선택 알고리즘

한국기술교육대학교 컴퓨터공학부 김상진

```
while(!morning){
    alcohol++
    dance++
} #party
```

```
while(!sleep){
    think++
    solve++
} #cse-mode
```

교육목표

- 선택 문제(selection problem)
  - 특정 순서에 있는 값을 찾는 문제
    - 정렬되어 있지 않은 리스트에서 k번째 요소 찾기
      - 예) [3, 4, 2, 8, 7, 1, 5, 6]에서 3번째 요소는?
  - 선택 문제는 정렬보다 쉬운 문제
  - 선택 문제의 하한과 반대자 논법(adversary argument)
  - 제한적 선택 문제 (찾는 위치 고정)
    - 최댓값과 최솟값 동시 찾기 문제
    - 두 번째로 큰 수 찾기 문제: 토너먼트 알고리즘
    - 중간값 찾기 문제
- 빠른 정렬에서 사용한 pivot을 기준으로 분할하는 방법을 활용하여 선형 시간 확률적 선택 알고리즘을 만들 수 있음
- 결정적 선택 알고리즘은?
  - 선형 시간 결정적 선택 알고리즘도 만들 수 있음

선택 문제

- 입력. n개의 서로 다른 수로 구성된 배열 A
- 출력. k번째 작은 수
- 정렬되어 있지 않은 리스트에서 k번째 요소 찾기
- 예) n번째 요소(가장 큰 값 찾기)
  - $n-1$번 비교 필요
    - 이보다 적게 비교하여 찾을 수 없음

```
max := A[1]
for i := 2 to n do
    if max < A[i] then
        max := A[i]
```

- 예) 첫 번째 요소(가장 작은 값 찾기)
  - $n-1$번 비교 필요
    - 이보다 적게 비교하여 찾을 수 없음
- 정렬한 후에 찾기: $O(n \log n)$
- 실제 선택 문제는 정렬 문제보다 쉬운 문제이며, $O(n)$으로 해결할 수 있음
- 선택 문제를 해결하는 알고리즘을 살펴보기 전에 선택 문제가 필요한 비교의 하한(최악의 경우 필요한 비교 수)을 알아보고자 함
  - 하한을 만족하는 알고리즘을 만들 수 있는지 살펴봄

선택 문제 하한과 반대자 논법 (1/2)

- 반대자 논법(adversary argument)
  - 친구와 추측 게임: 내가 선택한 1 ~ 100 사이의 수를 맞추는 게임
    - 친구가 질문을 많이 하도록 유도하기 위해 수를 선택하지 않고 진행함
    - 목표는 친구가 질문을 가장 많이 하도록 유도함
      - 예) 10보다 크니?
        - 질문을 많이 하도록 유도하기 위해서는 답은 예
        - 50보다 큰지 여부를 묻는 것이 가장 이득임
    - 답변에 일관성을 위해 부득이하게 결정해야 할 때 정함
  - 이 방법을 이용하면 선택 문제의 비교 하한을 구할 수 있음

선택 문제 하한과 반대자 논법 (2/2)

- 선택 문제의 비교 하한: 최악의 경우 필요한 비교 수
  - 어떤 입력이 최악의 경우에 해당하는지 알아야 비교 하한을 찾을 수 있음
  - 반대자 논법을 이용하면 최악의 경우에 해당하는 입력을 만들 수 있음
    - 이 논법은 어떤 결정을 해야 할 때 가장 일을 많이 하도록 유도함
  - 이 분석은 특정 알고리즘을 분석하는 것이 아니라 문제 해결하기 위해 필요한 비교 하한을 찾는 것임
- 비교 하한을 찾으면 그 다음 비교 하한으로 문제를 해결할 수 있는 알고리즘을 찾는 것임

최댓값과 최솟값을 동시에 (1/5)

- 최댓값과 최솟값을 각각 찾는 비용의 비교 하한은 $n-1$임
  - 이를 응용하여 최댓값을 찾은 후 이것을 제외하고 나머지에서 최솟값을 찾으면 총 $2n-3$번의 비교가 필요함
  - 하지만 이보다 더 효과적으로 할 수 있다는 것을 직관적으로 알 수 있음
  - 두 값을 비교할 때 그 결과를 최댓값과 최솟값을 찾을 때 동시에 활용할 수 있음
- n개의 서로 다른 수로 구성된 리스트에서 x가 최댓값, y가 최솟값임을 알기 위해 필요한 정보는?
  - x를 제외한 모든 요소는 어떤 비교에서 졌어야 하며, y를 제외한 모든 요소는 어떤 비교에서 이겼어야 함
    - 총 $2n-2$개의 정보가 필요함 $\Leftarrow$ 이 정보를 몇 번의 비교로 얻을 수 있을까?
- 참고. 정보와 비용은 다른 것임. 두 요소를 비교하면 하나는 winner, 하나는 loser가 되므로 2개를 정보를 얻을 수 있음. 항상 비교를 통해 우리가 필요한 2개의 정보를 얻을 수 있는 것은 아님

최댓값과 최솟값을 동시에 (2/5)

- 반대자 전략

요소 상태	의미
W	한 번도 진 적이 없고, 최소 한 번 이상 이겼음
L	한 번도 이긴 적이 없고, 최소 한 번 이상 졌음
WL	최소 한 번 이상 이긴 적도 있고, 진 적도 있음
N	비교에 한 번도 참여한 적이 없음

비교하는 두 요소 a, b의 상태	adversary response	새 상태	새 정보
N, N	a>b	W, L	2
W, N, or WL, N	a>b	W, L or WL, L	1
L, N	a<b	L, W	1
W, W	a>b	W, WL	1
L, L	a>b	WL, L	1
W, L or WL, L or W, WL	a>b	no change	0
WL, WL	일관성 유지	no change	0

- 리스트 요소의 값은 진행(비교할 때마다)하면서 확정함
- 진행 과정에서 일관성이 유지되면 바꿀 수 있음

최댓값과 최솟값을 동시에 (3/5)

- 예) $n = 6$: $2n - 2 = 10$개 정보, 이 예에서 사용한 비교 횟수는 8

비교	a		b		c		d		e		f		정보
	상태	값	상태	값	상태	값	상태	값	상태	값	상태	값	
a, b	W	10	L	5	N		N		N		N		2
a, e	W	10							L	3			3
c, d					W	12	L	6					5
c, f					W	12					L	9	6
c, a	W	15			WL	12							7
b, d			L	5			WL	6					8
e, f									WL	3	L	1	9
b, f			WL	5							L	1	10

최댓값과 최솟값을 동시에 (3/5)

- 예) $n = 6$: $2n - 2 = 10$개 정보, 이 예에서 사용한 비교 횟수는 7

비교	a		b		c		d		e		f		정보
	상태	값	상태	값	상태	값	상태	값	상태	값	상태	값	
a, b	W	10	L	5									2
c, d					W	8	L	4					4
e, f									W	7	L	3	6
a, c	W				WL								7
b, d			WL				L						8
a, e	W								WL				9
d, f							WL				L		10

- 가장 최선의 전략
- 반대자 전략을 알고 있으면 그것을 활용하는 알고리즘을 만들 수 있음

최댓값과 최솟값을 동시에 (4/5)

- 정리. n개의 서로 다른 수로 구성된 리스트에서 최댓값과 최솟값을 비교만을 이용하여 찾기 위해 필요한 최소 비교 수는 $3n/2 - 2$임
- 증명)
  - 가정. n은 짝수
    - 2개 정보를 얻을 수 있는 유일한 경우는 N과 N의 비교
      - 총 $n/2$개 존재 $\Longrightarrow$ n개 정보 획득
    - 그다음 비교에서는 최대 1개 정보만 얻을 수 있음
    - 총 필요한 정보 $2n - 2$
      - $n - 2$번 비교가 추가로 필요
    - 총 비교는 $\frac{n}{2} + n - 2 = \frac{3n}{2} - 2$
  - n이 홀수이면? $\frac{n-1}{2} + n - 1 = \frac{3n}{2} - \frac{3}{2}$

최댓값과 최솟값을 동시에

- 알고리즘

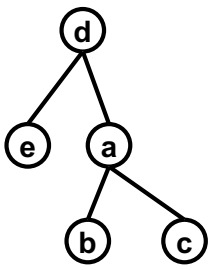
```
max, min := 0, 0
if A[1] < A[2] then
    max, min := A[2], A[1]
else
    max, min := A[1], A[2]
i := 3
while i ≤ n − 1 do
    if A[i] < A[i + 1] then
        if A[i] < min then min := A[i]
        if A[i + 1] > max then max := A[i + 1]
    else
        if A[i + 1] < min then min := A[i + 1]
        if A[i] > max then max := A[i]
    i += 2
```

- 요소 간 비교 횟수
- $1 + \left(\frac{n-2}{2}\right) \times 3 = \frac{3}{2}n - 2$

- 홀수이면?
  $max, min := A[1], A[1]$
  $i := 2$
  //
- $\left(\frac{n-1}{2}\right) \times 3 = \frac{3}{2}n - \frac{3}{2}$

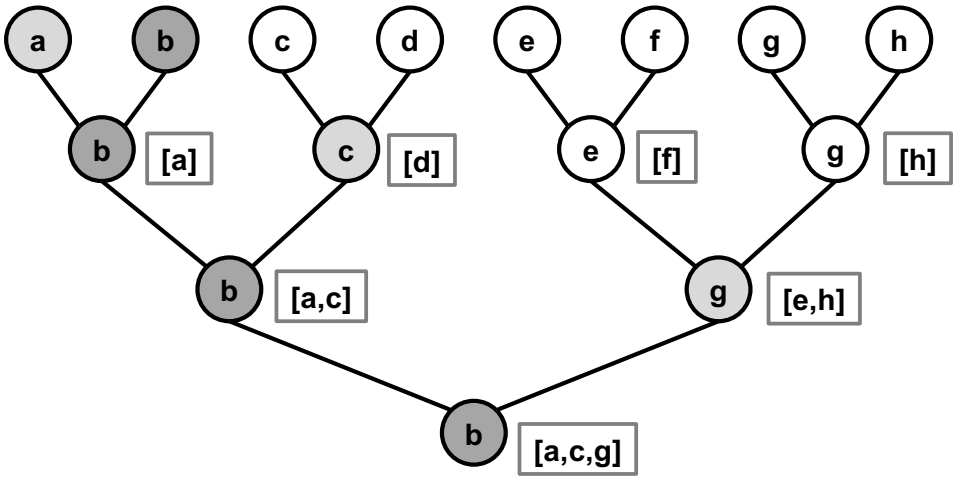
두 번째로 큰 수 찾기 (1/4)

- n개의 서로 다른 수로 구성된 리스트에서 두 번째로 큰 수 찾기
  - 최댓값을 찾은 후$(n-1)$, 남은 것에서 다시 최댓값을 찾으면$(n-2)$
    총 $2n-3$번의 비교로 두 번째로 큰 수를 찾을 수 있음
  - 최댓값과 최솟값을 동시에 찾는 방법을 활용하면 더 효과적으로 찾을 수 있음
  - 두 번째로 큰 수의 특징
    - 최댓값 외에 또 다른 값에 진 값은 두 번째로 큰 수가 될 수 없음
  - 최댓값을 찾는 과정에서 위와 같은 수를 제거함
  - 예) a, b, c, d, e
    - a, b: W, L
    - a, c: W, L
    - a, d: WL, W
    - d, e: W, L
      - max: d, second max는 a 아니면 e



두 번째로 큰 수 찾기 (2/4)

- 이 방법의 문제?
  - 비교 과정에서 b와 c를 배제해야 한다는 것을 어떻게 구현?
  - 더 심각한 문제: 이전 예에서 a가 max이면 배제할 것이 없음
- 토너먼트 방법:



두 번째로 큰 수 찾기 (3/4)

- 알고리즘
  - 토너먼트를 통해 max 찾기: $n-1$번 비교
    - n이 2의 거듭제곱이라고 가정하면 총 $\log n$ 라운드 진행
      - 1회전 $\frac{n}{2}$, 2회전 $\frac{n}{2^2}$, …, 최종 $\frac{n}{2^{\log n}} = 1$
        - $n\sum_{i=1}^{\log n}\left(\frac{1}{2}\right)^i = n-1$
  - max에 진 값들만 이용하여 second max 찾기: $\lceil \log n \rceil - 1$
  - 총 비용: $n + \lceil \log n \rceil - 2$
- 정리. 이 비용이 두 번째로 큰 수를 찾는 하한임
  - 증명) 반대자 논법을 이용하여 증명 가능

두 번째로 큰 수 찾기 (4/4)

- 구현 방법
  - 방법 1. 토너먼트
  - 방법 2. max에 진 요소들: 각 값마다 패자 리스트 유지
- 토너먼트 예) [3, 7, 5, 13]
  - 준비 $2n-1$크기의 배열, $n-1$위치부터 원래 값을 순서 대로 저장

			3	7	5	13

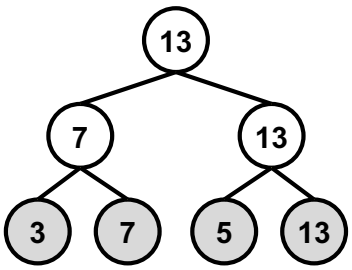
  - 뒤에서부터 2개씩 토너먼트를 진행함

13	7	13	3	7	5	13

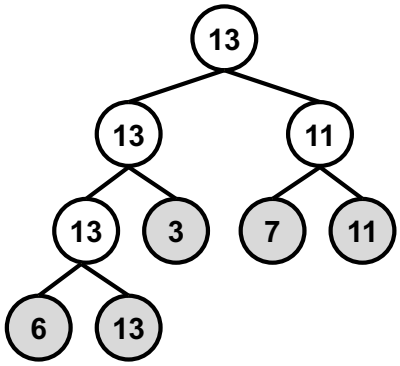
  - 첫 번째 색인에 위치한 것이 최댓값
  - 다시 아래로 내려가면서 최댓값에 진 것들 중에 최댓값을 찾음
- n이 홀수이어도 동일하게 진행함

13	13	11	13	3	7	11	6	13

이진 힙 모습



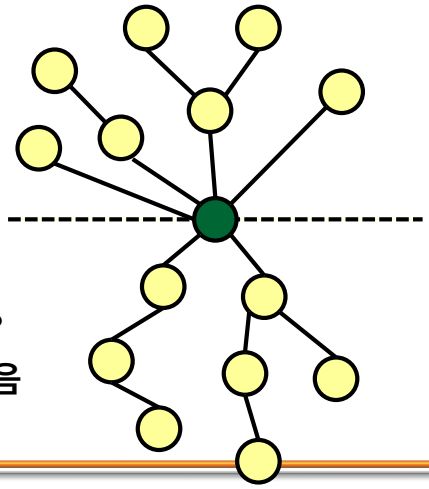
13	7	13	3	7	5	13



13	13	11	13	3	7	11	6	13

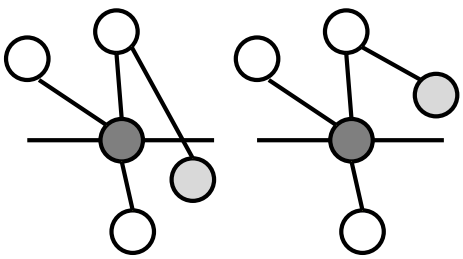
중간값에 대한 하한 (1/7)

- 입력. n개(홀수)의 서로 다른 수로 구성된 배열 A
- 출력. $(n+1)/2$번째 수
- 반대자 논법을 사용하여 최악의 경우 필요한 비교 수를 알아보자
  - 알고리즘은 비교 기반임
- 어떤 값이 중간값임을 확신하기 위해서는 이 값과 나머지 모든 값과의 관계를 알아야 함 (최소 $n-1$개 비교 필요)
  - 임의 x가 주어졌을 때 $x > median$ 또는 $x < median$인지 알아야 함
  - 이것을 그래프로 표현하면 옆 트리 형태처럼 표현됨 (간선: 두 노드의 비교 결과)
    - 위에 있는 노드일 수록 큰 값임
- 필요한 $n-1$개 비교 외에 얼마나 많은 불필요한 비교하도록 반대자가 유도할 수 있을까?
  - 이것을 알 수 있으면 비교 하한을 계산할 수 있음



중간값에 대한 하한 (2/7)

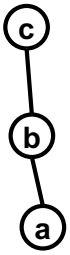
- 특정 값이 중간값임을 알게 해주는 비교를 핵심(crucial) 비교라 하고, 그 외에 모든 비교를 비핵심(non-crucial) 비교라 하자.
- x와 y의 비교
  - 핵심 비교: x와 중간값과의 관계를 형성하는 비교
    - $y \geq median$에 대해 $x > y$
    - $y \leq median$에 대해 $x < y$
    - 이 전 슬라이드에 제시된 트리에 있는 모든 간선은 핵심 비교임
  - 비핵심 비교:
    - $x > median$이고 $y < median$
- 반대자의 전략은 비교가 비핵심 비교가 되도록 답을 해야 함
  - 비핵심 비교가 되면 값을 수정하여 알고리즘의 답을 틀리게 할 수 있음



중간값에 대한 하한 (3/7)

- 알고리즘 입장
  - 예) a와 b의 비교
    - 어느 값이 크고 작은지는 알 수 있음
    - 중간값에 대해서는 어떤 정보도 알 수 없음
    - 이 비교 후 다음 비교는 어떤 것을?
  - 예) a와 b, b와 c를 비교한 후 d는 어떤 것과 비교하는 것이 유리하나?
    - a와 비교하여 a보다 작으면, c와 비교하여 c보다 크면 유익한 정보이지만 꼭 이 결과를 줄 것인지 알 수 없음
  - 알고리즘은 두 요소를 비교하였을 때 이 비교가 핵심 비교일지 비핵심 비교인지는 나중에 알게 됨

a	b	c	d	e	f	g



중간값에 대한 하한 (4/7)

- 반대자 전략
  - 중간값으로 사용할 값 결정 (위치는 결정하지 않음)
  - 어떤 위치에 있는 요소를 처음으로 비교하면 그 위치에 값을 할당하고 답변함
    - 모두 할당되어 있지 않으면 하나는 중간값보다 큰 값, 하나는 중간값보다 작은 값을 할당함
    - 하나는 할당되어 있고, 다른 하나는 할당되어 있지 않으면 비핵심 비교가 되도록 할당함
      - 예) $a = 15, median = 9$일 때 a와 b를 비교하면 b에 9보다 작은 값을 할당함. 반대자는 지금까지 비교 결과의 일관성을 유지한 상태에서 b에 할당된 값을 바꾸어 알고리즘의 선택을 틀리게 만들 수 있음
    - $\frac{n-1}{2}$보다 많은 수를 중간값보다 크게 또는 작게 할당할 수 없음
    - 값이 할당되지 않은 요소가 하나 남으면 그 위치에 중간값을 할당함
  - 항상 답변은 기존 답변과 일관성을 계속 유지해야 함

중간값에 대한 하한 (5/7)

- 반대자 전략 수행을 위해 각 요소의 상태를 다음과 같이 유지함

요소 상태	의미
L	중간값보다 큰 값이 할당된 요소 (Large)
S	중간값보다 작은 값이 할당된 요소 (Small)
N	아직 한 번도 비교에 참여하지 않은 요소

- 반대자의 전략 요약

비교	전략
N, N	S, L 또는 L, S
L, N or N, L	N에 중간값보다 작은 값 할당
S, N or N, S	N에 중간값보다 큰 값 할당

  - 이 전략을 수행하는 중에 $\frac{n-1}{2}$개의 요소가 L 또는 S가 되면 이 전략을 사용하지 않고 일관성이 유지되도록 새 요소에 값을 할당함
  - 제시한 반대자 전략은 모두 비핵심 비교임

중간값에 대한 하한 (6/7)

- 예) $n = 5$: 중간값 12

	a		b		c		d		e	
	상태	값	상태	값	상태	값	상태	값	상태	값
a, b	L	15	S	3						
c, d					L	13	S	5		12

	a		b		c		d		e	
	상태	값	상태	값	상태	값	상태	값	상태	값
a, b	L	15	S	3						
b, c					L	13				
c. d							S	5		12

  - 사용한 비교는 모두 비핵심 비교임
  - 이 이후 비교는 핵심 또는 비핵심일 수 있음

중간값에 대한 하한 (7/7)

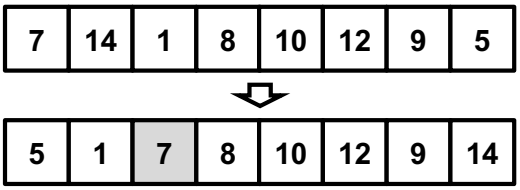
- 반대자가 얼마나 많은 비핵심 비교를 하도록 유도할 수 있나?
  - 각 비핵심 비교는 최대 1개의 요소를 L로 만듦
  - 반대자는 $\frac{n-1}{2}$개의 요소가 L 또는 S가 될 때까지 자유롭게 값 할당이 가능함
- 알고리즘은 처음에 $\frac{n-1}{2}$개의 N들 간의 비핵심 비교를 할 수 있음
  - 따라서 최소 $\frac{n-1}{2} + n - 1$개 비교가 필요함
    - 중간값을 알기 위한 최소 비교 $n - 1$개
    - $\frac{n-1}{2}$개의 비핵심 비교를 하도록 유도 가능
  - 이 비교를 통해 중간값을 찾을 수 있다는 것은 아님. 이보다 적게 비교하면 찾을 수 없다는 뜻임
- 실제 알고리즘은? 분할 정복은 어때?
  - 어떤 값을 기준으로 그것보다 작은 것과 큰 것으로 분할할 수 있다면

선택 문제 알고리즘

- 최댓값, 최솟값과 같은 특정 i번째 수는 그 문제 전용 알고리즘으로 해결하면 더 효과적임
- 선택하고자 하는 위치와 무관한 효율적인 알고리즘은?
  - 선형이 가능하다고 언급하였음
    - 정렬하는 것은 배제함
    - 여러 위치($\log n$ 위치 이상)를 동시에 찾아야 한다면 정렬하는 것이 더 효과가 있을 수 있음
  - 모든 요소를 정렬하지 않지만 정렬하는 과정을 이용하면 효과적으로 찾을 수 있지 않을까?
    - 특히, 빠른 정렬의 분할 알고리즘을 활용하면 …
      - 확률적 알고리즘
  - 정렬하는 것 또는 정렬하는 과정을 이용하는 것 모두 주어진 입력에 대한 수정이 불가피함

확률적 선택 알고리즘

- 빠른 정렬의 분할 방법을 이용하여 선형 알고리즘을 만들어보자.
  - 예) 5번째 요소를 찾고 싶음
    - 랜덤 피봇을 선택하여 1차 분할함
    - 피봇이 3번째 위치함
    - 5번째 요소는 어디에?
    - 와우. 빠른 정렬을 하는데 양쪽을 재귀해야 하는 것이 아니라 한쪽만… 그러면 빠른 정렬보다 성능이 더 좋을 수밖에



- pseudocode

```
RSelect(A, left, right, i)
if left = right then return A[left]
pLoc := partition(A, left, right)
if pLoc = i then return A[pLoc]
else if pLoc > i then
     return RSelect(A, left, pLoc − 1, i)
else return RSelect(A, pLoc + 1, right, i)
```


확률 선택 알고리즘의 시간복잡도

- RSelect 알고리즘에서 피봇이 가장 최악으로 선택되는 경우 시간 복잡도는? $O(n^2)$
- 거꾸로 피봇이 가장 최선으로 선택되는 경우 시간복잡도는?
  - 중간값을 피봇으로 선택한 경우
    - $T(n) \leq T\left(\frac{n}{2}\right) + O(n)$
    - 도사정리 경우 2에 해당함. 따라서 $T(n) = O(n)$임
  - 빠른 정렬 분석을 생각해보면 RSelect가 평균적으로 $O(n)$임을 알 수 있음

수학적 분석

- 참고. RSelect은 재귀 호출 밖에서 $\leq cn$ 연산 수행, $(c > 0)$: $O(n)$
  - 대부분 비용은 partition 비용
- RSelect의 특징
  - 매번 하나의 재귀 호출만 있음
  - 각 재귀호출마다 처리하는 입력의 크기는 작아짐
  - 이 크기를 이용하여 이 함수의 진행 정도를 측정하여 분석함 (트리의 높이)
- 표기법
  - phase j: 현재 다루는 배열의 크기가 $\left(\frac{3}{4}\right)^{j+1} \cdot n$에서 $\left(\frac{3}{4}\right)^{j} \cdot n$ 사이
    - 예) $j = 0$: 처리하고 있는 크기가 n의 75%이상일 경우
      - 가장 바깥 RSelect 호출은 phase 0에 있음
      - 피봇의 선택마다 phase가 증가하지 않을 수 있음
      - 특정 phase를 skip할 수 있음
  - X_j: phase j에서 일어난 재귀 호출의 수

수학적 분석

- 알고리즘의 시간복잡도
  - $\leq \sum_{\text{phase } j} X_j \cdot c \cdot \left(\frac{3}{4}\right)^{j} n = cn \sum_{\text{phase } j} \left(\frac{3}{4}\right)^{j} X_j$
- 25 ~ 75 split을 주는 피봇을 선택하면 항상 다음 phase로 이동
  - 이와 같은 피봇을 선택할 확률은 50%
- $E[X_j] \leq$ 동전 던지기에서 앞면이 나올 때까지 던진 수의 기댓값
- N: 동전 던지기에서 앞면이 나올 때까지 던진 수를 나타내는 확률변수
  - $E[N] = 1 + \frac{1}{2} E[N] \rightarrow E[N] = 2$
  - $E[X_j] \leq 2$
- $E\left[cn \sum_{\text{phase } j} X_j \cdot \left(\frac{3}{4}\right)^{j}\right] = cn \sum_{\text{phase } j} \left(\frac{3}{4}\right)^{j} \cdot E[X_j]$

$$= 2cn \sum_{\text{phase } j} \left(\frac{3}{4}\right)^{j} \leq 2cn \cdot \left(\frac{1}{1 - \frac{3}{4}}\right) \leq 8cn = O(n)$$

선형시간 결정적 선택 알고리즘 (1/3)

- 확률적 선택 알고리즘에서 가장 최선의 피봇은 중간값임
- 이 피봇을 결정적 방법으로 찾을 수 있으면 선형시간 결정적 선택 알고리즘을 만들 수 있음
  - 핵심 생각. median of medians???
- findMedians 알고리즘

findMedians$(A, left, right)$

$size := right - left + 1$
if $size \le 5$ *then sort* $A[left \ldots right]$ *and return median of* A
logically divide $A[left \ldots right]$ *into* $size/5$ *groups of size 5 each*
sort each group using $O(n \log n)$ *sorting algorithm*
copy $n/5$ *medians into new array* C
return findMedians$(C, left, left + size/5)$

- in-place로 이것을 구현해야 함.
- 어떻게???
- 새 배열 C를 사용하지 않고 구현하고, 피봇 값은 최종적으로 left 위치에

- 참고. 매우 유명한 컴퓨터공학자들이 고안한 알고리즘임 (1973)
  이 알고리즘을 고안한 5명 중 4명이 Turing Award를 받았음

Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald Rivest, Robert E. Tarjan

선형시간 결정적 선택 알고리즘 (2/3)

3	11	7	12	8	2	1	10	6	15	9	5	13	14	4

3	11	7	12	8	2	1	10	6	15	9	5	13	14	4

3	7	8	11	12	1	2	6	10	15	4	5	9	13	14

8	6	9	11	12	1	2	7	10	15	4	5	3	13	14

8	6	9	11	12	1	2	7	10	15	4	5	3	13	14

```
findMedians(A, left, right)
begin, j := left, 0
while begin ≤ right do
    end := begin + 4
    if end > right then end := right
    mid := begin + (end − begin)/2
    sort(A, begin, end)
    swap(A[left + j], A[mid])
    begin += 5
    ++j
```

```
DSelect(A, left, right, i)
size := right − left + 1
If size = 1 then return A[left]
findMedians(A, left, right)
If size > 5 then
    DSelect(C, left, left + size/5 − 1, left + size/10)
    swap(A[left], A[left + size/10])
pLoc := partition(A, left, right)
if pLoc = i then return p
else if pLoc > i then
    return DSelect(A, left, pLoc − 1, i)
else return DSelect(A, pLoc + 1, right, i)
```


DSelect 분석 (1/5)

- DSelect의 시간 복잡도는 $O(n)$임
  - RSelect와 시간 복잡도가 같지만 RSelect가 더 우수함
- findMedians부터 생각해보자
  - 5개 요소를 정렬하는 비용은? 선형 시간??? 정렬하는데???
    - Why?
      - 5개 요소를 정렬하는데 120연산 이하가 소요된다고 가정
        - 합병 정렬을 생각하면 120 연산 이하임을 알 수 있음
        - 보통은 삽입 정렬을 사용함
  - findMedians의 비재귀 부분의 비용
    - $\leq \left(\frac{n}{5}\right) \cdot 120 = 24n = O(n)$ $\frac{n}{5}$: 소그룹의 수

DSelect 분석 (2/5)

```
DSelect(A, left, right, i)
size := right − left + 1
If size = 1 then return A[left]
findMedians(A, left, right)
If size > 5 then
    DSelect(C, left, left + size/5 − 1, left + size/10)
    swap(A[left], A[left + size/10])
pLoc := partition(A, left, right)
if pLoc = i then return p
else if pLoc > i then
    return DSelect(A, left, pLoc − 1, i)
else return DSelect(A, pLoc + 1, right, i)
```

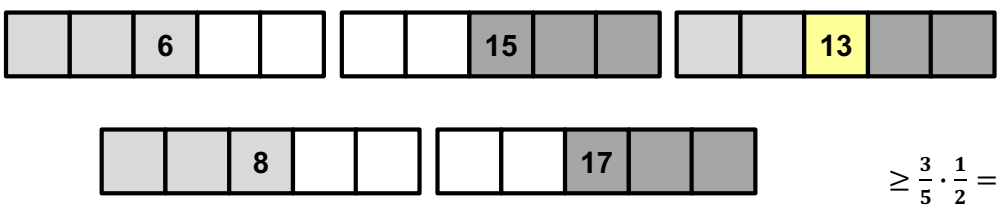
$O(n)$

$T\left(\frac{n}{5}\right)$

- DSelect 비용: $T(n) \leq T\left(\frac{n}{5}\right) + T(?) + cn$
  - 중간값의 중간값을 구하기 위한 DSelect 재귀비용: $T\left(\frac{n}{5}\right)$
  - 분할 정복을 위한 DSelect 재귀비용??? $T(?)$

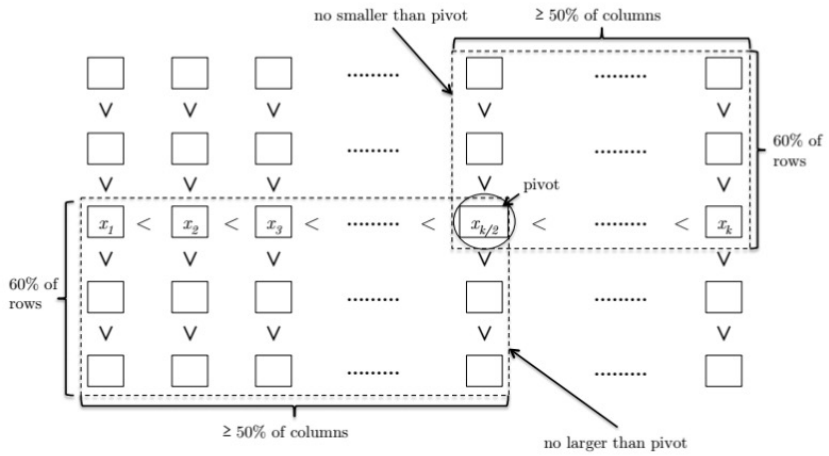
DSelect 분석 (3/5)

- DSelect 두 번째 재귀는 $\leq \frac{7}{10}n$임
  - Why?
    - $k = \frac{n}{5}$: 소그룹의 수
    - x_i: k개의 median들 중에 i번째로 작은 수 [6 8 13 15 17]
      - 피봇: $x_{k/2}$
    - 입력 배열의 요소들 중 30%은 이 피봇보다 작음 (반대도 성립)
      - 마음으로는 충분히 이해됨



$\geq \frac{3}{5} \cdot \frac{1}{2} = \frac{3}{10}$

DSelect 분석 (4/5)



- $T(n) \leq T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right) + cn$
  - Guess-and-check 방법으로 $T(n) \leq ln = O(n)$임을 증명
    - 귀납법으로…

DSelect 분석 (5/5)

- DSelect
  - $T(1) = 1$
  - $T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right)$
- $T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right) \leq 10cn$
  - 귀납법으로 증명
    - 귀납 출발. 성립
    - 귀납 가정. $T(k) \leq 10ck, \forall k < n$
    - 귀납 단계.
      - $T(n) \leq cn + T\left(\frac{n}{5}\right) + T\left(\frac{7}{10}n\right) \leq cn + 10c\left(\frac{n}{5}\right) + 10c\left(\frac{7}{10}n\right)$
        $= cn(1 + 2 + 7) = 10cn$