**<알고리즘>**

1. 알고리즘 이란?

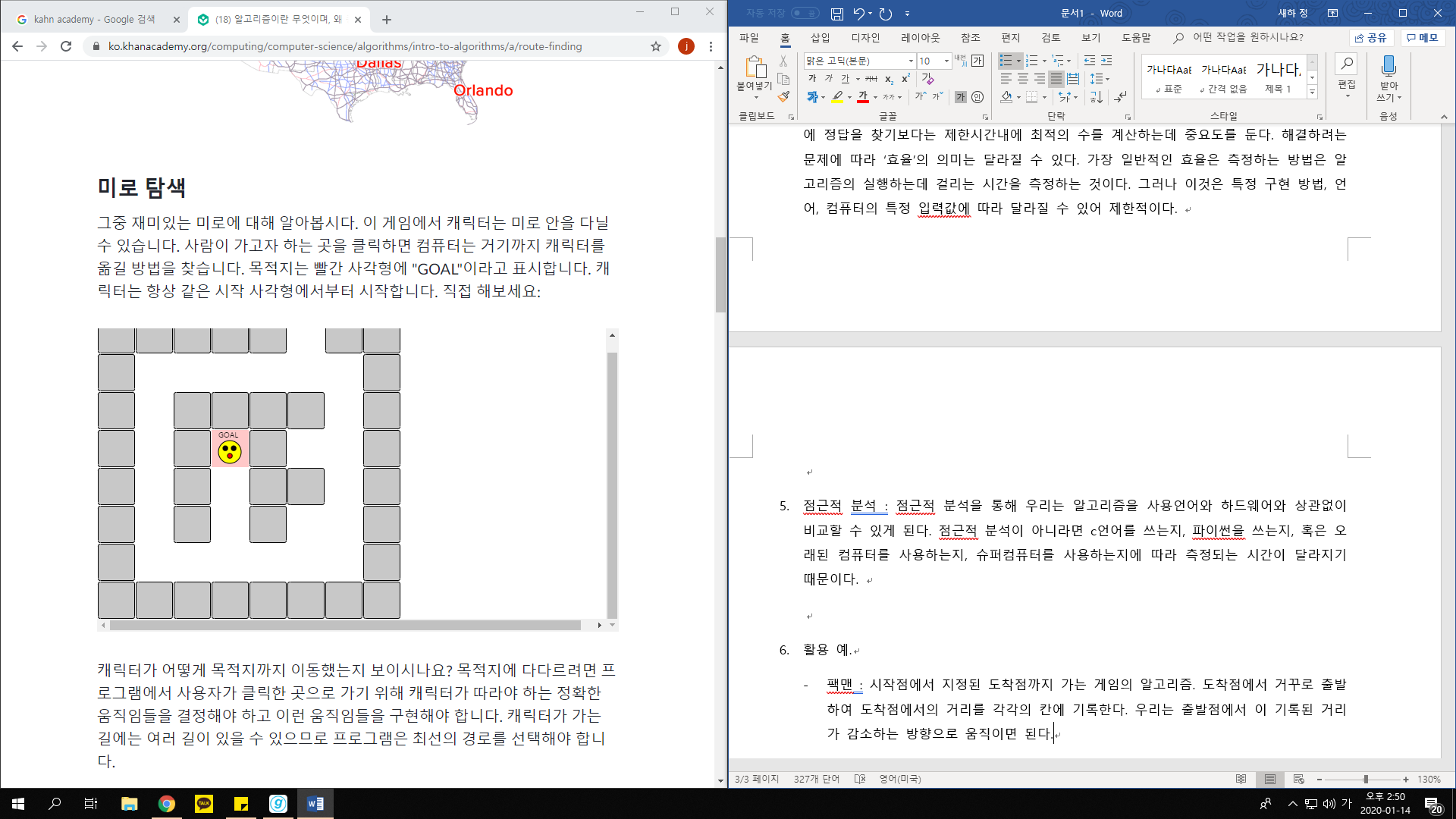
* 컴퓨터 프로그램이 어떤 문제를 해결하기 위해 필요한 명령어(절차)들의 집합
* 알고리즘이 컴퓨터 과학을 과학으로 만든다

1. 알고리즘 예

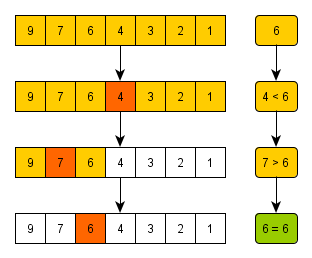
* 인터넷 간에 실시간 영상 전송( youtube, hangout) : 오디오, 비디오 압축 알고리즘
* 경로찾기 ( 구글맵스) : 경로찾기 알고리즘
* 3D 모델을 조명을 반영해 색칠하기 ( 픽사 애니메이션) : 렌더링 알고리즘
* NASA의 ISS의 태양광 패널 스케쥴링 : 최적화& 스케쥴 알고리즘
* 요리도 일련의 과정의 집합이므로 알고리즘으로 볼수 있다. Ex) 떡볶이 만들기
* 알파고나 체커게임과 같은 인공지능 알고리즘은 플레이어가 둘 수 있는 모든 경우의 수를 탐색합니다. 이런 경우의 수들을 큰 트리형태로 계산한뒤 미니맥스 알고리즘을 이용하여 빠른속도로 다음의 수를 계산할 수 있습니다. 바둑이나 체커와 비슷한 방식의 게임에는 이러한 알고리즘을 변형하여 적용할 수 있습니다.

1. 컴퓨터과학에서만 알고리즘이 쓰이는 것은 아니다. 생명공학에는 분자의 구조를 새로 디자인 하는 새 알고리즘을 개발하고, 물리학에서는 기후와 날씨 유형을 시뮬레이션하는 알고리즘을 만든다. **알고리즘은 어떠한 목적 혹은 문제를 해결하기 위한 일련의 절차를 정정의하는 것 같기 때문이다**. 이러한 알고리즘을 미리 정의해두면 우리는 갈래길에서, 수많은 결정점들 중 가장 좋은 결정점을 지능적으로 선택할 수 있다.
2. 좋은 알고리즘 : 문제를 해결할 수 있으며 효율적(빠르고,적은 메모리,적은 비용, 높은 정확률 등)인 알고리즘을 좋은 알고리즘 이라고 한다. 문제에 따라서 ‘효율적’이라는 의미는 달라질 수 있다. 우리는 보통 항상 정답을 도출하는 알고리즘은 좋은 알고리즘이라고 생각하지만, 정답을 도출해내기 위해서는 시간이 많이 걸리는 경우가 많다. 예를 들어 알파고 알고리즘에서는, 다음에 바둑을 두는 곳을 결정하는데 걸리는 시간이 제한적이기 때문에 정답을 찾기보다는 제한시간내에 최적의 수를 계산하는데 중요도를 둔다. 해결하려는 문제에 따라 ‘효율’의 의미는 달라질 수 있다. 가장 일반적인 효율은 측정하는 방법은 알고리즘의 실행하는데 걸리는 시간을 측정하는 것이다. 그러나 이것은 특정 구현 방법, 언어, 컴퓨터의 특정 입력값에 따라 달라질 수 있어 제한적이다.
3. 점근적 분석 : 점근적 분석을 통해 우리는 알고리즘을 사용언어와 하드웨어와 상관없이 비교할 수 있게 된다. 점근적 분석이 아니라면 c언어를 쓰는지, 파이썬을 쓰는지, 혹은 오래된 컴퓨터를 사용하는지, 슈퍼컴퓨터를 사용하는지에 따라 측정되는 시간이 달라지기 때문이다.
4. 활용 예.

* 팩맨 : 시작점에서 지정된 도착점까지 가는 게임의 알고리즘. 도착점에서 거꾸로 출발하여 도착점에서의 거리를 각각의 칸에 기록한다. 우리는 출발점에서 이 기록된 거리가 감소하는 방향으로 움직이면 된다.

1. 도착점을 0점으로 표시한다.
2. 도착점에서 1칸으로 움직일수 있는 곳을 1점으로 표시한다.
3. 1점으로 기록된 모든 칸에서 1칸으로 움직일 수 있으며, 어떤 점수도 기록되지 않은 칸을 2점으로 표시한다.
4. 이 과정을 발칸에 점수가 기록될때까지 반복한다.
5. 출발점의 플레이어는 각 칸에 기록된 점수가 감소하는 방향으로 계속 움직인다면 도착점에 도착할 수 있다.
6. 이진검색

-검색의 범위를 절반씩 줄여 나가는 것. 데이터가 정렬되어 있는 경우에만 이용가능

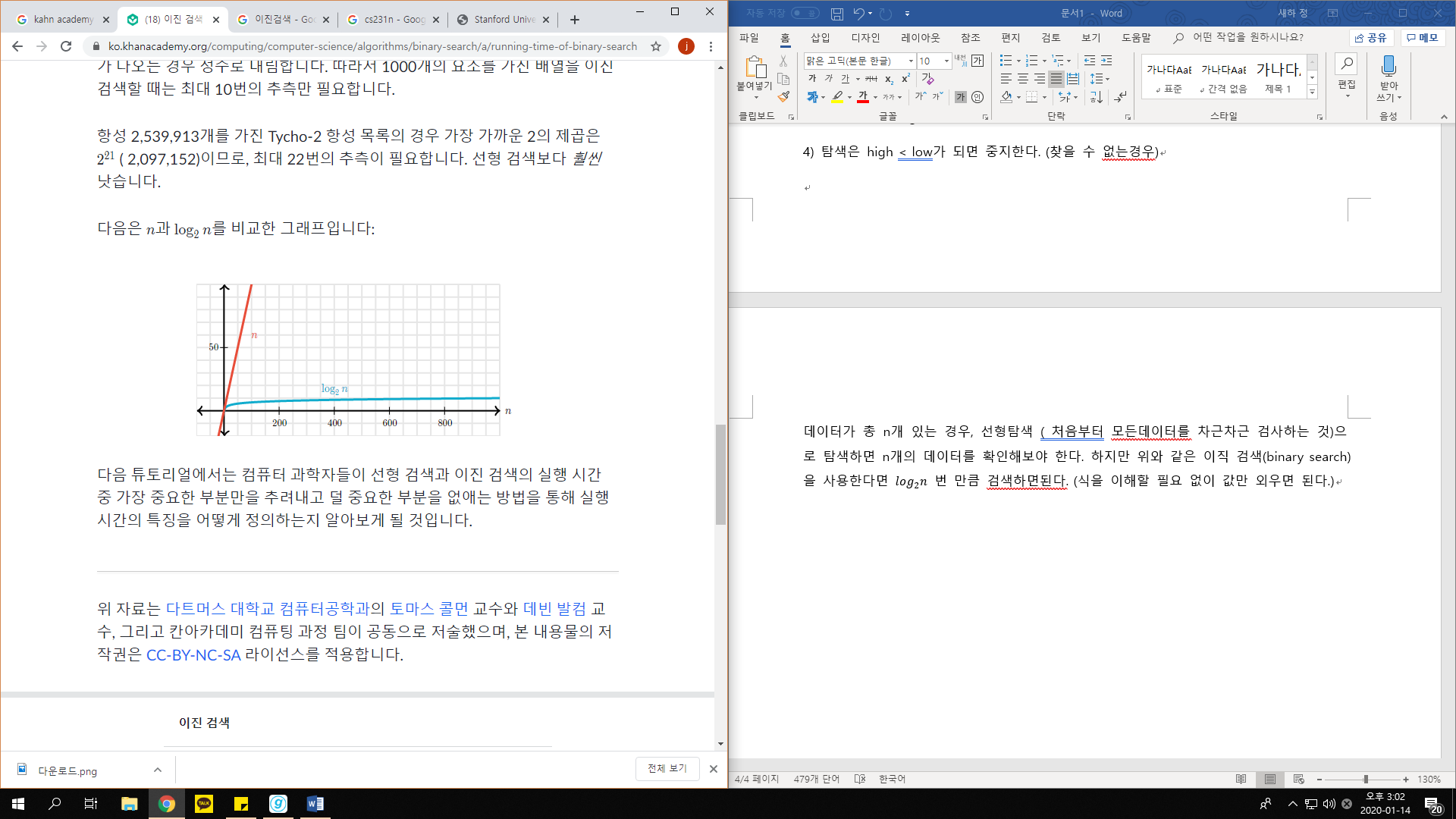
1) 데이터의 처음을 low 맨 끝을 high로 잡는다.

2) mid = (low+high)/2 즉 mid는 탐색영역 중간에 있는 데이터를 의미한다.

3) 찾으려는 데이터가 mid보다 크다면 mid의 오른쪽만 탐색한다. Low = mid + 1

3-2) 찾으려는 데이터가 mid보다 작다면 mid의 오른쪽만 탐색한다. High = mid -1

4) 탐색은 high < low가 되면 중지한다. (찾을 수 없는경우)

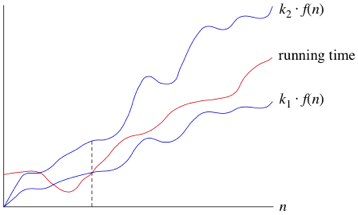
데이터가 총 n개 있는 경우, 선형탐색 ( 처음부터 모든데이터를 차근차근 검사하는 것)으로 탐색하면 n개의 데이터를 확인해보야 한다. 하지만 위와 같은 이직 검색(binary search)을 사용한다면 번 만큼 검색하면된다. (식을 이해할 필요 없이 값만 외우면 된다)

왼쪽의 그래프는 x축은 데이터의 개수, y축은 탐색하는데 걸리는 시간을 의미한다. log인경우 데이터가 늘어나도 탐색시간이 크게 늘어나지 않는 것을 볼 수 있다.

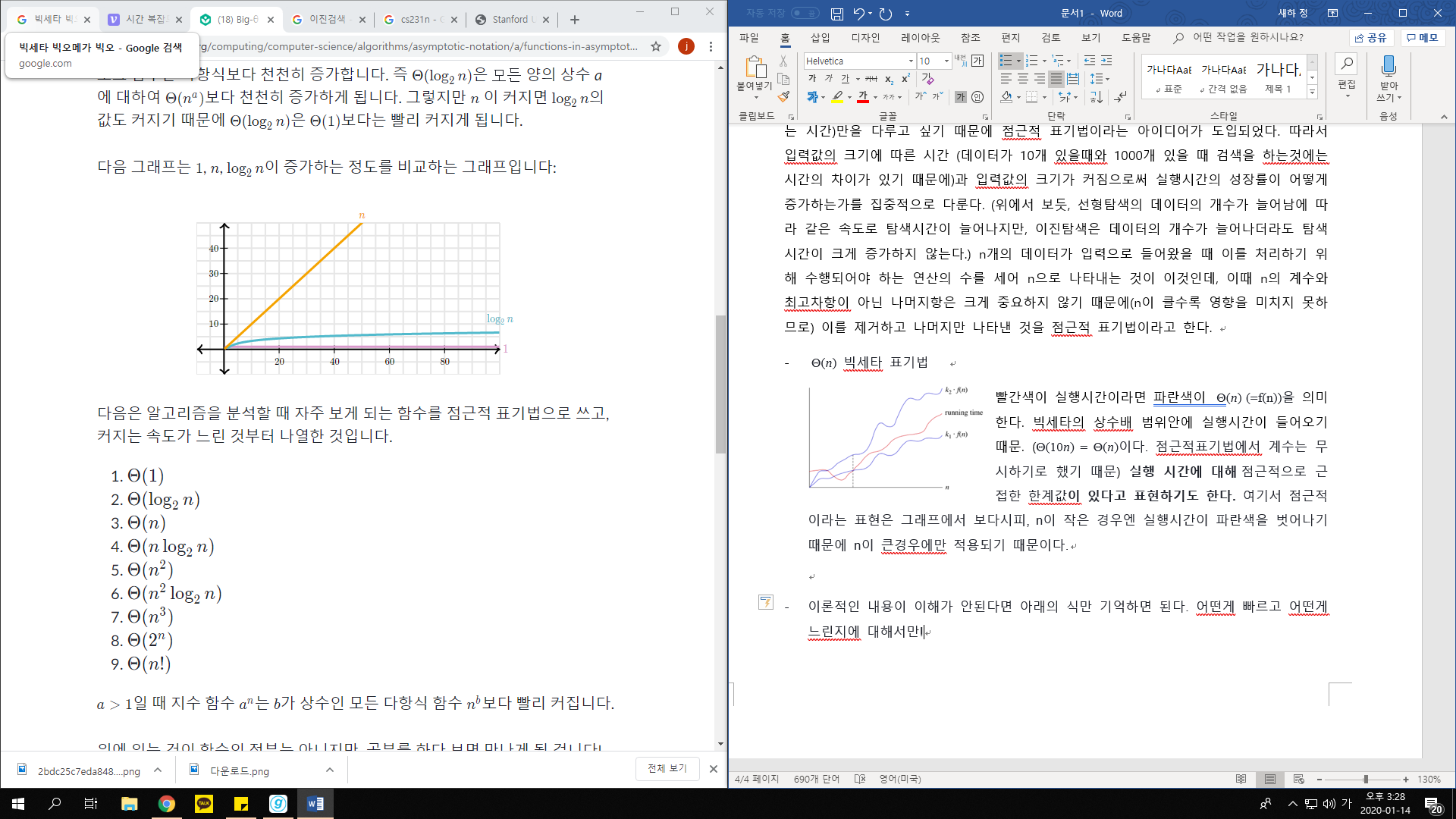
1. 점근적 표기법

앞에서 이야기한 점근적 표기법에 대해 이야기 해보자. 알고리즘의 실행시간은 컴퓨터의 언어나 컴퓨터의 하드웨어, 컴파일러의 속도와 같은 부분에 많은 영향을 받는다. 알고리즘의 성능을 이야기 할때에는 이러한 것을 제외하고 순수 알고리즘 자체의 복잡도 (걸리는 시간)만을 다루고 싶기 때문에 점근적 표기법이라는 아이디어가 도입되었다. 따라서 입력값의 크기에 따른 시간 (데이터가 10개 있을때와 1000개 있을 때 검색을 하는것에는 시간의 차이가 있기 때문에)과 입력값의 크기가 커짐으로써 실행시간의 성장률이 어떻게 증가하는가를 집중적으로 다룬다. (위에서 보듯, 선형탐색의 데이터의 개수가 늘어남에 따라 같은 속도로 탐색시간이 늘어나지만, 이진탐색은 데이터의 개수가 늘어나더라도 탐색시간이 크게 증가하지 않는다.) n개의 데이터가 입력으로 들어왔을 때 이를 처리하기 위해 수행되어야 하는 연산의 수를 세어 n으로 나타내는 것이 이것인데, 이때 n의 계수와 최고차항이 아닌 나머지항은 크게 중요하지 않기 때문에(n이 클수록 영향을 미치지 못하므로) 이를 제거하고 나머지만 나타낸 것을 점근적 표기법이라고 한다.

* Θ(*n*) 빅세타 표기법

빨간색이 실행시간이라면 파란색이  Θ(*n*) (=f(n))을 의미한다. 빅세타의 상수배 범위안에 실행시간이 들어오기 때문. (Θ(10*n*) = Θ(*n*)이다. 점근적표기법에서 계수는 무시하기로 했기 때문) **실행 시간에 대해 점근적으로 근접한 한계값이 있다고 표현하기도 한다.** 여기서 점근적이라는 표현은 그래프에서 보다시피, n이 작은 경우엔 실행시간이 파란색을 벗어나기 때문에 n이 큰경우에만 적용되기 때문이다.

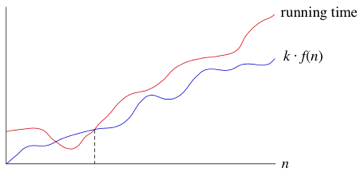
* 이론적인 내용이 이해가 안된다면 아래의 식만 기억하면 된다. 어떤게 빠르고 어떤게 느린지에 대해서만! 헷갈린다면 칸아카데미의 연습문제 풀어보기



# - Big-O (빅 오) 표기법

***점근적 상한선****에 대해서는 big-O 표기법을 사용하는데 이는 충분히 큰 입력 크기에 대하여 실행 시간에 상한값을 두고 제한하기 때문입니다.*

최악의 경우에 걸리는 실행시간을 포함하는 점근적 표기법을 빅 오 표기법이라고 한다. 예를 들어 최악의 경우 n번을 검색해야 하는 알고리즘이 있다면 O()은 n보다 크므로() 이 알고리즘의 빅오 표기법이된다. 또한 O(n)도 n을 포함하므로 이 알고리즘의 빅오 표기법이된다. (n보다 크기만 하면된다.)

-Big-Ω (빅 오메가) 표기법

**점근적 하한선. 예를 들어 동전을 던져 10번 연속으로 뒷면이 나올때까지 반복하는 알고리즘이 있다고 하자. 이때 최소한 동전을 10번은 던져야 한다. 이때 이 10번이 이 알고리즘의 점근적 하한선.**

1. 정렬

-선택정렬 의사코드

1)가장 작은 카드를 찾아서 첫 번째 카드와 바꿉니다.

2)두 번째로 작은 카드를 찾아서 두 번째 카드와 바꿉니다.

이때 두번째로 작은 카드는, 1번에서 찾은 첫번째 카드를 제외하면 가장 작은 카드로 볼 수 있다. 즉, 가장 작은 카드를 맨 앞으로 움직였으므로 남은 배열 **(= 하위배열)**에서 가장 작은 카드를 찾는것과 같음

3)세 번째로 작은 카드를 찾아서 세 번째 카드와 바꿉니다.

4)배열이 정렬될 때까지 그 다음으로 작은 카드를 올바른 위치로 옮기는 과정을 반복합니다.

-선택정렬 분석

선택 정렬에 소요되는 총 실행 시간은 세 부분으로 나눌 수 있습니다.

1)모든 indexOfMinimum 호출에 대한 실행시간. : **/2 + n/2 = Θ()**

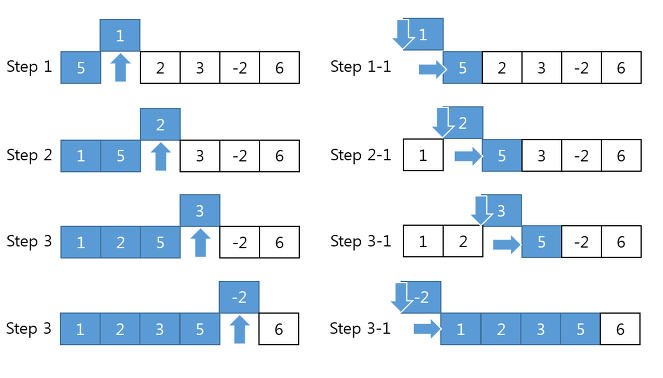
2)모든 swap 호출에 대한 실행시간. : **Θ(*n*)**

3)selectionSort함수 내 남아있는 나머지 반복문 실행시간 : **n번 반복loop- Θ(*n*)**

=> **Θ() = ) …** 최고차항이 아닌 수는 무시하므로

-삽입정렬

정렬을 한번 다르게 생각해봅시다. 어떤 사람이 카드 게임을 하고 있다고 상상해보세요. 손에 카드를 쥐고 있고 이미 정렬되어있는 상태입니다. 딜러가 새로운 카드 한 장을 건네줍니다. 그럼 이제 이 카드를 또 정렬하는 수고를 하지 않도록 새 카드를 한 번에 올바른 위치에 끼워 넣을 것입니다. 선택 정렬에서는 이미 정렬된 하위 배열에 새로이 추가되는 요소가 이 정렬된 하위 배열에 있는 요소보다 더 작지는 않습니다. 하지만 바로 위의 카드 예시에서 새로운 카드는 이미 정렬된 카드에 있는 것보다 값이 작을 수도 있습니다. 따라서 카드를 흝어보면서 새 카드와 손에 쥐고 있는 카드를 일일이 비교할 것입니다. 이렇게 새 카드를 알맞은 자리에 집어넣으면 쥐고 있는 카드 덱은 이미 잘 정렬된 상태입니다. 딜러가 또 다른 카드를 주면 이와 같은 과정을 반복합니다. 그다음 카드도 마찬가지로, 딜러가 더는 줄 카드가 없을 때까지 이 과정을 반복하게 됩니다.

1)insert를 호출하여 인덱스 0의 정렬된 하위 배열에 인덱스 1부터 시작하는 요소를 삽입합니다.

2)insert를 호출하여 인덱스 0에서 1까지 정렬된 하위 배열에 인덱스 2부터 시작하는 요소를 삽입합니다.

3)insert를 호출하여 인덱스 0부터 2까지 정렬된 하위 배열에 인덱스 3부터 시작하는 요소를 삽입합니다.

4) 반복…

5)마지막으로 insert를 호출하여 인덱스 0에서 n-2*n*−2n, minus, 2까지 정렬된 하위 배열에 인덱스 n-1*n*−1n, minus, 1부터 시작하는 요소를 삽입합니다.

-분석

삽입 정렬의 실행 시간을 정리하면 다음과 같습니다.

최악의 경우:  Θ()

최상의 경우:  Θ(*n*)

임의의 배열에 대한 평균적 경우: Θ()

1. 재귀

어떤 문제를 해결하기 위해 알고리즘을 설계할 때 동일한 문제의 조금 더 작은 경우를 해결함으로써 그 문제를 해결하는 것입니다. 문제가 간단해져서 바로 풀 수 있는 문제로 작아질 때까지 말이죠. 이런 테크닉을 **재귀**라고 합니다. 내가 나를 호출하는 함수를 재귀함수라고 한다.

예) 팩토리얼함수, 미로찾기

**재귀함수는 끊임없이 나를 호출하므로 함수가 종료하는 탈출조건이 꼭 정의되어 있어야 한다. 재귀함수는 크게 종료문(탈출조건 = basecase)과 내가 나를 호출하는 문장(재귀 조건) 두개로 나눌 수 있다.**

**규칙 :**

1. 재귀의 호출은 같은 문제 내에서 더 범위가 작은 값, 즉, 하위 문제에 대해 이루어져야 한다.
2. 재귀함수 호출은 더 이상 반복되지 않는 base case에 도달해야 한다.

예) 회문(palindrome) 찾기, 수의 제곱 구하기, 시어핀스키 가스켓(왼쪽 위 정사각형, 오른쪽 위 정사각형, 오른쪽 아래 정사각형에 시어핀스키 가스켓을 그리는 3개의 재귀 호출이 필요),하노이의 탑

* **하노이의 탑 및 점근적 표기법 연습문제 칸아카데미에서 풀어보기**

1. 분할정복

다음에서 다룰 병합정렬과 퀵소트 모두 분할정복 알고리즘을 이용한다. 분할정복 알고리즘은 재귀 알고리즘의 한종류이다.

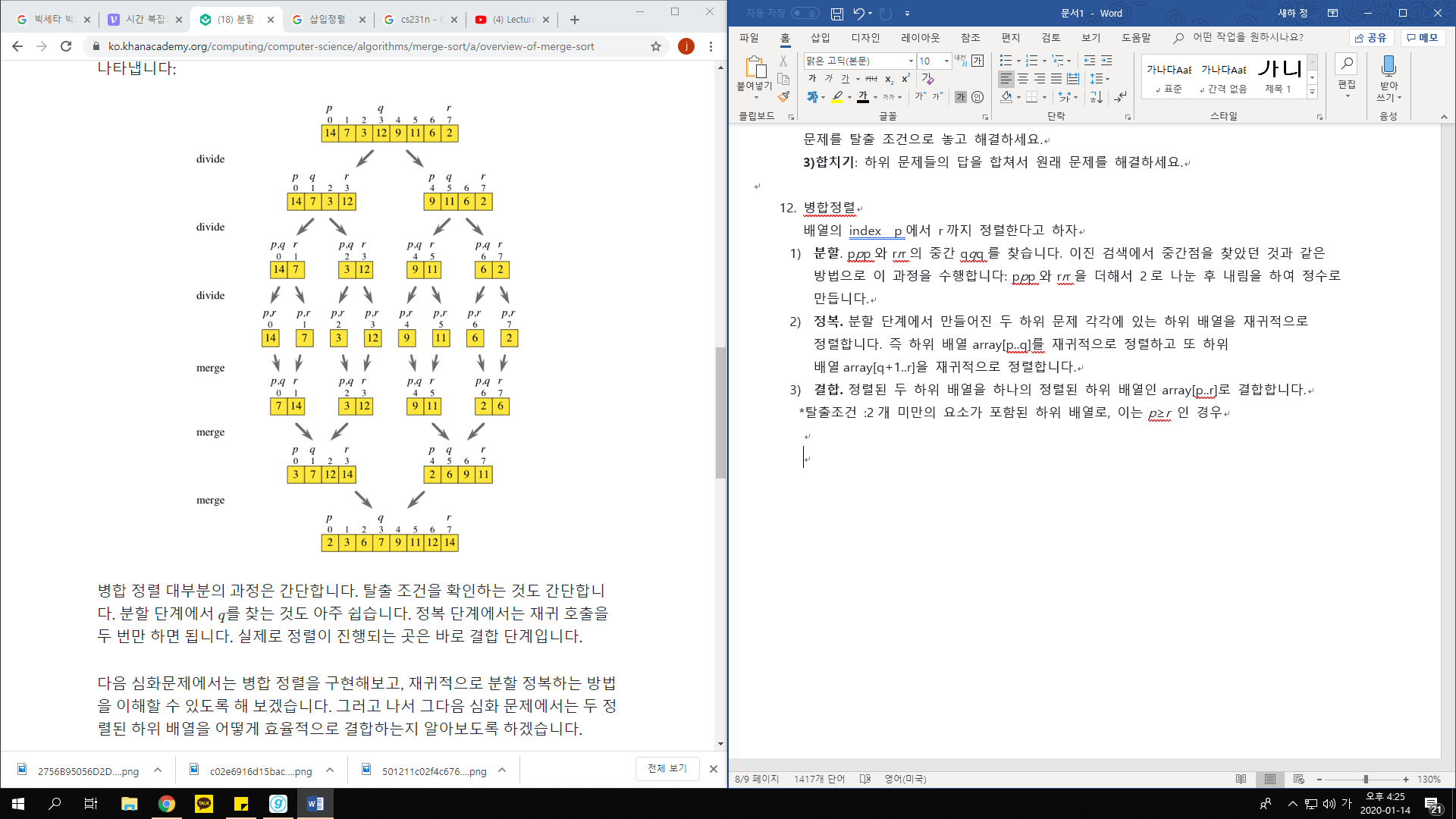
한 문제를 유형이 비슷한 여러 개의 하위 문제로 나누어 재귀적으로 해결하고 이를 합쳐 원래 문제를 해결합니다. 분할 정복 방식이 하위 문제를 재귀적으로 해결하기 때문에 하위 문제 각각은 원래 문제보다 범위가 작아야 하며 하위 문제는 각 문제마다 탈출 조건이 존재해야 합니다

1)**분할**: 원래 문제를 분할하여 비슷한 유형의 더 작은 하위 문제들로 나누세요.

**2)정복**: 하위 문제 각각을 재귀적으로 해결하세요. 하위 문제의 규모가 충분히 작으면 문제를 탈출 조건으로 놓고 해결하세요.

**3)합치기**: 하위 문제들의 답을 합쳐서 원래 문제를 해결하세요.

1. 병합정렬

배열의 index p에서 r까지 정렬한다고 하자

1. **분할**. p*p*p와 r*r*r의 중간 q*q*q를 찾습니다. 이진 검색에서 중간점을 찾았던 것과 같은 방법으로 이 과정을 수행합니다: p*p*p와 r*r*r을 더해서 2로 나눈 후 내림을 하여 정수로 만듭니다.
2. **정복.** 분할 단계에서 만들어진 두 하위 문제 각각에 있는 하위 배열을 재귀적으로 정렬합니다. 즉 하위 배열 array[p..q]를 재귀적으로 정렬하고 또 하위 배열array[q+1..r]을 재귀적으로 정렬합니다.
3. **결합.** 정렬된 두 하위 배열을 하나의 정렬된 하위 배열인 array[p..r]로 결합합니다.

\*탈출조건 :2개 미만의 요소가 포함된 하위 배열로, 이는 *p*≥*r* 인 경우

13.퀵정렬

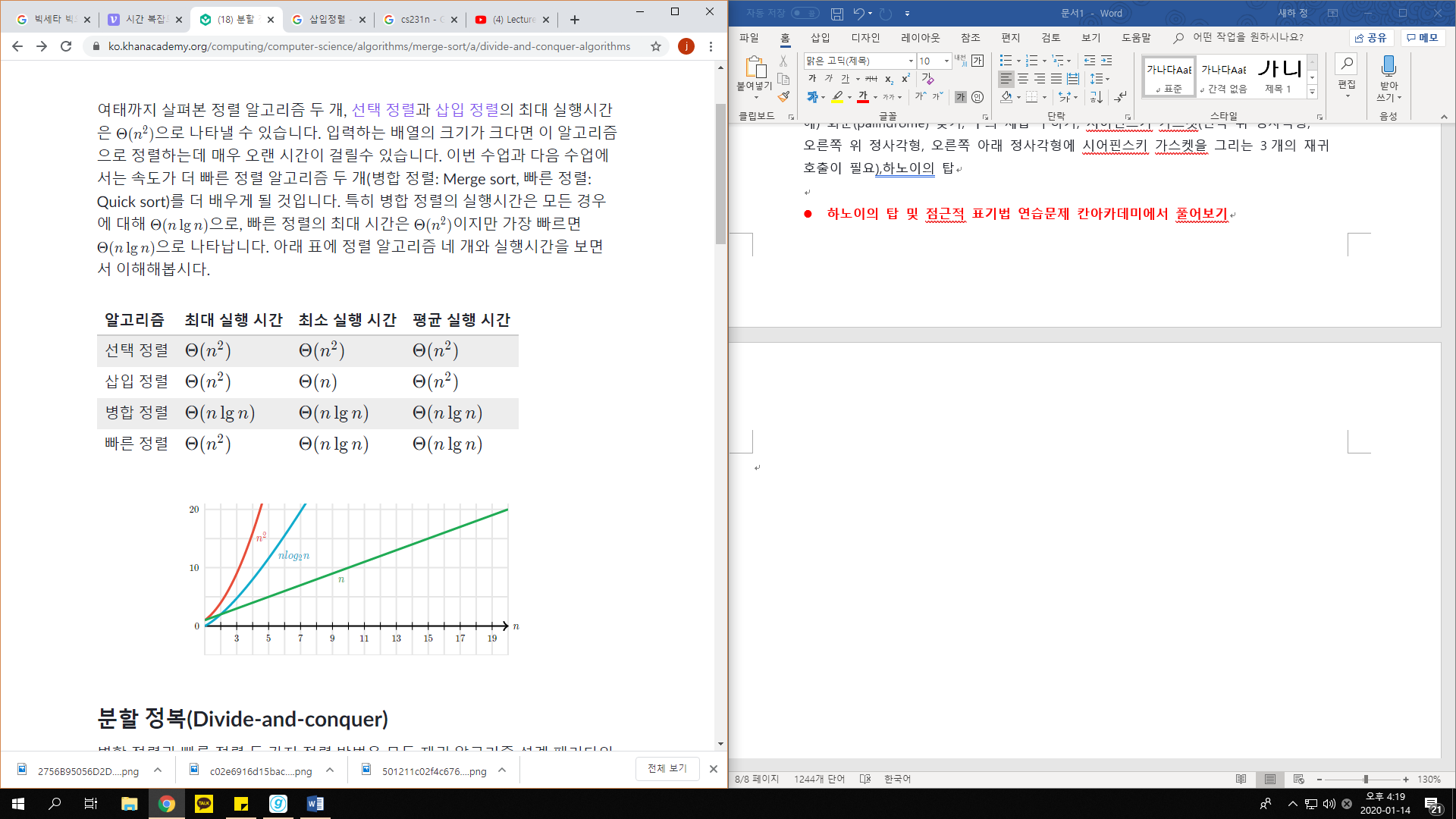
1) 파티션 : pivot을 (랜덤하게 고름) 골라 pivot보다 작은수는 왼쪽에 큰수는 오른쪽으로 분할

2) 나뉜 두개의 그룹을 다시 파티션 합니다

3) 정복 : 이과정을 재귀적으로 더 이상 파티션 할 수 없을때까지 반복합니다

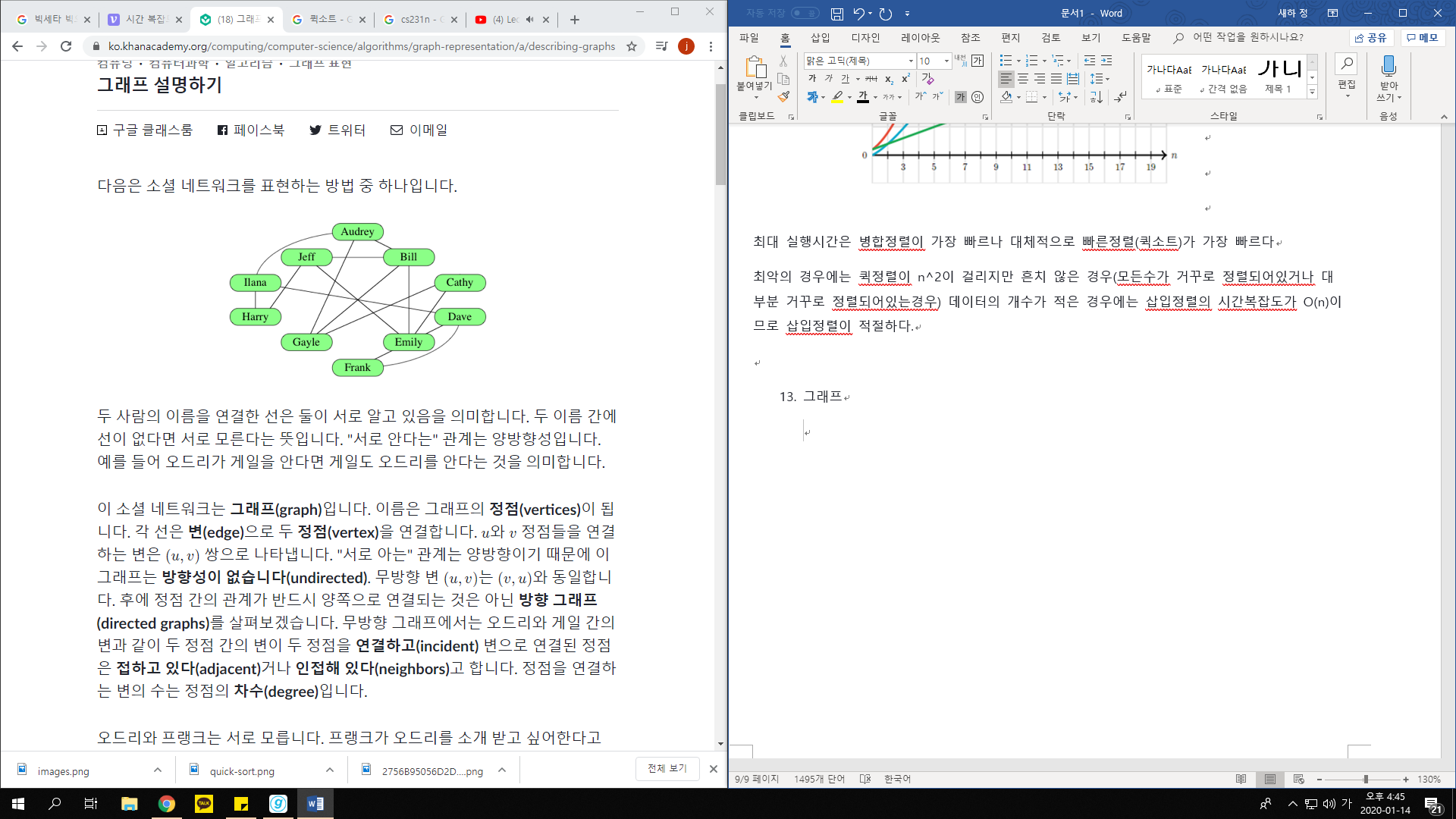
4) 결합 : 3에서 나온 결과를 그대로 결합합니다.

\* 피벗은 움직이지 않습니다.



최대 실행시간은 병합정렬이 가장 빠르나 대체적으로 빠른정렬(퀵소트)가 가장 빠르다

최악의 경우에는 퀵정렬이 n^2이 걸리지만 흔치 않은 경우(모든수가 거꾸로 정렬되어있거나 대부분 거꾸로 정렬되어있는경우) 데이터의 개수가 적은 경우에는 삽입정렬의 시간복잡도가 O(n)이므로 삽입정렬이 적절하다.

1. 그래프

왼쪽과 같은 그림을 **그래프** 라고 한다

**그래프는 연관관계나 데이터들간의 관계가 의미를 가지는 데이터들을 저장하는데 쓰인다.**

초록색원, 즉 **데이터는 하나의 정점**

**변**은 두 정점을 연결한다. 정점 u와 정점 v를 연결하는 변은 (u,v)로 나타내기도 한다.

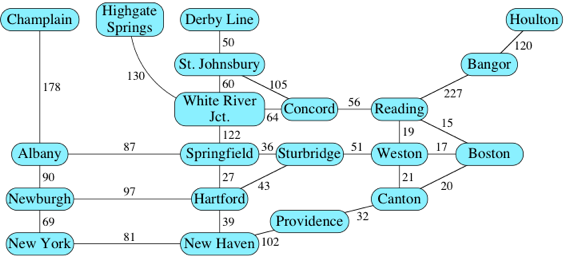
데이터간의 관계가 양방향인 경우 **방향성이 없다**고 한다. (예, A가B의 친구이면 B도 A의 친구이므로 방향성이 없다.)

방향이 있는 경우는 (그녀는 나의 엄마입니다. 이 아이는 저의 자식입니다. 와 같이 양방향으로 적영되지 않는 경우를 방향이 있다고 한다.)

**무방향 그래프**에서, 두 정점 간의 변이 두 정점을 **연결**하고 있다고 표현하고, 변에 연결된 정점은 **접하고 있다** 거나 **인접**해 있다고 표현합니다. 정점을 연결하는 변의 수는 정점의 **차수** 입니다.

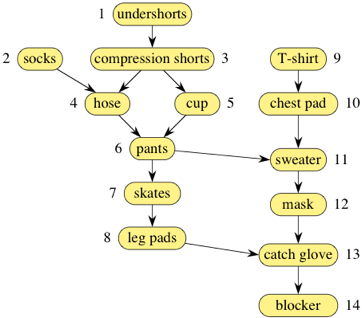
위의 그림에서 Audrey 에서 Emily로 Bill을 거쳐서 갈수 있습니다. 이렇게 한정점에서 다른정점 으로 갈 수 있는 경우 **경로**가 있다고 합니다. **최단경로**는 이러한 A에서 B로가는 여러 경로중에서 가장 짧은 경로를 의미합니다.

오드리-빌-에밀리-제프-해리-일래나-오드리와 같이 어떤 정점에서 시작하여 다시 자신에게 돌아오는 경로가 있다면 이를 **사이클**이라고 합니다.

데이터들간의 관계에 따라

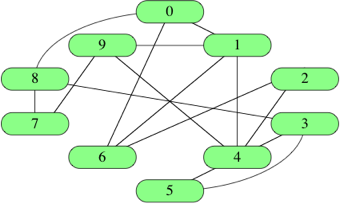
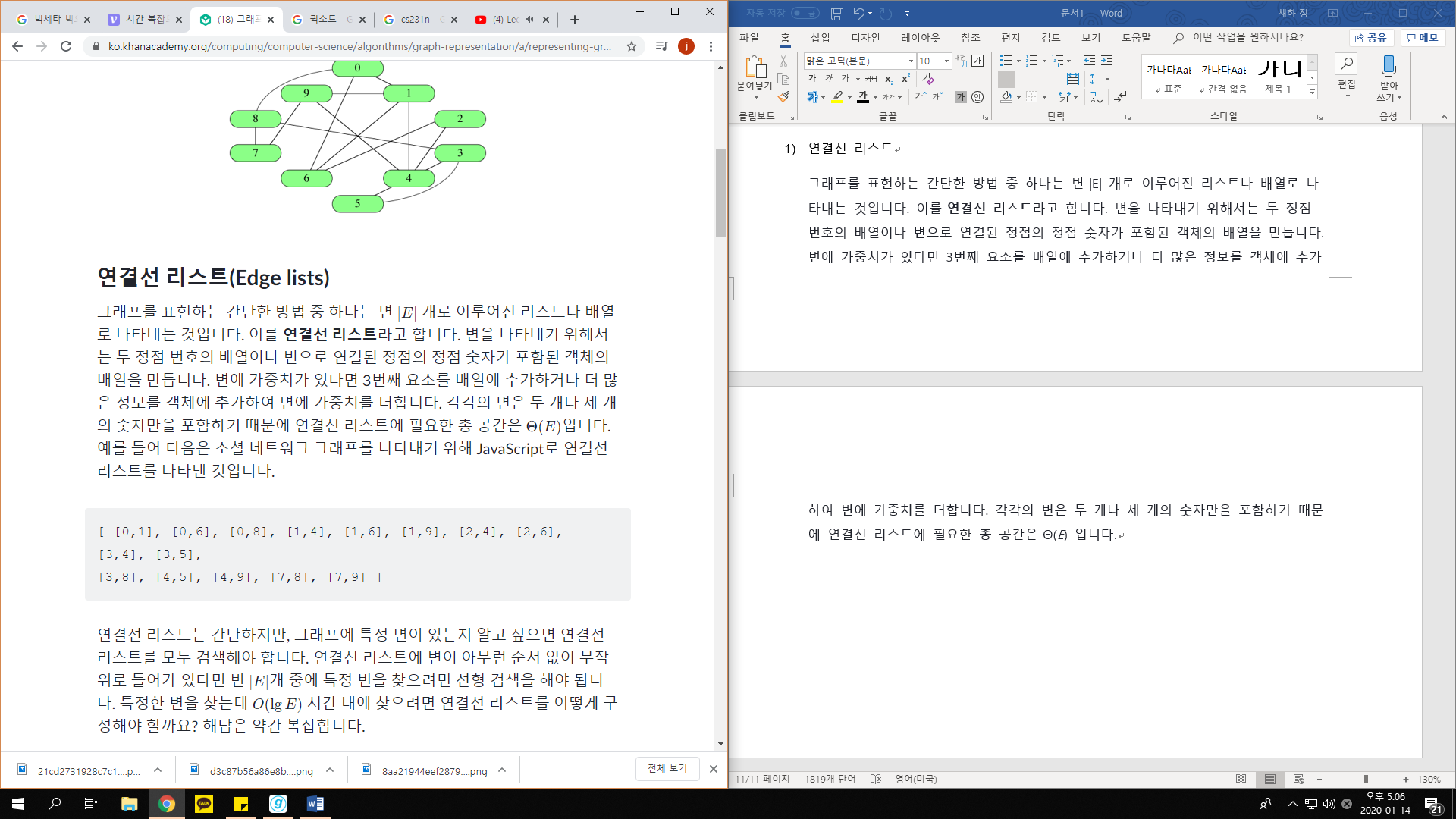
정점을 잇는 변이 **가중치**를 가질수도 있습니다. 이러한 값을 가중치라고 하고 이러한 그래프를 **가중그래프** 라고 합니다.

이런경우, 최단경로는 변의 수가아니라 가중치의 합으로 결정됩니다.

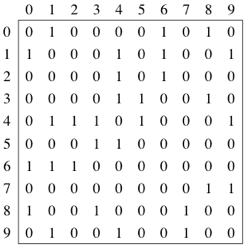
이와 같은 그래프는 화살표, 즉 방향성을 가집니다. 이러한 그래프를 **방향 그래프**라고 합니다. 왼쪽 그래프는 어떤 정점에서 시작해도 다시 그 정점으로 돌아갈 수 없습니다. 즉, 사이클이 없다는 것인데, 이런 그래프는 **방향성 비순환 그래프 입**니다.(= dag)

방향성이 있는 그래프에서 방향성이 있는 **변은 한 정점을 떠나 다른 정점으로 들어간다**고 표현합니다. 예를들어서 왼쪽 그래프에서는 undershorts와 compression shorts를 잇는 변은 undershorts를 떠나 compression shorts로 들어갑니다. 이를 **(undershorts, compression short)**로 나타냅니다. 방향성 그래프이므로 이 쌍의 **순서는 중요**합니다. 정점을 떠나는 변의 숫자는 이 정점의 **출력 차수**이며 들어가는 변의 숫자는 **입력 차수**입니다. **정점 집합을 V으로, 변 집합을 E로 나타냅니다**

1. 그래프의 표현
2. 연결선 리스트

그래프를 표현하는 간단한 방법 중 하나는 변 |E| 개로 이루어진 리스트나 배열로 나타내는 것입니다. 이를 **연결선 리스트**라고 합니다. 변을 나타내기 위해서는 두 정점 번호의 배열이나 변으로 연결된 정점의 정점 숫자가 포함된 객체의 배열을 만듭니다. 변에 가중치가 있다면 3번째 요소를 배열에 추가하거나 더 많은 정보를 객체에 추가하여 변에 가중치를 더합니다. 각각의 변은 두 개나 세 개의 숫자만을 포함하기 때문에 연결선 리스트에 필요한 총 공간은 Θ (*E*) 입니다.

연결선 리스트는 간단하지만, 그래프에 특정 변이 있는지 알고 싶으면 연결선 리스트를 모두 검색해야 합니다. 연결선 리스트에 변이 아무런 순서 없이 무작위로 들어가 있다면 변 ∣*E*∣개 중에 특정 변을 찾으려면 선형 검색을 해야 됩니다. (O(E))

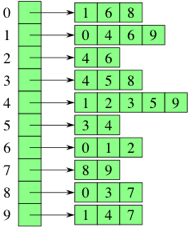
1. 인접행렬

∣*V*∣ 정점이 있는 그래프에서 **인접 행렬**은 0과 1로 이루어진 ∣*V*∣×∣*V*∣ 행렬입니다. 여기서 변 (i,j)(*i*,*j*)가 그래프에 있을 경우에만 i행과 j열의 값이 1이 됩니다. 변 가중치를 지정하고자 한다면 i, j 항목에 가중치를 주고 변이 없을 경우를 나타내기 위한 특수값(보통 null값) 을 지정해 놓습니다.

인접 행렬에서는 어떤 변의 존재 여부를 일정 시간 내에 파악할 수 있습니다. 행렬에서 해당 항목을 찾아보기만 하면 됩니다. 예를 들어 인접 행렬 이름이 graph고 변 (i,j)가 그래프에 있는지 알아보려면 graph[i][j]를 탐색해보면 됩니다. 그렇다면 인접 행렬의 단점은 무엇일까요? 두 가지가 있습니다. 먼저 그래프가 변이 몇 개 없는 **희소** 그래프일 경우라도 공간을 Θ(한다는 것입니다. 즉, 희소 그래프에서 인접 행렬은 대부분 0으로 이루어지는데 겨우 변 몇 개를 나타내는데 지나치게 많은 공간을 사용하게 되는 것입니다. 둘째로 어떤 정점이 주어진 정점 i와 인접해 있는지 알기 위해서 i정점과 인접한 정점들의 수가 적을 때도 i의 모든 |V| 항목을 찾아봐야 한다는 것입니다.

1. 인접리스트

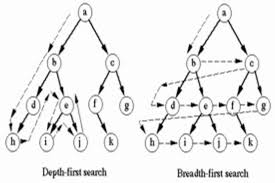
**인접 리스트**로 그래프를 나타내려면 인접 행렬과 연결선 리스트를 결합해야 합니다. 각 정점 i*i*i에 대해 그 정점과 인접한 정점의 배열을 저장합니다. 일반적으로 한 정점 당 하나의 인접 리스트가 존재하여 총 |V|개의 배열을 가집니다

각 정점의 인접 리스트는 배열을 찾아보기만 하면 되기 때문에 일정 시간 내에 얻을 수 있습니다. 변 (i,j) 가 그래프에 있는지 알아보려면 일정 시간 내에 i의 인접 리스트로 가서 i의 인접 리스트에서 j를 찾아보면 됩니다. 최악의 경우에는 얼마나 걸릴까요? 정답은 Θ(*d*) 입니다. 여기서 d는 정점 i의 차수입니다. 왜냐하면 바로 이 수치가 i의 인접 리스트의 길이를 알려주기 때문입니다. 정점 i의 차수는 |V|-1만큼 커지거나(i 이 다른 모든 |V|-1 정점들과 인접한 경우) 0으로 작아질 수( i가 인접한 변 없이 고립된 경우) 있습니다. 비방향 그래프에서 정점 j는 i가 j의 인접 리스트에 있을 경우에만 정점 i의 인접 리스트에 있습니다. 그래프에 가중치가 있다면 각 인접 리스트 내 각 항목은 정점 번호와 변 가중치의 두 항목으로 이루어진 배열이나 객체이어야 합니다.

인접 리스트가 차지하는 공간은 얼마나 될까요? 리스트가 |V| 개만큼 있고 각 리스트는 |V|-1개만큼 정점을 가질 수 있지만 무방향 그래프의 인접 리스트에는 2|E|개의 항목이 있습니다. 왜 2|E| 일까요? 각 변 (i,j) 는 *i*의 리스트에서 한 번, 그리고 j의 리스트에서 한 번, 모두 합쳐서 두 번 나타나며 리스트에는 |E| 개의 변이 있기 때문입니다. 방향 그래프에서 인접 리스트는 방향성 있는 변 하나 당 한 요소씩 총 |E| 개의 요소를 가지고 있습니다.

1. 너비우선 탐색

너비우선탐색(Breadth-first Search)은 **BFS**라고도 쓰며, 주어진 **소스 정점**에서 다른 모든 정점에 이르기까지 거치는 변의 수를 계산해 가장 짧은 경로를 찾습니다.

그림에서 볼 수 있듯, 오른쪽의 BFS는 너비를 우선으로 탐색합니다.

방문한 정점을 이웃한 정점들을 방문해햐할 정점에 추가합니다. 또한 이미 방문한 정점은 다시 방문하지 않습니다.

1. A를 방문하고 방문해야할 정점 목록에 A의 이웃인 B와 C를 추가합니다. 방문해야할리스트 : {B, C }
2. B를 방문후 방문해야할 리스트에 B의 이웃인 D와 E를 추가합니다. 방문해야 할 리스트 : {C, D, E} (B는 방문했으므로 삭제)
3. C를 방문후 방문해야할 리스트에 C의 이웃인 F와 G를 추가합니다. 방문해야 할 리스트 : {D, E, F, G}…
4. 이와 같은 과정을 반복하면 모든 노드를 방문할수 있습니다.

이와 같이 BFS는 모든 정점을 방문하므로 완전탐색의 방법 중 하나입니다. 그림에서 왼쪽을 보면 깊이를 우선으로 탐색하는 것을 알 수 있습니다. 이러한 탐색은 깊이우선탐색, 즉 DFS라고 합니다. BFS의 예로는 앞에서 언급했던 팩맨게임과 케빈-베이컨 게임 등이 있습니다. 최단거리를 구하는데에도 쓰입니다.

정점이 V개이고 변이 E개인 그래프를 너비 우선 탐색하는데 걸리는 시간은 얼마나 될까요? 답은 O(V+E) 입니다.