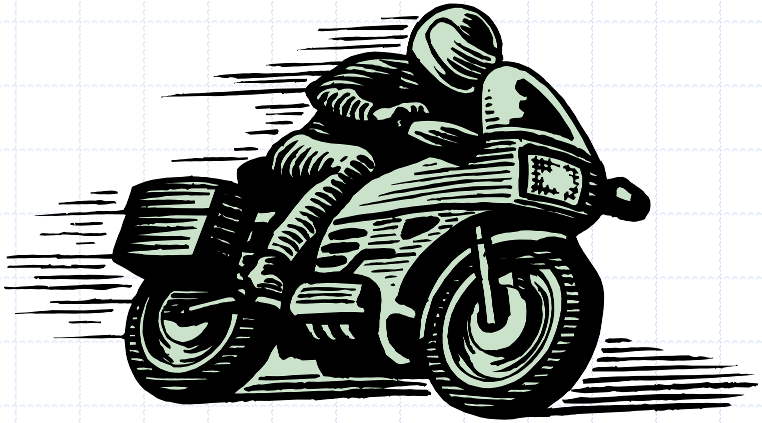
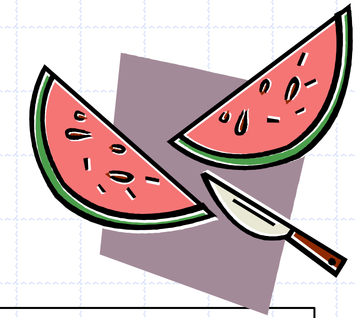
퀵 정렬



Outline

- 8.1 퀵 정렬
- 8.2 무작위 퀵 정렬
- 8.3 제자리 퀵 정렬
- 8.4 합병 정렬과 퀵 정렬 비교
- 8.5 응용문제

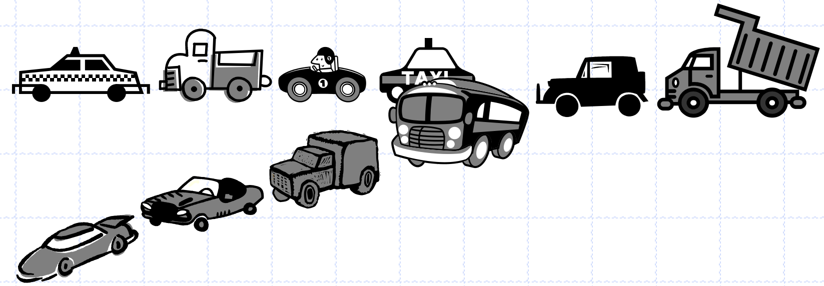
퀵 정렬



- **퀵 정렬**(quick-sort): **분할통치법**에 기초한 정렬 알고리즘
  1. **분할**(divide): 기준원소 ***p***를(pivot, 보통은 마지막 원소)를 택하여 ***L***을 다음 세 부분으로 분할
     - ***LT*** (***p***보다 작은 원소들)
     - ***EQ*** (***p***와 같은 원소들)
     - ***GT*** (***p***보다 큰 원소들)
  2. **재귀**(recur): ***LT***와 ***GT***를 정렬
  3. **통치**(conquer): ***LT***, ***EQ***, ***GT***를 결합

Alg ***quickSort(L)***
    input list ***L*** with ***n*** elements
    output sorted list ***L***

1. **if** (***L.size***() > 1)
        k ← *a position in* ***L***
        LT, ***EQ***, ***GT*** ← ***partition***(***L***, ***k***)
        quickSort(***LT***)
        quickSort(***GT***)
        L ← ***merge***(***LT***, ***EQ***, ***GT***)
2. **return**



리스트 분할

- 입력 리스트를 다음과 같이 분할
  1. L로부터 각 원소 e를 차례로 삭제
  2. e를 **기준원소** p와의 비교 결과에 따라 부리스트 LT, EQ, GT에 삽입
- 삽입과 삭제를 리스트의 맨 앞이나 맨 뒤에서 수행하므로 **O**(1) 시간 소요
- 따라서, quick-sort의 분할 단계는 **O**(n) 시간 소요

Alg *partition*(L, k)
  input list L with n elements,
    position k of pivot
  output sublists LT, EQ, GT of the
    elements of L, less than, equal to,
    or greater than pivot, resp.

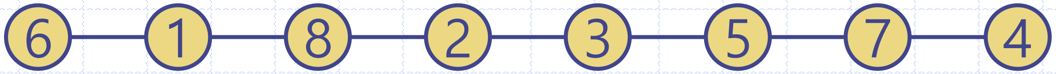
1. $p \leftarrow L.get(k)$ {pivot}
2. $LT, EQ, GT \leftarrow$ *empty list*
3. **while** (!$L.isEmpty()$)
    $e \leftarrow L.removeFirst()$
    if ($e < p$)
      $LT.addLast(e)$
    elseif ($e = p$)
      $EQ.addLast(e)$
    else {$e > p$}
      $GT.addLast(e)$
4. **return** LT, EQ, GT

기준원소 선택

- 리스트 원소 가운데 **기준원소**(pivot) 선택
  - **결정적**이며 쉬운 방법
    - 맨 앞 원소
    - 맨 뒤 원소
    - 중간 원소
  - **결정적**이며 조금 복잡한 방법
    - 맨 앞, 중간, 맨 뒤 위치의 세 원소의 중앙값(median)
    - 0/4, 1/4, 2/4, 3/4, 4/4 위치 다섯 원소의 중앙값
    - 전체 원소의 중앙값
  - **무작위**한 방법
    - 무작위 방식으로 원소 선택
- 기준원소 선택의 영향
  - 분할 결과
  - 퀵 정렬 수행 성능

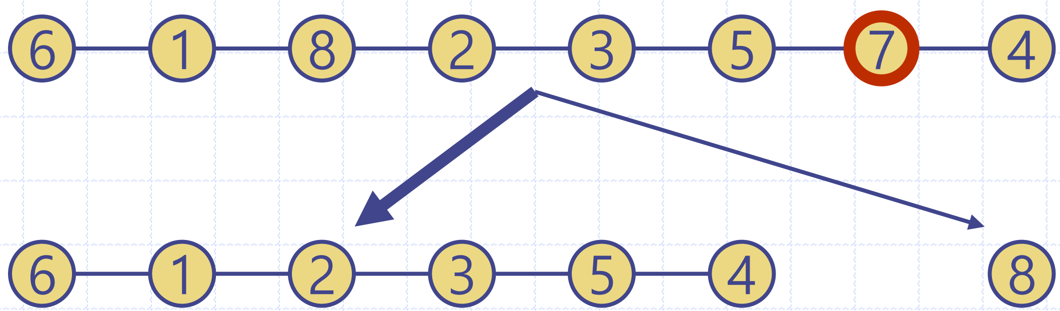
퀵 정렬 트리

- quick-sort의 실행을 이진트리로 보이기
  - 이진트리의 각 노드는 quick-sort의 **재귀호출**을 표현하며 다음을 저장
    - 실행 이전의 무순 리스트 및 기준원소
    - 실행 이후의 정렬 리스트
  - 루트는 **초기 호출**을 의미
  - 잎들은 크기 0 또는 1의 부리스트에 대한 호출을 의미
- 실행예를 위한 입력 리스트



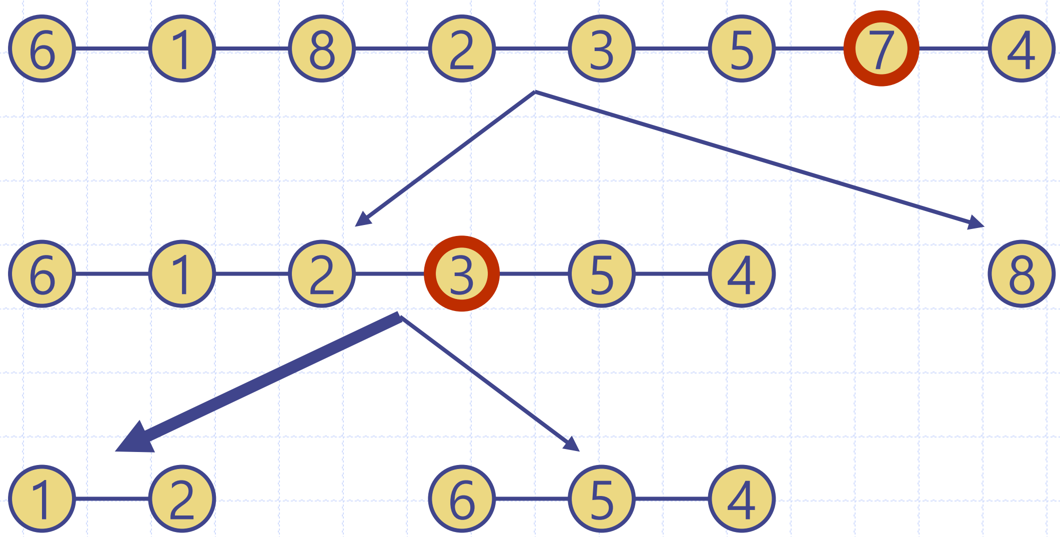
퀵 정렬 수행 예

- 초기 호출, 분할, 재귀 호출



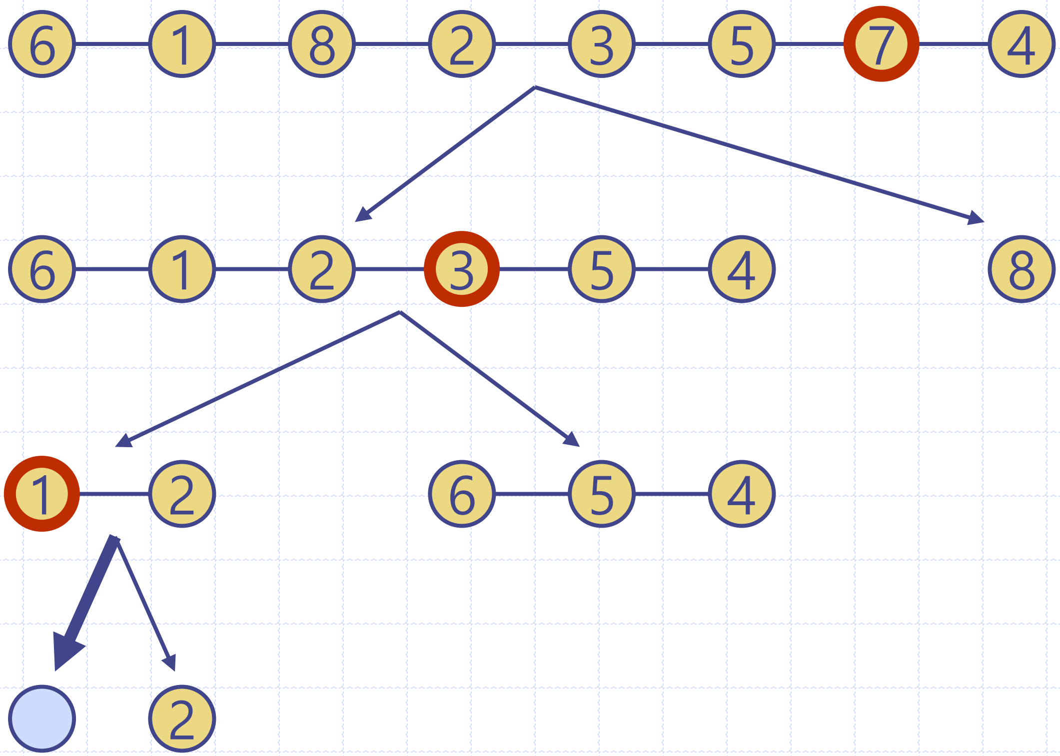
퀵 정렬 수행 예 (conti.)

- 분할, 재귀호출



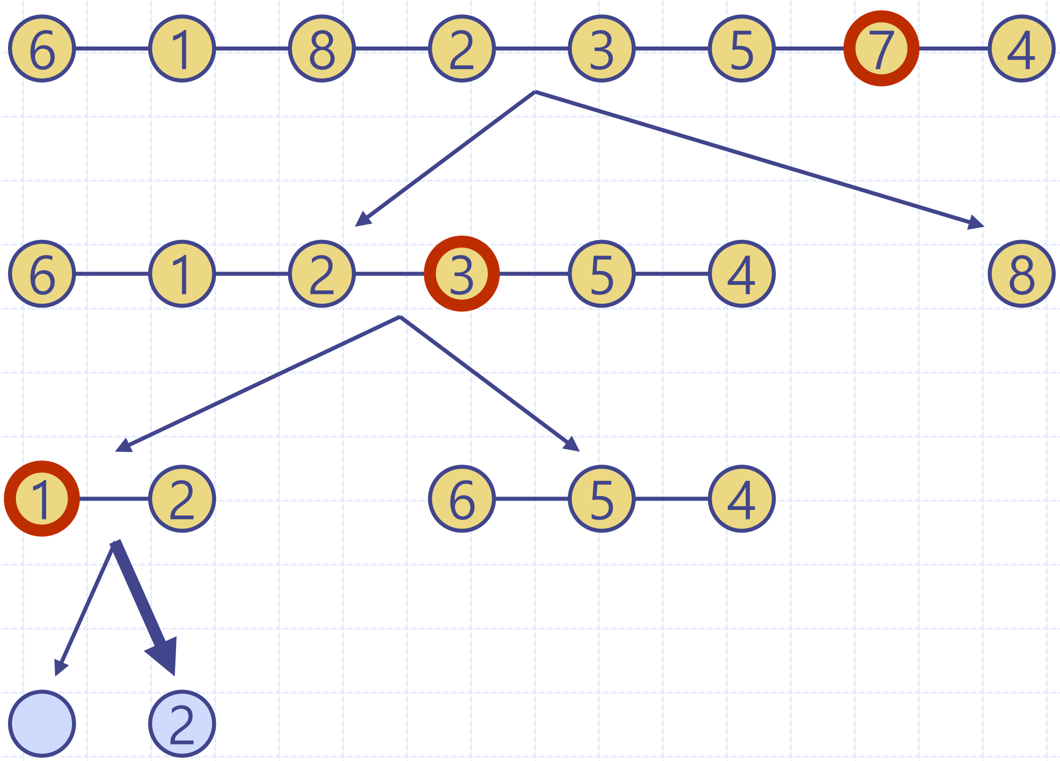
퀵 정렬 수행 예 (conti.)

- 분할, 재귀 호출, 베이스 케이스



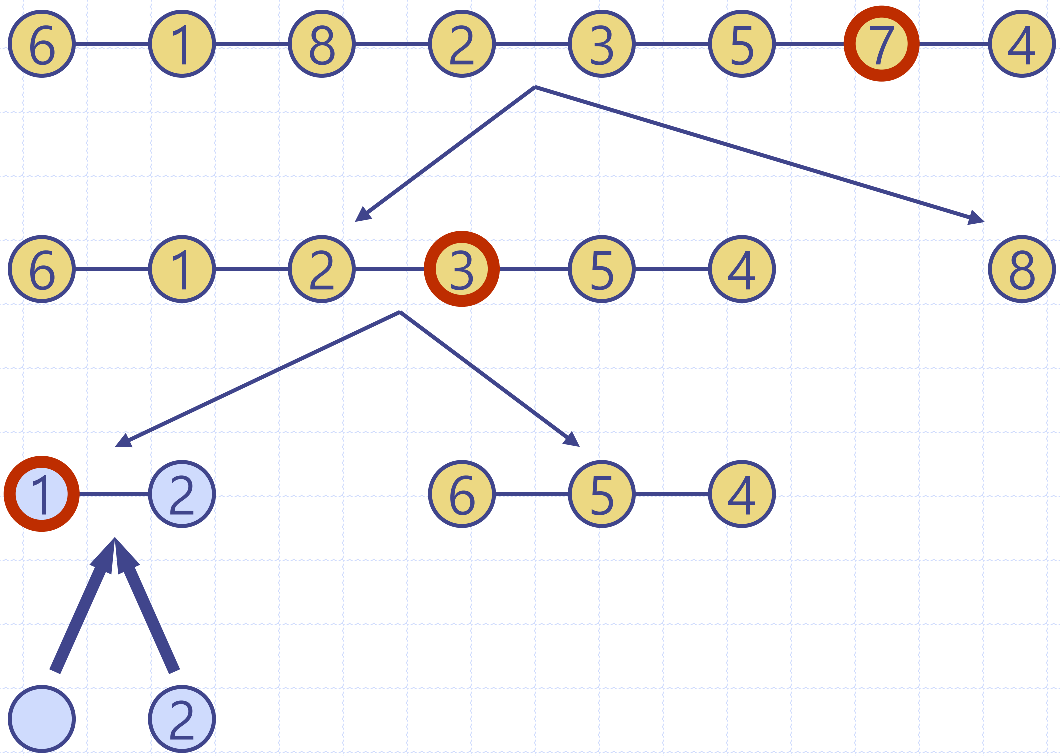
퀵 정렬 수행 예 (conti.)

- 재귀 호출, 베이스 케이스



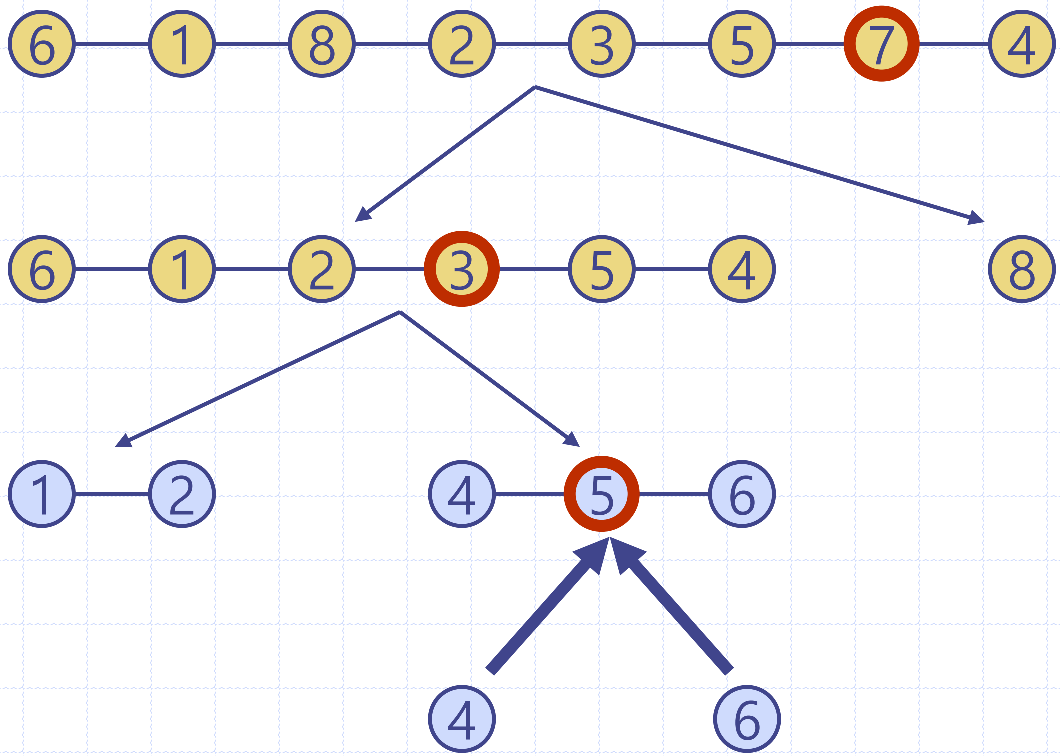
퀵 정렬 수행 예 (conti.)

- 결합



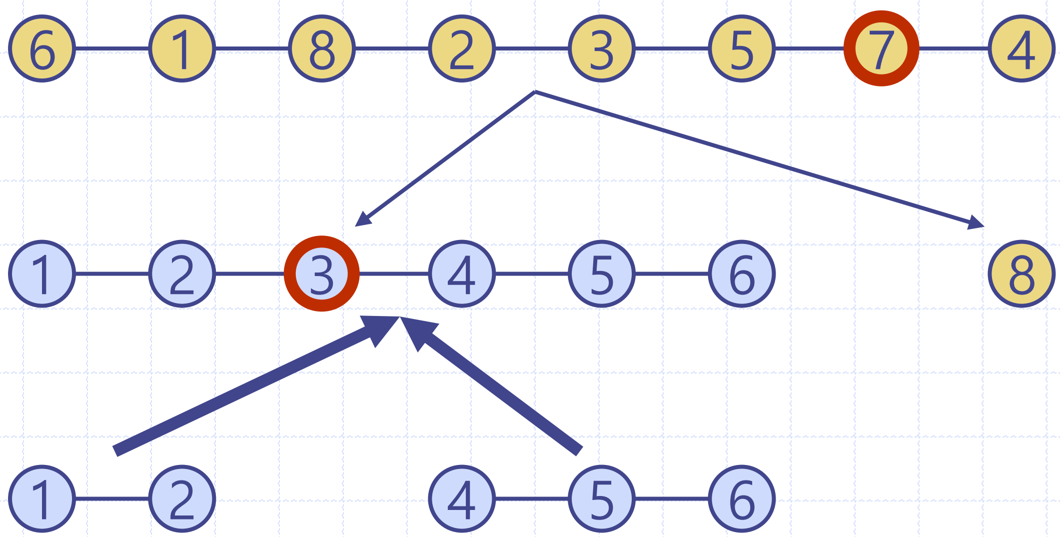
퀵 정렬 수행 예 (conti.)

- 재귀 호출, 분할, … , 결합



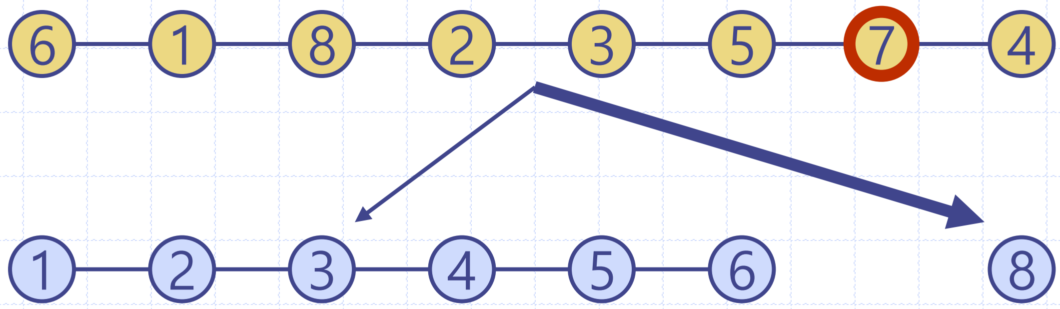
퀵 정렬 수행 예 (conti.)

- 결합



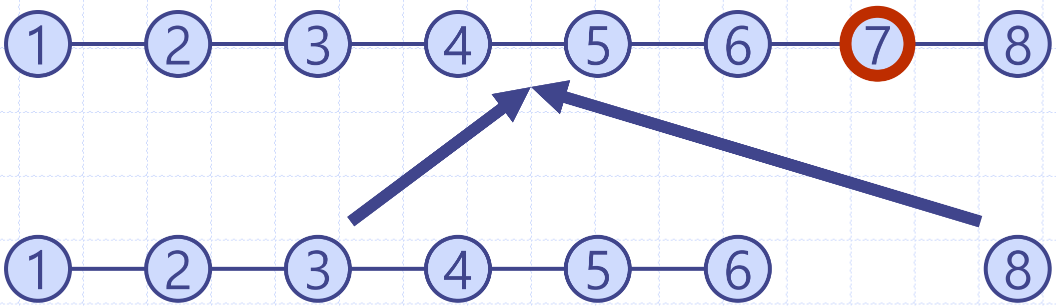
퀵 정렬 수행 예 (conti.)

- 재귀 호출, 베이스 케이스



퀵 정렬 수행 예 (conti.)

- 결합

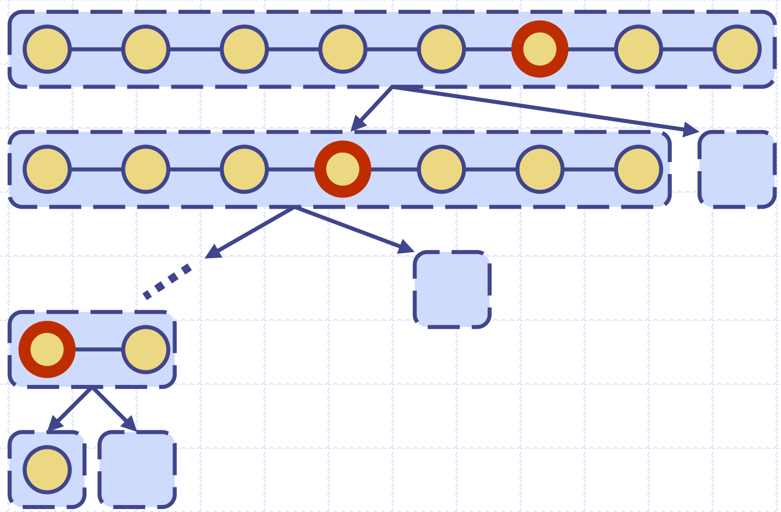


최악실행시간

- quick-sort의 **최악**은 기준원소가 항상 유일한 최소이거나 최대 원소일 경우
- 이 경우 LT와 GT 가운데 하나는 크기가 $n - 1$이며, 다른 쪽은 크기가 0
- 실행시간은 다음 합에 비례

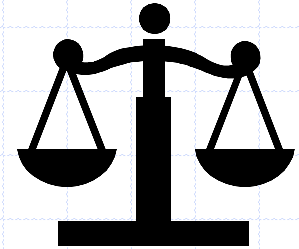
$$n + (n - 1) + \ldots + 2 + 1$$

- 따라서, quick-sort의 **최악실행시간: $O(n^2)$**

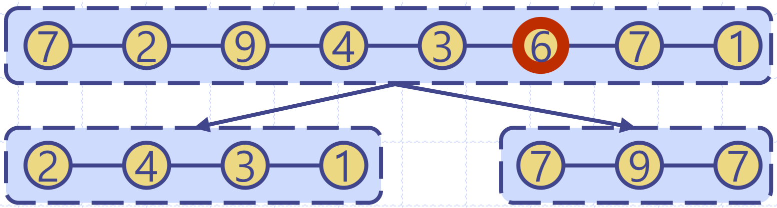


depth	time
0	n
1	$n - 1$
…	…
$n - 1$	1

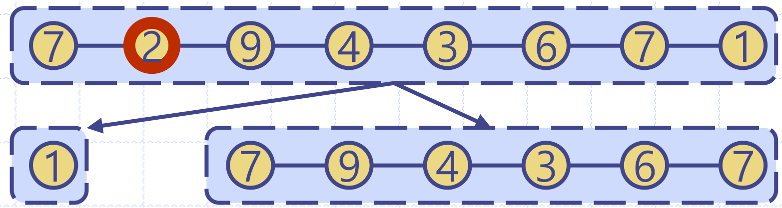
기대실행시간



- 크기 *s*의 리스트에 대한 quick-sort의 재귀 호출을 고려하면,
  - **좋은 호출:** ***LT***와 ***GT***의 크기가 모두 (3/4)***s*** 보다 작다
  - **나쁜 호출:** ***LT***와 ***GT***의 가운데 하나의 크기가 (3/4)***s*** 보다 크다

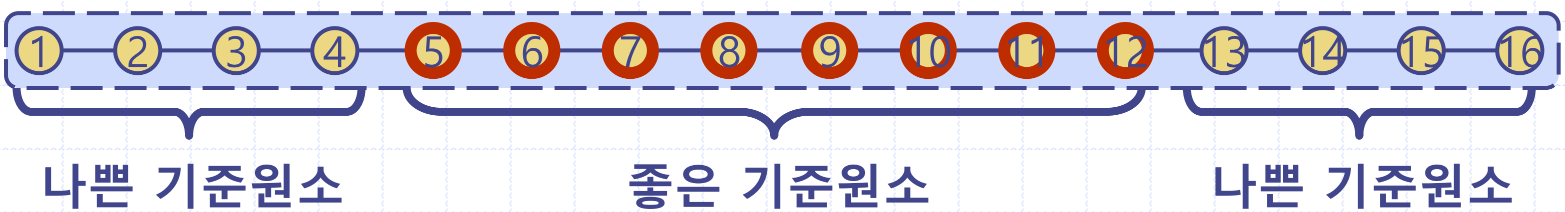


좋은 호출

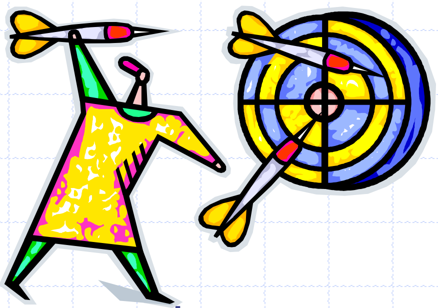


나쁜 호출

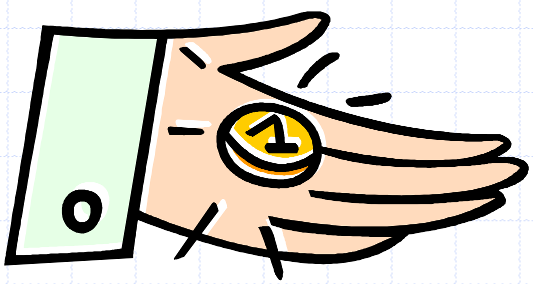
- 호출이 **좋을** 확률은 ½ (**예:** 동전던지기)
  - 가능한 기준원소의 ½은 좋은 호출을 부른다



무작위 퀵 정렬

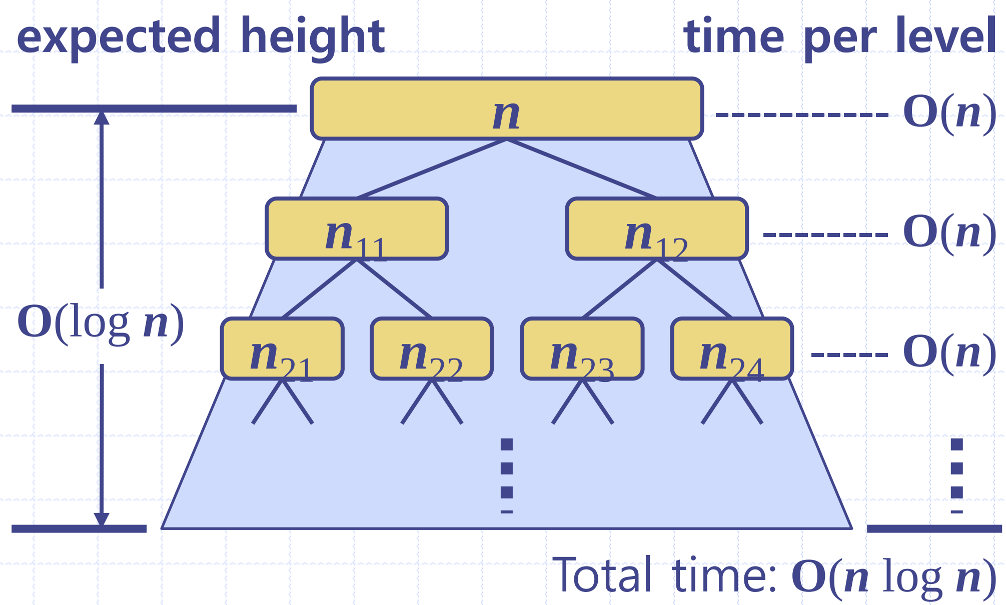


- quick-sort의 **결정적** 버전에서는, 기준원소로서 리스트로부터의 특정한 원소, 즉 **마지막 원소**를 선택하였다
- 기준원소 선택을 위한 새로운 규칙:
  "입력 리스트의 **무작위**(random) **원소**를 선택하라"
- **확률적 상식**: k개의 헤드를 얻기 위한 동전던지기의 기대회수는 $2k$다

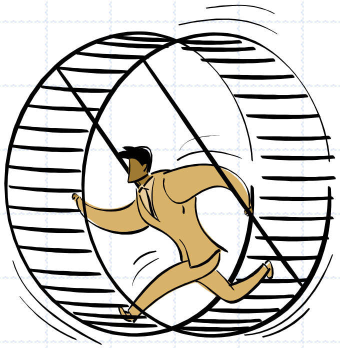


무작위 퀵 정렬의 기대실행시간

- 깊이 i의 노드에 대해 다음을 **기대**할 수 있다
  - $i/2$개의 조상: **좋은 호출**
  - 현재 호출을 위한 입력 리스트의 크기: 최대 $(3/4)^{i/2}n$
- 따라서,
  - 깊이 $2\log_{4/3}n$의 노드에 대해*, 기대 입력 크기: 1
  - 퀵 정렬 트리의 기대 높이: $O(\log n)$
- 같은 깊이의 노드들에 대해 수행되는 작업량: $O(n)$
- 따라서, quick-sort의 **기대실행시간:** $O(n \log n)$



제자리 퀵 정렬



- quick-sort를 **제자리**에서 수행되도록 구현 가능
- 분할 단계에서, 입력 리스트의 원소들을 재배치하기 위해 **대체**(replace) 작업을 사용 – 즉,
  - ***LT*** (***a*** 보다 아래의, 기준원소보다 작은 원소들)
  - ***EQ*** (***a***와 ***b*** 사이의, 기준원소와 같은 원소들)
  - ***GT*** (***b*** 보다 위의, 기준원소보다 큰 원소들)

Alg ***inPlaceQuickSort(L, l, r)***
input list ***L***, position ***l***, ***r***
output list ***L*** with elements of position from ***l*** to ***r*** rearranged in increasing order

1. **if** (***l*** $\geq$ ***r***)
   return
2. ***k*** $\leftarrow$ *a position between* ***l*** *and* ***r***
3. ***a***, ***b*** $\leftarrow$ ***inPlacePartition***(***L***, ***l***, ***r***, ***k***)
4. ***inPlaceQuickSort***(***L***, ***l***, ***a*** – 1)
5. ***inPlaceQuickSort***(***L***, ***b*** + 1, ***r***)

- 재귀호출은 ***LT***와 ***GT*** 부리스트 대해 수행

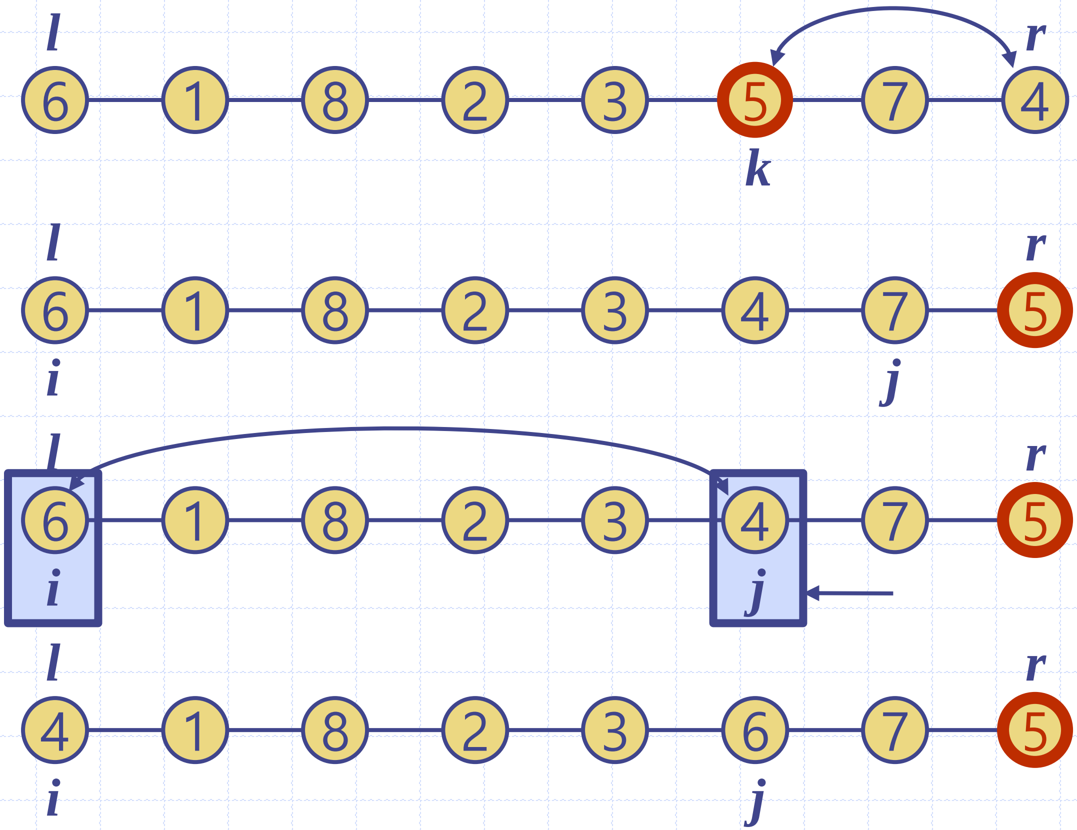
제자리 분할

Alg ***inPlacePartition***(A, l, r, k)
  input array $A[l..r]$ of *distinct* elements, index l, r, k
  output final index of the pivot resulting from partitioning $A[l..r]$ into LT, pivot, GT

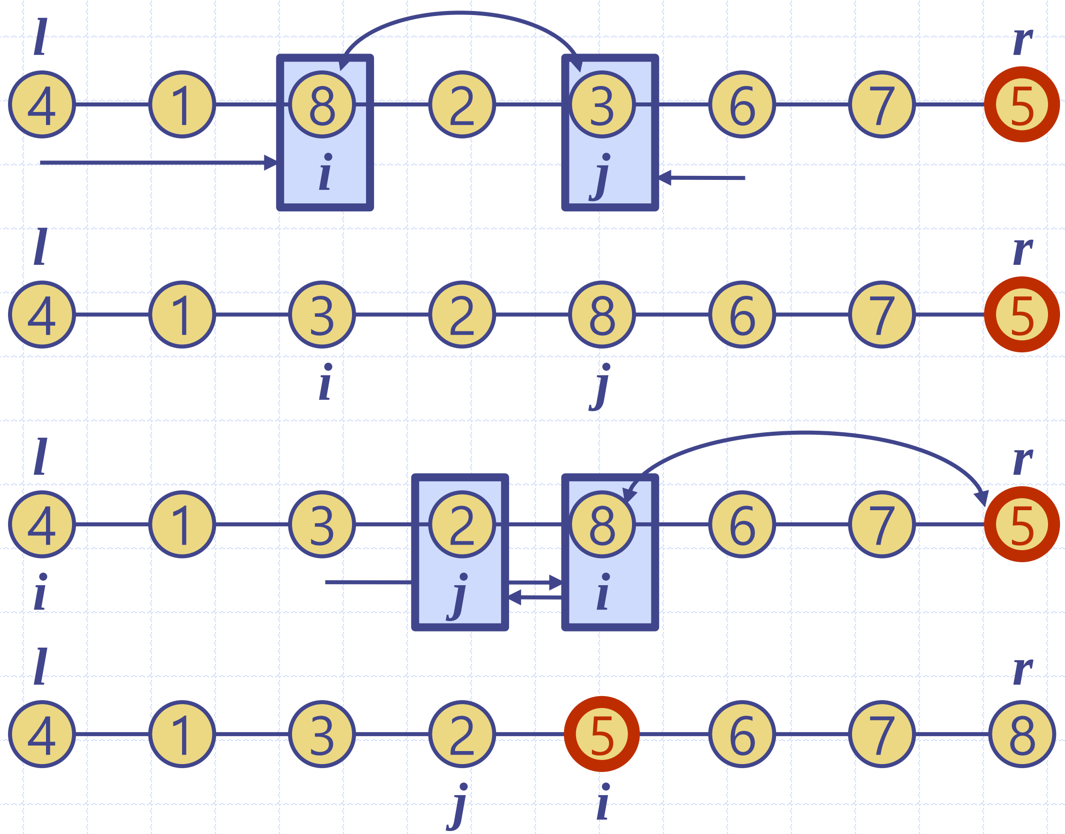
1. $p \leftarrow A[k]$ {pivot}
2. $A[k] \leftrightarrow A[r]$ {hide pivot}
3. $i \leftarrow l$
4. $j \leftarrow r - 1$
5. **while** $(i \le j)$
    while $(i \le j \ \& \ A[i] \le p)$
      $i \leftarrow i + 1$
    while $(j \ge i \ \& \ A[j] \ge p)$
      $j \leftarrow j - 1$
    if $(i < j)$
      $A[i] \leftrightarrow A[j]$
6. $A[i] \leftrightarrow A[r]$ {replace pivot}
7. **return** i {index of pivot}

- 위의 inPlacePartition 버전은 입력 배열 A가 **유일한** 원소로만 이루어졌다고 전제함

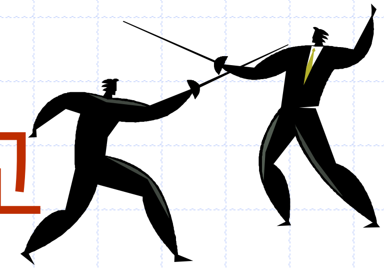
제자리 분할 예



제자리 분할 예 (conti.)



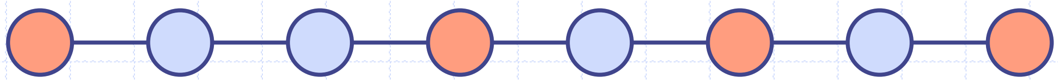
합병 정렬과 퀵 정렬 비교



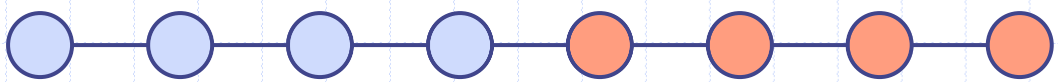
	합병 정렬	퀵 정렬
기법	분할통치법	분할통치법
실행시간	$\mathbf{O}(\boldsymbol{n} \log \boldsymbol{n})$ 최악실행시간	$\mathbf{O}(\boldsymbol{n}^2)$ 최악실행시간 $\mathbf{O}(\boldsymbol{n} \log \boldsymbol{n})$ 기대실행시간
분할 vs. 결합	분할은 쉽고, 합병은 어렵다	분할은 어렵고, 합병은 쉽다
제자리 구현	제자리 합병이 어렵다	제자리 분할이 쉽다
실제 작업 순서	작은 것에서 점점 큰 부문제로 진행	큰 것에서 점점 작은 부문제로 진행

응용문제: 색 분리

- n개의 원소로 이루어진 리스트 L이 있다 – 여기서 원소는 각각 **빨강** 또는 **파랑**으로 색칠되어 있다

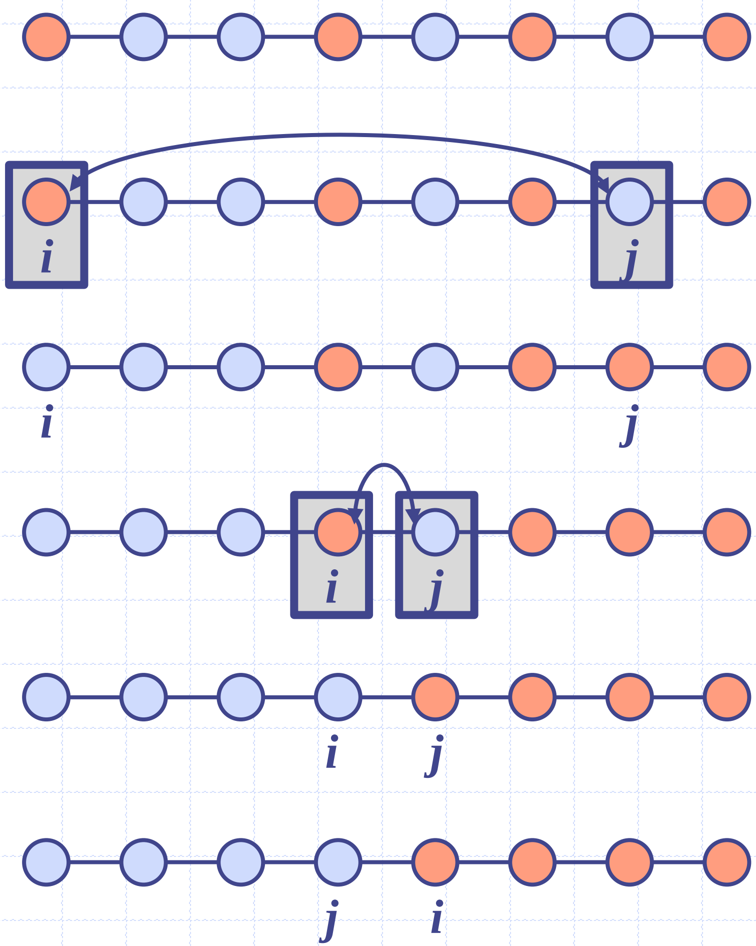


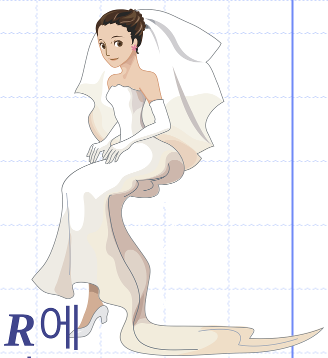
- L이 배열로 표현되었다고 전제하고, L의 모든 파랑 원소들이 빨강 원소들의 앞에 오도록 재배치하는 제자리, $\mathbf{O}(n)$-시간 메쏘드를 설명하라



해결: 색 분리

- **파랑**과 **빨강** 원소에 대해 다음을 수행
  1. 배열의 왼쪽과 오른쪽 양쪽 끝에서 각각 포인터 출발
  2. 왼쪽 포인터가 **파랑** 원소를 가리키는 동안 첨자를 계속 **증가**
  3. 마찬가지로, 오른쪽 포인터가 **빨강** 원소를 가리키는 동안 첨자를 계속 **감소**
  4. 왼쪽 포인터가 빨강 원소에 이르고 오른쪽 포인터가 파랑 원소에 이르면, 두 원소를 **맞교환**
  5. 두 포인터가 만날 때까지 포인터를 계속 진행하면서 원소를 맞교환
  6. 두 포인터가 만난 시점에 배열 원소의 재배치 완료
- **O(*n*)** 시간 소요





응용문제: 신부와 반지

- n명의 신부 집합 B(rides)와 n개의 반지 집합 R(ings)가 있다 – R에 속한 반지 각각은 B에 속하는 신부들 중 단 한 명의 신부에게만 크기가 딱 들어 맞는다
- 공교롭게도, B에 속한 신부들은 모두 닮았고, R에 속한 반지 역시 모두 닮았다 – 즉, 겉만 봐서는 신부 간에 손가락 굵기 차이를 구별하거나 반지 간에 크기 차이를 구별할 수 없다
- 오직 가능한 비교는, **신부-반지** 쌍 (b, r)를 택하여 (여기서 물론 $b \in B$, $r \in R$), 신부 b의 손가락이 반지 r의 크기보다 **큰지**, **작은지**, **딱 맞는지** 끼어 보는 수밖에 없다
- 모든 신부와 반지의 짝을 찾기 위한 효율적인 메쏘드 match를 설계하라
  - match(B, R): 각각 n개의 원소로 구성된 집합 B와 R의 원소들에 대한 매치 쌍들의 집합을 반환
- **신부-반지** 끼어 보는 시도 회수를 기준으로 메쏘드의 실행시간을 구하라

해결

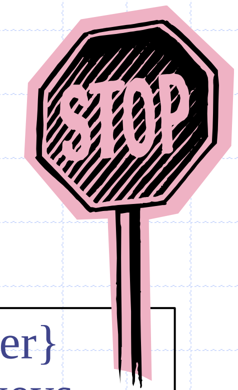
- 아무 신부 하나를 택하여 모든 반지와 비교 – 그러면서 반지 집합을 작은 반지와 큰 반지, 딱 맞는 반지로 나눈다
- 딱 맞는 반지를, 딱 맞는 신부를 제외한 모든 신부와 비교 – 비교를 진행하면서 작은 신부와 큰 신부로 나눈다
- 이제 문제가 **두 개의 부문제**로 나뉘어졌다
  - 한 문제: 딱 맞는 쌍보다 **작은** 신부들과 반지들로 구성
  - 다른 문제: 딱 맞는 쌍보다 **큰** 신부들과 반지들로 구성
- 이와 같은 방식으로 반복한다면 quick-sort와 유사한 알고리즘을 얻게 된다
- 알고리즘의 분석 역시 quick-sort에서 했던 것처럼 기대실행시간을 분석할 수 있으며 그 결과는 $\mathbf{O}(\boldsymbol{n} \log \boldsymbol{n})$이 된다

해결 (conti.)

Alg ***match***(B, R)
　input set B, R
　output matchup set of pairs of elements from B and R

1. **if** (B.*isEmpty*())
　　return *an empty set*
2. **if** (B.*size*() = 1)
　　return *a singleton set with element* (B.*get*(1), R.*get*(1))
3. $b \leftarrow$ *an element in* B
4. $R_{LT}, R_{EQ}, R_{GT} \leftarrow$ ***partition***(R, b)
5. $r \leftarrow R_{EQ}$.*get*(1)
6. $B_{LT}, B_{GT} \leftarrow$ ***partition***$(B - \{b\}, r)$
7. $M_{LT} \leftarrow$ ***match***(B_{LT}, R_{LT})
8. $M_{GT} \leftarrow$ ***match***(B_{GT}, R_{GT})
9. $M \leftarrow$ ***merge***$(M_{LT}, \{(b, r)\}, M_{GT})$
10. **return** M

응용문제: 퀵 정렬 변형



- 오른 쪽의 quickSort(A, n) 함수는 크기 ***n***의 배열 ***A***에 무순으로 저장된 ***n***개의 **유일한** 키들을 오름차순으로 퀵 정렬한다
- partition(A, l, r, k)는 ***A*[*k*]**를 기준원소로 삼아, 배열 ***A***의 원소들을 **기준원소**보다 작은 원소들과 큰 원소들이 각각 배열 ***A***의 왼 편과 오른 편에 위치하도록 이동시키고, 그 사이에 기준원소를 위치시킨 후, 기준원소의 **위치**(즉, 배열 첨자)를 반환한다

Alg ***quickSort*(*A*, *n*)** {driver}
    input array ***A*** of ***n*** distinct keys
    output sorted array ***A***

1. ***rQuickSort*(*A*, 0, *n*** – 1)
2. **return**

Alg ***rQuickSort*(*A*, *l*, *r*)** {recursive}
    input array ***A*** of size ***n***, index ***l*, *r***
    output array ***A*** with elements of index from ***l*** to ***r*** rearranged in increasing order

1. **if** (***l*** < ***r***)
    k ← *a position between* ***l*** *and* ***r***
    m ← ***partition*(*A*, *l*, *r*, *k*)**
    rQuickSort*(*A*, *l*, *m – 1)
    rQuickSort*(*A*, *m + 1, ***r*)**

응용문제: 퀵 정렬 변형 (conti.)

- 퀵 정렬의 수행 원리 상, 무순의 초기 배열 A는 시간이 지남에 따라 거시적인 관점에서 점점 정렬된 상태로 옮겨가게 된다
- 이 점에 착안하여 정렬이 상당히 진행된 적당한 시점에 퀵 정렬을 **중단**하고 (“**거의**” 정렬된 입력에 대해서는 퀵 정렬보다 더 우수한 시간성능을 가지는) **삽입 정렬**로 전환하여 정렬을 완성할 수도 있다
- 그렇게 하기 위한 quickSort 또는 rQuickSort 알고리즘의 **수정 내용**을 작성하라
- **사용 가능**
  - insertionSort(A, n): n개의 유일한 키를 저장한 배열 A를 삽입 정렬

해결

- 퀵 정렬을 더 이상 적용하지 않을 대상 원소 수의 크기(**예:** ***Limit*** = 100)를 정하여 이를 rQuickSort에 반영
- rQuickSort 완료 후 거의 정렬된 상태의 배열을 insertionSort가 처리하도록 quickSort를 수정

Alg ***quickSort*(*A*, *n*)** {driver}
  input array ***A*** of ***n*** distinct keys
  output sorted array ***A***

1. ***rQuickSort***(***A***, 0, ***n*** – 1)
2. ***insertionSort*(*A*, *n*)**
3. **return**

Alg ***rQuickSort*(*A*, *l*, *r*)** {recursive}
  input array ***A*** of size ***n***, integer ***l***, ***r***
  output array ***A*** with elements of index from ***l*** to ***r*** rearranged in increasing order

1. **if** (***r*** – ***l*** ≥ ***Limit***)
    k ← *a position between* ***l*** *and* ***r***
    m ← ***partition***(***A***, ***l***, ***r***, ***k***)
    rQuickSort(***A***, ***l***, ***m*** – 1)
    rQuickSort(***A***, ***m*** + 1, ***r***)