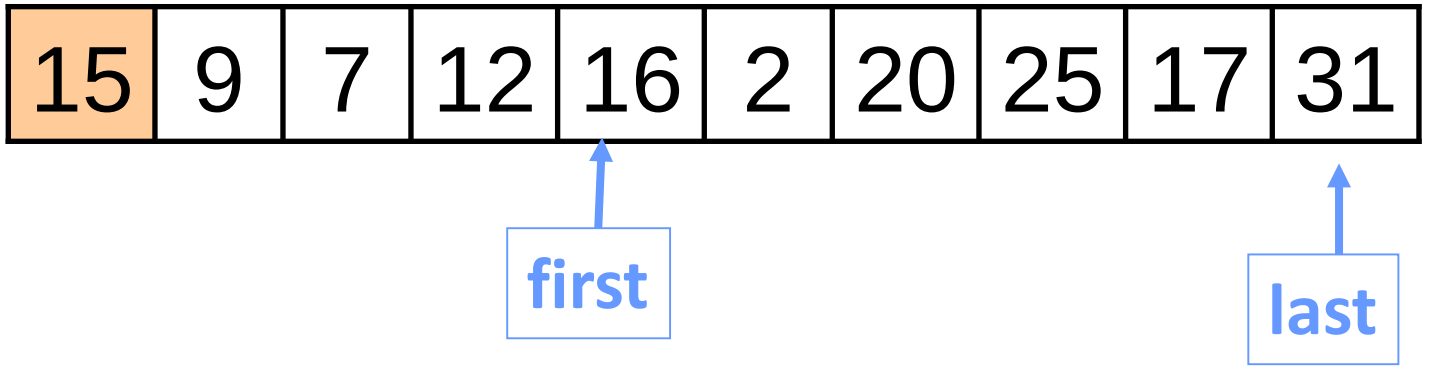


ทำการเปรียบเทียบค่าของข้อมูลที่ชี้โดย first กับ Pivot
ถ้าน้อยกว่า pivot ให้ทำการเลื่อน firstไปยังข้อมูลต่อไป
และเปรียบเทียบไปจนกว่าจะไม่น้อยกว่าจึงหยุดการเปรียบเทียบ

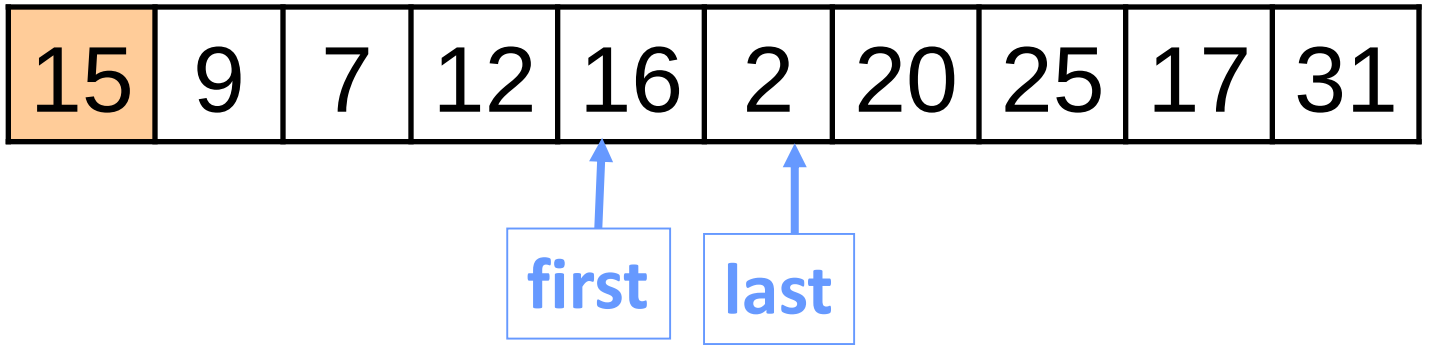


เมื่อพบตำแหน่งแล้ว จะหันมาพิจารณาที่ last ชี้อยู่

หากค่าที่ last ชี้อยู่มีค่าน้อยกว่าที่ first ชี้อยู่ให้สลับตำแหน่ง

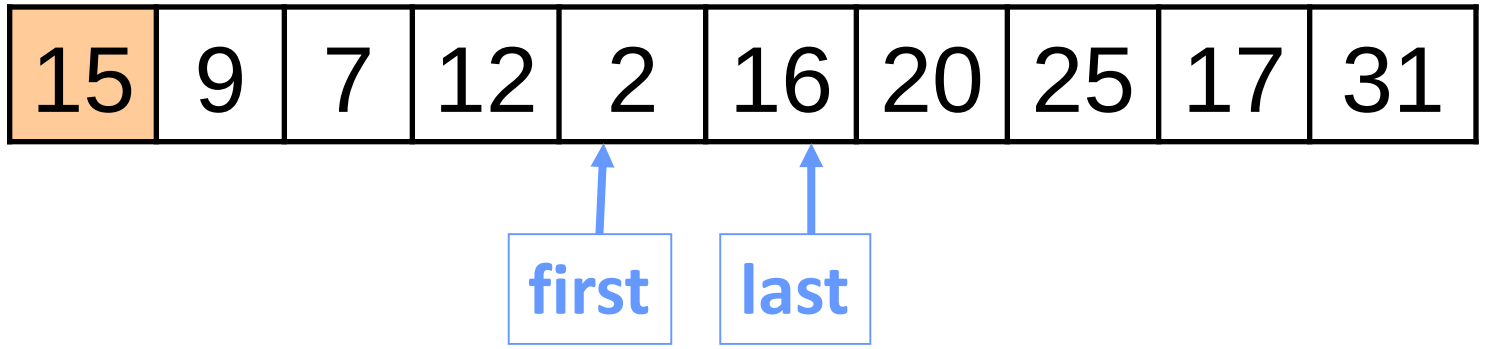


ย้อนกระบวนการไปพิจารณา first และ last จะได้

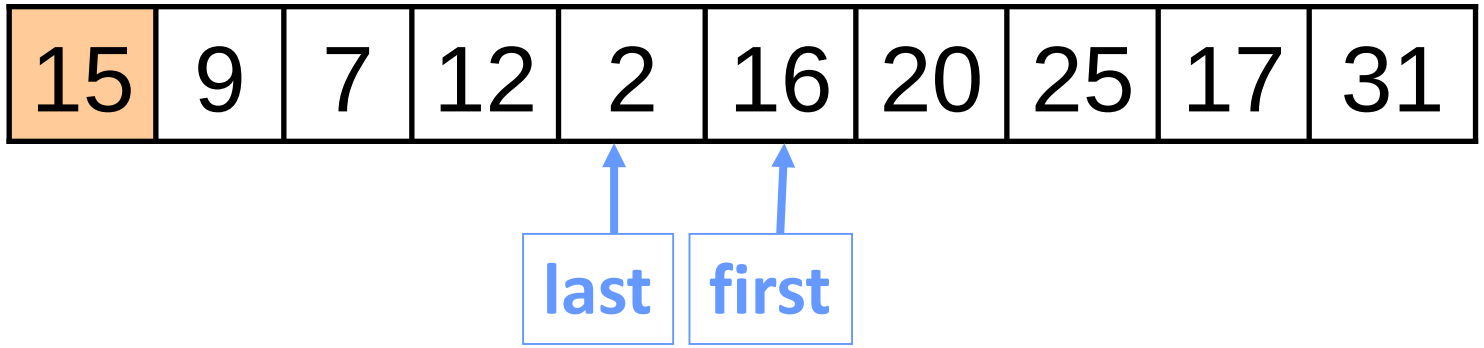


First มากกว่า pivot พิจารณา last เลื่อนมาด้านซ้าย

สลับตำแหน่งข้อมูล



ย้อนกระบวนการไปพิจารณา first และ last จะได้



พบตำแหน่งที่จะใช้แบ่งชุดข้อมูลแล้ว

จึงทำการแบ่งข้อมูลออกเป็นสองชุด

ทำการแบ่งข้อมูลออกเป็นสองชุด

15	9	7	12	2

16	20	25	17	31

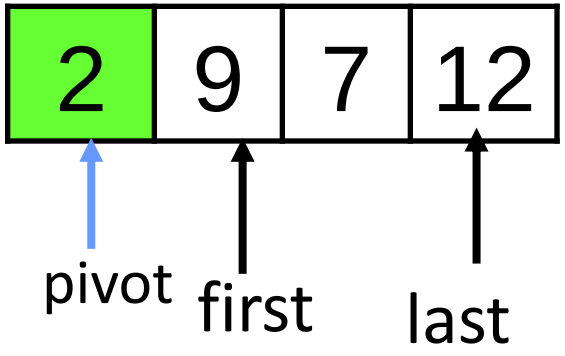
สลับค่าข้อมูลตัวแรกกับสุดท้ายของข้อมูลชุดซ้าย

2	9	7	12	15

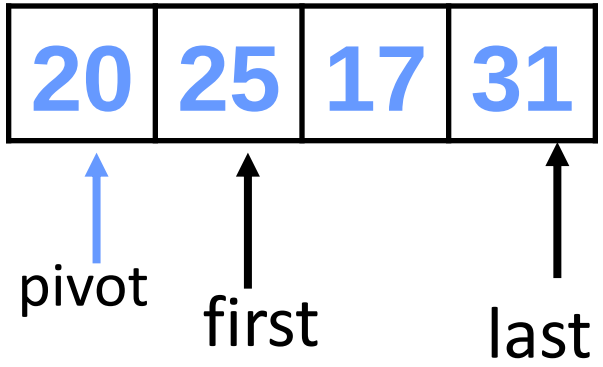
16	20	25	17	31

จะพบว่า 15 และ 16 อยู่ในตำแหน่งที่ถูกต้อง(ตน.5และ ตน.6) เรียบร้อยแล้วไม่ต้องนำมาพิจารณาอีก ดำเนินการกับข้อมูลที่เหลือเหมือนเดิมจนกว่าจะได้ข้อมูลในตำแหน่งต่างๆ จนครบ และดำเนินการแบบเดียวกับกับข้อมูลชุดขวามือ

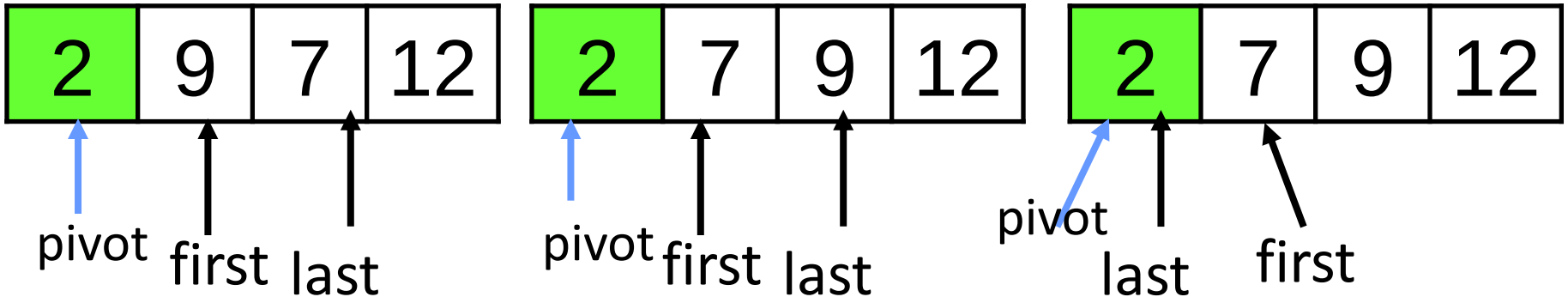
ข้อมูลเริ่มต้นชุดซ้าย



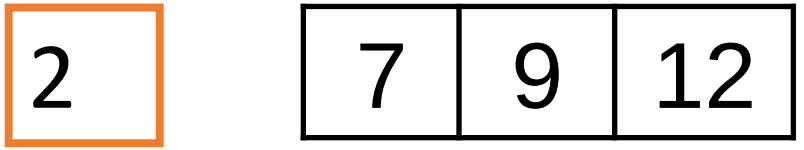
ข้อมูลเริ่มต้นชุดขวา

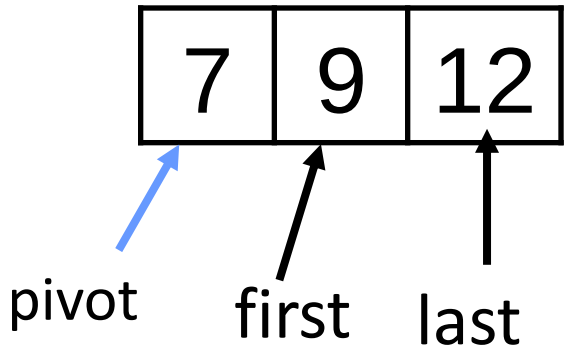


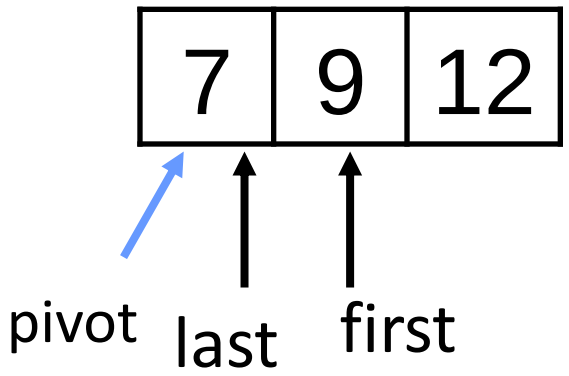
พิจารณาชุดข้อมูลด้านซ้ายมือก่อน



- First มากกว่า pivot พิจารณา last เลื่อนมาด้านซ้ายจนพบ 7 ซึ่งน้อยกว่า 9 จึงหยุด ทำการสลับตำแหน่งข้อมูล
- First มากกว่า pivot พิจารณา last เลื่อนมาด้านซ้ายจนมาชี้ที่ pivot พบว่าไม่มีข้อมูลใดต่อไปอีก ทำการแบ่งข้อมูลออกเป็นสองชุดแสดงว่า pivot อยู่ในตำแหน่งที่ถูกต้องแล้ว (2 อยู่ ตน. 1)

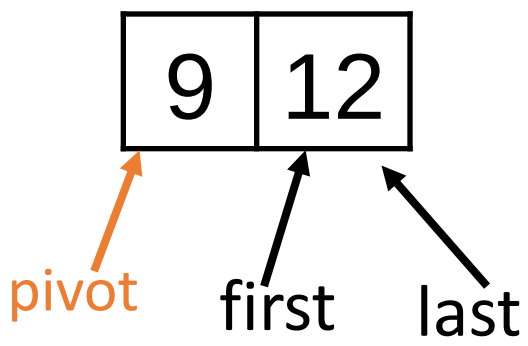


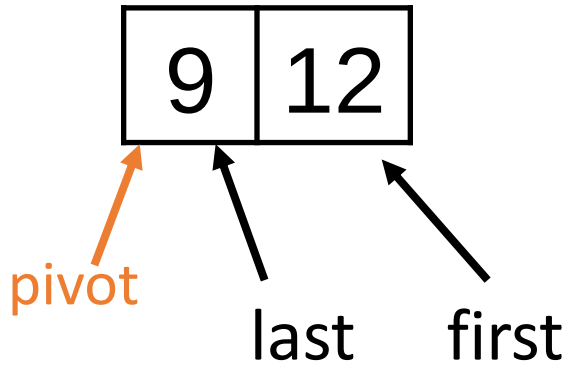




เปรียบเทียบ first กับ pivot พบว่า first มากกว่า พิจารณา last พบว่ามากกว่า first จึงเลื่อนมาด้านซ้ายจนมาชี้ที่ pivot และไม่มีตัวอื่นอีกจะได้ตำแหน่งในการแยก list

พบว่า pivot (7) อยู่ในตำแหน่งที่ถูกต้องแล้ว





สรุปตำแหน่งข้อมูลที่ทราบแล้วคือตำแหน่งที่ 1 ถึง 5 ได้แก่ 2 7 9 12 และ 15 ตามลำดับ

ดำเนินการกับข้อมูลชุดด้านขวามือต่อ

Quick Sort

62	21	24	98	87	22	74	85	76	45
P	F								L

ทำการเปรียบเทียบค่าของข้อมูลที่ชี้โดย first กับ Pivot
ถ้าน้อยกว่า pivot ให้ทำการเลื่อน first ไปยังข้อมูลต่อไป

Quick Sort

62	21	24	98	87	22	74	85	76	45
P	F								L

62	21	24	98	87	22	74	85	76	45
P		F							L

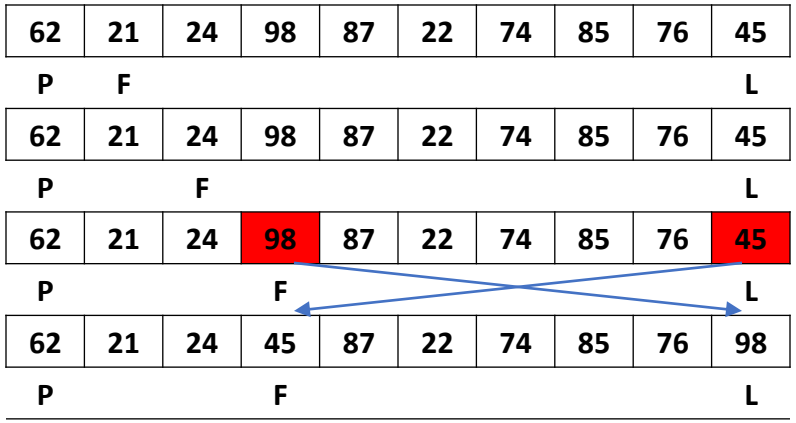
ทำการเปรียบเทียบค่าของข้อมูลที่ชี้โดย first กับ Pivot
ถ้าน้อยกว่า pivot ให้ทำการเลื่อน first ไปยังข้อมูลต่อไป

Quick Sort

62	21	24	98	87	22	74	85	76	45
P	F								L
62	21	24	98	87	22	74	85	76	45
P		F							L
62	21	24	98	87	22	74	85	76	45
P			F						L

เมื่อพบตำแหน่งแล้ว จะหันมาพิจารณาที่ last ชี้อยู่ หากค่าที่ last ชี้อยู่มีค่าน้อยกว่าที่ first ชี้อยู่ให้สลับตำแหน่ง

Quick Sort



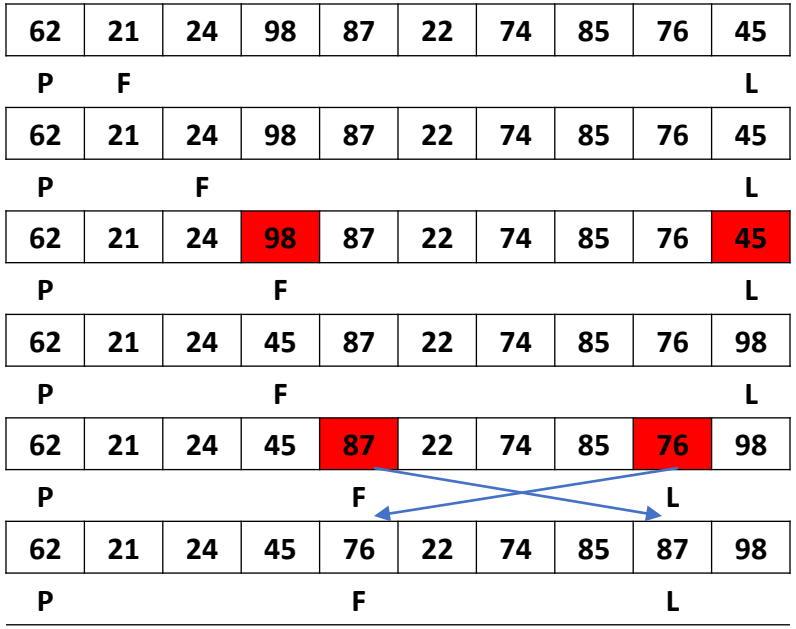
ทำการเปรียบเทียบค่าของข้อมูลที่ชี้โดย first กับ Pivot
ถ้าน้อยกว่า pivot ให้ทำการเลื่อน first ไปยังข้อมูลต่อไป

Quick Sort

62	21	24	98	87	22	74	85	76	45
P	F								L
62	21	24	98	87	22	74	85	76	45
P		F							L
62	21	24	98	87	22	74	85	76	45
P			F						L
62	21	24	45	87	22	74	85	76	98
P			F						L
62	21	24	45	87	22	74	85	76	98
P				F				L	

First มากกว่า pivot พิจารณา last เลื่อนมาด้านซ้าย

Quick Sort



เมื่อพบตำแหน่งแล้ว จะหันมาพิจารณาที่ last ชี้อยู่
หากค่าที่ last ชี้อยู่มีค่าน้อยกว่าที่ first ชี้อยู่ให้สลับตำแหน่ง

Quick Sort

62	21	24	98	87	22	74	85	76	45
P	F								L
62	21	24	98	87	22	74	85	76	45
P		F							L
62	21	24	98	87	22	74	85	76	45
P			F						L
62	21	24	45	87	22	74	85	76	98
P			F						L
62	21	24	45	87	22	74	85	76	98
P				F				L	
62	21	24	45	76	22	74	85	87	98
P				F				L	
62	21	24	45	76	22	74	85	87	98
P				F			L		

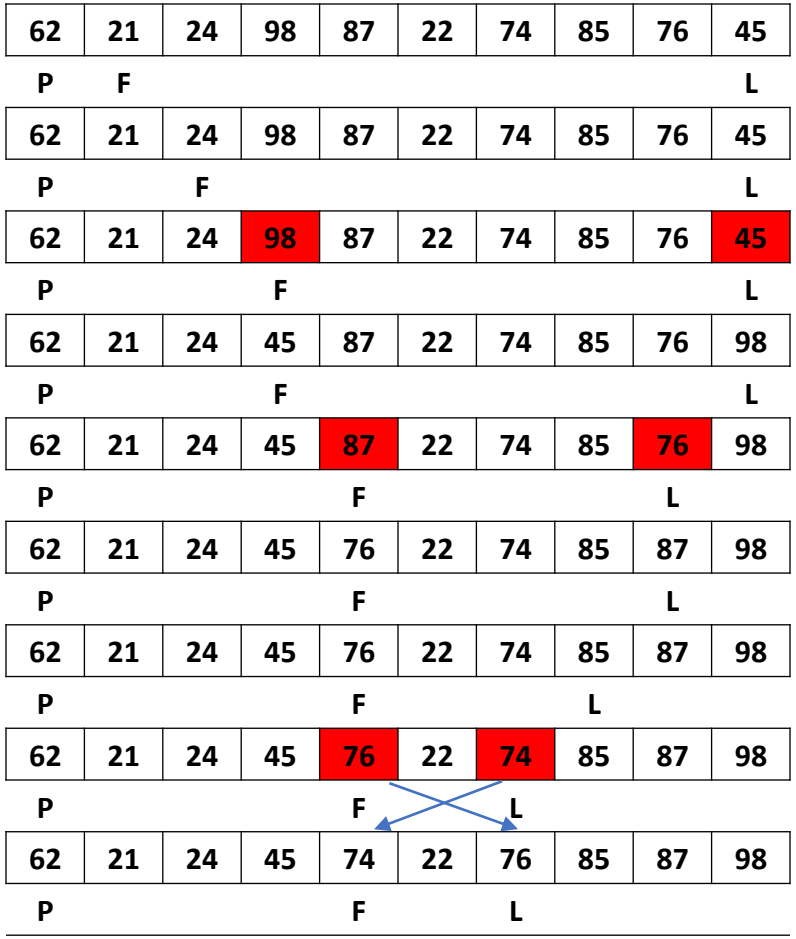
First มากกว่า pivot พิจารณา last เลื่อนมาด้านซ้าย

Quick Sort

62	21	24	98	87	22	74	85	76	45
P	F								L
62	21	24	98	87	22	74	85	76	45
P		F							L
62	21	24	98	87	22	74	85	76	45
P			F						L
62	21	24	45	87	22	74	85	76	98
P			F						L
62	21	24	45	87	22	74	85	76	98
P				F				L	
62	21	24	45	76	22	74	85	87	98
P				F				L	
62	21	24	45	76	22	74	85	87	98
P				F			L		
62	21	24	45	76	22	74	85	87	98
P				F		L			

First มากกว่า pivot พิจารณา last เลื่อนมาด้านซ้าย

Quick Sort



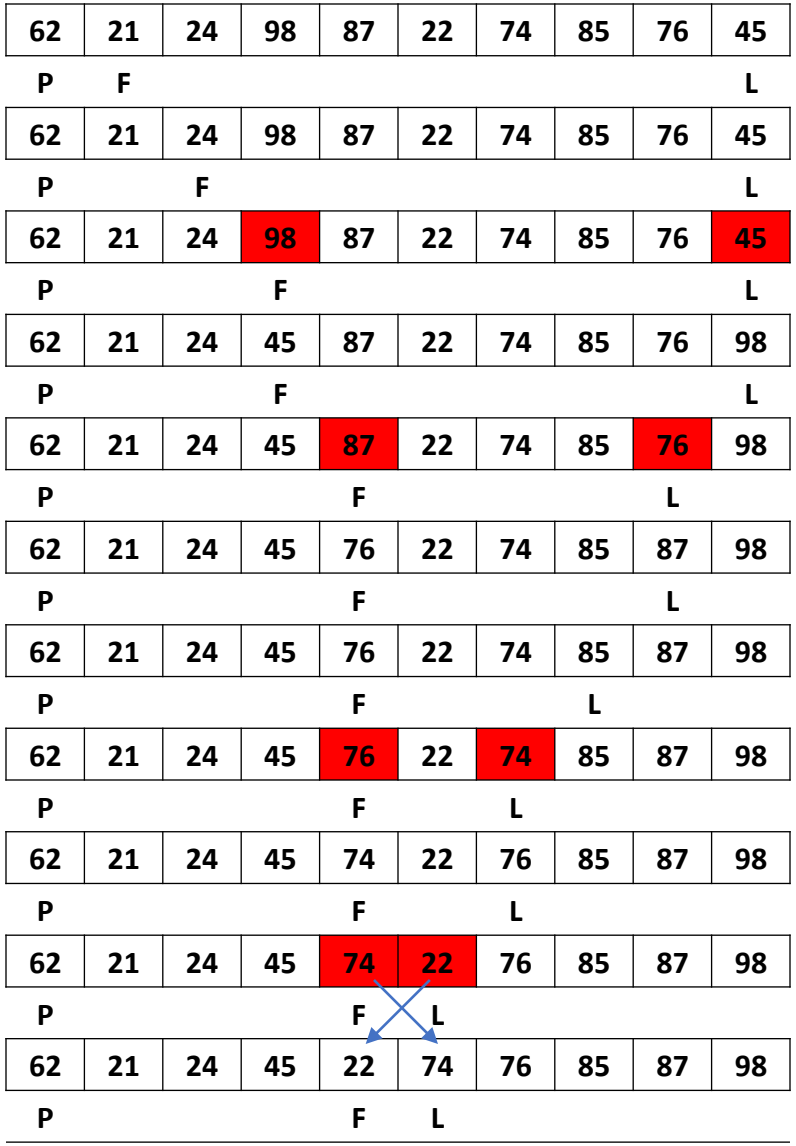
เมื่อพบตำแหน่งแล้ว จะหันมาพิจารณาที่ last ชี้อยู่
หากค่าที่ last ชี้อยู่มีค่าน้อยกว่าที่ first ชี้อยู่ให้สลับตำแหน่ง

Quick Sort

62	21	24	98	87	22	74	85	76	45
P	F								L
62	21	24	98	87	22	74	85	76	45
P		F							L
62	21	24	98	87	22	74	85	76	45
P			F						L
62	21	24	45	87	22	74	85	76	98
P			F						L
62	21	24	45	87	22	74	85	76	98
P				F				L	
62	21	24	45	76	22	74	85	87	98
P				F				L	
62	21	24	45	76	22	74	85	87	98
P				F			L		
62	21	24	45	76	22	74	85	87	98
P				F		L			
62	21	24	45	74	22	76	85	87	98
P				F		L			
62	21	24	45	74	22	76	85	87	98
P				F	L				

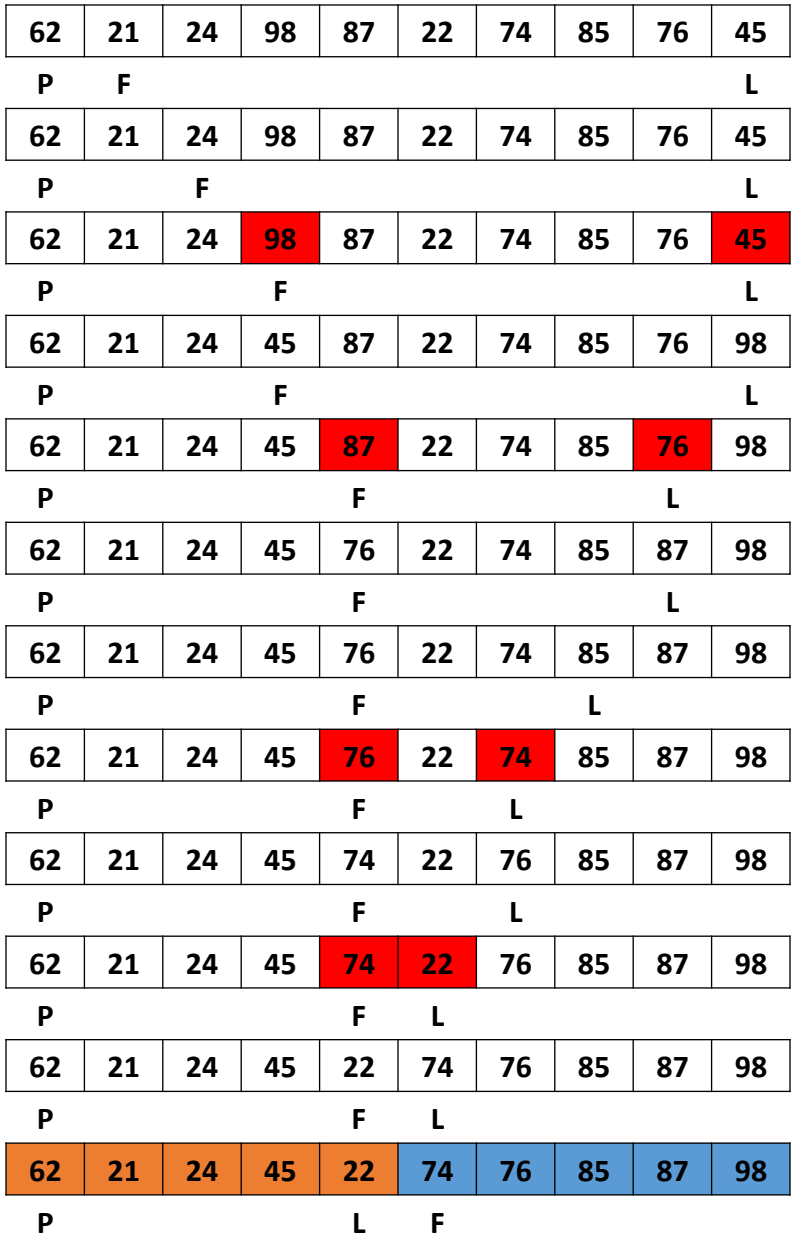
First มากกว่า pivot พิจารณา last เลื่อนมาด้านซ้าย

Quick Sort



เมื่อพบตำแหน่งแล้ว จะหันมา
พิจารณาที่ last ชี้อยู่
หากค่าที่ last ชี้อยู่มีค่าน้อยกว่าที่
first ชี้อยู่ให้สลับตำแหน่ง

Quick Sort



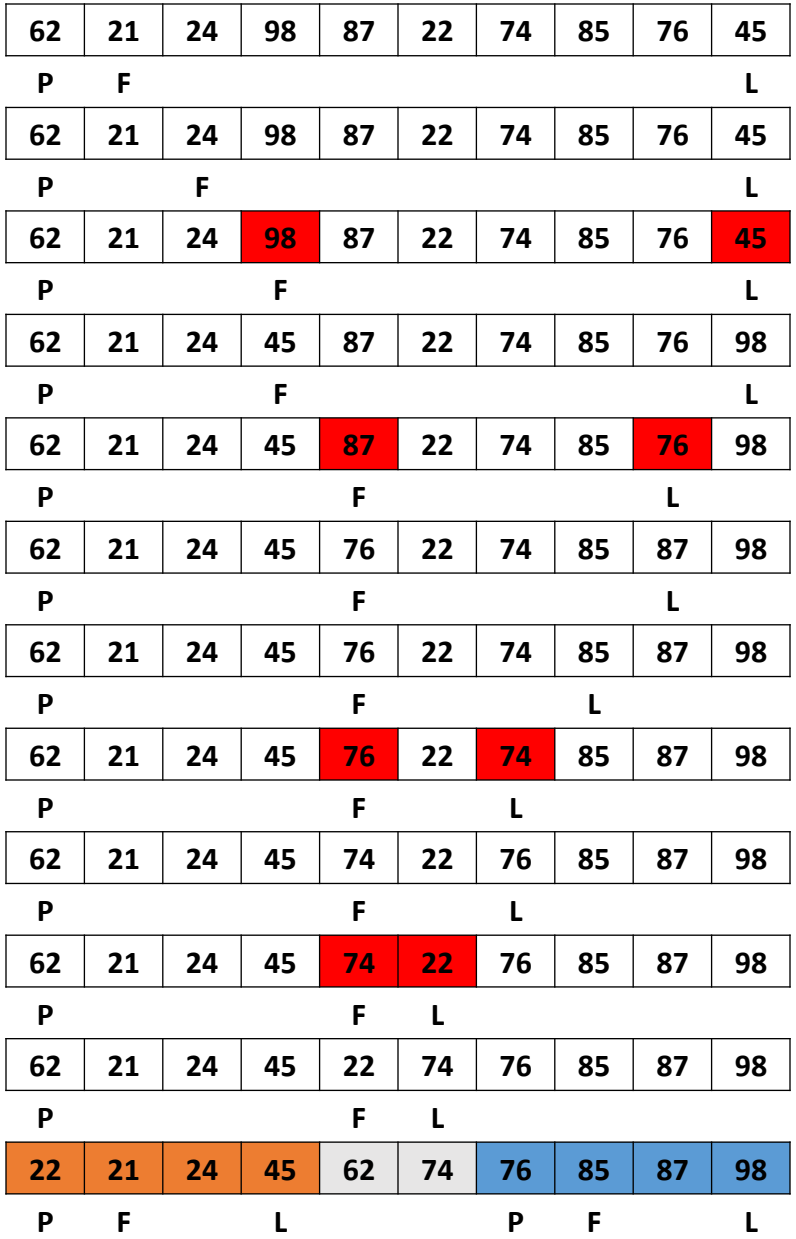
ทำการเลื่อนตำแหน่ง
จนกระทั่ง First กับ Last
ไขว้ตำแหน่งกันจนได้จุดตัด

Quick Sort

62	21	24	98	87	22	74	85	76	45
P	F								L
62	21	24	98	87	22	74	85	76	45
P		F							L
62	21	24	98	87	22	74	85	76	45
P			F						L
62	21	24	45	87	22	74	85	76	98
P			F						L
62	21	24	45	87	22	74	85	76	98
P				F				L	
62	21	24	45	76	22	74	85	87	98
P				F				L	
62	21	24	45	76	22	74	85	87	98
P				F			L		
62	21	24	45	76	22	74	85	87	98
P				F		L			
62	21	24	45	74	22	76	85	87	98
P				F		L			
62	21	24	45	74	22	76	85	87	98
P				F	L				
62	21	24	45	22	74	76	85	87	98
P				F	L				
22	21	24	45	62	74	76	85	87	98

สลับ Pivot กับ First จะได้
ตำแหน่งตรงกลางที่ตัดแล้ว

Quick Sort



เริ่มการทำงานรอบใหม่โดย
ใช้ Recursive

```
void quicksort(int *arr, const int left, const int right){

    if (left >= right) {
        return;
    }

    int part = partition(arr, left, right);//O(?)

    quicksort(arr, left, part - 1, sz);  //O(n/2)
    quicksort(arr, part + 1, right, sz); //O(n/2)
}
```

```
int partition(int *arr, const int left, const int right) {
    const int mid = left + (right - left) / 2;
    const int pivot = arr[mid];
    // move the mid point value to the front.
    std::swap(arr[mid],arr[left]);
    int i = left + 1;
    int j = right;
    while (i <= j) {                                //O(n)
        while(i <= j && arr[i] <= pivot) {          //O(x), x<n
            i++;
        }
        while(i <= j && arr[j] > pivot) {           //O(y), y+x = n
            j--;
        }
        if (i < j) {
            std::swap(arr[i], arr[j]);
        }
    }
    std::swap(arr[i - 1],arr[left]);
    return i - 1;
}
```

ประสิทธิภาพ

- ความเร็ว Big O = ?
- Linked หรือว่า Array ต่างกันไหม
- Singly Linked List กับ Doubly Linked List ล่ะ

Quick Sort - Best and Average Case

Pivot อยู่ตรงกลางเสมอ

$T(N) = T(i) + T(N\text{-}i\text{-}1) + cN$

$T(N) = 2T(N/2 - 1) + cN$

$$< 2T(N/2) + cN$$

$$< 4T(N/4) + c(2N/2 + N)$$

$$< 8T(N/8) + cN(1+1+1)$$

$$< kT(N/k) + cN\log(k) = O(N\log N)$$

Quick Sort

10	9	8	7	6	5	4	3	2	1
P	F								L

Quick Sort

10	9	8	7	6	5	4	3	2	1
P	F								L
10	9	8	7	6	5	4	3	2	1
P		F							L

Quick Sort

10	9	8	7	6	5	4	3	2	1
P	F								L
10	9	8	7	6	5	4	3	2	1
P		F							L
10	9	8	7	6	5	4	3	2	1
P			F						L

Quick Sort

10	9	8	7	6	5	4	3	2	1
P	F								L
10	9	8	7	6	5	4	3	2	1
P		F							L
10	9	8	7	6	5	4	3	2	1
P			F						L
10	9	8	7	6	5	4	3	2	1
P				F					L

Quick Sort

10	9	8	7	6	5	4	3	2	1
P	F								L
10	9	8	7	6	5	4	3	2	1
P		F							L
10	9	8	7	6	5	4	3	2	1
P			F						L
10	9	8	7	6	5	4	3	2	1
P				F					L
10	9	8	7	6	5	4	3	2	1
P					F				L

Quick Sort

10	9	8	7	6	5	4	3	2	1
P	F								L
10	9	8	7	6	5	4	3	2	1
P		F							L
10	9	8	7	6	5	4	3	2	1
P			F						L
10	9	8	7	6	5	4	3	2	1
P				F					L
10	9	8	7	6	5	4	3	2	1
P					F				L
10	9	8	7	6	5	4	3	2	1
P						F			L

Quick Sort

10	9	8	7	6	5	4	3	2	1
P	F								L
10	9	8	7	6	5	4	3	2	1
P		F							L
10	9	8	7	6	5	4	3	2	1
P			F						L
10	9	8	7	6	5	4	3	2	1
P				F					L
10	9	8	7	6	5	4	3	2	1
P					F				L
10	9	8	7	6	5	4	3	2	1
P						F			L
10	9	8	7	6	5	4	3	2	1
P							F		L

Quick Sort

10	9	8	7	6	5	4	3	2	1
P	F								L
10	9	8	7	6	5	4	3	2	1
P		F							L
10	9	8	7	6	5	4	3	2	1
P			F						L
10	9	8	7	6	5	4	3	2	1
P				F					L
10	9	8	7	6	5	4	3	2	1
P					F				L
10	9	8	7	6	5	4	3	2	1
P						F			L
10	9	8	7	6	5	4	3	2	1
P							F		L
10	9	8	7	6	5	4	3	2	1
P								F	L

Quick Sort

10	9	8	7	6	5	4	3	2	1
P	F								L
10	9	8	7	6	5	4	3	2	1
P		F							L
10	9	8	7	6	5	4	3	2	1
P			F						L
10	9	8	7	6	5	4	3	2	1
P				F					L
10	9	8	7	6	5	4	3	2	1
P					F				L
10	9	8	7	6	5	4	3	2	1
P						F			L
10	9	8	7	6	5	4	3	2	1
P							F		L
10	9	8	7	6	5	4	3	2	1
P								F	L
10	9	8	7	6	5	4	3	2	1
P									FL

Quick Sort

10	9	8	7	6	5	4	3	2	1
P	F								L
10	9	8	7	6	5	4	3	2	1
P		F							L
10	9	8	7	6	5	4	3	2	1
P			F						L
10	9	8	7	6	5	4	3	2	1
P				F					L
10	9	8	7	6	5	4	3	2	1
P					F				L
10	9	8	7	6	5	4	3	2	1
P						F			L
10	9	8	7	6	5	4	3	2	1
P							F		L
10	9	8	7	6	5	4	3	2	1
P								F	L
10	9	8	7	6	5	4	3	2	1
P									FL
1	9	8	7	6	5	4	3	2	10
P	F							L	

Quick Sort - Worst case

Pivot มีขนาดเล็กจนหั่นข้อมูลแทบไม่ได้ $i = 0$:

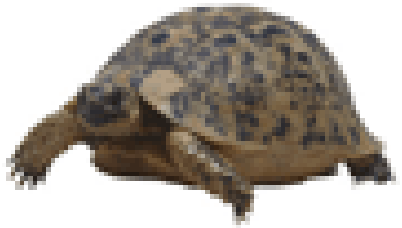
$$T(N) = T(i) + T(N\text{-}i\text{-}1) + cN$$

$$T(N) = T(N\text{-}1) + cN$$

$$= T(N\text{-}2) + c(N\text{-}1) + cN$$

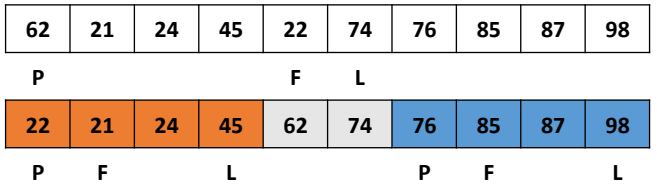
$$= T(N\text{-}k) + c\sum_{i=0}^{k-1}(N-i)$$

$$= O(N^2)$$

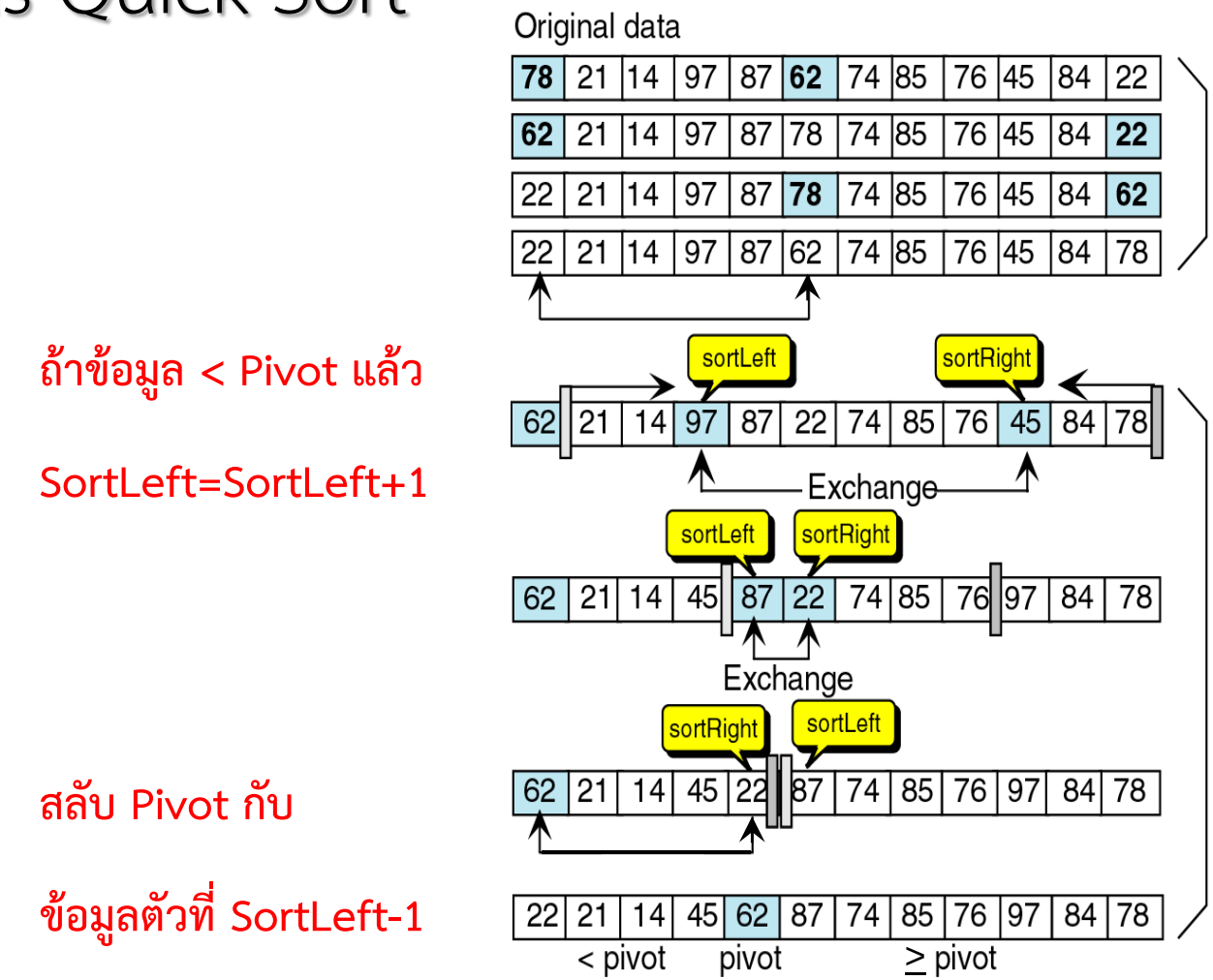


ทำไม Quick Sort จึงเร็วกว่า Merge Sort

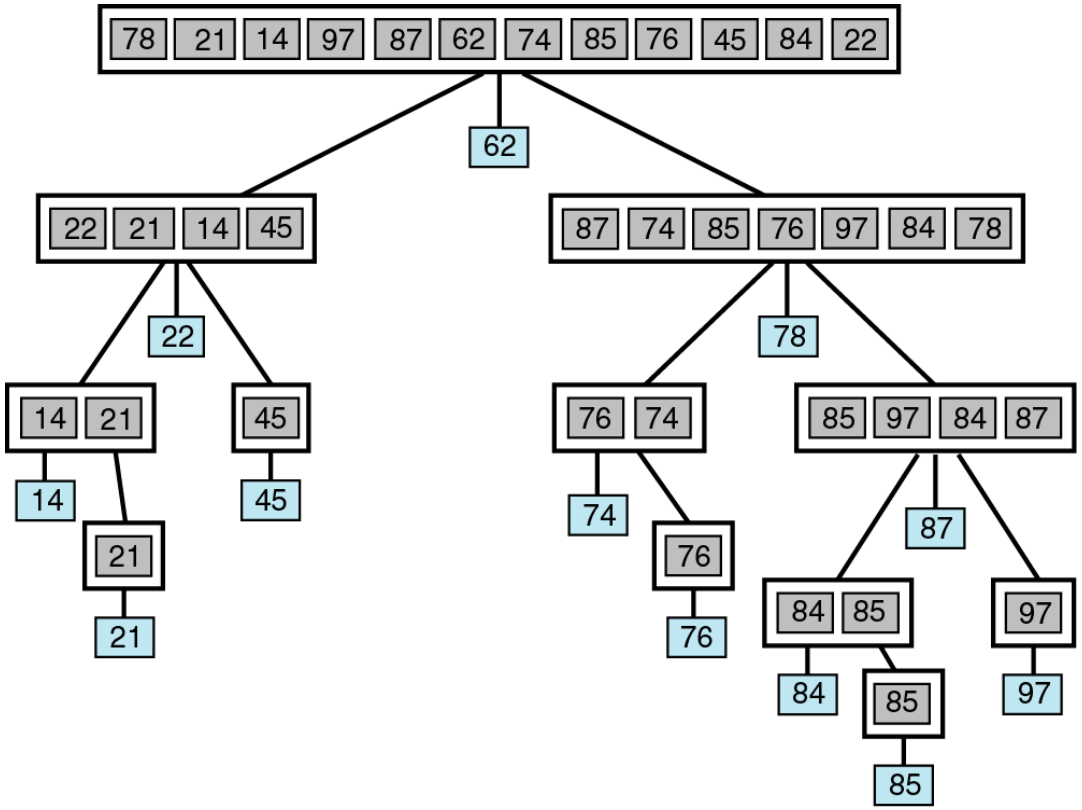
- ตามปกติแล้ว Quicksort จะทำการเปรียบเทียบมากกว่า Mergesort เนื่องจากพาร์ติชั่นไม่ได้สมดุลกันอย่างสมบูรณ์
  - Mergesort – เปรียบเทียบประมาณ $n \log n$ ครั้ง
  - Quicksort – เปรียบเทียบประมาณ $1.38\, n \log n$ ครั้ง โดยเฉลี่ย
- แต่ Quicksort ทำสำเนา (copy) น้อยลงมาก เพราะโดยเฉลี่ยแล้วครึ่งหนึ่งขององค์ประกอบจะอยู่ฝั่งที่ถูกต้องของพาร์ติชั่น (บวกฝั่งซ้ายและขวา) – ในขณะที่ Mergesort จะคัดลอกทุกองค์ประกอบเมื่อทำการรวม
  - Mergesort – คัดลอก $2n \log n$ ชุด (using “temp array”)
    คัดลอก $n \log n$ ชุด (using “alternating array”)
  - Quicksort – คัดลอก $2/n \log n$ ชุด โดยเฉลี่ย



Las Vegas Quick Sort



Las Vegas Quick Sort

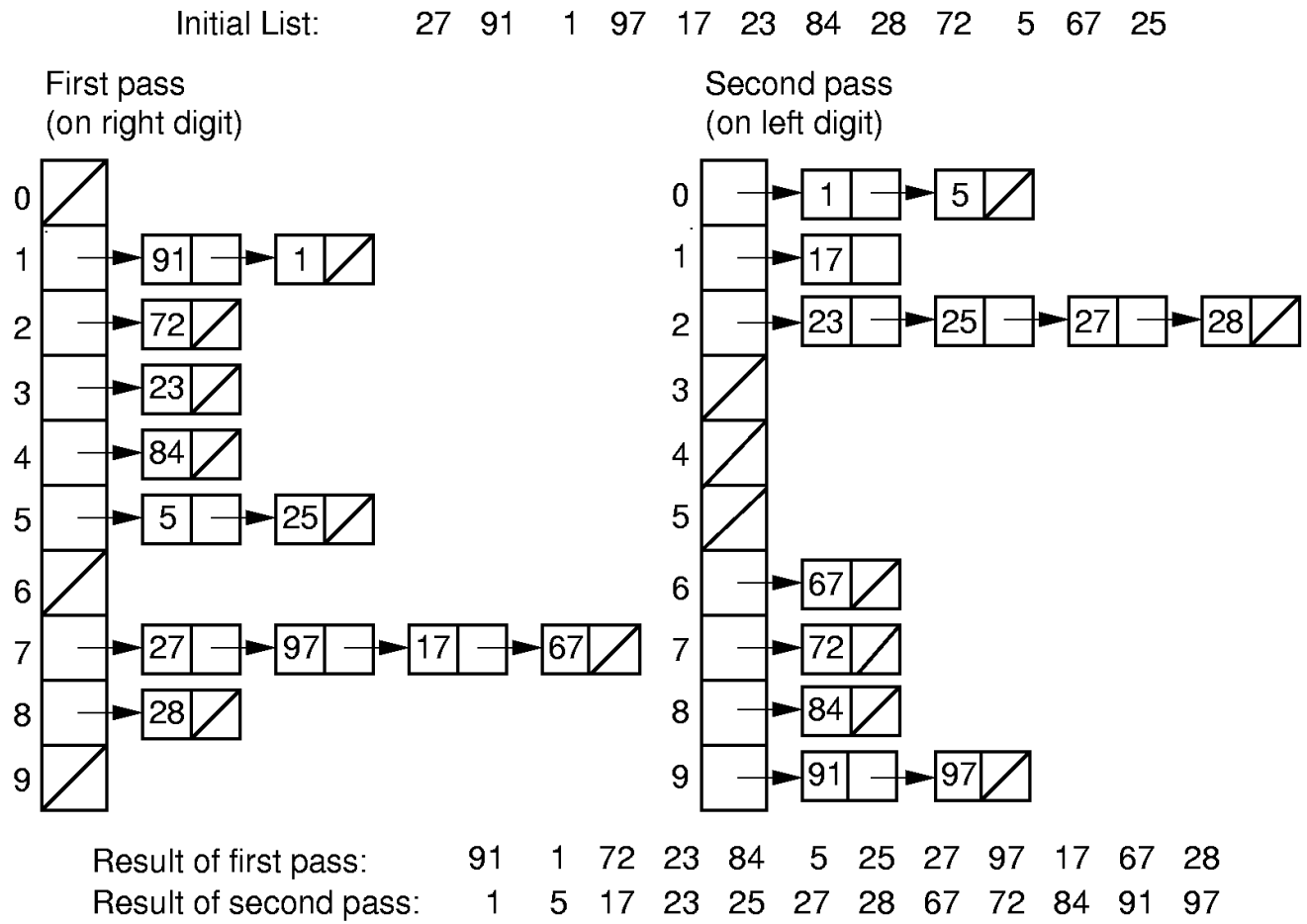


Radix Sort

การเรียงลำดับแบบฐาน -- เป็นการเรียงลำดับโดยการพิจารณาข้อมูลทีละหลัก

1. เริ่มพิจารณาจากหลักที่มีค่าน้อยที่สุดก่อน นั่นคือถ้าข้อมูลเป็นเลขจำนวนเต็มจะพิจารณาหลักหน่วยก่อน
2. การจัดเรียงจะนำข้อมูลทีละตัว แล้วนำไปเก็บไว้ที่ซึ่งจัดไว้สำหรับค่านั้น เป็นกลุ่มๆ ตามลำดับการเข้ามา
3. ในแต่ละรอบเมื่อจัดกลุ่มเรียบร้อยแล้ว ให้รวบรวมข้อมูลจากทุกกลุ่มเข้าด้วยกัน โดยเริ่มเรียงจากกลุ่มที่มีค่าน้อยทีสุดก่อนแล้วเรียงไปเรื่อยๆ จนหมดทุกกลุ่ม
4. ในรอบต่อไปนำข้อมูลทั้งหมดที่ได้จัดเรียงในหลักหน่วยเรียบร้อยแล้วมาพิจารณาจัดเรียงในหลักสิบต่อไป ทำไปเรื่อยๆ จนกระทั่งครบทุกหลัก

Radix Sort


Initial List: 27 91 1 97 17 23 84 28 72 5 67 25
First pass
(on right digit)
Second pass
(on left digit)
0
1
2
3
4
5
6
7
8
9
91
1
72
23
84
5
25
27
97
17
67
28
1
5
17
23
25
27
28
67
72
84
91
97
Result of first pass: 91 1 72 23 84 5 25 27 97 17 67 28
Result of second pass: 1 5 17 23 25 27 28 67 72 84 91 97

ประสิทธิภาพ

- ความเร็ว Big O = ?
- Linked หรือว่า Array ต่างกันไหม
- ถ้าจำนวน key เพิ่มขึ้นล่ะ
- เหมาะกับการจัดเรียงอะไร ไม่เหมาะกับการจัดเรียงอะไร

Running time of Radix Sort

- N items, K digit keys in base B
- How many passes?
- How much work per pass?

- Total time?

Running time of Radix Sort

- N items, K digit keys in base B
- How many passes? K
- How much work per pass? $N + B$

  just in case $B > N$, need to account for time to empty out buckets between passes
- Total time? $O(\ K(N+B)\ )$

การวัดประสิทธิภาพ

Algorithm	Best	Worst
Bubble Sort	n^2	n^2
Selection Sort	n^2	n^2
Insertion Sort	n	n^2
Shell Sort	$n \log n$	$n \log n$
Merge Sort	$n \log n$	$n \log n$
Quick Sort	$n \log n$	n^2
Bucket Sort	n	$n^2/k = n^2$
Radix Sort	n	$n \log_k n$

การวัดประสิทธิภาพ

Sort	10	100	1K	10K	100K	1M	Up	Down
Insertion	.0011	.051	4.55	447.7	48790	–	0.3	916.0
Bubble	.0018	.114	11.36	1250.6	143819	–	584.2	1012.2
Selection	.0015	.073	5.84	566.3	66510	–	561.9	589.1
Shell	.0018	.040	0.64	10.4	177	2980	3.7	6.8
Shell/O	.0017	.035	0.57	9.8	154	2680	2.9	5.2
Quick	.0026	.037	0.41	4.9	57	640	2.9	3.0
Quick/O	.0010	.022	0.30	3.9	47	560	1.5	1.5
Merge	.0039	.057	0.72	9.1	118	1490	6.7	6.7
Merge/O	.0012	.330	0.50	6.7	95	1250	6.8	6.6
Heap	.0034	.490	0.65	8.8	129	2080	7.9	7.4
Radix/4	.0379	.350	3.48	35.5	379	3990	35.4	35.4
Radix/8	.0345	.191	1.77	17.8	189	2010	17.8	17.7

Evaluating Sorting Algorithms

- ปัจจัยอื่น นอกเหนือจาก asymptotic complexity ส่งผลต่อประสิทธิภาพมีอะไรบ้าง

- สมมติว่าอัลกอริทึมสองชุดทำงานจำนวนคำสั่งเท่ากันทุกประการ อย่างใดอย่างหนึ่งจะดีกว่าอื่น ๆ ?

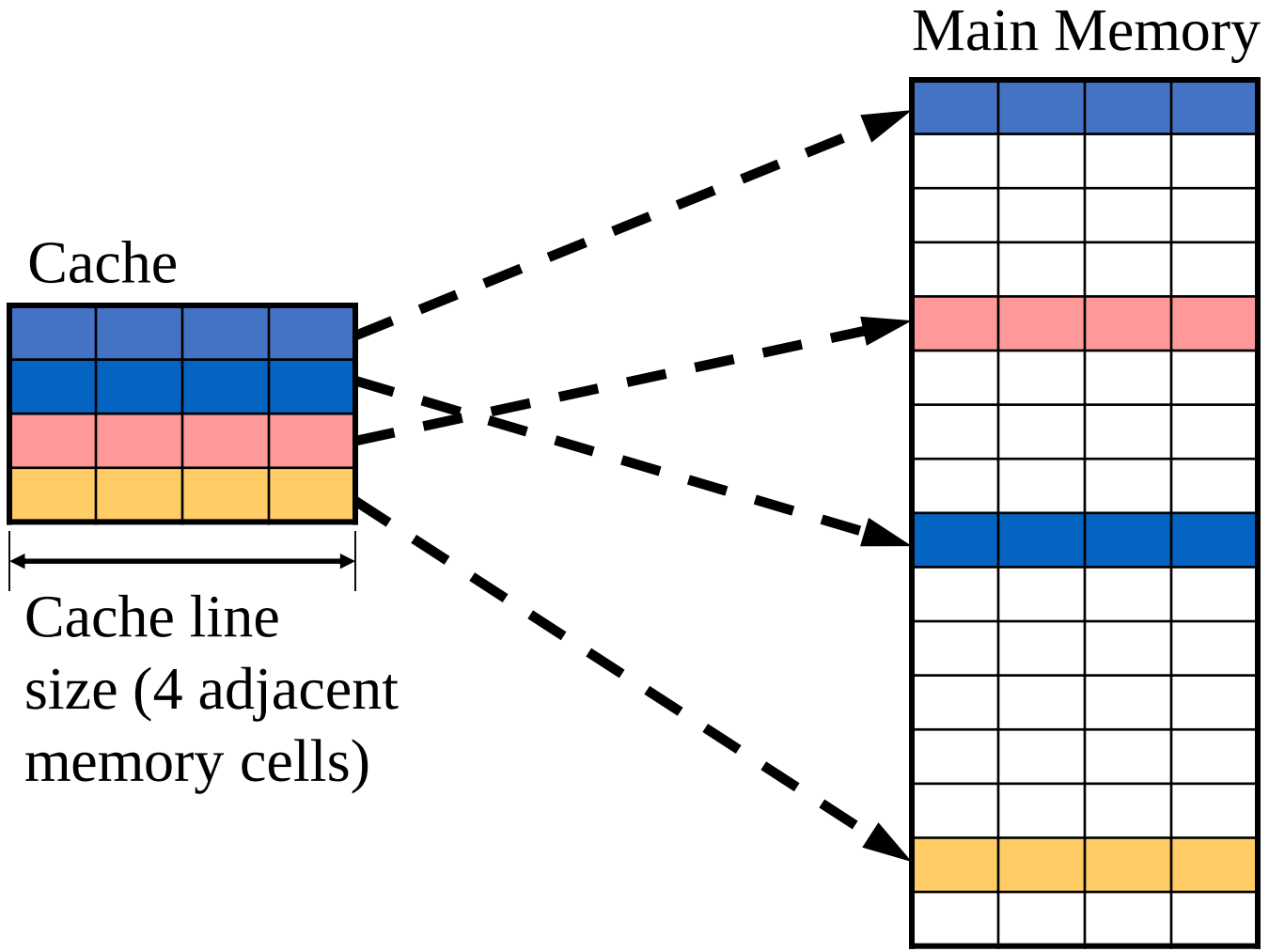
Example Memory Hierarchy Statistics

Name	Extra CPU cycles used to access	Size
L1 (on chip) cache	0	32 KB
L2 cache	8	512 KB
RAM	35	256 MB
Hard Drive	500,000	8 GB

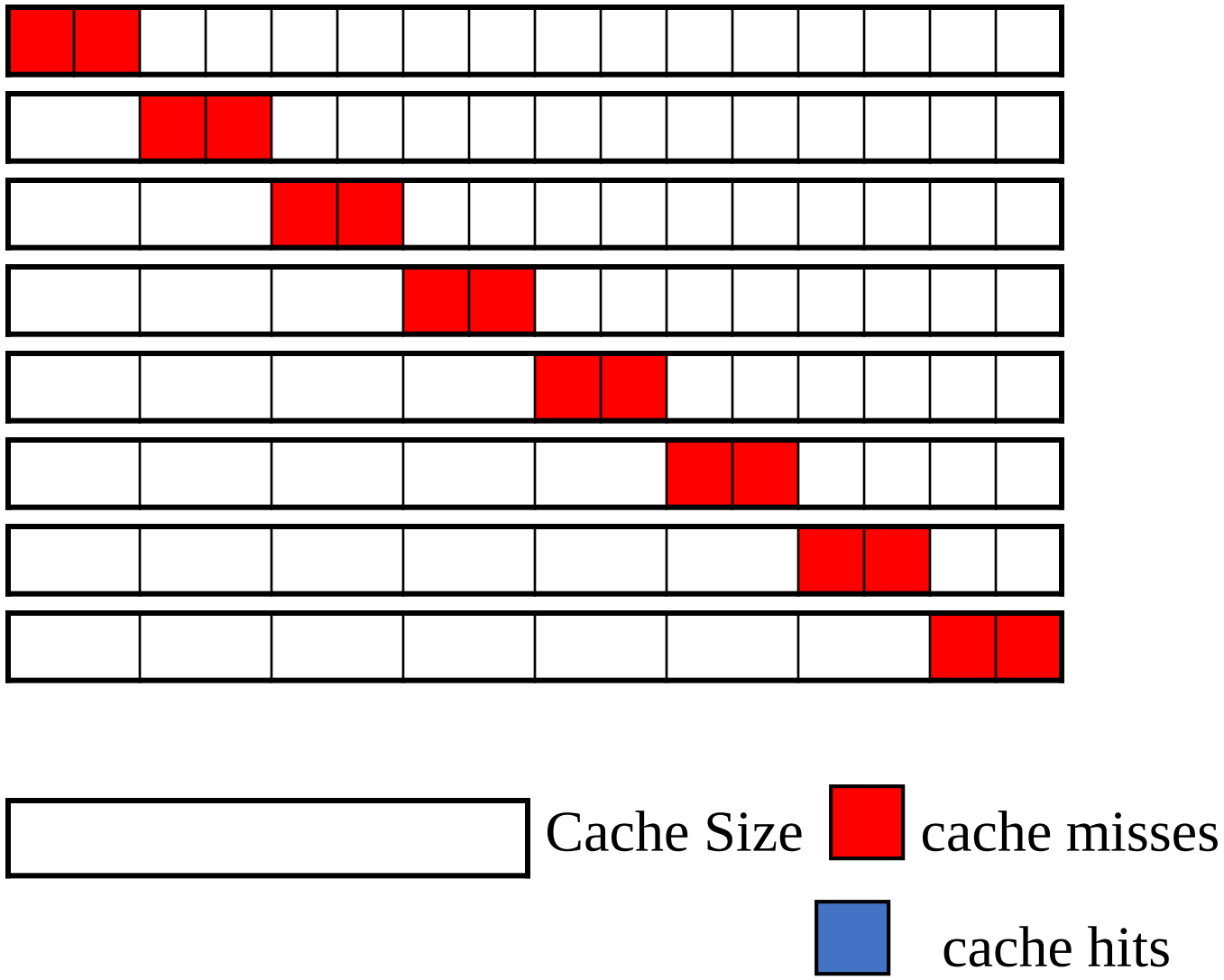
The Memory Hierarchy Exploits Locality of Reference

- Idea: *small* amount of *fast* memory
- Keep *frequently* used data in the *fast* memory
- LRU replacement policy
  - Keep recently used data in cache
  - To free space, remove Least Recently Used data
- Often significant practical reduction in runtime by minimizing cache misses

Cache Details (simplified)



Iterative Merge Sort

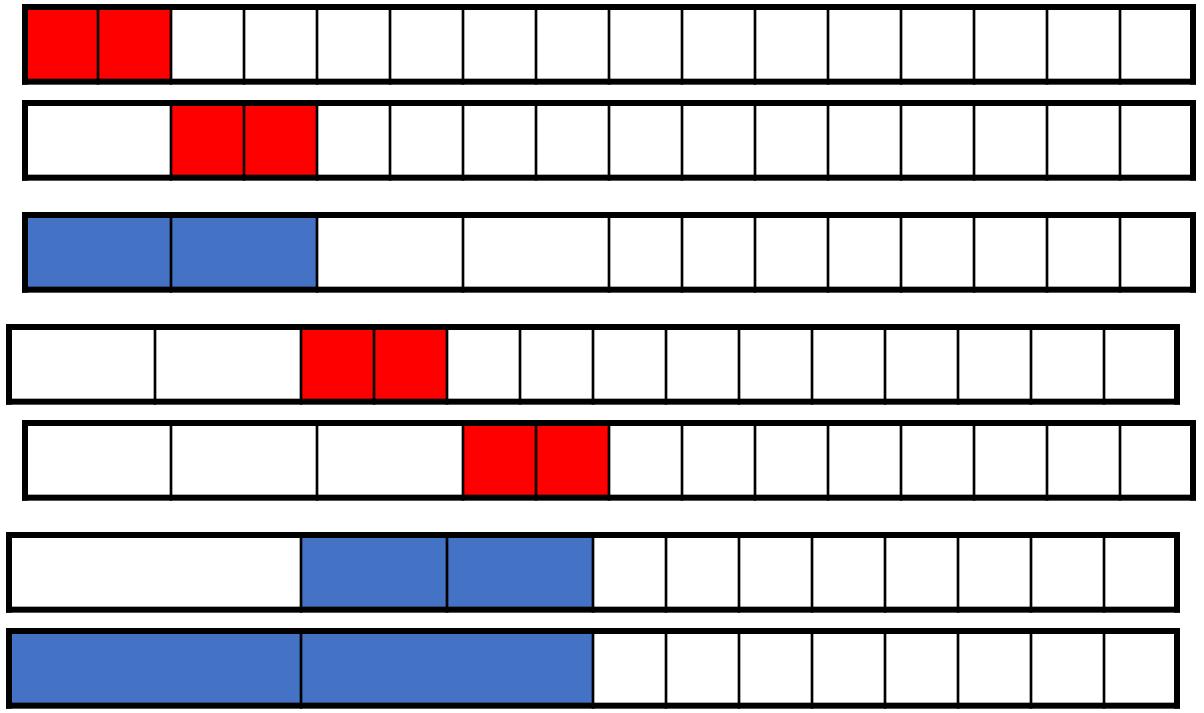


Iterative Merge Sort – cont'd

Cache Size

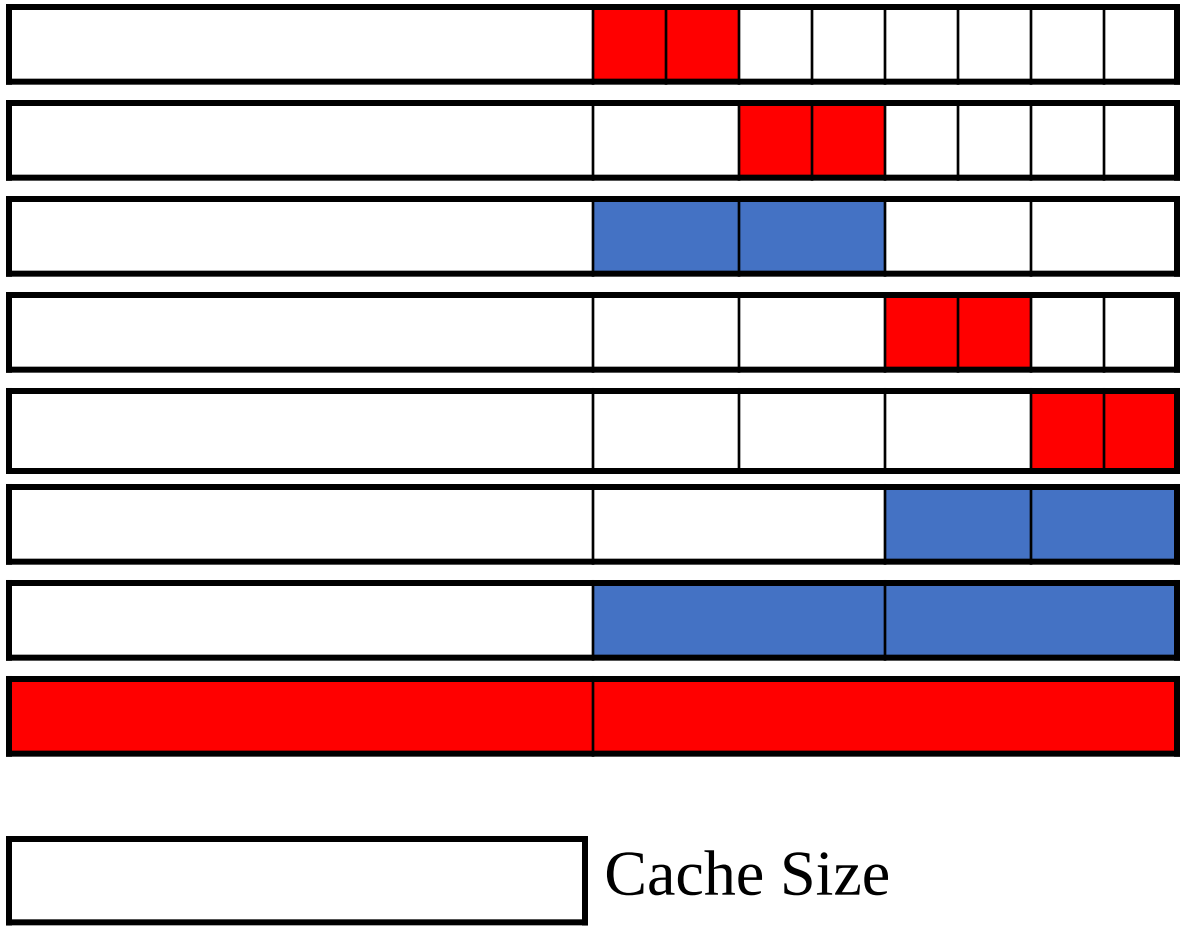
no temporal locality!

“Tiled” Merge Sort – better



Cache Size

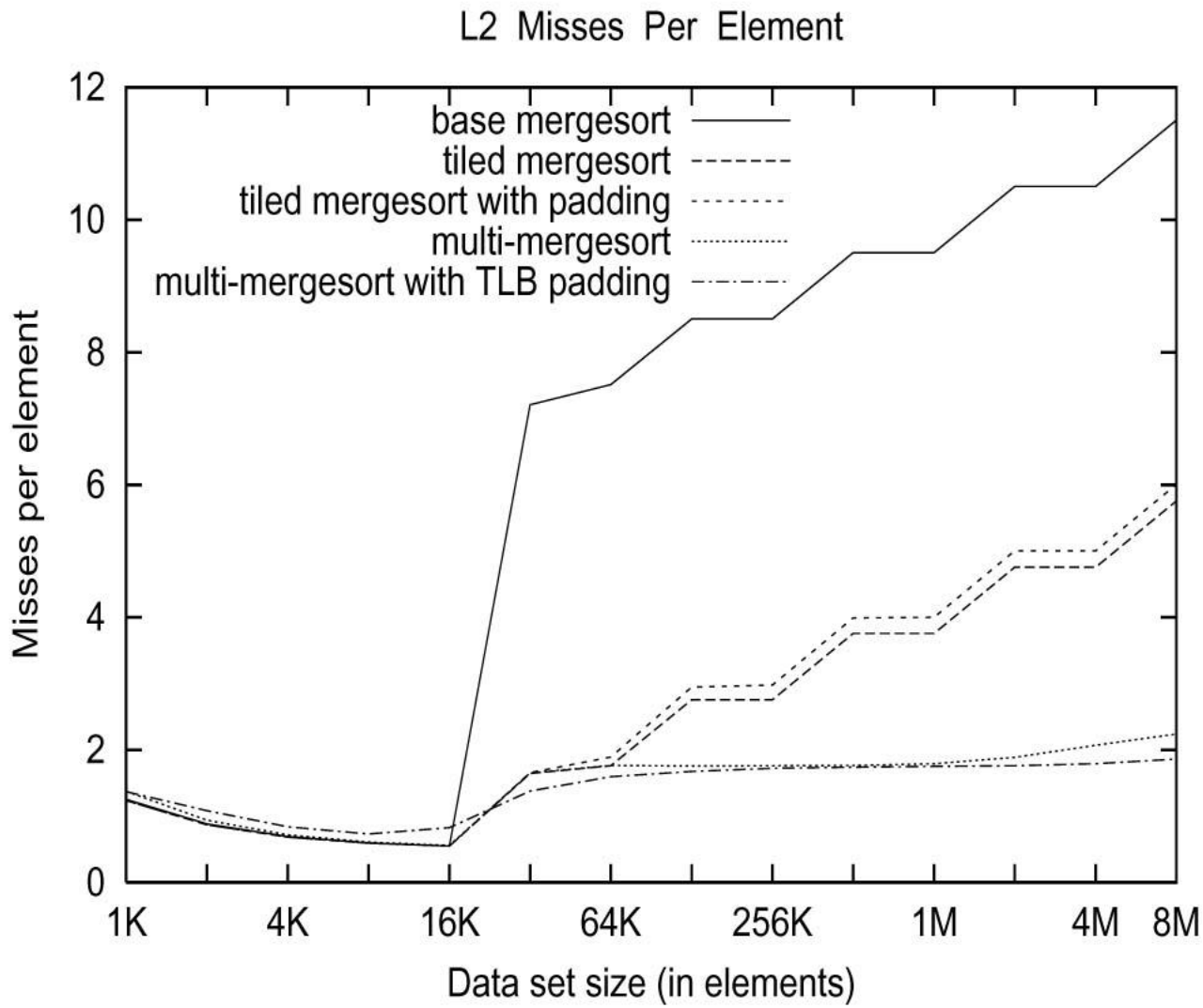
“Tiled” Merge Sort – cont’d

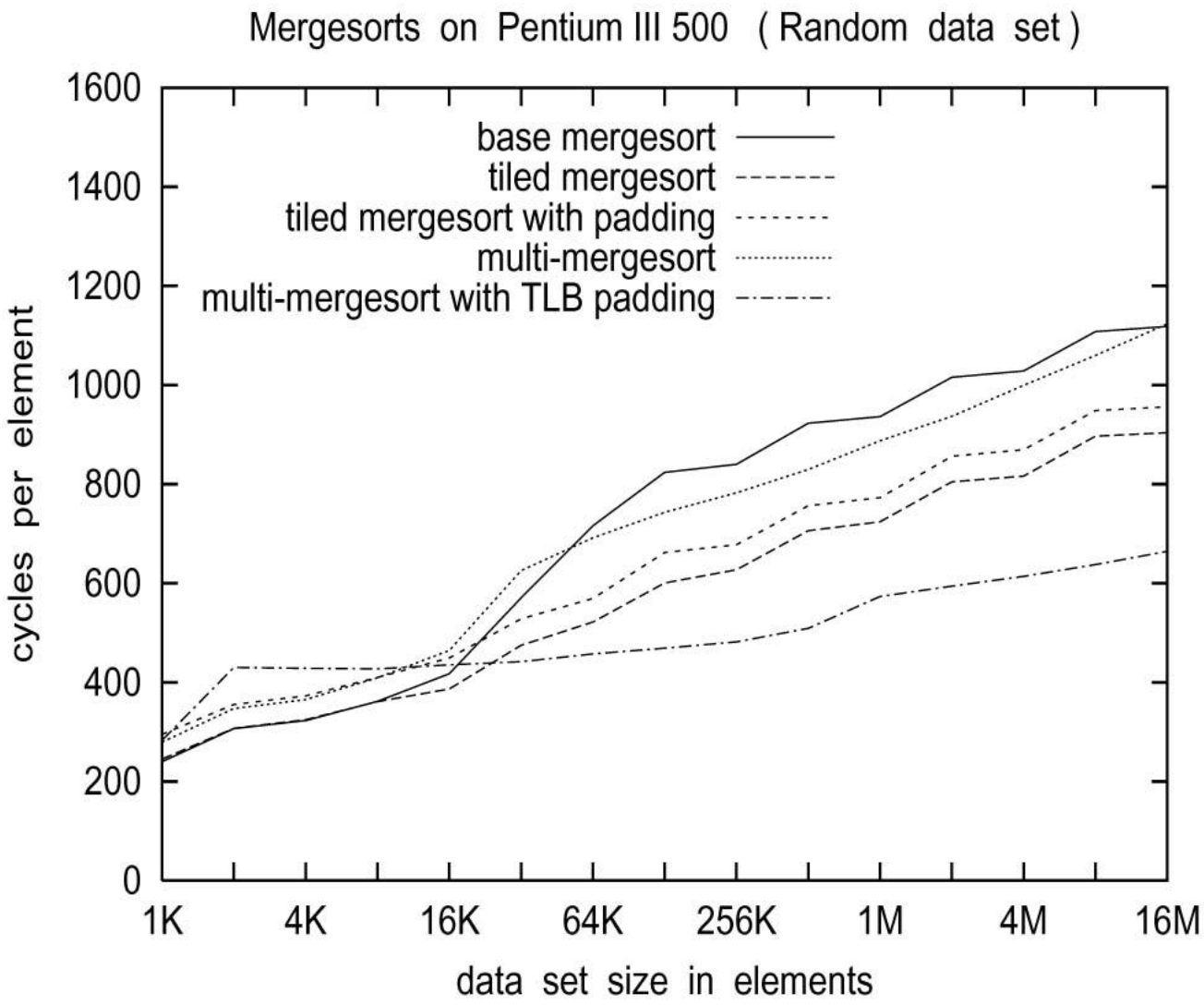


Cache Size

Additional Cache Optimizations

- "TLB Padding" – optimizes virtual memory
  - insert a few unused cells into array so that sub-problems fit into separate *pages* of memory
  - Translation Lookaside Buffer
- Multi-MergeSort – merge all "tiles" simultaneously, in a big (n/tilesize) multi-way merge
- Lots of tradeoffs – L1, L2, TBL cache, number of instructions


L2 Misses Per Element
base mergesort
tiled mergesort
tiled mergesort with padding
multi-mergesort
multi-mergesort with TLB padding
Misses per element
0
2
4
6
8
10
12
1K
4K
16K
64K
256K
1M
4M
8M
Data set size (in elements)


Mergesorts on Pentium III 500 (Random data set)
base mergesort
tiled mergesort
tiled mergesort with padding
multi-mergesort
multi-mergesort with TLB padding
cycles per element
0
200
400
600
800
1000
1200
1400
1600
1K
4K
16K
64K
256K
1M
4M
16M
data set size in elements

Other Sorting Algorithms

- Quicksort - สามารถเพิ่มประสิทธิภาพแคชที่คล้ายกันได้ – ยังดีกว่า Mergesort ที่ปรับแต่งได้ดีที่สุดเล็กน้อย
- Radix Sort – การใช้งานทั่วไปทำให้ใช้แคชได้ไม่ดี: ในแต่ละ Bucket
  - กวาดผ่าน input list- แคชหายไประหว่างทาง (ไม่ดี!)
  - ต่อท้าย output list – จัดทำ index ด้วยตัวเลขสุ่ม (ดีกว่า)
- ออกแรงเยอะหน่อยก็แข่งกับ QuickSort ได้

Timings(sec)						
n	PLSB 11	EBT 11	LSB 6	LSB 11	QSort	MSort
1M	**0.52**	0.67	0.68	0.90	0.70	1.02
2M	**1.05**	1.37	1.37	1.86	1.50	2.22
4M	**2.13**	2.78	2.76	3.86	3.24	4.47
8M	**4.25**	6.27	5.44	7.68	6.89	9.71
16M	**8.52**	10.83	10.87	15.23	14.65	19.47
32M	**17.03**	21.57	21.73	31.71	31.69	41.89

Fig. 4. Comparison of TLB-tuned LSB radix sorts: PLSB with radix 11 (PLSB 11), explicit block transfer with radix 11 (EBT 11) and normal LSB with 6-bit radix (LSB 6) versus normal LSB with 11-bit radix (LSB 11), memory-tuned quicksort (QSort) and tiled mergesort (MSort), on random 32-bit unsigned integers, M=10^6.

Conclusions

- ความเร็วของแคช, RAM และหน่วยความจำภายนอกมีผลอย่างมากต่อการจัดเรียง (และอัลกอริทึมอื่นๆ ด้วย)
- อัลกอริทึมที่มีความซับซ้อนเท่ากันอาจไม่ได้เร็วเท่ากันสำหรับหน่วยความจำประเภทต่างๆ
- การปรับแต่ง Cache สามารถปรับปรุงประสิทธิภาพของอัลกอริทึมได้อย่างมาก