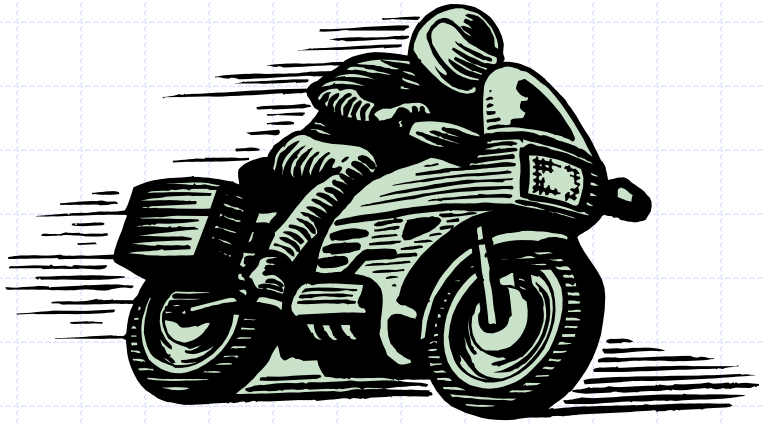
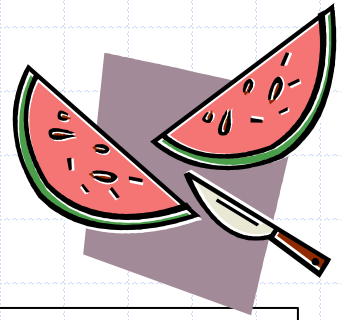
퀵 정렬



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

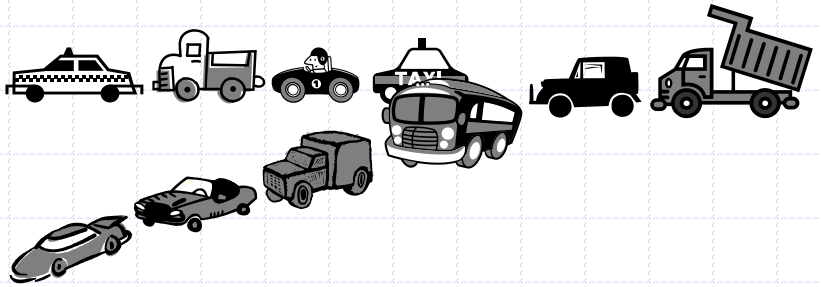


- **퀵 정렬**(quick-sort): **분할통치법**에 기초한 정렬 알고리즘
  1. **분할**(divide): 기준원소 p를(pivot, 보통은 마지막 원소)를 택하여 L을 다음 세 부분으로 분할
     - LT (p보다 작은 원소들)
     - EQ (p와 같은 원소들)
     - GT (p보다 큰 원소들)
  2. **재귀**(recur): LT와 GT를 정렬
  3. **통치**(conquer): LT, EQ, GT를 결합

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. ***L***로부터 각 원소 ***e***를 차례로 삭제
  2. ***e***를 **기준원소** ***p***와의 비교 결과에 따라 부리스트 ***LT***, ***EQ***, ***GT***에 삽입
- 삽입과 삭제를 리스트의 맨 앞이나 맨 뒤에서 수행하므로 **O**(1) 시간 소요
- 따라서, quick-sort의 분할 단계는 **O**(***n***) 시간 소요

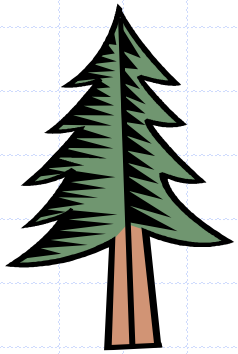
Alg ***partition***(***L***, ***k***)
  input list ***L*** with ***n*** elements,
    position ***k*** of pivot
  output sublists ***LT***, ***EQ***, ***GT*** of the
    elements of ***L***, less than, equal to,
    or greater than pivot, resp.

1. ***p*** ← ***L.get***(***k***) {pivot}
2. ***LT***, ***EQ***, ***GT*** ← *empty list*
3. **while** (!***L.isEmpty***())
    e ← ***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***

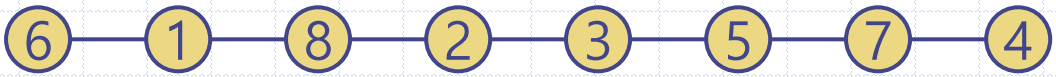
기준원소 선택

- 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

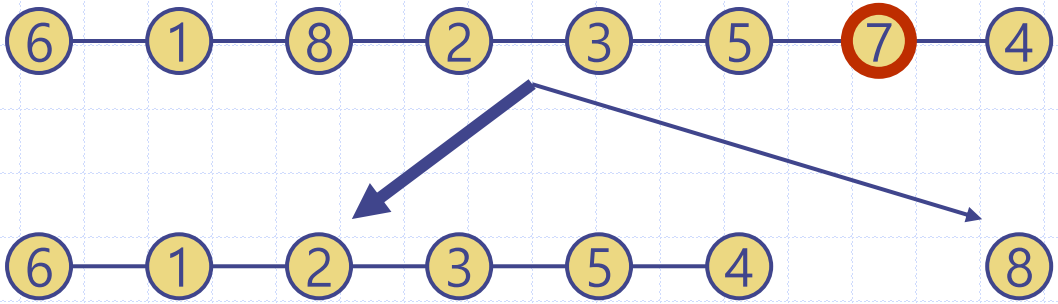


- quick-sort의 실행을 이진트리로 보이기
  - 이진트리의 각 노드는 quick-sort의 **재귀호출**을 표현하며 다음을 저장
    - 실행 이전의 무순 리스트 및 기준원소
    - 실행 이후의 정렬 리스트
  - 루트는 **초기 호출**을 의미
  - 잎들은 크기 0 또는 1의 부리스트에 대한 호출을 의미
- 실행예를 위한 입력 리스트



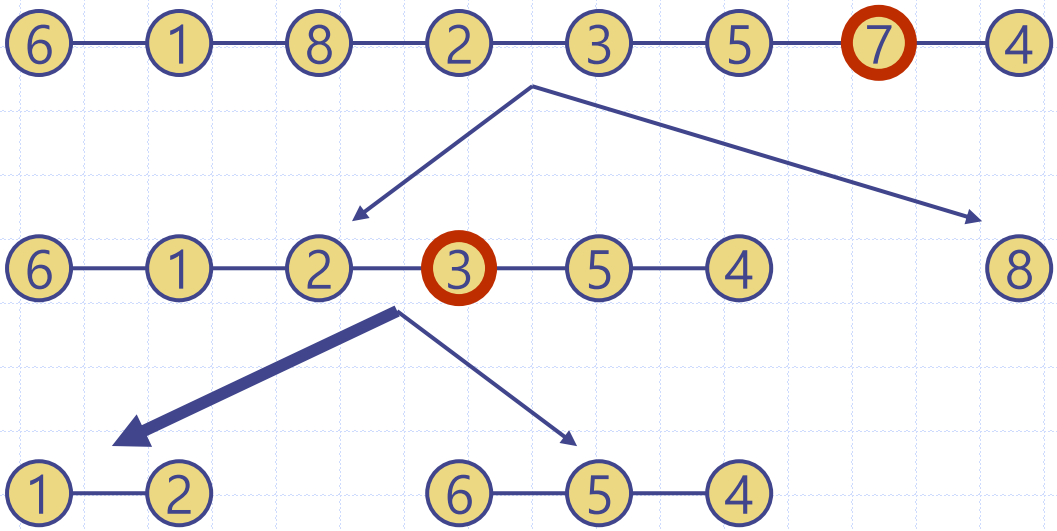
퀵 정렬 수행 예

- 초기 호출, 분할, 재귀 호출



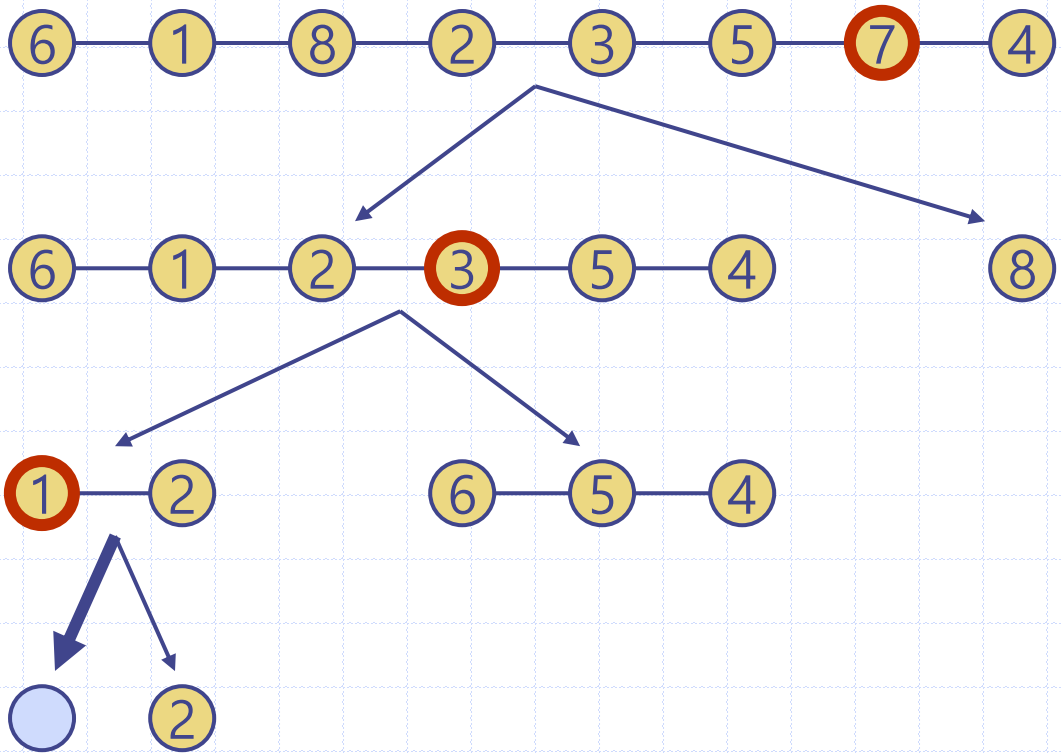
퀵 정렬 수행 예 (conti.)

- 분할, 재귀호출



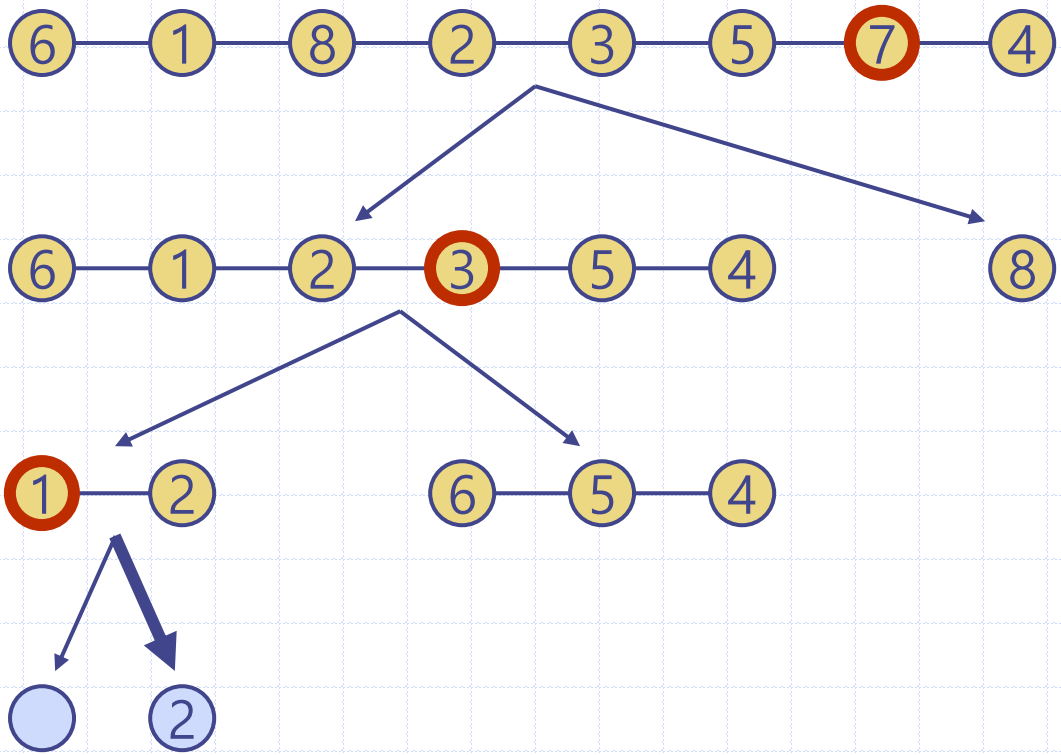
퀵 정렬 수행 예 (conti.)

- 분할, 재귀 호출, 베이스 케이스



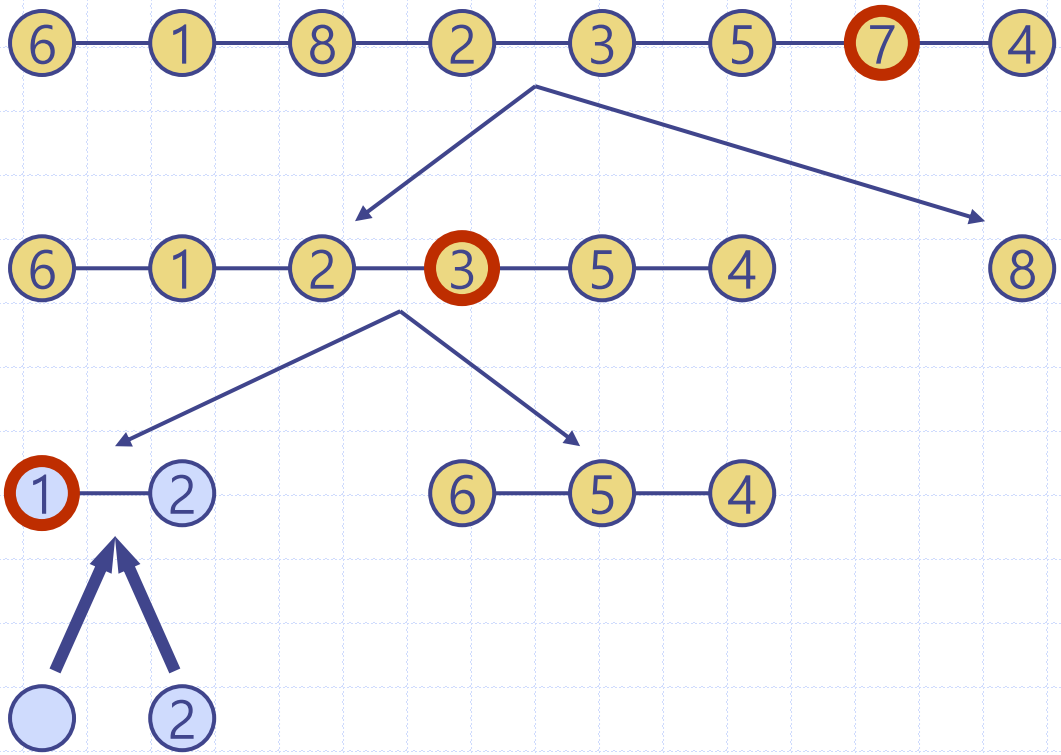
퀵 정렬 수행 예 (conti.)

- 재귀 호출, 베이스 케이스



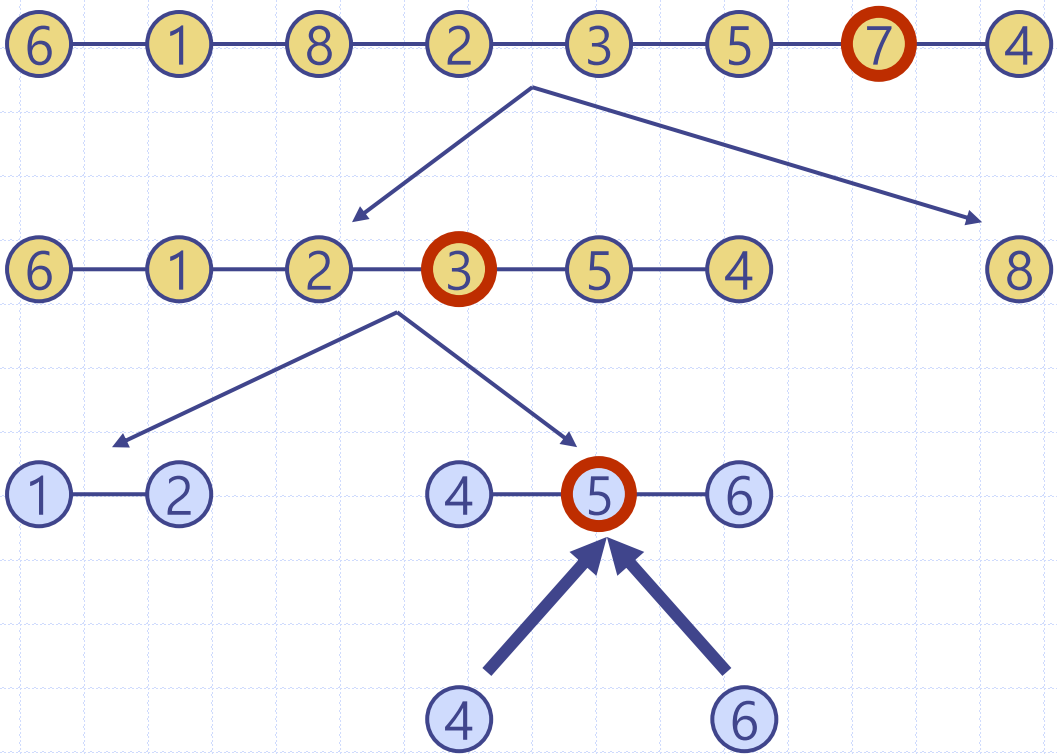
퀵 정렬 수행 예 (conti.)

- 결합



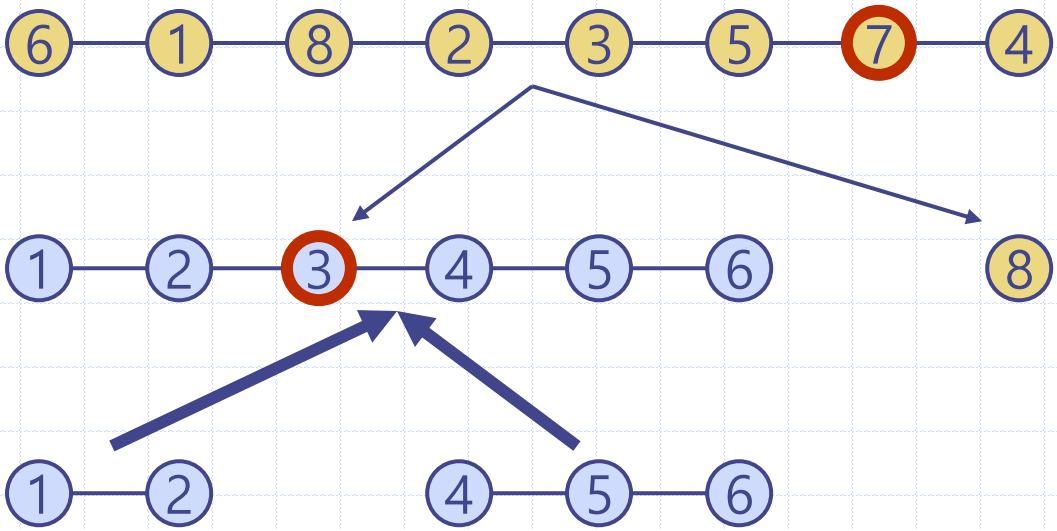
퀵 정렬 수행 예 (conti.)

- 재귀 호출, 분할, … , 결합



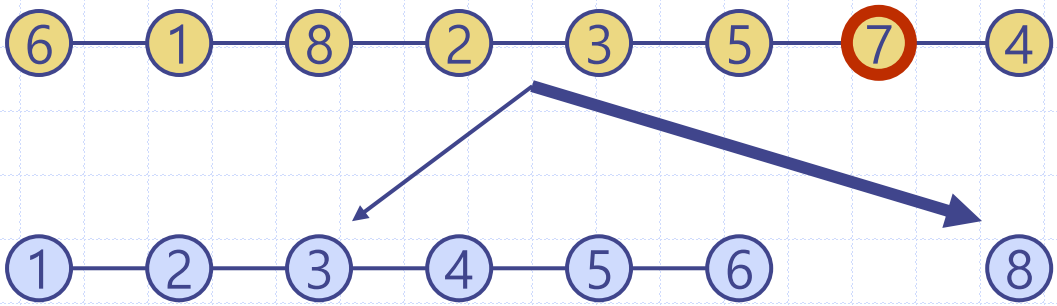
퀵 정렬 수행 예 (conti.)

- 결합



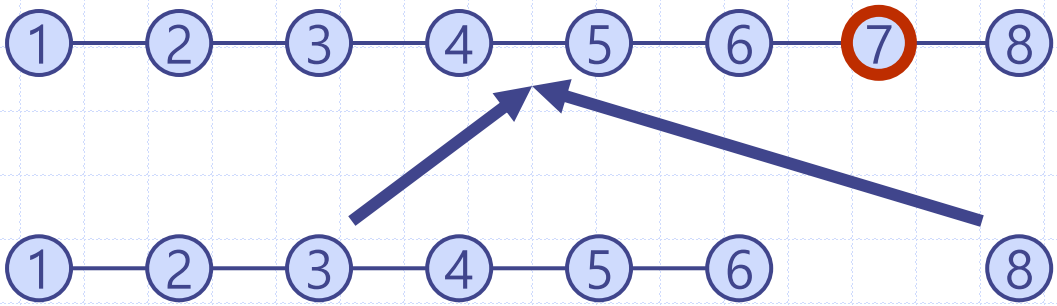
퀵 정렬 수행 예 (conti.)

- 재귀 호출, 베이스 케이스

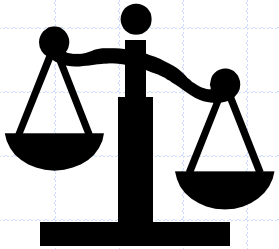


퀵 정렬 수행 예 (conti.)

- 결합



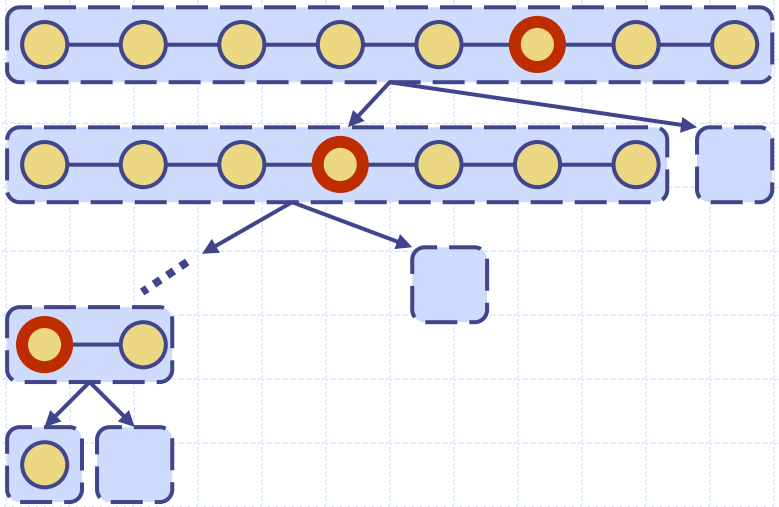
최악실행시간



- 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

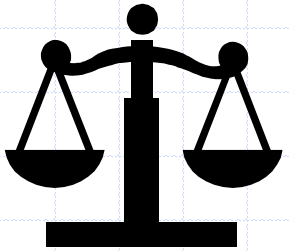
$$n + (n-1) + \ldots + 2 + 1$$

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

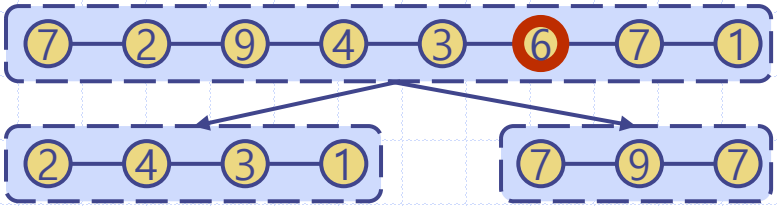


depth	time
0	n
1	$n-1$
…	…
$n-1$	1

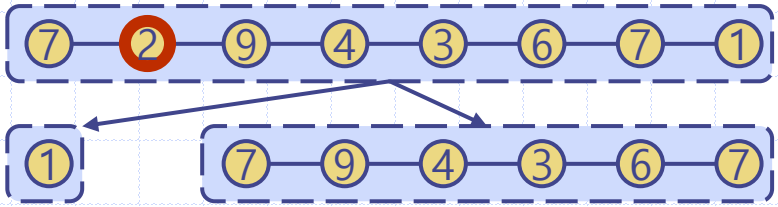
기대실행시간



- 크기 s의 리스트에 대한 quick-sort의 재귀 호출을 고려하면,
  - **좋은 호출:** LT와 GT의 크기가 모두 $(3/4)s$ 보다 작다
  - **나쁜 호출:** LT와 GT의 가운데 하나의 크기가 $(3/4)s$ 보다 크다

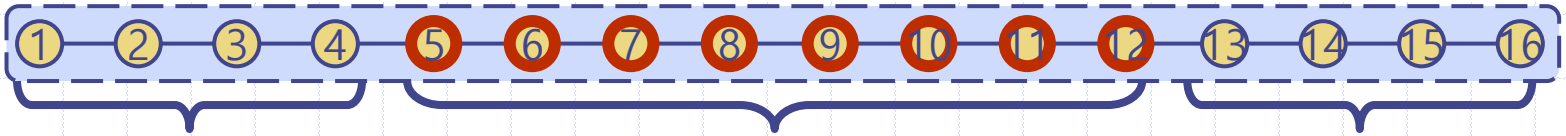


좋은 호출



나쁜 호출

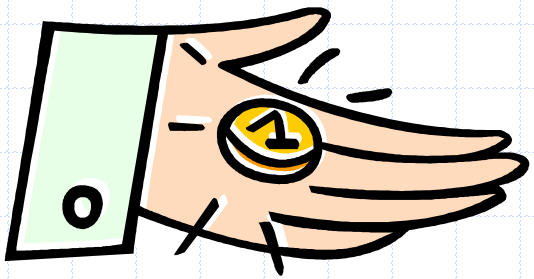
- 호출이 **좋을** 확률은 ½ (**예:** 동전던지기)
  - 가능한 기준원소의 ½은 좋은 호출을 부른다



나쁜 기준원소 **좋은 기준원소** **나쁜 기준원소**

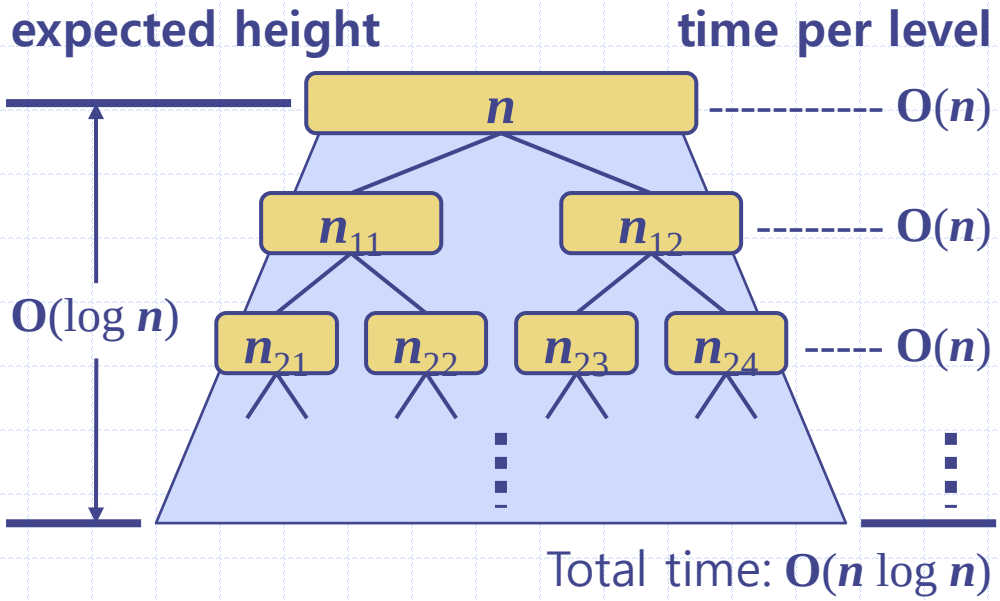
무작위 퀵 정렬

- 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

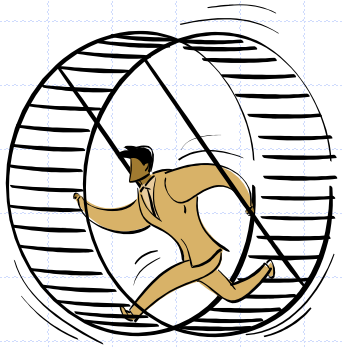


무작위 퀵 정렬의 기대실행시간

- 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*** ≥ ***r***)
   return
2. ***k*** ← *a position between* ***l*** *and* ***r***
3. ***a***, ***b*** ← ***inPlacePartition***(***L***, ***l***, ***r***, ***k***)
4. ***inPlaceQuickSort***(***L***, ***l***, ***a*** – 1)
5. ***inPlaceQuickSort***(***L***, ***b*** + 1, ***r***)

- 재귀호출은 ***LT***와 ***GT*** 부리스트 대해 수행

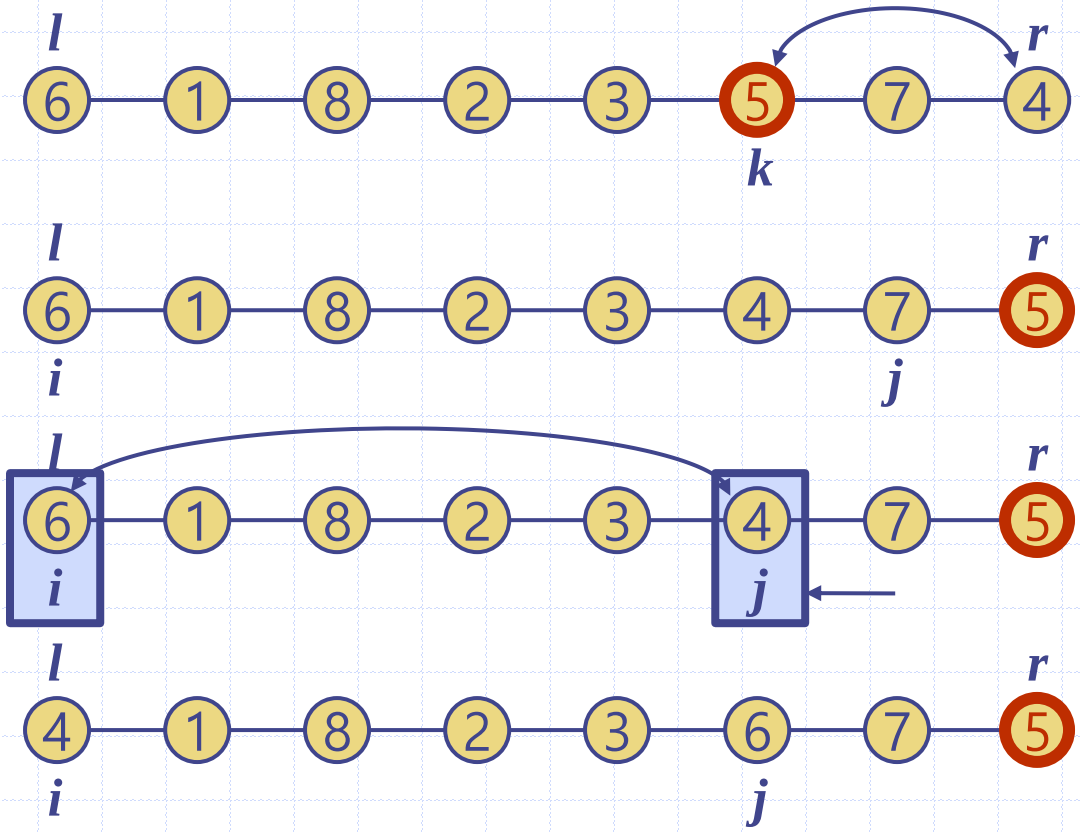
제자리 분할

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***

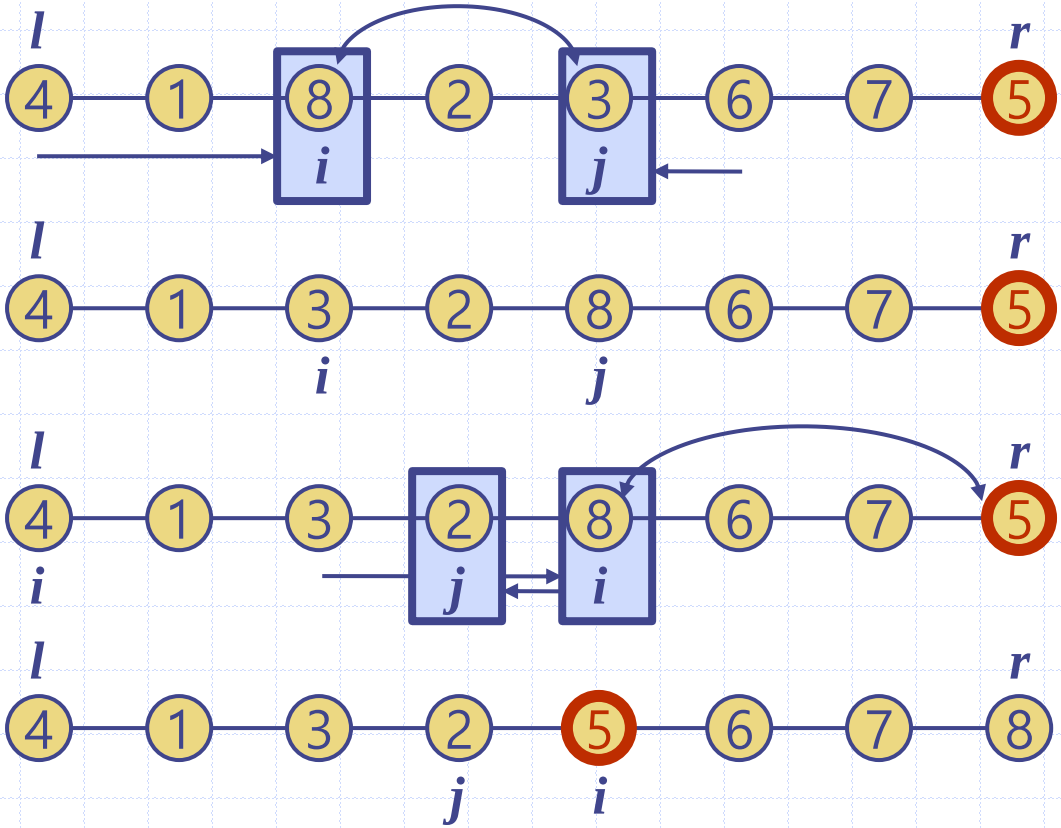
1. $p \leftarrow A[k]$ {pivot}
2. $A[k] \leftrightarrow A[r]$ {hide pivot}
3. $i \leftarrow l$
4. $j \leftarrow r - 1$
5. **while** ($i \leq j$)
   while ($i \leq j$ & $A[i] \leq p$)
    $i \leftarrow i + 1$
   while ($j \geq i$ & $A[j] \geq p$)
    $j \leftarrow j - 1$
   if ($i < j$)
    $A[i] \leftrightarrow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

- 위의 inPlacePartition 버전은 입력 배열 ***A***가 **유일한** 원소로만 이루어졌다고 전제함

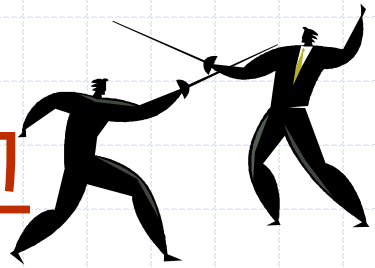
제자리 분할 예



제자리 분할 예 (conti.)



합병 정렬과 퀵 정렬 비교



	합병 정렬	퀵 정렬
기법	분할통치법	분할통치법
실행시간	$O(n \log n)$ 최악실행시간	$O(n^2)$ 최악실행시간 $O(n \log n)$ 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