알고리즘및실습

제5장 빠른 정렬

1. 확률적 알고리즘

요소를 서로 비교하여 정렬하는 비교 기반 정렬 알고리즘은 $\Theta(n \log n)$보다 좋을 수 없다. 따라서 합병 정렬은 가장 우수한 정렬 알고리즘 중 하나이다. 하지만 합병 정렬은 공간 복잡도가 좋지 않은 단점이 있다. 따라서 실제 각종 언어 라이브러리에서 제공하는 정렬 함수는 합병 정렬을 사용하지 않는다.

이 장에서는 합병 정렬과 시간 복잡도 같지만 추가 공간을 사용하지 않는 **빠른 정렬**(quicksort) 알고리즘을 살펴본다. 이 알고리즘은 합병 정렬과 마찬가지로 분할 정복 알고리즘이다. 하지만 지금까지 살펴본 알고리즘과 다르게 **확률적**(randomized) 알고리즘이다. 확률적 알고리즘은 같은 입력에 대해 알고리즘을 실행할 때마다 다르게 동작한다. 하지만 알고리즘이기 때문에 다르게 동작하더라도 결과는 동일하다.

따라서 이 장의 핵심 내용은 확률적 알고리즘이다. 우리는 확률적 알고리즘을 설계하여 기존 알고리즘의 성능을 개선할 수 있다. 하지만 확률적 알고리즘은 결정적 알고리즘보다 성능을 분석하기 어렵다.

2. 빠른 정렬

Tony Hoarse가 개발한 빠른 정렬의 기본적 생각은 요소 중 임의의 값을 기준으로 그것보다 작은 것과 큰 것으로 요소를 재배치하는 것이다. 이때 기준으로 사용한 요소를 피봇(pivot)이라 하고, 이 과정을 분할(partition)이라 한다. 이 재배치를 통해 각 요소가 정렬된 위치로 이동하는 것은 아니다. 하지만 피봇은 정렬된 위치로 이동하게 된다. 이 분할의 이점은 다음과 같다.

- 첫째, 선형 시간에 분할할 수 있고, 추가 메모리 공간이 필요하지 않다.
- 둘째, 문제의 크기가 줄일 수 있다.

여기서 문제의 크기를 줄인다는 것은 피봇을 중심으로 피봇보다 작은 것은 피봇의 왼쪽, 큰 것은 피봇의 오른쪽으로 이동하게 되는데, 피봇을 제외한 왼쪽에 있는 것을 정렬하고, 오른쪽에 있는 것을 정렬하면 전체가 정렬이 된다. 따라서 분할을 통해 두 개의 소문제를 얻게 되며, 이를 재귀적으로 정렬하여 문제를 해결할 수 있다. 합병 정렬과 비슷하지만 차이점은 두 개의 소문제 크기가 같지 않다는 것이다. 빠른 정렬 알고리즘은 알고리즘 5.1과 같다.

알고리즘 5.1 빠른 정렬 알고리즘

```
1: procedure SORT(A[])
2:     QUICKSORT(A, 1, LEN(A))
3: procedure QUICKSORT(A[], lo, hi)
4:     if lo ≥ hi then return
5:     CHOOSEPIVOT(A, lo, hi)
6:     pivotLoc := PARTITION(A, lo, hi)
7:     QUICKSORT(A, lo, pivotLoc − 1)
8:     QUICKSORT(A, pivotLoc + 1, hi)
```

2.1 분할 알고리즘

피봇이 주어졌을 때 이를 기준으로 피봇보다 작은 것을 피봇의 왼쪽, 피봇보다 큰 것을 피봇의 오른쪽으로 이동하는 분할 알고리즘은 추가 공간을 사용하지 않고 구현할 수 있다. inline 알고리즘을 만드는 것이 어렵게 느껴지면 추가 공간을 사용하면 어떻게 만들 수 있는지 생각하여 보고, 이를 활용 및 변환하여 inline 알고리즘을 만드는 것을 시도해 볼 필요가 있다. 특히, 추가 공간에서 사용한 색인을 원 공간에 적용하면 보통 쉽게 inline 알고리즘으로 바꿀 수 있다.

추가 공간을 사용하지 않고 어떻게 할 수 있는지 먼저 생각하여 보자. 피봇을 임시 변수에 저장하고, 피봇보다 작으면 1부터 순방향으로 차례로 저장하고, 피봇보다 크면 n부터 역방향으로 차례로 저장하면 된다. 피봇을 만나면 건너뛴 후에 모든 요소를 처리하였으면 피봇값을 순방향의 다음 위치에 아니면 역방향의 다음 위치에 저장하면 분할을 완료할 수 있다. 이 알고리즘의 비용은 $O(n)$이다.

피봇과 같은 값이 여러 개 있을 수 있다. 이 문제는 피봇을 임시 변수로 옮기지 않고 1번 위치에 저장한 후에 피봇보다 작거나 같으면 2부터 순방향으로 차례로 저장하는 형태로 알고리즘을 바꾸고, 모든 요소를 처리한 후에 순방향 마지막 요소와 1번 위치에 있는 pivot과 교환한다. 이 분할 알고리즘의 경우 두 개의 색인 변수를 사용하고 있다. 이를 L과 R이라 하면 $L = 2$, $R = n$로 초기화하여 사용해야 하며, L은 1씩 증가하고, R은 1씩 감소한다.

그러면 L과 R을 원 배열 전체에 적용하여 분할하는 알고리즘을 생각하여 보자. 우선 피봇은 1번 위치에 옮겼고, 1번 위치에 있던 요소는 피봇 위치로 옮긴 후부터의 알고리즘을 생각하여 보자. 추가 공간을 사용한 알고리즘의 경우 색인 변수를 이용하여 원 배열에 있는 요소를 하나씩 처리한다. inline 알고리즘에서는 L 변수를 2로 설정하여 하나씩 요소를 검사한다. $L = 2$일 때를 생각하여 보자. $A[L]$ 요소가 피봇보다 작거나 같으면 L을 1 증가하면 된다. 반대로 $A[L]$가 피봇보다 크면 R 위치에 있는 요소와 교환하고 R을 1 감소하면 된다. 하지만 이 경우에는 L은 그대로 유지한다. R 위치의 요소를 옮겨왔기 때문에 다시 L 위치부터 검사해야 한다. 이 과정은 $L > R$이 참일 때까지 반복하며 최종적으로 $A[1]$과 $A[R]$를 교환하면 된다. 이 알고리즘은 알고리즘 5.2과 같다.

알고리즘 5.2 분할 알고리즘

```
1: function PARTITION(A[], lo, hi)
2:     pivot := A[lo]
3:     L, R := lo + 1, hi
4:     while L <= R do
5:         if A[L] ≤ pivot then
6:             ++L
7:         else
8:             SWAP(A[L], A[R−−])
9:     SWAP(A[lo], A[R])
10:    return hi
```

위 알고리즘에서 진행할 때 검사하는 요소가 피봇가 같거나 작으면 SWAP 없이 L만 증가한다. 마찬가지로 $A[\texttt{hi}]$ 위치에 있는 요소가 피봇보다 크면 그 위치에 계속 있어도 분할의 조건을 만족한다. 따라서 분할 알고리즘을 알고리즘 5.3과 같이 개선할 수 있다. 이 알고리즘은 이중 반복문으로 구성되어 있지만 각 요소를 최대 한번만

검사하기 때문에 시간 복잡도는 $O(n)$이다.

알고리즘 5.3 분할 알고리즘

```
1: function PARTITION(A[], lo, hi)
2:     pivot := A[lo]
3:     L, R := lo + 1, hi
4:     while L <= R do
5:         while L <= R and A[L] ≤ pivot do ++L
6:         while L <= R and A[R] > pivot do −−R
7:         if L <= R then
8:             SWAP(A[L++], A[R−−])
9:     SWAP(A[lo], A[R])
10:    return hi
```

앞서 설명한 2가지 방법 외에 다른 방법으로 분할할 수도 있다. 예를 들어 알고리즘 5.4와 같이 분할 수 있다.

알고리즘 5.4 분할 알고리즘

```
1: function PARTITION(A[], lo, hi)
2:     pivot := A[lo]
3:     L := lo + 1
4:     for i := lo + 1 to hi do
5:         if A[i] <= pivot then
6:             SWAP(A[i], A[L++])
7:     SWAP(A[lo], A[L − 1])
8:     return L − 1
```

2.2 분할 알고리즘의 정확성

지금까지 알고리즘을 제시하고 설명하면서 알고리즘의 정확성을 증명하지는 않았다. 알고리즘의 정확성을 증명하는 것은 매우 중요하다. 하지만 개발 과정에서는 종종 알고리즘을 수학적으로 증명하지 않고 사용하는 경우가 많다.

예를 들어 프로그래밍 경시대회 문제를 풀 때 문제를 해결하는 알고리즘을 고안하고 이를 증명한 후에 제출하지는 않는다. 몇 가지 테스트 데이터를 이용하여 검증한 후에 제출하는 것이 일반적이다. 이때 통과하지 못하면 그것의 문제는 다음 중 하나이다.

- 경우 1. 알고리즘이 정확하지 않은 경우
- 경우 2. 알고리즘이 정확하지만 구현을 잘못한 경우
- 경우 3. 알고리즘이 정확하지만 테스트 데이터에 문제가 있는 경우

보통 잘 구축된 사이트에서 경우 3은 발생하지 않는다. 따라서 대부분의 경우는 경우 1 아니면 경우 2이다. 물론 통과를 하더라도 알고리즘이 정확하다고 단정할 수는 없다. 준비된 테스트 데이터가 모든 경우를 검증하지 못할 수 있기 때문이다. 전문적으로 운영되는 사이트의 경우에는 이와 같은 상황도 배제할 수 있다.

그럼에도 불구하고 알고리즘의 정확성을 증명할 수 있으면 알고리즘에 대한 신뢰성을 높일 수 있다. 전수 조사하는 알고리즘은 전수 조사만 정확하게 한다면 알고리즘의 정확성은 보장된다. 따라서 전수 조사를 하지 않는 알고리즘은 더욱 더 정확성에 대한 증명이 필요하다. 예를 들어 8장에서 살펴보는 탐욕적 알고리즘은 꼭 정확성 증명이 필요한 알고리즘 형태이다. 빠른 정렬 알고리즘은 확률적 알고리즘이기 때문에 결정적 알고리즘보다는 정확성 증명이 더 중요할 수 있다.

보통 정확성 증명할 때 많이 사용하는 기법 중 하나는 귀납법이다. 어떤 n에 대한 귀납법은 $n = 1$인 기저 사례에 대해 성립한다는 것을 보이고, $k < n$인 k에 대해 성립한다고 가정을 한 후 $k + 1$일 때 이 가정을 이용하여 증명함으로써 모든 n에 대해 성립한다는 것을 증명하는 방법이다.

프로그래밍에서 반복문으로 구성된 알고리즘은 반복문의 불변 조건을 이용하여 증명한다. 이때에도 귀납법을 사용하여 증명한다. 반복 전에 불변 조건이 만족됨을 보이고, 귀납 가정을 통해 k번 반복하였을 때 불변 조건이 성립한다는 것을 가정한 후, 다음 단계에서도 불변 조건이 성립한다는 것을 보임으로써 반복문 종료 후에 불변 조건이 성립한다는 것을 증명하면 된다.

이전 절에서 제시한 3개 알고리즘의 불변 조건을 살펴보자.

- 알고리즘 5.2: $[\texttt{lo}, L)$은 `pivot`과 같거나 작으며, $(R, \texttt{hi}]$은 `pivot`보다 크다.
- 알고리즘 5.3: $[\texttt{lo}, L)$은 `pivot`과 같거나 작으며, $(R, \texttt{hi}]$은 `pivot`보다 크다.
- 알고리즘 5.4: $[\texttt{lo}, L)$은 `pivot`과 같거나 작으며, $[L, i)$는 `pivot`보다 크다.

이 불변 조건이 계속 만족된다는 것을 증명하면 분할 알고리즘의 정확성을 증명할 수 있다.

알고리즘 5.2에 대해서는 귀납법으로 이 불변 조건이 항상 성립한다는 것을 증명하여 보자. 반복문이 시작하기 전 $[\texttt{lo}, L)$ 범위에 피봇만 위치하고 있고, $(R, \texttt{hi}]$ 범위는 빈 구간이므로 성립한다. $k-1$번 반복한 후에 $[\texttt{lo}, L)$ 범위와 $(R, \texttt{hi}]$ 범위에 있는 요소들이 불변 조건을 만족한다고 가정하자. k번째 반복에서 알고리즘은 $A[L]$과 `pivot`을 비교한다. $A[L]$이 `pivot`보다 같거나 작은 경우와 $A[L]$이 `pivot`보다 큰 경우, 두 가지 경우로 나누어진다. 전자의 경우 L만 1 증가한다. 귀납 가정에 의해 $[\texttt{lo}, L-1)$은 성립하였고 $A[L]$은 `pivot`보다 같거나 작기 때문에 불변 조건이 계속 만족하며, $(R, \texttt{hi}]$에서 R은 이번 반복에서 변하지 않았으므로 귀납 가정에 의해 여전히 성립한다. 후자의 경우 $A[L]$에 있는 요소를 $A[R]$과 바꾸고 R를 1 감소한다. 따라서 $(R, \texttt{hi}]$ 구간에 새 요소가 하나 추가되었지만 이 요소는 `pivot`보다 큰 값이고, 기존 요소는 귀납 가정에 불변 조건을 만족하므로 새 구간도 불변 조건을 계속 만족한다. 또 이 경우 L은 변하지 않기 때문에 $[\texttt{lo}, L)$은 귀납 가정에 의해 계속 불변 조건을 만족한다. 결론적으로 불변 조건은 계속 만족하며 마지막 반복 후 $A[R]$ 위치는 $L-1$ 위치이며, 이 위치와 $A[\texttt{lo}]$에 피봇과 교환하면 피봇을 기준으로 피봇과 같거나 작으면 피봇 왼쪽에 피봇보다 큰 것은 오른쪽에 위치하게 된다.

빠른 정렬 알고리즘 자체의 증명은 분할 알고리즘만 증명하면 쉽게 귀납법을 이용하여 증명할 수 있다. 정렬할 배열의 길이에 대한 귀납법으로 증명하여 보자. 기저 사례는 당연히 성립한다. 요소가 하나밖에 없으면 그 자체가 정렬된 상태이며, 이 알고리즘을 적용하였을 때 배열의 상태는 변하지 않는다. 배열 크기가 n보다 작을 때 빠른 정렬이 성립한다고 가정하자. 이를 이용하여 배열 크기가 n일 때를 정렬된다는 것을 증명하여 보자. 배열을 정렬하기 위해 먼저 분할 알고리즘을 적용하게 된다. 분할 알고리즘은 이전에 증명하기 때문에 피봇은 올바른 정렬 위치로 이동하게 되며, 피봇과 같거나 작은 것은 피봇 왼쪽에, 피봇보다 큰 요소는 오른쪽 위치하게 된다. 이 두 부분의 크기는 모두 n보다 작기 때문에 이들을 재귀적으로 빠른 정렬을 이용하여 정렬하면 귀납 가정에 의해 올바르게 정렬된다. 따라서 n일 때도 정확하게 정렬되므로 빠른 정렬은 정확하다는 것을 알 수 있다.

3. 피봇의 중요성

빠른 정렬 알고리즘의 정확성을 이전 절에서 증명하였다. 지금부터 빠른 정렬의 성능을 분석하고 싶다. 하지만 빠른 정렬은 쉽게 분석할 수 없다. 재귀 알고리즘의 점화식이 표준 점화식으로 표현되지 않기 때문에 이전 장에서 학습한 도사 정리를 이용하여 분석할 수 없다.

빠른 정렬의 시간 복잡도를 분석하기 위해 어떤 피봇을 선택하여야 효과적인지부터 분석하여 보자. 피봇으로 가장 효과적인 것은 배열의 중간값이다. 항상 정렬해야 하는 요소들 중 중간값을 피봇으로 선택할 수 있다면 분할

알고리즘을 적용한 후에 피봇 왼쪽에 있는 요소의 개수와 오른쪽에 있는 요소의 개수 차이가 1 이하가 된다. 이와 같은 경우에는 표준 점화식으로 표현할 수 있으며, 점화식은 다음과 같다.

$$T(n) \leq 2T\left(\frac{n}{2}\right) + O(n)$$

이 점화식을 도사 정리에 적용하면 경우 1에 해당하며, 합병 정렬과 점화식이 같으므로 시간 복잡도는 $O(n \log n)$이 된다.

중간값을 찾는 것은 선택 문제(selection problem)의 일종이며, 중간값은 $O(n)$에 찾을 수 있다. 따라서 항상 중간값을 찾은 후, 이 값을 피봇으로 사용하여 분할하는 빠른 정렬 알고리즘을 만들 수 있으며, 이 형태의 빠른 정렬 알고리즘의 시간 복잡도는 $O(n \log n)$이 된다. 하지만 중간값을 찾는 것은 $O(n)$이기는 하지만 비용이 소요되며, 알고리즘의 구현의 복잡성도 증가한다. 따라서 실제 중간값을 찾지 않고 효과적으로 피봇을 선택하는 방법이 있다면 빠른 정렬의 성능을 더 향상할 수 있고, 코드의 간결성도 향상할 수 있다.

정렬해야 하는 요소 중 하나를 랜덤하게 선택하면 성능은 어떻게 될까? 랜덤하게 선택한다는 것은 n개 요소 중 하나를 피봇을 선택할 확률이 $1/n$이라는 것을 의미한다. 따라서 절대 매번 중간값을 선택할 수는 없다. 하지만 다수의 경우에는 좋은 피봇을 선택한다면 성능은 매번 중간값을 선택하는 것과 비슷한 성능을 얻을 수 있다. 더욱이 랜덤으로 선택하면 피봇을 선택하는 비용이 $O(n)$이 아니라 $O(1)$이 된다. 실제 매번 중간값을 선택한다면 빠른 정렬의 시간 복잡도는 $O(n \log n)$이 되지만 반대로 매번 최악의 경우를 선택하면 빠른 정렬의 시간 복잡도는 $O(n^2)$이 된다. 여기서 최악의 경우는 n개를 1과 $n-2$개로 분할한 경우이다.

중간값을 선택하지 못하더라도 매번 $25 \sim 75$ 분할을 할 수 있다면 빠른 정렬의 시간 복잡도는 $O(n \log n)$이 된다. $25 \sim 75$ 분할이란 한쪽이 전체의 75%이하가 된다는 것이다. 이렇게 하였을 때 $O(n \log n)$되는 이유는 다음 절에서 자세히 분석한다. 우선 랜덤하게 피봇을 선택하였을 때 매번 $25 \sim 75$ 분할을 할 수 있는지 생각하여 보자. 100개의 요소가 있다고 생각하였을 때 26번째부터 75번째 사이에 있는 요소를 선택하면 $25 \sim 75$ 분할이 되며, 이와 같은 요소를 선택할 확률은 50%이다.

4. 빠른 정렬에 대한 분석

랜덤하게 피봇을 선택하는 빠른 정렬 알고리즘의 시간 복잡도를 분석하여 보자. 랜덤하게 선택하므로 확률적 분석이 필요하다. 기본 확률적 지식은 이 장 부록에 간단히 제시되어 있다. 분석을 쉽게 하기 위해 정렬해야 하는 모든 요소가 다르다고 가정하자. 즉, 중복된 요소가 없다고 가정한다.

크기가 n인 배열을 정렬하고자 한다. 확률적 분석을 위해 먼저 표본 공간과 확률 변수를 정의하여 보자. 사용할 표본 공간 Ω는 가능한 모든 랜덤 피봇의 선택이고, 확률 변수 $C(\sigma)$는 $\sigma \in \Omega$에 대해 빠른 정렬이 수행한 비교 수이다. 이 분석을 통해 증명하고자 하는 것은 $E[C] = O(n \log n)$이다.

z_i는 배열에서 i번째로 작은 요소라 하자. 즉, z_i는 정렬한 후 i번째 색인에 위치하게 되는 요소이다. $\sigma \in \Omega$가 주어졌을 때, $X_{ij}(\sigma)$는 빠른 정렬에서 z_i와 z_j를 서로 비교한 횟수이다. X_{ij} 값의 범위는 0 아니면 1이다. 왜 0 아니면 1일까? 빠른 정렬에서 요소 간 비교는 분할할 때 이루어진다. 분할 알고리즘은 선정된 피봇을 나머지 요소와 비교한다. 비교를 통해 피봇보다 작은 것과 큰 것으로 구간을 분할하며, 이 과정에서 피봇과 어떤 요소와 비교는 한번만 이루어진다. 그다음에는 피봇은 다시 비교에 참여하지 않는다.

따라서 알고리즘에서 이루어지는 전체 비교 횟수는 다음과 같이 정의할 수 있다.

$$C(\sigma) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}(\sigma)$$

그러면 구하고자 하는 기대값은 다음과 같다.

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} E[X_{ij}]$$

그런데 $E[X_{ij}] = 0 \times \Pr[X_{ij} = 0] + 1 \times \Pr[X_{ij} = 1] = \Pr[X_{ij} = 1]$이다. 따라서 다음이 성립한다.

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \Pr[X_{ij} = 1]$$

그러면 $\Pr[X_{ij}] = 1$ 확률만 구하면 모든 분석은 끝난다. $i < j$이고 피봇으로 z_k를 선택한 경우를 생각하여 보자. 총 다음과 같은 4가지 경우로 나눌 수 있다.

- 경우 1. $z_k < z_i$: 둘 다 같은 두 번째 재귀 호출로 전달되고, 이번 분할 과정에서는 비교가 이루어지지 않은 경우이다.
- 경우 2. $z_k > z_j$: 둘 다 같은 두 번째 재귀 호출로 전달되고, 이번 분할 과정에서는 비교가 이루어지지 않은 경우이다.
- 경우 3. $z_i < z_k < z_k$: 서로 다른 재귀 호출로 전달되고, 이 둘은 절대 비교되지 않는 경우이다.
- 경우 4. $z_i = z_k$ 또는 $z_j = z_k$: 서로 한번 비교가 이루어지고, 이 이후는 다시 비교가 이루어지지 않는다.

이 관찰을 통해 $\Pr[X_{ij}] = 1$은 z_{i+1}부터 z_{j-1} 전에 z_i 또는 z_j가 피봇으로 선택될 확률이다. 따라서 다음이 성립한다.

$$\Pr[X_{ij}] = \frac{2}{j - i + 1}$$

이를 이용하여 기대값을 구하면 다음과 같다.

$$E[C] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{1}{j-i+1} \leq 2n \sum_{k=2}^{n} \frac{1}{k} \leq 2n \ln n = O(n \log n)$$

참고로 $i = 1$이면 $\sum_{j=i+1}^{n} \frac{1}{j-i+1}$은 다음과 같다.

$$\sum_{j=2}^{n} \frac{1}{j} = \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \leq \ln n$$

따라서 랜덤한 피봇을 선택하는 빠른 정렬의 평균 시간 복잡도는 $O(n \log n)$이다.

5. 비교 기반 정렬 알고리즘의 한계

합병 정렬, 빠른 정렬처럼 요소를 서로 비교하여 정렬하는 알고리즘은 $O(n \log n)$보다 빠를 수 없다. 이것을 증명하기 위해 먼저 다음을 관찰하여 보자. 두 개의 배열 [1 2 3 4]와 [4 3 2 1]이 있을 때 한 번도 비교하지 않으면 이 두 배열을 구분할 수 없다. 구분할 수 없으면 알고리즘은 동일하게 동작하기 때문에 출력의 차이가 없으며, 둘 중 하나는 오답을 주기 때문에 0번 비교하면 정렬을 할 수 없다. 같은 이유로 비교를 1번만 하면 [1 2 3 4]와 [1 2 4 3]을 구분할 수 없으며, 비교를 2번만 하면 [1 2 3 4]와 [1 3 2 4]를 구분할 수 없다.

n개의 서로 다른 값으로 배열을 구성할 수 있는 경우의 수는 $n!$이다. k번 비교하였을 때 이들을 모두 구분할 수 있어야 올바르게 정렬할 수 있다. k번 비교하면 2^k개의 서로 다른 입력을 구분할 수 있다. 따라서 크기가 n인 배열을 정렬하기 위한 최소 비교 횟수 k는 $2^k \geq n!$를 만족해야 한다. 다음이 성립하기 때문에

$$2^k \geq n^n \geq n!$$

$k \geq n \log n$이 성립한다. 따라서 총 비교 횟수는 $O(n \log n)$이다.

6. 비교 기반이 아닌 정렬 알고리즘

비교 기반이 아닌 대표적인 정렬 알고리즘은 버킷 정렬이다. 버킷 정렬은 입력 데이터가 알려진 범위 내에 균일하게 분포되어 있는 경우 사용할 수 있는 정렬 알고리즘이다. 버킷 정렬은 일정한 수의 버킷을 준비한다. 보통 정렬할 값의 범위를 n등분하여 각 영역을 나타내는 버킷을 준비한다. 입력 배열을 스캔하면서 각 요소를 그 요소 범위에 맞는 버킷에 삽입한다. 이 비용은 $O(n)$이다. 이때 버킷에 삽입되는 요소의 수는 매우 작아 버킷 내에 요소를 정렬하는 비용은 $O(1)$이 되어야 한다. 그다음 각 버킷 내에 있는 요소를 정렬한다. 버킷을 이용하여 최종 출력을 만든다. 이 비용은 $O(n)$이다. 따라서 전체 비용은 $O(n)$이다.

다음 문제를 생각하여 보자.

산술적 3쌍 개수 찾기 문제

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

알고리즘 5.5 산술적 3쌍 개수 찾기 알고리즘

```
1:  function ARITHMETICTRIPLETS(A[], d)
2:      N := [false] × 201
3:      for all n ∈ A do
4:          N[n] := true
5:      max := A[LEN(A)]
6:      count := 0
7:      for i := 0 to LEN(A) − 2 do
8:          if A[i] + 2 × d ≤ max and N[A[i] + d] and N[A[i] + 2 × d] then
9:              ++count
10:     return count
```

예를 들어 $[0, 1, 4, 6, 7, 10]$ 배열과 $d = 3$이 주어졌다고 하자. 그러면 $0+3$과 $0+6$이 존재하는지 찾아야 한다. 이미 주어진 배열은 정렬되어 있다. 그런데 $A[i]$는 0부터 200 사이의 수이며, 그 범위가 제한적이므로 버킷 정렬 개념을 여기에 적용할 수 있다. 이를 이용하여 위 문제를 해결하는 알고리즘은 알고리즘 5.5와 같다. 배열 N에 입력 배열에 등장하는 모든 수 정보를 버킷 정렬 개념을 이용하여 설정하였기 때문에 선형 검색을 하면서 문제에서 요구하는 3쌍의 존재 여부를 검사할 수 있다.

7 확률 기초

확률을 설명할 때 가장 많이 사용하는 예 중 하나가 주사위이다. 빠른 정렬 알고리즘은 기존 장에서 살펴본 알고리즘과 달리 확률 알고리즘이며, 확률 알고리즘의 성능을 분석하기 위해서는 기초적인 확률 지식이 필요하다. 복습하는 차원에서 간단한 설명을 포함한다.

확률을 살펴볼 때 제일 먼저 생각해야 하는 것이 **표본 공간**(sample space)이다. 6면 주사위를 던졌을 때 나타나는 수의 표본 공간은 $\{1, 2, 3, 4, 5, 6\}$d이다. 표본 공간은 보통 Ω를 이용하여 나타낸다. 알고리즘을 분석할 때 사용하는 표본 공간은 유한하다. 이 절에서 확률을 설명할 때 사용하는 표본 공간은 6면 주사위 2개를 던졌을 때 나오는 결과이며, 다음과 같이 정의할 수 있다.

$$\Omega = \{(1,1), (1,2), \ldots, (6,6)\}$$

표본 공간에 있는 각 결과가 나타날 확률 $\Pr[i] \geq 0$이며, $\sum_{i \in \Omega} \Pr[i] = 1$이다. 6면 주사위 2개 던지기에서 한 결과가 나타날 확률 $\Pr[i] = \frac{1}{36}$이다.

우리는 확률을 이용하여 특정 **사건**(event)이 발생할 확률을 알고 싶은 경우가 많다. 사건은 표본 공간의 부분 집합이며, 특정 사건 S가 일어날 확률은 사건에 포함된 모든 결과가 나타날 확률을 더하면 된다. 즉, $\Pr[S] = \sum_{i \in S} \Pr[i]$이다. 예를 들어 6면 주사위 2개 던지기에서 결과의 합이 7인 사건의 확률은 $\frac{1}{6}$이다. 우리는 합이 7이 아니라 다양한 합을 살펴보고 싶을 수 있다.

이와 같은 사건을 분석하기 위해 사용하는 것이 **확률 변수**(random variable)이다. 확률 변수는 표본 공간의 한 결과를 실수값으로 매핑하여 주는 함수이다. 예를 들어 주사위 2개 던지기에서 결과의 합을 나타내는 확률 변수를 정의하여 사용할 수 있다. 이 변수를 X라 하면 $\Pr[X = 7] = \frac{1}{6}$이다. 즉, 사건의 확률을 조금 더 수학적으로 표현하고 분석하기 위해 우리는 확률 변수를 사용한다.

확률 변수가 평균적으로 가지게 되는 값을 기대값(expectation, expected value)라 하며, 다음과 같이 정의한다.

$$E[X] = \sum_{i \in \Omega} X(i) \Pr[i]$$

예를 들어 확률 변수 X가 6면 주사위 2개 던지기에서 결과의 합을 나타내면 $E[X] = 7$이다. $X(1,1) = 2$, $X(1,2) = 3$, ..., $X(6,6) = 36$을 모두 합한 후에 36으로 나누면 기대값을 얻을 수 있다.

기대값의 중요한 특성은 **기대값의 선형성**(linearity of expectation)이다. 기대값의 선형성이란 X_1, X_2, ..., X_n이 Ω에 정의된 확률 변수이면 다음이 성립한다는 것을 말한다.

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

위 식은 n개의 확률 변수가 서로 독립적이지 않아도 성립한다. 예를 들어 X_1이 6면 주사위 두 개 던지기에서 첫 번째 주사위 값이고, X_2가 6면 주사위 두 개 던지기에서 두 번째 주사위 값이면 $E[X_i] = 3.5$이다. 기대값의 선형성을 적용하면 $E[X_1 + X_2] = E[X_1] + E[X_2] = 7$이다.

확률을 분석할 때 조건부 확률을 살펴보아야 하는 경우가 많다. $A, B \in \Omega$에 대해 **조건부 확률** $\Pr[A|B]$는 다음과 같이 정의한다.

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

예를 들어 두 주사위를 던진 결과의 합이 7일 때 두 주사위 중 하나가 1일 확률은 조건부 확률로 나타낼 수 있다. 이 조건부 확률에서 A는 두 주사위 중 하나가 1인 사건이고, B는 두 주사위를 던진 결과의 합이 7인 사건으로

정의되며, 확률은 다음과 같다.

$$\Pr[A|B] = \frac{\Pr[A \cap B]}{\Pr[B]} = \frac{2/36}{6/36} = \frac{1}{3}$$

조건부 확률을 살펴볼 때 독립 사건 개념이 필요하다. 사건 $A, B \in \Omega$가 독립 사건이기 위한 필요충분조건은 다음과 같다.

$$\Pr[A \cap B] = \Pr[A] \times \Pr[B]$$

독립 개념은 확률 변수에도 적용되는 개념이다. 한 표본 공간에 정의된 확률 변수 X와 Y가 독립이기 위한 필요충분조건은 모든 x, y에 대해 $\Pr[X = x]$와 $\Pr[Y = y]$가 독립이어야 한다. 확률 변수 X와 Y가 독립이면 다음이 성립한다.

$$E[X \times Y] = E[X] \times E[Y]$$

기대값의 선형성은 확률 변수가 독립이 아니어도 성립하지만 위 식은 독립인 경우에만 성립한다.

예를 들어 2비트 표본공간 $\Omega = \{00, 10, 01, 11\}$을 생각하여 보자. 이 표본공간에서 다음과 같은 3개의 확률 변수가 있을 때,

- X_1: 첫 번째 비트 값
- X_2: 두 번째 비트 값
- X_3: 첫 번째 비트 값과 두 번째 비트 값을 XOR한 값
- X_4: 첫 번째 비트 값과 첫 번째와 두 번째 비트 값의 XOR한 값을 곱한 값

X_1과 X_3은 독립 확률 변수이다. 그 이유는 다음이 성립한다.

$$E[X_1 \times X_3] = E[X_1] \times E[X_3]$$

하지만 $E[X_2 \times X_4] \neq E[X_2] \times E[X_4]$이므로 X_2와 X_4는 독립 확률 변수가 아니다.

퀴즈

1. 빠른 정렬과 관련된 다음 설명 중 틀린 것은?
   ① 랜덤하게 피봇을 선택하면 50%의 확률로 25~75 분할하는 피봇을 선택하게 된다.
   ② 평균 두 번 중 한 번은 25~75 분할하는 피봇 선택할 수 있기 때문에 재귀 트리의 높이는 $\log_{3/4} n$에 비례한다.
   ③ 두 개의 요소가 빠른 정렬 과정 중에 최대 2번까지 서로 비교가 이루어질 수 있다.
   ④ 빠른 정렬은 정렬 과정에서 추가 메모리 공간을 사용하지 않는다. 즉, 입력 데이터 외에 추가로 필요한 공간은 $O(1)$이다.
2. 입력 배열이 다음과 같을 때 빠른 정렬을 적용하기 위해 4번째 요소를 피봇으로 선택하였다.

   $$[12, 4, 7, 5, 11, 9, 2, 6]$$

   그 피봇을 중심으로 알고리즘 5.2를 이용하여 분할한 후에 모습은 다음 중 어떤 것인가?
   ① [2, 4, 5, 11, 9, 12, 6, 7]
   ② [4, 2, 5, 6, 9, 11, 7, 12]
   ③ [2, 4, 5, 12, 11, 9, 7, 6]
   ④ [4, 2, 5, 11, 9, 12, 6, 7]

3. 입력 배열이 다음과 같을 때 빠른 정렬을 적용하기 위해 4번째 요소를 피봇으로 선택하였다.

$$[12, 4, 7, 5, 11, 9, 2, 6]$$

그 피봇을 중심으로 알고리즘 5.3를 이용하여 분할한 후에 모습은 다음 중 어떤 것인가?

① [2, 4, 5, 11, 9, 12, 6, 7]
② [4, 2, 5, 6, 9, 11, 7, 12]
③ [2, 4, 5, 12, 11, 9, 7, 6]
④ [4, 2, 5, 11, 9, 12, 6, 7]

연습문제

1. 빠른 정렬 알고리즘은 랜덤으로 피봇을 선택하기 때문에 확률적 알고리즘이다. 피봇을 랜덤으로 선택하는 것과 항상 구간의 맨 왼쪽값을 피봇으로 선택하는 것의 차이를 설명하시오.

2. 정수 n개로 구성된 배열이 주어진다. 이 배열에는 항상 $\lfloor n/2 \rfloor$보다 많은 요소가 반드시 하나 존재한다. 이 요소를 찾는 확률적 알고리즘을 제시하라. 제시한 알고리즘의 정확성과 성능을 분석하시오.

3. C++ 표준 라이브러리는 IntroSort라는 알고리즘을 사용한다. 이 알고리즘은 빠른정렬, 삽입정렬, 힙정렬을 혼합하여 사용하는 알고리즘이다. IntroSort는 다음과 같이 동작한다.

   - 정렬할 구간의 크기가 16보다 작으면 삽입정렬을 이용한다.
   - 진행 중인 빠른 정렬의 재귀 호출 깊이가 정해진 기준 $(2 \log n)$을 초과하면 힙정렬을 이용한다.
   - 위 두 가지 경우가 아니면 빠른 정렬을 수행한다.

   이것을 알고리즘으로 표현하면 다음과 같다.

```
1:  procedure INTROSORT(A[], lo, hi, d)
2:      if hi − lo + 1 < 16 then
3:          INSERTIONSORT(A, lo, hi)
4:      else if d == 0 then
5:          HEAPSORT(A, lo, hi)
6:      else
7:          pivotLoc := PARTITION(A, lo, hi)
8:          INTROSORT(A, lo, pivotLoc − 1, d − 1)
9:          INTROSORT(A, pivotLoc + 1, hi, d − 1)
10: procedure INTROSORT(A[])
11:     d := 2 × log n
12:     INTROSORT(A, 1, n, d)
```

   IntroSort를 구현하라.

4. 주어진 일련의 정수를 다음 조건을 만족하도록 재배치하라.

$$A[0] < A[1] > A[2] < A[3] > \cdots$$

   주어진 배열은 모두 이 조건을 만족하도록 재배치할 수 있다.