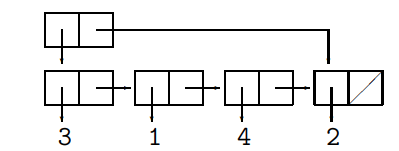
**큐[Queues]**

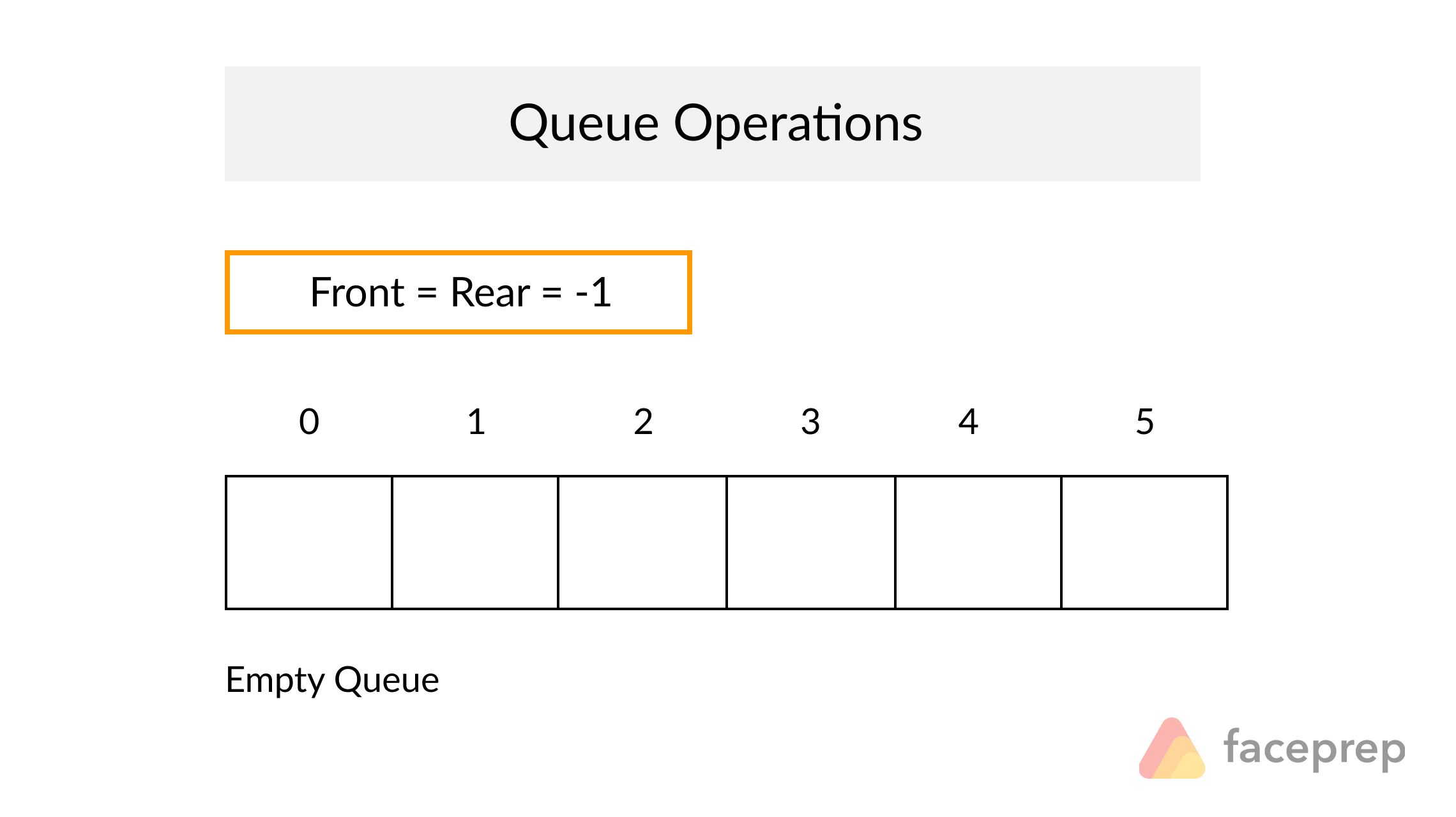
큐는 선입선출(FIFO: First-In-First-Out) 전략을 모델링하는 데 사용되는 데이터 구조입니다. 개념적으로 큐의 끝에 추가하고 앞에서 요소를 제거합니다.

<Graphical Representation: q=[3, 1, 4, 2]>



큐는 리스트나 스택과 비슷하게 그래픽으로 표현할 수 있습니다. 하지만 추가적으로 두 개의 셀이 있어, 첫 번째 셀은 큐의 첫 번째 요소를 가리키고, 두 번째 셀은 마지막 요소를 가리킵니다. 예를 들어, 초기 상태가 빈 큐에 요소 [3, 1, 4, 2]를 삽입하면 다음과 같이 표현됩니다, 이러한 구조는 큐의 첫 번째 요소를 가져오거나 큐의 맨 끝에 요소를 추가하는 작업을 효율적으로 수행할 수 있게 합니다.

<Graphical Representation: gif>



**front**는 **큐의 삭제**가 발생하는 지점, **rear**는 **큐의 삽입**이 발생하는 지점

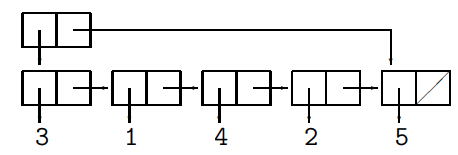
**추상 데이터 형식[Abstract Data Type “Queue”]**

*생성자 [constructors]*

**EmptyQueue:** 빈 큐

**push(element, queue):** 요소와 큐를 받아서 요소가 원래 큐의 끝에 추가된 큐를 반환합니다.

<Graphical Representation:push(5,q)>



*선택자 [selectors]*

**top(queue):** 큐의 맨 위 요소(예제에서는 3)를 반환합니다.

**pop(queue):** 맨 위 요소를 제거한 큐를 반환합니다.

#선택자는 비어 있지 않은 큐에 대해서만 작동하므로, 큐가 비어 있는지 여부를 반환하는 조건[condition]이 필요합니다.

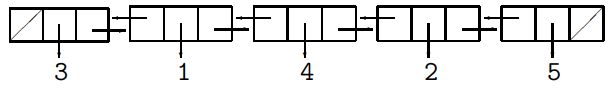
**isEmpty(queue)**

#큐에 데이터를 추가하는 작업을 **enqueue**, 데이터를 꺼내는 작업은 **dequeue**

**이중 연결 리스트[Doubly Linked List]**

양방향 연결 리스트는 각 노드가 이전 노드와 다음 노드를 가리키는 포인터를 가지고 있는 연결 리스트입니다. 연결 리스트와 양방향 연결 리스트는 [3, 1, 4, 2, 5]와 같이 동일하게 보일 수 있습니다. 그러나 양방향 연결 리스트는 다음 요소뿐만 아니라 이전 요소에도 쉽게 접근할 수 있습니다.

<Graphical Representation: q=[3, 1, 4, 2]>



비어 있지 않은 양방향 연결 리스트는 세 개의 셀로 표현할 수 있습니다. 첫 번째 셀은 또 다른 세 셀 또는 빈 리스트를 가리키는 포인터를 포함하고, 두 번째 셀은 리스트 요소를 가리키는 포인터를 포함하며, 세 번째 셀은 또 다른 세 셀 또는 빈 리스트를 가리키는 포인터를 포함합니다.

이러한 구조를 통해 양방향으로 이동하고, 노드를 삽입하거나 삭제하는 작업을 효율적으로 수행할 수 있습니다.

**추상 데이터 형식[Abstract Data Type “Doubly Linked List”]**

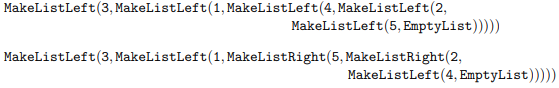
*생성자 [constructors]*

**EmptyList:** 빈 리스트

**MakeListLeft(element, list):** 요소와 양방향 연결 리스트를 받아서, 요소가 원래 양방향 연결 리스트의 왼쪽에 추가된 새로운 양방향 연결 리스트를 반환합니다.

**MakeListRight(element, list):** 요소와 양방향 연결 리스트를 받아서, 요소가 원래 양방향 연결 리스트의 오른쪽에 추가된 새로운 양방향 연결 리스트를 반환합니다.

< [3, 1, 4, 2, 5]를 생성하는 예시-양방향 연결 리스트는 *여러 가지 방법*으로 구성될 수 있습니다.>



첫 번째 예시는 모든 요소를 왼쪽에 추가하는 방식으로 생성하고, 두 번째 예시는 왼쪽과 오른쪽을 혼합하여 생성합니다.

*선택자 [selectors]*

**firstLeft(list):** 리스트의 왼쪽 첫 번째 요소를 반환합니다.

**restLeft(list):** 리스트의 왼쪽 첫 번째 요소를 제외한 나머지 리스트를 반환합니다.

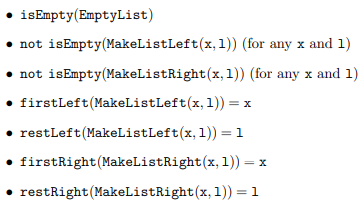
**firstRight(list):** 리스트의 오른쪽 첫 번째 요소를 반환합니다.

**restRight(list):** 리스트의 오른쪽 첫 번째 요소를 제외한 나머지 리스트를 반환합니다.

##선택자는 비어 있지 않은 리스트에 대해서만 작동하므로, 리스트가 비어 있는지 여부를 반환하는 조건이 필요합니다

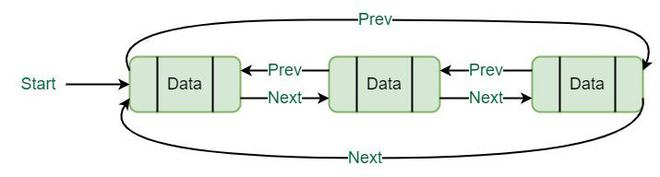
**isEmpty(list)**

이로 인해 다음과 같은 자동으로 참인 관계가 생깁니다.

****

**순환 양방향 연결 리스트[Circular Doubly Linked List]**

표준 양방향 연결 리스트의 간단한 확장으로, 왼쪽 끝 요소가 오른쪽 끝 요소를 가리키고, 그 반대의 경우도 마찬가지인 순환 양방향 연결 리스트를 정의할 수 있습니다. 리스트가 원형으로 연결되어 있어 어느 노드에서 시작하든 리스트를 순환하며 모든 노드를 방문할 수 있습니다.



**추상 데이터 형식의 장점[Advantage of Abstract Data Types]**

추상 데이터 형식의 연산을 사용하여 프로그램을 작성하는 훈련을 받으면 기본 데이터 구조에 대한 액세스를 찾기 위해 모든 프로그램을 검색하는 대신 연산을 재구현하여 프로그램을 더 쉽게 수정할 수 있습니다. *추상 데이터 형식*이라는 것은 데이터를 어떻게 저장하고 관리하는지에 대한 *일종의 틀*이라고 생각하면 됩니다. 데이터를 어떻게 저장하고 관리할지에 대한 *일반적인 규칙*을 정의한 것입니다

*더 쉽게 설명하자면*, 레고 블록과 같습니다. 레고 블록을 이용하여 다양한 모형을 만들 수 있지만, 각 블록의 내부 구조까지 알 필요는 없습니다.

**탐색[Searching]**

컴퓨팅에서 중요하고 반복되는 문제 중 하나는 정보를 찾는 것입니다. 더 간단히 말해서, 이 문제를 탐색(searching)이라고 합니다.

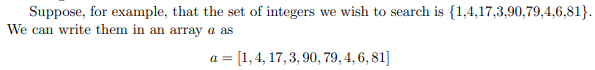
**검색을 위한 요구 사항[Requirements for searching]**

정보를 검색하려면 먼저 정보를 적절히 **표현(또는 인코딩)** 해야 합니다. 이 과정에서 **자료 구조(Data Structure)** 가 중요한 역할을 합니다. 컴퓨터에서 모든 데이터는 궁극적으로 0과 1로 이루어진 **이진수(비트)** 로 표현되지만, 이는 너무 저수준(low-level)이어서 대부분의 목적에 적합하지 않습니다. 우리가 개발하고 연구해야 할 자료 구조는 인간이 이해하기 쉬운 방식에 더 가깝거나, 단순한 비트의 나열보다 구조화된 것이어야 합니다. 적절한 표현 방법을 선택한 후에는, 표현된 정보를 **처리**해야 합니다. 이 과정에서 **알고리즘**이 필요합니다.

ex#두가지를 고려할수있다.

*가장 단순하고 명확한 표현 방법.*

*해당 표현을 처리하기 위한 두 가지 잠재적 알고리즘.*

****

이 배열에서 숫자 17이 위치한 인덱스를 물어보면, 답은 2입니다. 이는 배열에서 17이 위치한 자리(index)를 나타냅니다. 반면, 숫자 91이 배열에 있는지 묻는다면, "없다"고 답해야 합니다. 배열 인덱스는 0부터 시작하므로, 음수 값을 "없음"을 나타내는 값으로 사용할 수 있습니다. 여기서는 -1.

**검색 문제의 명세[Specification of the search problem]**

이제 우리는 그 데이터 구조를 사용하여 검색 문제의 명세를 공식화할 수 있습니다.



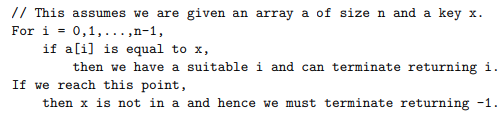
a = [17, 13, 17]에서 x = 17인 경우, j는 0이나 2가 될 수 있습니다. 명세상 어느 값이든 반환해도 문제가 없습니다. 따라서 명세가 모호(ambiguity)하다는 점을 알 수 있습니다. 실제로, 하나의 명세를 만족하는 서로 다른 알고리즘들이 존재할 수 있습니다. 예를 들어, 하나의 알고리즘은 x가 가장 먼저 나타나는 위치를 반환할 수 있고, 또 다른 알고리즘은 마지막 위치를 반환할 수 있습니다.

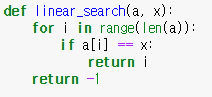
명세의 모호성은 문제 해결 과정에서 다양한 알고리즘 설계를 허용하는 유연성을 제공합니다.

알고리즘 구현 시, 문제 명세의 모호성을 인식하고 원하는 동작을 명확히 정의해야 합니다.

**간단한 알고리즘: 선형 탐색/검색[A simple algorithm: Linear Search]**

가장 간단한 알고리즘은 **선형 검색**입니다. 이를 의사 코드(pseudocode)로 표현하면 다음과 같습니다.



 최악의 경우, 배열의 크기 n만큼 비교가 필요합니다.

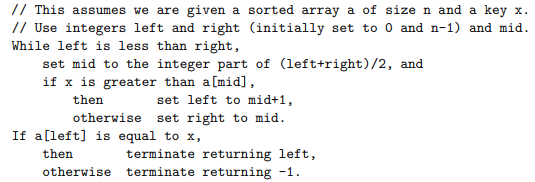
평균적으로 n/2번의 비교가 이루어집니다.

더 효율적인 알고리즘이 가능하도록 컬렉션을 구성하려고 노력해야 하며, 효율성 측면에서 더 많은 것을 요구할수록 컬렉션을 나타내는 데이터 구조가 더 복잡해지는 경향이 있다.

**더 효율적인 알고리즘: 이진 검색/탐색[A more efficient algorithm:** **Binary Search]**

**이진 검색**은 선형 검색보다 훨씬 더 효율적입니다. 이를 위해 배열은 **정렬된 상태**여야 합니다. 예를 들어, 배열 [1, 4, 17, 3, 90, 79, 4, 6, 81]을 [1, 3, 4, 4, 6, 17, 79, 81, 90]과 같이 오름차순으로 정렬합니다.

<pseudocode: Binary Search >

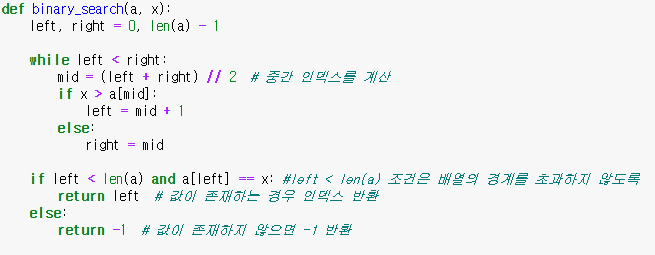


배열의 크기를 매번 절반으로 줄이므로, 시간 복잡도는 O(log n)입니다.

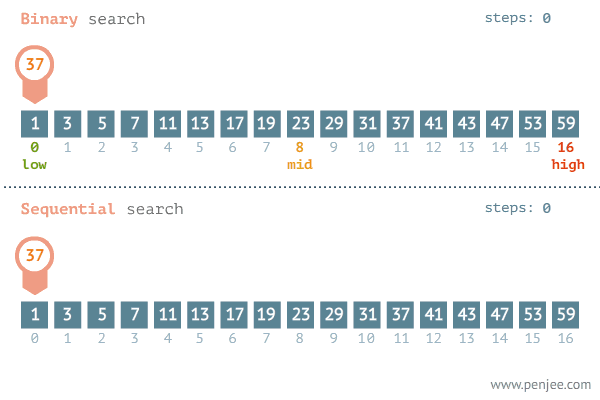
예를 들어, 배열 크기가 1,000,000일 때, 이진 검색의 최악의 경우 비교 횟수는 약 20회입니다. 선형 검색의 1,000,000회에 비해 훨씬 효율적입니다.

이진 검색을 사용하려면 배열을 먼저 정렬해야 합니다. 배열을 정렬하는 데 드는 시간 복잡도는 일반적으로 O(n log n)입니다.  
데이터에 대한 검색이 한 번만 이루어진다면, 정렬 비용 때문에 선형 검색이 더 유리할 수 있습니다. 반대로 여러 번 검색해야 한다면, 정렬 후 이진 검색을 사용하는 것이 효율적입니다.  
자료 구조와 알고리즘의 선택은 문제의 특성과 요구사항에 따라 달라집니다.

<Representation by python>



<Graphical Representation: gif>



**효율성과 복잡도[Efficiency and Complexity]**

알고리즘을 개발할 때는 그 효율성을 고려하는 것이 중요합니다. 이를 통해 특정 상황에 가장 적합한 알고리즘을 선택할 수 있습니다.

**시간 복잡도 vs 공간 복잡도[Time versus space complexity]**

**시간 복잡도(Time Complexity)**

소프트웨어를 개발할 때, 특히 중요한 작업은 알고리즘이 주어진 작업을 얼마나 빨리 완료할 수 있는지를 판단하는 것입니다. 예를 들어, 항공권 예약 시스템을 프로그래밍하는 경우, 여행사나 고객이 거래가 완료되기를 30분 동안 기다리는 것은 허용될 수 없습니다. 문제 크기에 비례하여 실행 시간이 합리적이어야 하며, 일반적으로 더 빠른 실행이 더 나은 것으로 간주됩니다. **시간 복잡도**는 **데이터 구조의 크기에 따라 실행 시간이 어떻게 달라지는지**를 나타내는 지표입니다.

**공간 복잡도(Space Complexity)**

또 다른 중요한 요소는 주어진 작업을 수행하기 위해 필요한 메모리 양입니다. 현대의 컴퓨터는 과거에 비해 메모리가 풍부해졌기 때문에 메모리 사용량은 상대적으로 덜 중요한 문제가 되었습니다. 그러나 여전히 **공간 복잡도**는 **데이터 구조의 크기에 따라 메모리 요구량이 어떻게 달라지는지**를 설명합니다.

**시간과 공간의 상호 교환(Trade-off)**

특정 작업에 대해, 시간과 공간을 맞바꾸는 알고리즘이 존재하는 경우가 많습니다. 알고리즘의 실행 속도를 높이기 위해 더 많은 메모리를 사용하거나, 반대로 메모리 사용량을 줄이기 위해 실행 속도를 희생해야 할 수 있습니다. 이러한 현상을 **시간과 공간의 트레이드오프**라고 합니다. 즉 어느 한쪽을 선택해야 하는 경우가 생길때 시간과 공간을 비교하면서 최적의 타협점을 찾아야하고, 알고리즘 설계자는 애플리케이션의 요구사항에 따라 시간과 공간의 균형을 잘 맞춰야 합니다.

**최악의 경우 복잡도 vs 평균 복잡도[Worst versus average complexity]**

알고리즘의 효율성을 평가할 때, **평균적인 경우의 성능**이 더 중요한지, 아니면 **최악의 경우에도 일정 성능을 보장**하는 것이 중요한지 결정해야 합니다.

많은 애플리케이션의 경우, 평균적인 경우가 더 중요합니다.그러나 항공기의 특정 구역을 추적하는 것과 같은 시간에 민감한 문제의 경우, 최악의 경우에 소프트웨어가 너무 오래 걸리는 것은 완전히 받아들일 수 없습니다.

알고리즘/프로그램은 종종 평균적인 경우의 효율성과 최악의 경우의 효율성을 트레이드오프(**Trade-off)**합니다.

**성능의 구체적인 측정 방법[Concrete measures for performance]**

**단순 실행 시간 측정의 문제**

알고리즘의 실행 시간을 측정하기 위해 단순히 알고리즘을 구현하고 실행 시간을 측정하는 방법을 사용할 수도 있습니다. 그러나 이러한 **경험적 측정 방법**에는 여러 한계가 있습니다.

1. 여러 알고리즘을 비교하려면 모두 프로그래밍해야 하므로 많은 시간을 낭비할 수 있습니다.
2. 실행 시간은 프로그램이 실행되는 **하드웨어 및 컴파일러**에 따라 영향을 받을 수 있습니다.
3. 테스트 데이터가 애플리케이션에 적합한지 확인해야 하며, 이는 특히 큰 애플리케이션의 경우 현실적으로 어렵습니다.

**알고리즘 중심의 복잡도 분석[**구현의 효율성보다는 알고리즘의 효율성을 측정하는 것이 좋습니다.]

알고리즘의 효율성을 평가할 때는 특정 프로그래밍 언어나 하드웨어 환경에 의존하지 않고, **알고리즘 자체의 논리적 구조**에 초점을 맞추는 것이 더 유용합니다. 이를 통해 알고리즘의 본질적인 성능을 평가할 수 있습니다. 이를 위해서는 알고리즘을 특정 프로그래밍 언어나 하드웨어에 의존하지 않는 **의사 코드(pseudocode)** 형태로 표현합니다.

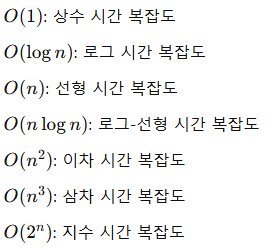
알고리즘의 시간 복잡도를 결정하기 위해 우리는 각 연산이 몇 번 발생하는지 세야 하며, 이는 일반적으로 **문제 크기**에 따라 달라집니다. 문제의 크기는 일반적으로 조작되는 항목의 수로 표현됩니다. 예를 들어:

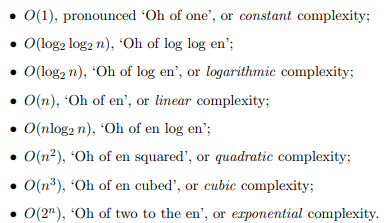
* 검색 알고리즘의 경우, **검색 대상 항목의 수**가 문제의 크기가 됩니다.
* 정렬 알고리즘의 경우, **정렬할 항목의 수**가 문제의 크기입니다.

**복잡도 클래스와 Big-O 표기법[Big-O notation for complexity class]**

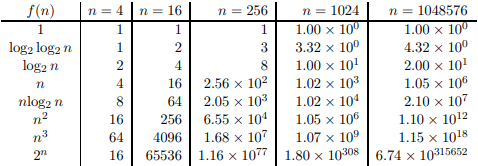
대부분의 경우, 우리는 시간 복잡도를 나타내는 정확한 함수 C(n) 자체보다는 복잡도 **클래스(complexity class)**에만 관심을 둡니다. 복잡도 클래스는 **상수 오버헤드** 및 **작은 상수 요소**를 무시하고, 문제 크기에 따른 주요 성장 속도만 나타냅니다. 복잡도 클래스는 루프의 개수와 루프의 내용이 실행되는 횟수에 의해 결정됩니다.

상수 계수와 다른 중요하지 않은 세부 사항이 무시된다는 사실을 표현하는 표준 표기법이 있는데, 이를 **빅-O 표기법[Big-O notation]**이라고 합니다.

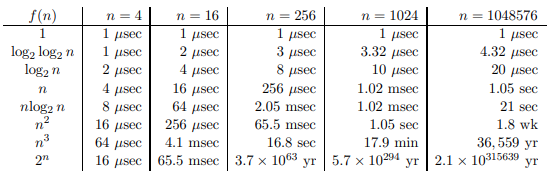
**대표적인 복잡도 클래스**



문제 크기에 대해 얼마나 많은 연산이 필요한지를 다음 표에 나타낼 수 있습니다.

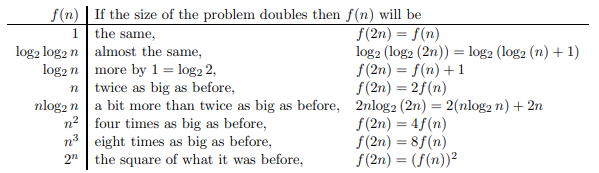


이 중 일부 숫자는 너무 커서 그 시간 범위가 얼마나 긴지 상상하기 어렵습니다. 따라서 다음 표는 1MIP(million instructions per second: 1백만 개의 명령어/초)의 속도로 작동하는 컴퓨터를 가정하여 명령어 수 대신 시간 범위를 제공합니다.



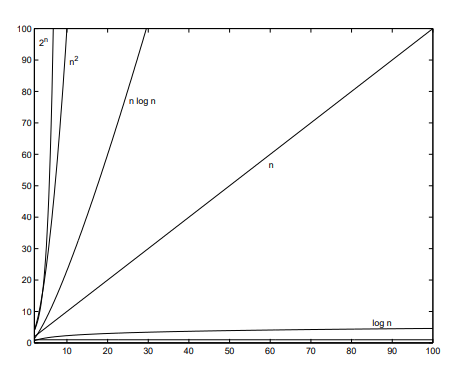
지수적 복잡도 O(2^n)을 가진 알고리즘의 경우, 적당한 크기의 문제라도 실행 시간이 우주의 나이(약 1.4 × 10^10년)보다 길어질 수 있으며, 현재의 컴퓨터는 몇 년 이상 중단 없이 실행되는 경우가 거의 없습니다. 이것이 복잡도 클래스가 매우 중요한 이유입니다.

또 다른 유용한 방법은 문제 크기가 두 배가 될 때 컴퓨팅 시간이 어떻게 변하는지 고려하는 것입니다.



전체 크기의 절반 또는 4분의 1 또는 8분의 1인 문제에 대해 프로그램을 테스트하고, 전체 크기의 문제를 완료하는 데 얼마나 오래 기다려야 할지 대략적인 아이디어를 얻을 수 있습니다. 또한, 그 추정치는 성장 클래스를 계산할 때 무시된 상수 계수나 특정 컴퓨터의 속도에 영향을 받지 않을 것입니다.

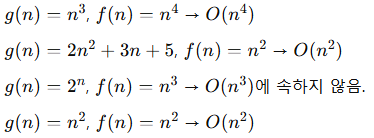
<그래프를 통한 복잡도 클래스의 시각화 >

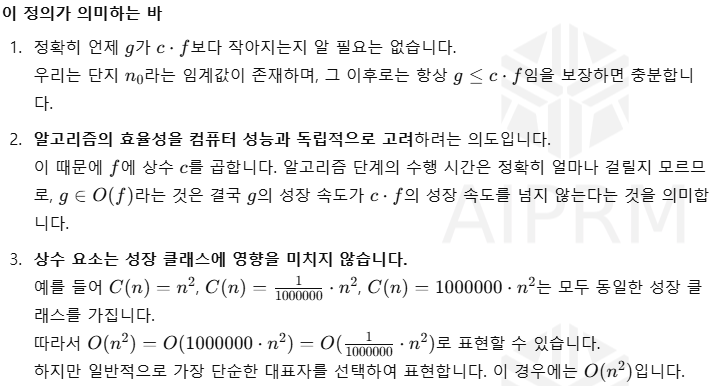


**복잡도 클래스의 공식적인 정의[Formal definition of complexity classes]**



ex)





**복잡도 클래스 간 관계**

복잡도 클래스는 성장 속도에 따라 다음과 같은 포함 관계를 가집니다.



**주요 성분만 고려하기**

여러 성장 클래스를 포함하는 함수의 경우, 가장 큰 성장 클래스를 기준으로 복잡도 클래스를 정합니다. 이를 통해 더 간단하게 표현할 수 있습니다.



이 함수의 가장 큰 성장 클래스는 4n^2입니다

따라서, 상수 요소를 무시하고 이 함수는O(n^2) 에 속합니다.

**알고리즘의 복잡도 클래스 예시**

알고리즘이 어떤 클래스 O(f)에 '속한다'고 말할 때, 그 알고리즘이 f보다 빠르게 성장하지 않는다는 의미입니다.

# '선형 검색'(정렬되지 않은 데이터 항목 집합에서 검색하는 경우)이 선형 복잡도, 즉 O(n) 성장 클래스에 속한다는 것을 보았습니다. 이것은 평균적인 경우와 최악의 경우 모두에 해당합니다. 필요한 연산은 검색하려는 항목과 데이터 집합에 나타나는 모든 항목의 비교입니다. 최악의 경우에는 올바른 항목을 찾을 때까지 모든 n개의 항목을 확인해야 하므로 n번의 비교를 수행합니다. 그러나 평균적으로는 올바른 항목에 도달할 때까지 n/2개의 항목만 확인하면 되므로 n/2개의 연산이 남습니다. C(n) = n과 C(n) = n/2인 두 함수는 모두 O(n)이라는 동일한 복잡도 클래스에 속합니다. 그러나 알고리즘이 O(n^2)에 속한다고 말하는 것도 똑같이 정확할 것입니다. 왜냐하면 그 클래스는 O(n)을 모두 포함하기 때문입니다.

@@

**정확한 복잡도 분석의 어려움**

복잡도 분석은 효율성을 평가할 때 반복적으로 사용되며, 복잡도를 단순히 **성장 클래스**로만 평가하면 분석이 훨씬 간단해집니다. 그러나 정확한 복잡도가 무엇인지 확신하기 어려운 경우가 있는데 (유명한 NP = P 문제의 경우와 같이), 이 경우 알고리즘이 '최대' 이차적이라고 말할 수 있습니다(위 예시에대하여, 선형탐색).

**요약**

* **Big-O 클래스 정의**는 함수의 성장 속도를 기준으로 g(n)이 결국 c⋅f(n)을 초과하지 않는다는 것을 보장합니다.
* **상수 요소와 작은 성분은 무시**하여, 함수의 주된 성장 성분만 고려합니다.
* 복잡도 클래스는 알고리즘의 성능을 간단하고 명확하게 나타내며, 서로 다른 알고리즘을 비교할 때 매우 유용합니다.
* 정확한 복잡도 분석은 쉬운 작업은 아니지만, 문제를 효율적으로 해결하는 데 필수적인 도구입니다.