Dual-pivot Quicksort

Dual Pivot Quick Sort

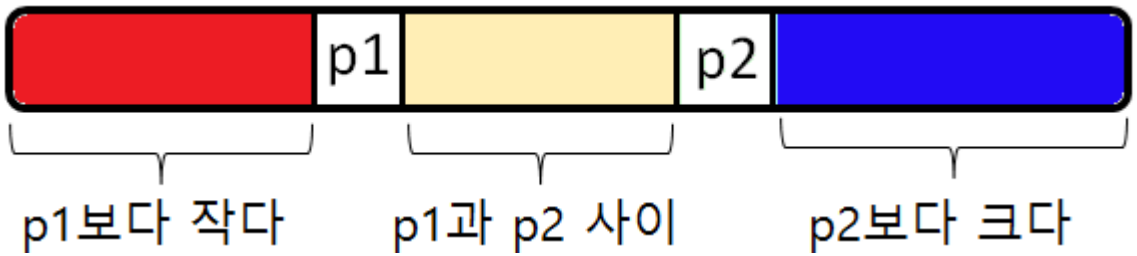
Vladimir Yaroslavskiy, 2009

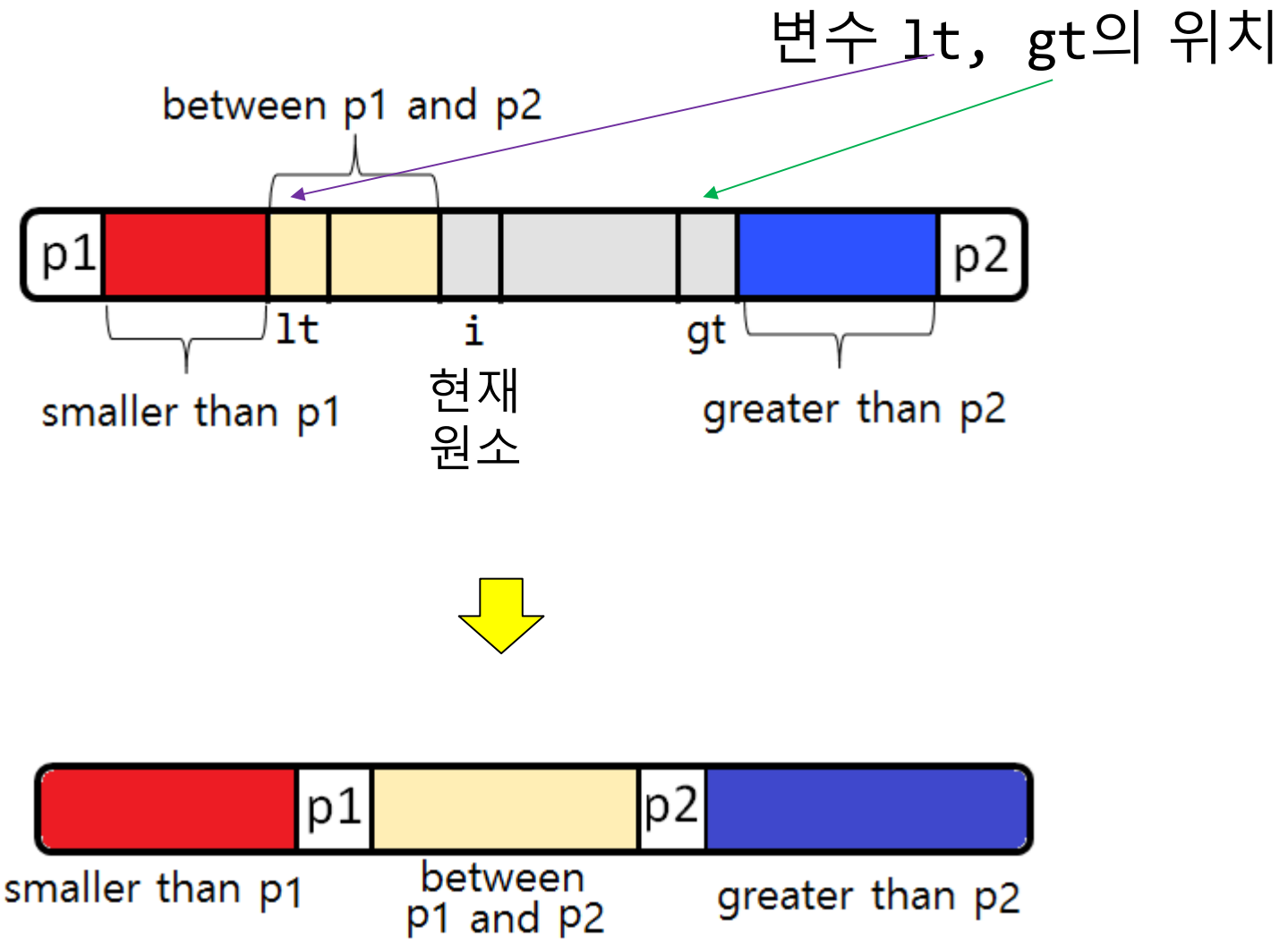
Java SE 7 이후 Arrays.sort() 원시 타입 데이터 정렬

순환 호출 시 입력 크기가 27 이하이면 삽입 정렬 호출

[핵심 아이디어]

p1 < p2




변수 lt, gt의 위치
between p1 and p2
p1
p2
lt
i
gt
smaller than p1
현재
원소
greater than p2
p1
p2
smaller than p1
between
p1 and p2
greater than p2

```
01 def dpqsort(a, low, high):
02     if high > low:
03         p1 = a[low]
04         p2 = a[high]
05         if p1 > p2:
06             a[low], a[high] = a[high], a[low]
07             p1 = a[low]
08             p2 = a[high]
09         i  = low+1
10         lt = low+1
11         gt = high-1
12         while i <= gt:
13             if a[i] < p1:
14                 a[i], a[lt] = a[lt], a[i]
15                 i  += 1
16                 lt += 1
17             elif p2 < a[i]:
18                 a[i], a[gt] = a[gt], a[i]
19                 gt -= 1
20             else:
21                 i += 1
22         lt -= 1
23         a[low], a[lt] = a[lt], a[low]
24         gt += 1
25         a[high], a[gt] = a[gt], a[high]
26         dpqsort(a, low,  lt-1)
27         dpqsort(a, lt+1, gt-1)
28         dpqsort(a, gt+1, high)
```

p1 < p2가 되도록

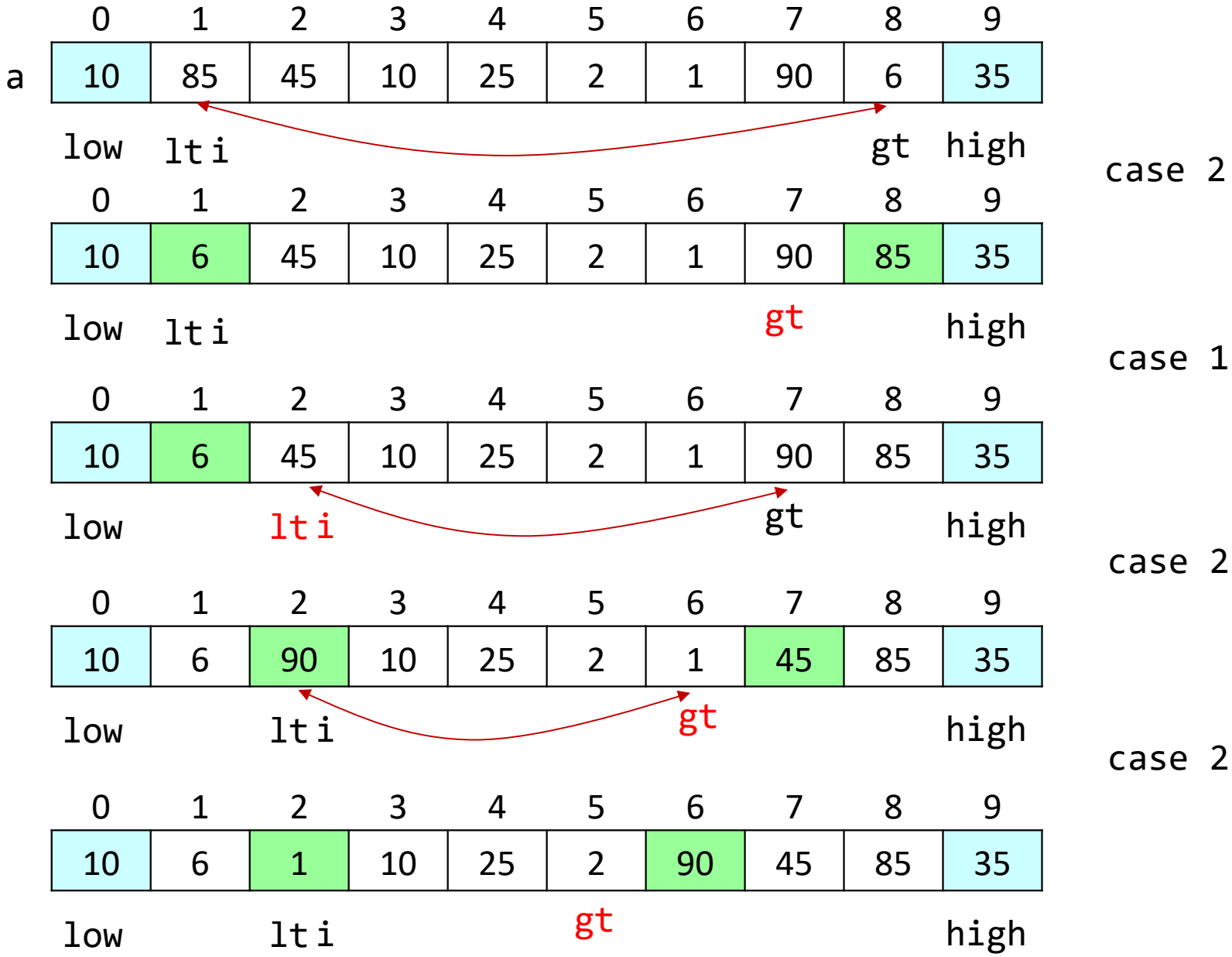
case 1

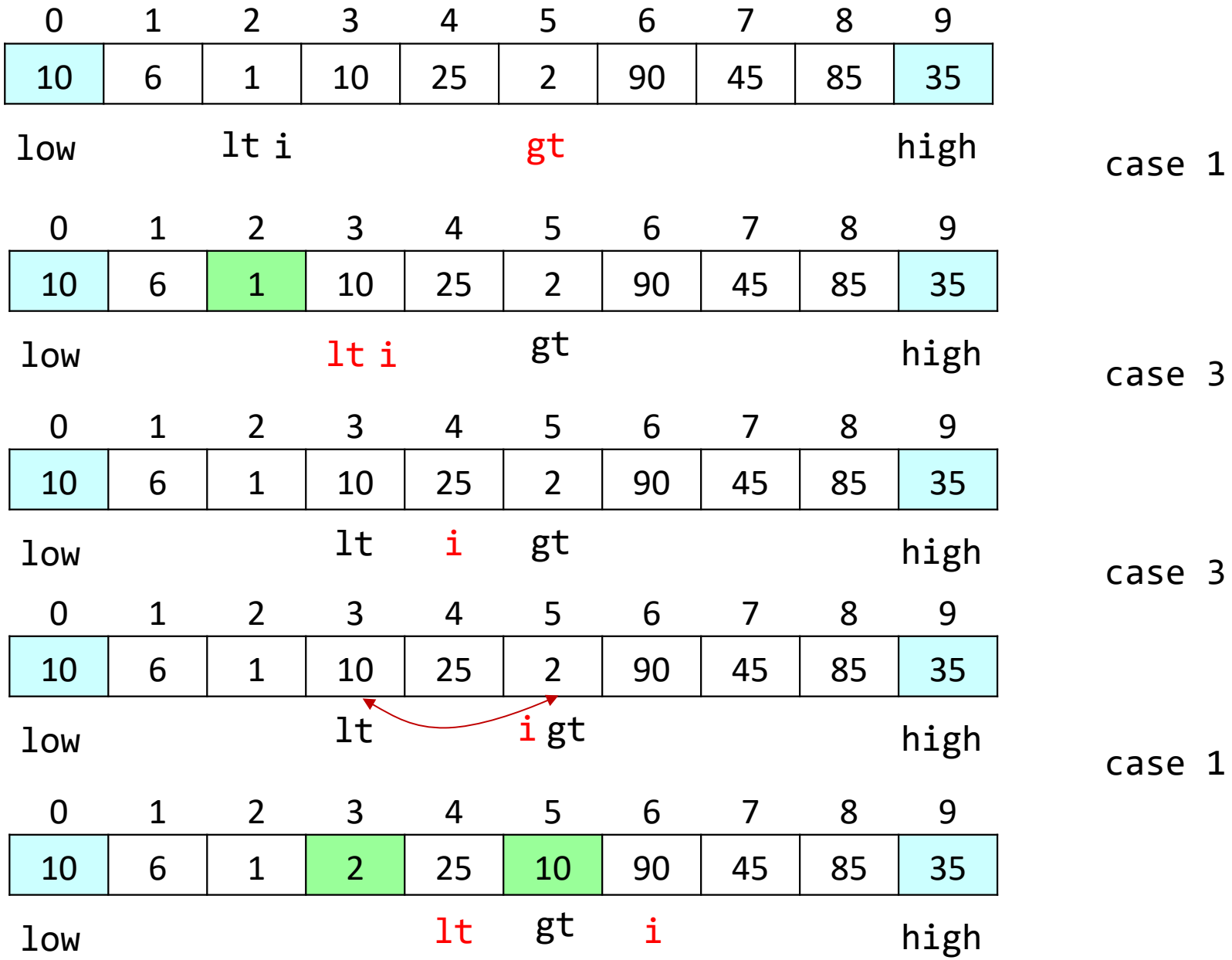
case 2

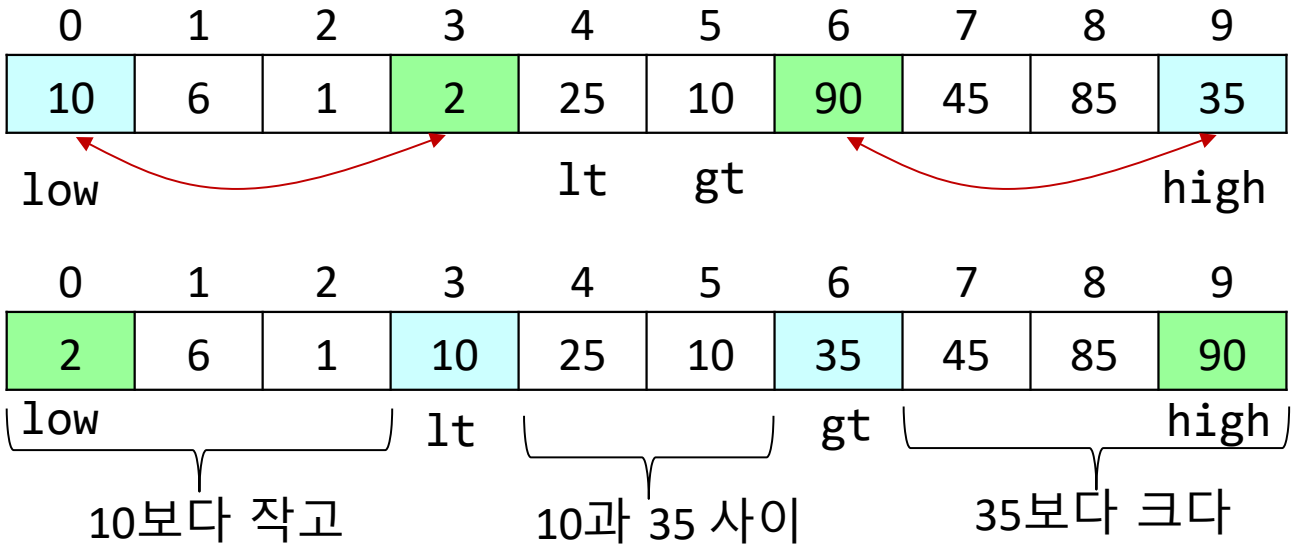
case 3

3부분 각각
순환 호출

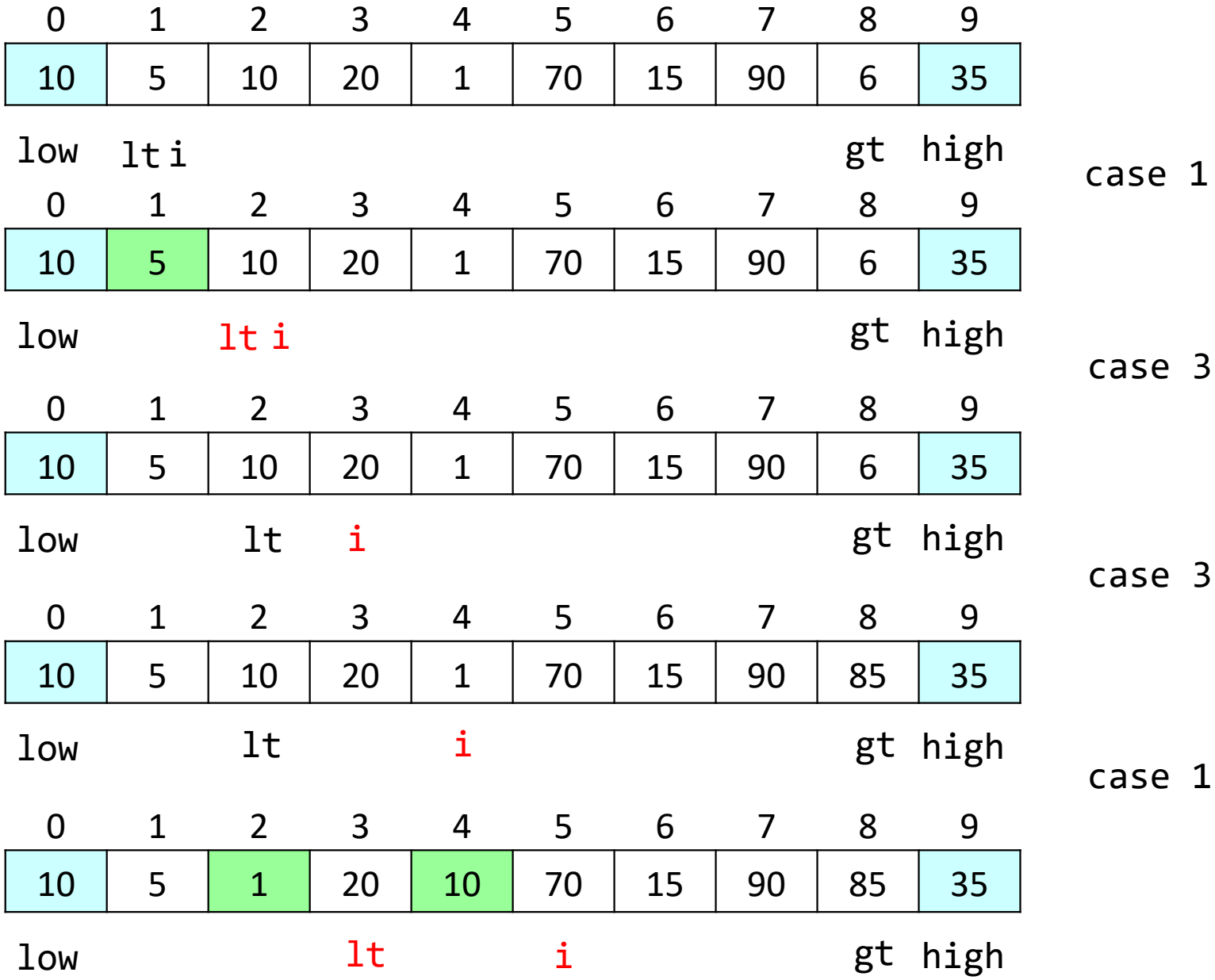
[예제]


0 1 2 3 4 5 6 7 8 9
a
10 85 45 10 25 2 1 90 6 35
low lt i gt high
case 2
0 1 2 3 4 5 6 7 8 9
10 6 45 10 25 2 1 90 85 35
low lt i gt high
case 1
0 1 2 3 4 5 6 7 8 9
10 6 45 10 25 2 1 90 85 35
low lt i gt high
case 2
0 1 2 3 4 5 6 7 8 9
10 6 90 10 25 2 1 45 85 35
low lt i gt high
case 2
0 1 2 3 4 5 6 7 8 9
10 6 1 10 25 2 90 45 85 35
low lt i gt high


0 1 2 3 4 5 6 7 8 9
10 6 1 10 25 2 90 45 85 35
low
lt i
gt
high
case 1
0 1 2 3 4 5 6 7 8 9
10 6 1 10 25 2 90 45 85 35
low
lt i
gt
high
case 3
0 1 2 3 4 5 6 7 8 9
10 6 1 10 25 2 90 45 85 35
low
lt
i
gt
high
case 3
0 1 2 3 4 5 6 7 8 9
10 6 1 10 25 2 90 45 85 35
low
lt
i gt
high
case 1
0 1 2 3 4 5 6 7 8 9
10 6 1 2 25 10 90 45 85 35
low
lt
gt
i
high


0 1 2 3 4 5 6 7 8 9
10 6 1 2 25 10 90 45 85 35
low
lt
gt
high
0 1 2 3 4 5 6 7 8 9
2 6 1 10 25 10 35 45 85 90
low
lt
gt
high
10보다 작고
10과 35 사이
35보다 크다

[예제 2]


0 1 2 3 4 5 6 7 8 9
10 5 10 20 1 70 15 90 6 35
low lt i gt high
case 1
0 1 2 3 4 5 6 7 8 9
10 5 10 20 1 70 15 90 6 35
low lt i gt high
case 3
0 1 2 3 4 5 6 7 8 9
10 5 10 20 1 70 15 90 6 35
low lt i gt high
case 3
0 1 2 3 4 5 6 7 8 9
10 5 10 20 1 70 15 90 85 35
low lt i gt high
case 1
0 1 2 3 4 5 6 7 8 9
10 5 1 20 10 70 15 90 85 35
low lt i gt high

성능

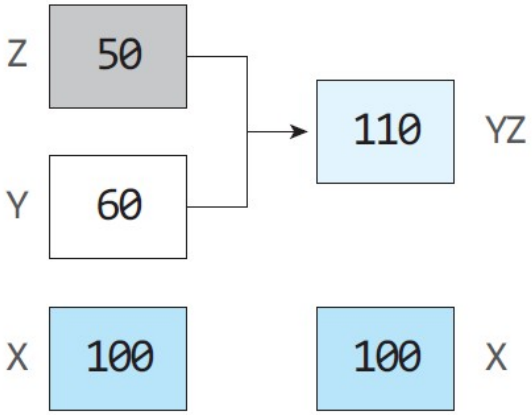
- 이중 피봇 퀵 정렬은 실제 수행 성능이 퀵 정렬보다 우수
- 자바 SE7 이후 원시 타입(primitive type) 시스템 정렬로 사용
- 이론적 비교: 이중 피봇 퀵 정렬의 평균 비교 횟수가 ~1.9nlnn이고, 퀵 정렬은 ~2.0nlnn
- 이중 피봇 퀵 정렬이 캐시 메모리(cache memory)의 접근이 훨씬 효율적이기 때문에 좋은 성능을 보임

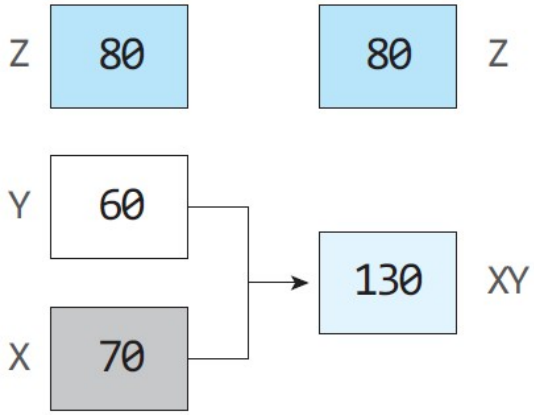
Tim Sort

Tim Sort

입력에 대해 삽입 정렬을 수행하여 일정 크기의 런(run)을 만들어 일정 조건에 따라 합병하는 정렬 알고리즘

- 스택에 push한 후에 가장 위에 있는 3개의 런(정렬된 부분 입력)의 크기를 차례로 Z, Y, X라고 하면, 다음의 두 가지 조건이 충족되도록 런들을 [알고리즘 V-1]과 같이 합병한다.
  (1) X > Y + Z
  (2) Y > Z



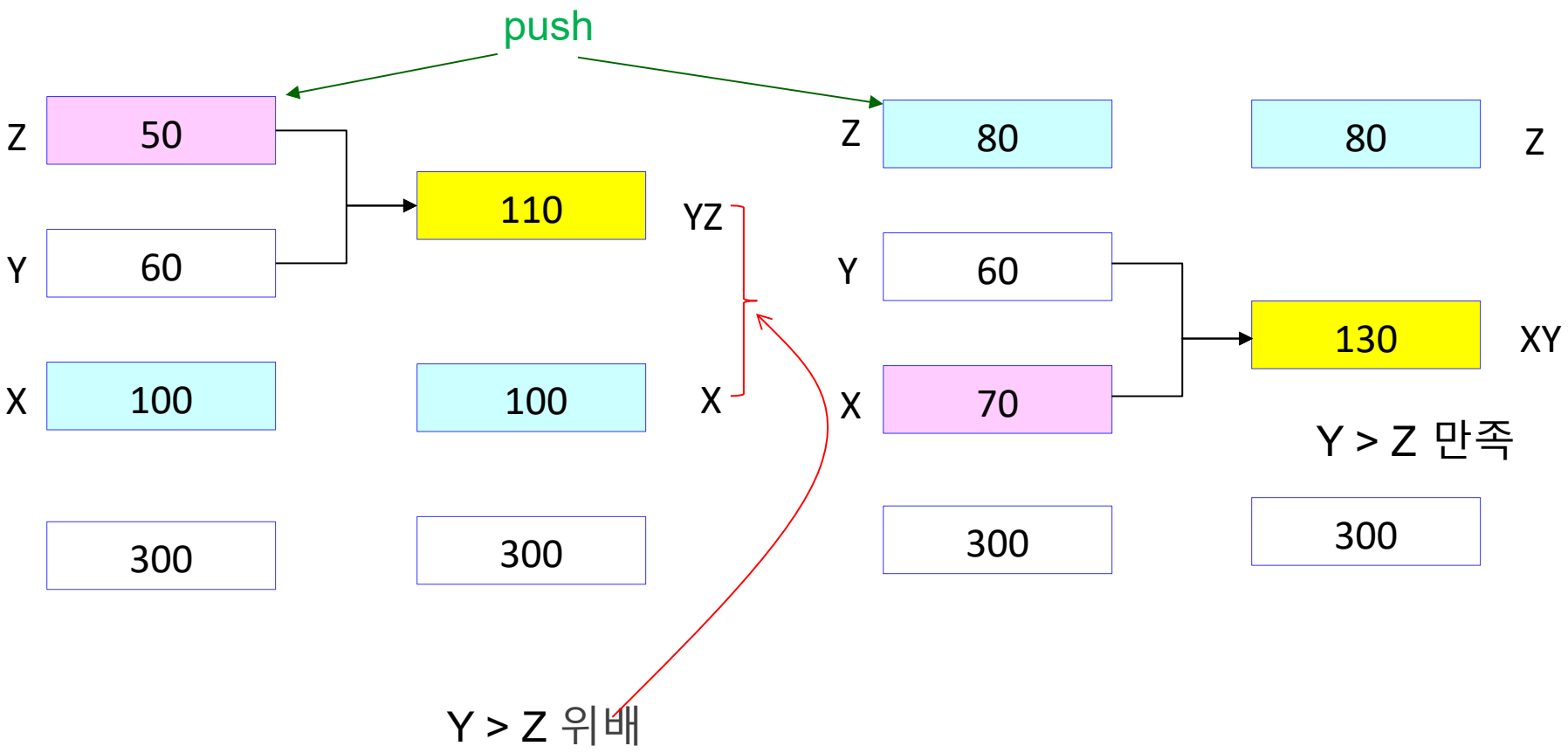


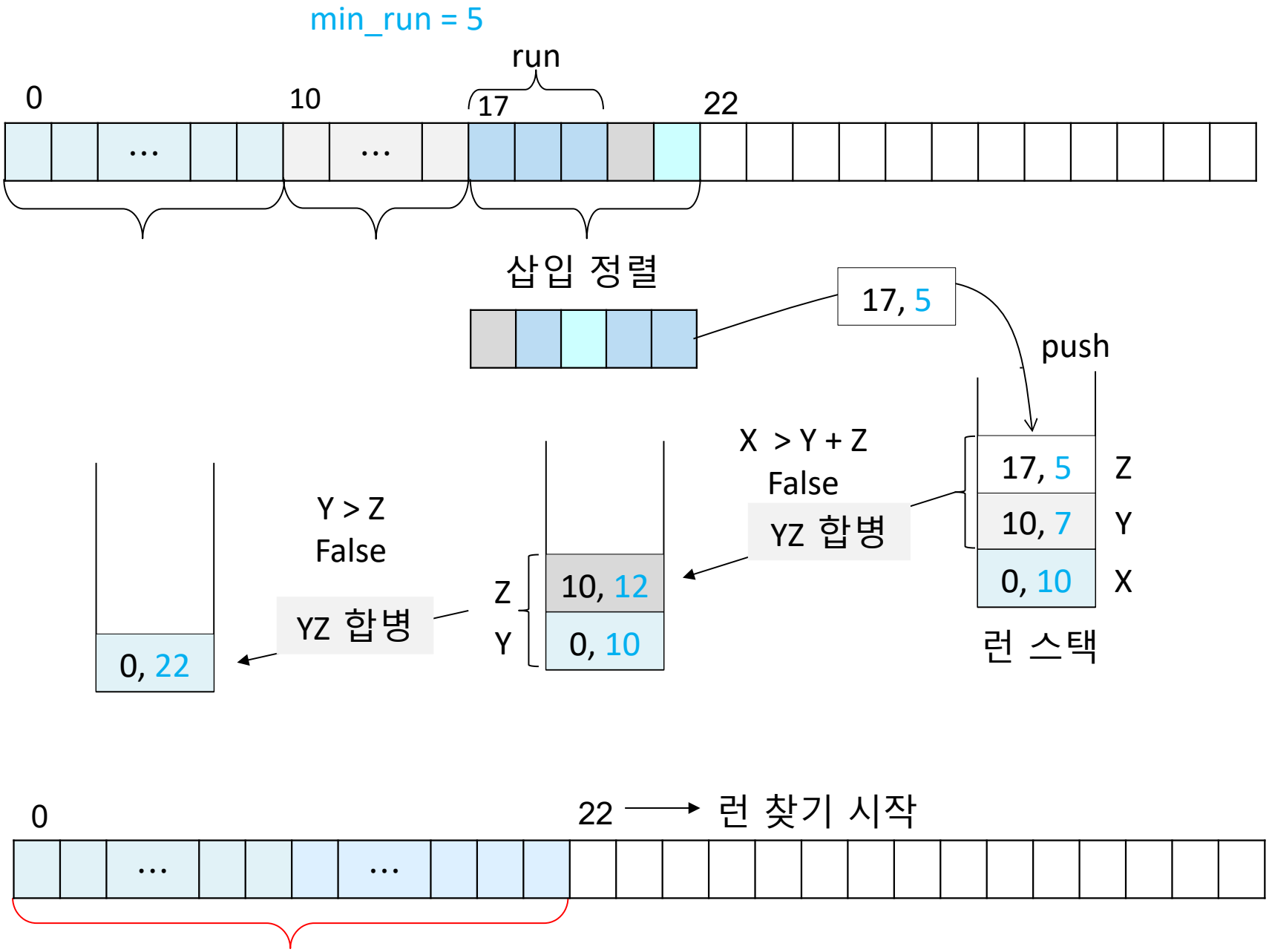
http://cr.openjdk.java.net/~martin/webrevs/openjdk7/timsort/raw_files/new/src/share/classes/java/util/TimSort.java

```
while stack_size > 1:
  top = stack_size-1
  if top>2 and run[top-2]<=run[top-1]+run[top]: # 조건(1)이 위배
      if run[top-2]<run[top]:
          merge(top-2,top-1)
      else:
          merge(top-1,top)
  elif run[top-1]<=run[top]: # 조건(2)가 위배
      merge(top-1, top)
  else:                      # 두 조건 모두 만족
      break
```

[알고리즘 V-1]

[예제 2]


push
Z
50
Y
60
X
100
300
110
100
300
YZ
X
Y > Z 위배
Z
80
Y
60
X
70
300
80
Z
130
XY
Y > Z 만족
300


min_run = 5
run
0
10
17
22
삽입 정렬
17, 5
push
17, 5
10, 7
0, 10
Z
Y
X
런 스택
X > Y + Z
False
YZ 합병
10, 12
0, 10
Z
Y
Y > Z
False
YZ 합병
0, 22
0
22
런 찾기 시작

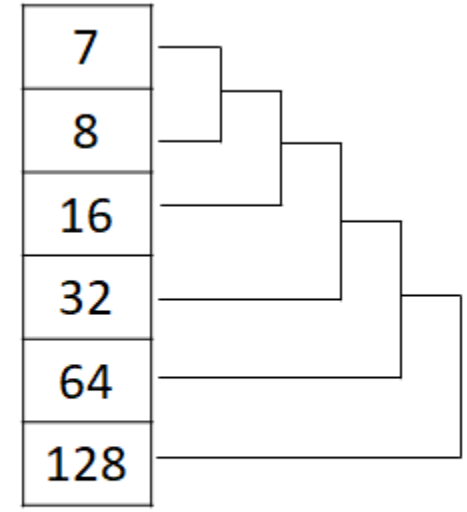
min_run 크기 정하기

- min_run이 256이면 너무 크고, 8이면 너무 작음
- 실험 검증 결과: 32에서 65가 좋은 성능을 보임

```
def min_run_len(N):
    r = 0
    while N >= 64 :
        r = r | (N & 1)
        N >>= 1
    return N + r
```

- 입력 크기인 n의 이진수에서 최상위 6비트를 추출한 수에 r을 더한다. 여기서 n의 이진수의 최상위 6비트를 제외한 나머지 비트에서 적어도 1개의 1비트가 있으면 r =1이 된다.

➢ 런 스택에 1개의 항목이 남을 때까지 합병



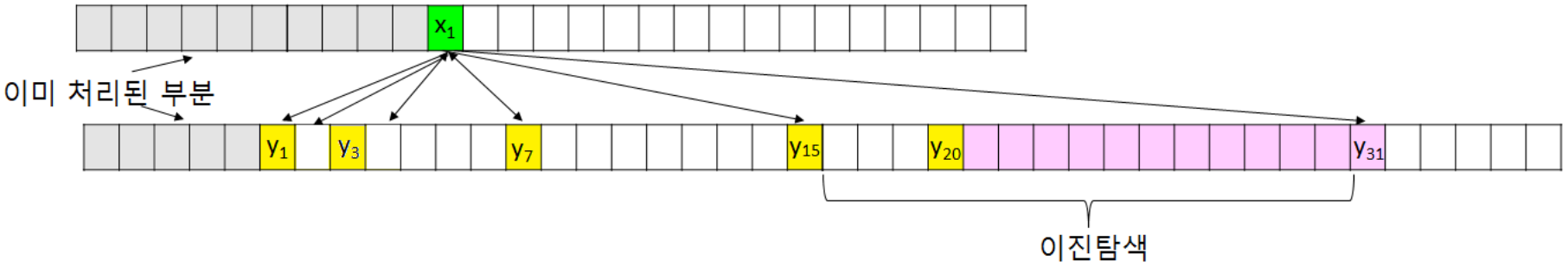
런스택

완전히 균형된 합병

가장 이상적인 합병

Galloping 합병

[예측] X와 Y를 합병하는 중에 한쪽에서 예를 들어 X에서 승자가 연속해서 많이 나왔다면 이젠 Y에서 승자가 연속해서 많이 나올 것이다.



- y_{20}은 y_{15}와 y_{31}사이에서 x_1과 같거나 작은 수들 중에 가장 큰 수
- 이진 탐색으로 y_{20}을 찾은 후 y_1부터 y_{20}을 한꺼번에 (승자로서) 출력

Tim Sort 성능

- 최선 경우 O(n): 입력이 이미 정렬된 경우
- 최악, 평균 경우 O(nlogn)
- 안정적인(stable) 정렬

정렬 알고리즘의 성능 비교

	최선 경우	평균 경우	최악 경우	추가 공간	안정성
선택 정렬	n^2	n^2	n^2	$O(1)$	X
삽입 정렬	n	n^2	n^2	$O(1)$	○
쉘 정렬	$nlogn$	?	$n^{1.5}$	$O(1)$	X
힙 정렬	$nlogn$	$nlogn$	$nlogn$	$O(1)$	X
합병 정렬	$nlogn$	$nlogn$	$nlogn$	n	○
퀵 정렬†	$nlogn$	$nlogn$	n^2	$O(1)$*	X
Tim Sort	n	$nlogn$	$nlogn$	n	○

* 퀵 정렬에서 수행되는 순환 호출까지 고려한다면 추가 공간은 $O(\log n)$이다.
  단, 순환 호출 시 작은 부분을 먼저 호출한 경우의 분석이다.

† 이중 피벗 퀵 정렬의 이론적인 성능은 퀵 정렬과 같아서 생략