빠른 정렬

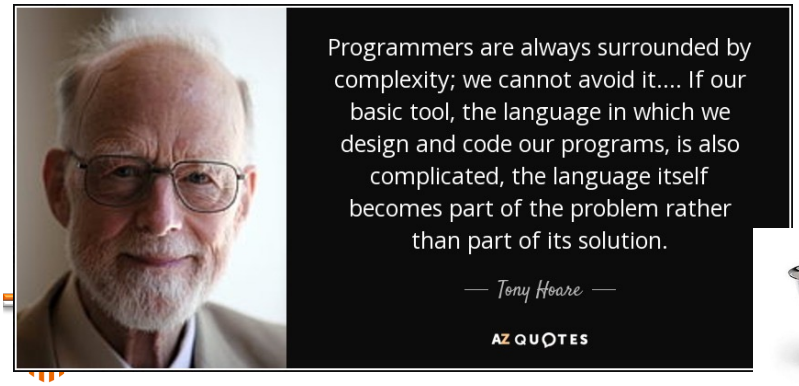
한국기술교육대학교 컴퓨터공학부 김상진

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

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

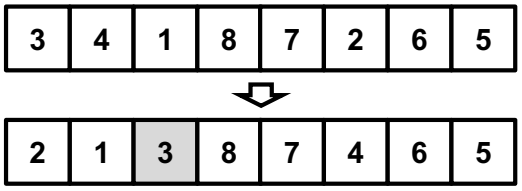
교육목표

- 빠른 정렬(quicksort)
  - 가장 훌륭한 알고리즘 중 하나
    - 실제 가장 많이 사용하는 정렬 알고리즘
      - 합병 정렬 알고리즘과 비교
  - 분할 정복 알고리즘
  - 확률적(randomized) 알고리즘
  - 평균적으로 $O(n\log n)$이며, 추가적인 공간을 거의 사용하지 않음
- 비교 기반 정렬 알고리즘은 $O(n\log n)$보다 빠를 수 없음



Pivot을 중심으로 분할

- 기본 생각
  - Pivot(정렬해야 하는 값 중 하나)을 선택
  - 요소들을 재배치하여 pivot 왼쪽에는 pivot보다 작은 값들, 오른쪽에는 pivot 보다 큰 값들이 오도록 함
    - 요소들이 정렬 위치로 재배치되는 것은 아님
    - (👍) 하지만 pivot은 올바른 위치로 이동하게 됨



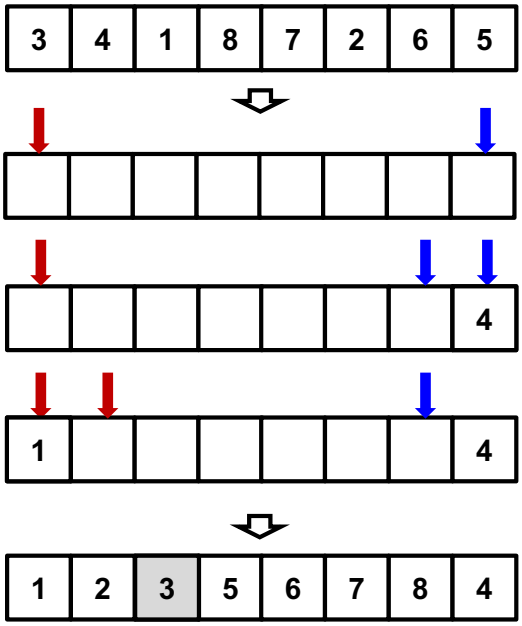
- 분할 관련 2가지 쿨한 사실
  - Fact 1. 선형 시간이 소요되며, 추가 메모리 공간이 필요하지 않음
  - Fact 2. 문제 크기를 줄임 (같은 크기는 아님)

빠른 정렬

- 분할 정복 알고리즘: quicksort(A[], left, right)
  left 색인부터 right 색인까지 정렬
  - 단계 1. if $\mathrm{left} \geq$ right then return
  - 단계 2. $p \coloneqq$ choosePivot(A, left, right)
  - 단계 3. Partition A around p
    - $\mathrm{pivotLoc} \coloneqq$ location of p after partition
  - 단계 4. quicksort(A, left, $\mathrm{pivotLoc} - 1$)
  - 단계 5. quicksort(A, $\mathrm{pivotLoc} + 1$, right)

분할 알고리즘

- 참고. $O(n)$ 공간을 추가로 사용하면 $O(n)$ 시간에 분할할 수 있음
  - How?

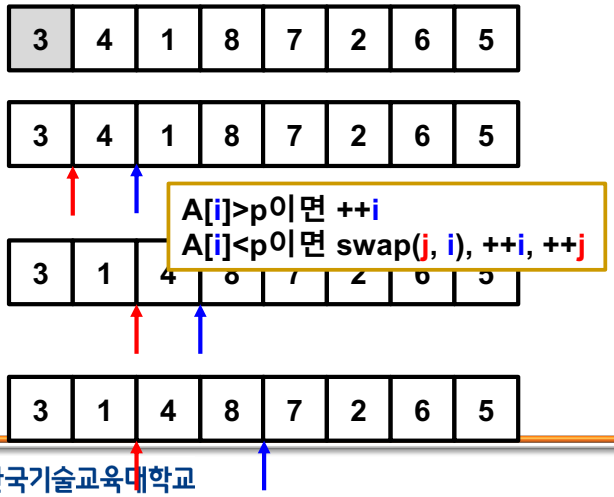


- 3을 pivot
- left = 1, right = n
- idx = 2

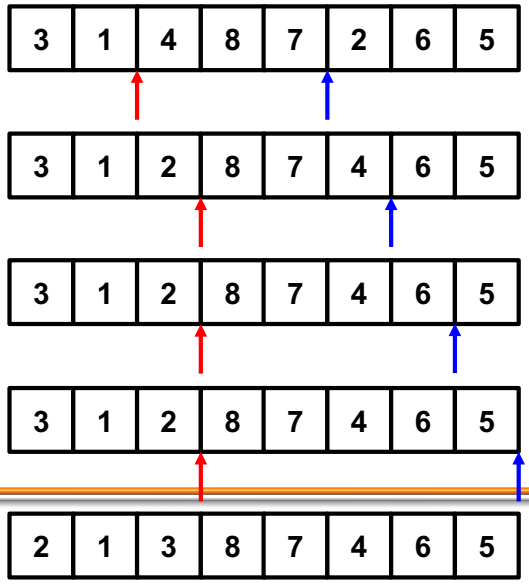
- inline 알고리즘을 만들어야 할 경우
  먼저 추가 공간을 사용하는 알고리즘을 만들고
  추가 공간에서 사용한 색인을 원 공간에
  적용하여 inline 알고리즘으로 변환

분할 알고리즘

- In-place 알고리즘
  - pivot 요소를 첫 번째 요소로 이동(swap)
  - High Level Idea
    - 단일 스캔을 함. 그래야 $O(n)$
    - 불변 조건. 특정 위치를 스캔한 결과는 pivot을 중심으로 분할되어 있어야 함 (어떻게?)



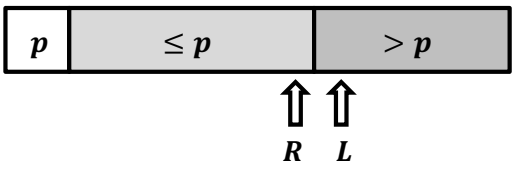
A[i]>p이면 ++i
A[i]<p이면 swap(j, i), ++i, ++j

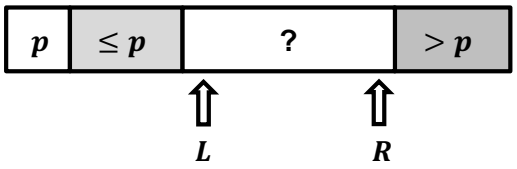


분할 알고리즘

- pseudocode

```
partition(A, left, right)                    버전 1
p := A[left]
L, R := left + 1, right
while L ≤ R do
    if A[L] ≤ p then ++L
    else
        swap(A[L], A[R--])
swap(A[left], A[R])
```

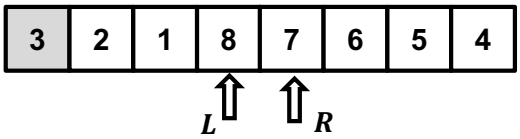





```
p := A[left]
pivotLoc := left
L, R := left + 1, right
while L ≤ R do
    while L ≤ R and p ≤ A[L] do
        L := L + 1
    while R ≥ L and p < A[R] do
        R := R − 1
    if L < R then
        swap(A[L], A[R])
        L := L + 1
        R := R − 1
swap(A[pivotLoc], A[R])                      버전 2
```


알고리즘의 정확성

- 알고리즘의 반복문의 불변 조건을 이용하여 증명함
  - 버전 1의 불변 조건. $[\text{left}, L)$ 위치에 있는 요소는 pivot보다 같거나 작고, $(R, \text{right}]$ 사이에 있는 요소는 pivot보다 큼



  - 버전 2의 불변 조건. $[\text{left}, L)$ 위치에 있는 요소는 pivot보다 같거나 작고, $(R, \text{right}]$ 사이에 있는 요소는 pivot보다 큼 (차이가 없음)
- 이것이 성립하면 마지막 swap을 통해 정확하게 분할된다는 것을 알 수 있음
  - 귀납법(다음 슬라이드)으로 증명 가능
  - 버전 1의 경우 각 반복마다 한 요소를 처리함
    - 이 요소가 pivot보다 같거나 작은 경우: L만 증가
    - 이 요소가 pivot보다 큰 경우: swap 후 R 감소

귀납법(induction)을 이용한 증명

- $P(n)$: 양의 정수가 매개 변수인 주장
- $n \geq 1$인 모두 n에 대해 $P(n)$이 성립함을 귀납법으로 증명하는 방법
  - 기저 사례(base case)인 $P(1)$ 증명
  - 귀납 단계(inductive step): 모든 $n \geq 2$에 대해 증명
    - 귀납 가정(inductive hypothesis): $k < n$인 모든 k에 대해 $P(k)$가 성립한다고 가정
    - 귀납 가정을 이용하여 $P(k+1)$에 대해 증명

빠른 정렬의 정확성 증명 (1/2)

- 불변조건: $\forall x \in [\text{left}, L) \Rightarrow x \leq \text{pivot}, \forall x \in (R, \text{right}] \Rightarrow x < \text{pivot}$
- 증명) 귀납법 (버전1)
  - 귀납 출발. 반복문 시작 전 $[\text{left}, L)$ 구간에는 피봇만 하나 있고, $(R, \text{right}]$은 빈 구간이므로 성립함
  - 귀납 가정. $k-1$ 반복 후 불변 조건 만족
  - 귀납 단계. k번째 반복
    - $A[L]$과 피봇을 비교함
      - 같거나 작으면 L만 증가 $\Rightarrow$ 귀납 가정에 $A[L](\leq \text{pivot})$만 $[\text{left}, L)$ 구간에 추가됨 $\Rightarrow$ 불변 조건 계속 만족
      - 크면 $A[L]$과 $A[R]$을 swap한 후에 R 감소 $\Rightarrow$ 귀납 가정에 $A[R](> \text{pivot})$만 $(R, \text{right}]$ 구간에 추가됨 $\Rightarrow$ 불변 조건 계속 만족
- 종료 후 $A[R]$과 $A[\text{left}]$를 swap하면 피봇을 중심으로 피봇과 같거나 작은 것은 왼쪽에 큰 것은 오른쪽에 위치함
  - 분할 알고리즘은 정확함

빠른 정렬의 정확성 증명 (2/2)

- 귀납법 증명
  - 귀납 출발. $P(1)$은 당연히 성립
    - 왜? 요소가 하나면 그 자체가 정렬되어 있음
  - 귀납 가정. 모든 $k < n$ 에 대해 $P(k)$가 성립함
    - n보다 작은 크기의 배열은 빠른 정렬이 올바르게 정렬함
  - 귀납 단계.
    - 분할 알고리즘은 정확하기 때문에 n크기의 배열이 pivot 중심으로 분할한 후 모습은 다음과 같음

    - 그런데 귀납 가정에 의해 quicksort는 n보다 작은 것을 올바르게 정렬하기 때문에 왼쪽 부분과 오른쪽 부분은 재귀 호출에 의해 올바르게 정렬됨

정확성 증명의 중요성

- judge와 같은 프로그래밍 경시대회 문제를 풀 때
  - 알고리즘을 고안하고 구현한 다음 몇 가지 데스트 후 제출하여 통과하면 알고리즘이 정확하다고 생각함
  - 실제 보통 증명 후 제출하지는 않음
- 통과를 못하는 경우
  - 알고리즘이 부정확한 경우
  - 알고리즘은 정확하지만 구현을 잘못한 경우
  - 테스트 데이터에 문제가 있는 경우 (전문 사이트는 이와 같은 경우가 없음)
- 통과를 한 경우
  - 알고리즘이 여전히 부정확할 수 있음
    - 테스트 데이터가 모든 경우를 검증하지 못할 수 있음

피봇의 중요성

- 질문. 빠른 정렬의 시간 복잡도는?
  - 쉽게 분석할 수 없음
- Why? 피봇의 선택에 따라 필요한 비용이 많이 달라짐
  - 어떤 피봇이 좋을까?
    - 배열을 반으로 잘 나누는 놈: 중간값(median element)
- 정렬된 배열에서 첫 번째 요소를 피봇으로 사용하였을 때 소요되는 시간 복잡도의 빅O는? $O(n^2)$
- 매 번 중간값을 피봇으로 사용할 수 있다고 가정하면 시간 복잡도는? $O(n\log n)$
  - 이것의 점화식은? $T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$ 도사정리에서 경우 1에 해당
  - n개의 요소가 있을 때 중간값을 찾는 비용은? $O(n)$
    - 비재귀적 비용: 중간값 찾기 + 분할 = $O(n)$

피봇의 선택

- 어떻게 선택해야 가장 좋을까?
  - 랜덤 피봇: 모든 요소를 선택할 확률이 같음
    - 좋을까?
      - 절대 매번 중간값을 선택할 수는 없음
      - 다수의 경우에는 좋은 피봇를 선택하지 않을까?
      - 극단으로 잘못 선택하면 $O(n^2)$, 잘 선택하면 $O(n\log n)$
    - Fact. 항상 25 ~ 75 분할을 할 수 있다면 빠른 정렬은 $O(n\log n)$
      - 재귀 트리의 높이를 생각하여 보자
    - Fact. 보통 절반의 요소가 25 ~ 75 분할을 할 수 있음
      - 100개의 요소가 있으면 26번째부터 75번째 사이에 오는 요소를 선택하면 25 ~ 75 분할이 이루어짐
- 빠른 정렬은 추가 공간을 사용하지 않고, 우리가 빅O 계산에서 무시한 상수도 비교적 작기 때문에 성능이 매우 우수함

빠른 정렬: 평균 시간 복잡도

- 빠른 정렬 정리: 크기가 n인 모든 배열에 대해 랜덤 피봇을 사용하는 빠른 정렬의 평균 시간 복잡도는 $O(n\log n)$임
- 참고.
  - 입력 배열에 대한 어떤 가정도 하지 않음 (모든 경우에 성립)
  - 동일 데이터에 대해서도 매번 진행 과정은 다름
  - 빠른 정렬의 시간 복잡도의 범위는 $n\log n$에서 n^2임
    - 하지만 이 정리에 의하면 대부분 $n\log n$이라는 것임
      - 왜 평균이 $n\log n$이기 때문

빠른 정렬에 대한 수학적 분석

- 확률에 대한 기본적 지식이 필요
- Preliminaries
  - 크기가 n인 배열 A
  - 표본 공간(sample space) Ω: 가능한 모든 랜덤 pivot의 선택 (pivot sequences)
  - 확률 변수(random variable): $\sigma \in \Omega$에 대해 $C(\sigma)$
    - $C(\sigma)$: 빠른 정렬이 수행한 비교 수
- 우리가 증명해야 하는 것은 $E[C] = O(n\log n)$
- 참고. 도사 정리를 적용할 수 없음
  - Why? 줄어드는 문제의 크기가 고정되어 있지 않고, 두 개의 소문제의 크기가 같지 않음

빠른 정렬에 대한 수학적 분석

- z_i: 배열에서 i번째로 작은 요소 (정렬 후 i번째 색인에 있어야 하는 요소)
- $\sigma \in \Omega$이 주어졌을 때,
  $X_{ij}(\sigma)$: 빠른 정렬에서 z_i와 z_j를 서로 비교한 횟수
  - $X_{ij}(\sigma)$ 값의 범위는? 0 또는 1 두 요소는 최대 한번만 비교함
- 언제 비교가 일어나는가?
  - 빠른 정렬에서는 항상 pivot은 정렬해야 하는 다른 모든 값과 비교함
  - pivot과 비교된 후에 다시는 pivot과 해당 값을 비교하지 않음
  - 다른 쪽으로 분할된 이후에는 서로 비교하지 않음
- 모든 σ에 대해 $C(\sigma) = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}(\sigma)$임
- $E[C] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}]$
- $E[X_{ij}] = 0 \cdot \Pr[X_{ij}= 0] + 1 \cdot \Pr[X_{ij} = 1] = \Pr[X_{ij}= 1]$
- $E[C] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$

빠른 정렬에 대한 수학적 분석

- $\Pr[X_{ij} = 1]$????? $\Pr[X_{ij} = 1] = \frac{2}{j-i+1}$
- $i < j$인 z_i, z_j, 피봇 z_k
  - 경우 1. $z_k < z_i$: 둘 다 두 번째 재귀 호출로 전달 (미래에 비교 가능)
  - 경우 2. $z_k > z_j$: 둘 다 첫 번째 재귀 호출로 전달 (미래에 비교 가능)
    - z_i와 z_j 사이에 있는 요소는 항상 같은 분할로 이동함
  - 경우 3. $z_i < z_k < z_j$: 서로 다른 재귀 호출로 전달
    - 둘은 절대 비교되지 않음
  - 경우 4. z_i와 z_j 중 하나가 피봇으로 선택
    - 비교가 이루어지고, 다시는 비교가 일어나지 않음
- $\Pr[X_{ij} = 1] = \Pr[z_i$ 또는 z_j가 $z_{i+1}, \dots z_{j-1}$전에 피봇으로 선택]
  - 모든 요소를 피봇으로 선택할 확률은 같음
  - 비교가 일어나는 경우는 z_i 또는 z_j가 피봇으로 선택된 경우
  - 총 경우의 수: $j - i + 1$

$z_a(< z_i)$와 $z_b(> z_j)$는 z_i와 z_j 비교에 영향을 주지 않음

빠른 정렬에 대한 수학적 분석

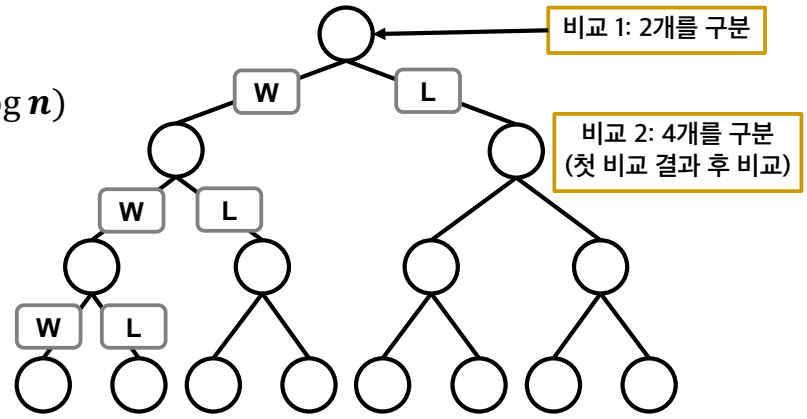
- $E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{j-i+1}$
- 참고.
  - $i = 1$이면 내부 합은 $\sum_{j=2}^{n} \frac{1}{j} = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$
  - $i = 2$이면 내부 합은 $\sum_{j=3}^{n} \frac{1}{j-1} = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n-1}$
  - …
- $E[C] \leq 2n \sum_{k=2}^{n} \frac{1}{k} \leq 2n \ln n$
  - $\sum_{k=2}^{n} \frac{1}{k} \leq \ln n$

비교 기반 정렬 알고리즘의 한계 (1/2)

- 결론. 비교 기반 정렬 알고리즘은 $O(n \log n)$보다 빠를 수 없음
- Why? 증명?
- 비교를 0번
  - [1 2 3 4]와 [4 3 2 1]을 구분할 수 없음
    - 구분할 수 없으면 알고리즘은 동일하게 동작하기 때문에 출력의 차이가 있을 수 없음
    - 따라서 둘 중 하나는 오답. 정렬할 수 없음
- 비교를 1번
  - 첫 2개를 비교한다고 가정하면 [1 2 3 4]와 [1 2 4 3]을 구분할 수 없음
- 비교를 2번
  - [1 2 3 4]와 [1 3 2 4]를 구분할 수 없음
- Fact. 비교 기반 정렬은 비교를 통해 크고 작음만 알 수 있음

비교 기반 정렬 알고리즘의 한계 (2/2)

- n개 서로 다른 값으로 구성된 배열: 가능한 경우의 수 $n!$
- 비교를 k번
  - 2^k개의 서로 다른 입력을 구분할 수 있음
- 다음이 성립해야 k번 비교로 크기가 n인 배열을 정렬할 수 있음
  - $2^k \geq n!$
    - $2^k \geq n^n \geq n!$
    - $k \geq n \log n = O(n \log n)$



비교 기반이 아닌 정렬 알고리즘

- 버킷 정렬
  - 입력 데이터가 알려진 범위 내에 균일하게 분포되어 있는 경우 사용할 수 있는 정렬 알고리즘
  - 일정한 수의 버킷 준비
    - 범위를 n등분하여 각 영역을 나타내는 버킷
  - 입력 배열을 스캔하면서 각 요소를 해당 버킷에 삽입: $O(n)$
    - 각 버킷에 삽입하는 요소의 수는 매우 작아야 함
    - 다음 과정이 선형 시간 내에 정렬할 수 있을 정도로 작아야 함
  - 각 버킷 내 정렬: $O(1)$
  - 버킷을 이용하여 최종 배열 출력: $O(n)$
  - 전체 비용은 $O(n)$

버킷 정렬 응용

- 입력. n개의 순 증가하는 정수로 구성된 배열 A, 양의 정수 d, $0 \le A[i] \le 200$
- 출력. 다음을 만족하는 $(A[i], A[j], A[k])$ 3쌍의 수
  - 조건 1. $i < j < k$
  - 조건 2. $A[j] - A[i] = d$, $A[k] - A[j] = d$

```
count := 0
for i := 1 to n − 2 do
    found := false
    for j := i + 1 to n − 1 do
        if A[j] = A[i] + d then found := true
    if not found then continue
    for  k := j + 1 to n do
        if A[k] = A[j] + d then found := true
    if found  then ++count
```

$A[i]$의 범위가 비교적 작음

```
N := [false]×201 // 0 색인
for i := 1 to n do
    N[A[i]] := true
```


부록. 확률 (1/4)

- 표본 공간(sample space) Ω: 가능한 모든 결과
  - 알고리즘 세계에서 표본 공간은 유한
  - 예) 6면 주사위 2개를 던지기. $\Omega = \{(1,1), (1,2), \ldots, (6,6)\}$
- 각 결과가 나타날 확률 $\Pr[i] \ge 0$임
  - Fact. $\sum_{i \in \Omega} \Pr[i] = 1$
  - 예) 6면 주사위 2개 던지기: $\Pr[i] = 1/36$
- 사건(event): $S \subseteq \Omega$
  - $\Pr[S] = \sum_{i \in S} \Pr[i]$
  - 예) 6면 주사위 2개 던지기에서 결과의 합이 7인 경우
    - $p(S) = 1/6$
- 확률변수: Ω에서 실수값으로 매핑하여 주는 함수
  - 예) X: 6면 주사위 2개 던지기에서 결과의 합
    - $\Pr[X = 7] = 1/6$

부록. 확률 (2/4)

- 확률변수 X의 기댓값(Expectation, Expected Value): $E[X]$
  - X의 평균값
  - $E[X] = \sum_{i \in \Omega} X(i)\Pr[i]$
  - 예) X: 6면 주사위 2개 던지기에서 결과의 합
    - $E[X] = 7$
- 기댓값의 선형성(linearity of expectation)
  - $X_1, X_2, \ldots, X_n$이 Ω에 정의된 확률 변수이면 다음이 성립함

$$E\left[\sum_{j=1}^{n} X_j\right] = \sum_{j=1}^{n} E[X_j]$$

    - 서로 독립적인 확률 변수가 아니어도 성립
  - 예) 6면 주사위 두 개 던지기에서 X_1이 첫 번째 주사위 값, X_2가 두 번째 주사위 값
    - $E[X_i] = 3.5$
    - 기대값의 선형성에 의해 $E[X_1 + X_2] = E[X_1] + E[X_2] = 7$

부록. 확률 (3/4)

- 조건부 확률: 사건 $A, B \in \Omega$에 대해 조건부 확률 $P[A|B]$는 다음과 같이 정의함 (A given B)

$$\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]}$$

- 예) 두 주사위를 던진 결과의 합이 7일 때 두 주사위 중 하나가 1일 확률은? $\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]} = \frac{2/36}{6/36} = \frac{1}{3}$
- 독립 사건(independence of events)
  - 사건 $A, B \in \Omega$가 독립 사건이기 위한 필요충분조건은
  - $\Pr[A \cap B] = \Pr[A] \cdot \Pr[B]$
- 독립 확률 변수(independence of random variable)
  - Ω에 정의된 확률변수 X, Y가 독립이기 위한 필요충분조건은 모든 x, y에 대해 $P[X = x]$와 $P[Y = y]$가 독립적이어야 함
  - 확률변수 X와 Y가 독립이면 $E[X \cdot Y] = E[X] \cdot E[Y]$임
    - 기댓값의 선형성과 달리 특정 조건하에서만 성립

부록. 확률 (4/4)

	00	10	01	11	E
X_1	0	1	0	1	½
X_2	0	0	1	1	½
X_3	0	1	1	0	½
X_4	0	1	0	0	¼
X_1X_3	0	0	0	1	¼
X_2X_4	0	0	0	0	0

- 예) $\Omega = \{00, 10, 01, 11\}$
  - 확률변수 X_1: 첫 번째 비트 값
  - 확률변수 X_2: 두 번째 비트 값
  - 확률변수 X_3: 첫 번째 비트 값과 두 번째 비트 값을 XOR한 값
  - 확률변수 X_4: 첫 번째 비트값과 첫 번째와 두번째 비트 값의 XOR한 값을 곱한 값
  - X_1과 X_3는 독립 확률변수임
    - Why? $E[X_1X_3] = E[X_1] \cdot E[X_3]$가 성립해야 함. 양변 모두 ¼임
  - X_2와 X_4는 독립 확률변수가 아님
    - $E[X_2X_4] = 0 \neq E[X_2] \cdot E[X_4] = 1/8$