알고리즘및실습

2020년도 질의응답

알고리즘 구현과 프로그래밍 언어

알고리즘 문제를 풀 때 여러 언어를 사용해서 풀 수가 있는데 알고리즘 문제를 풀 때 언어가 큰 영향을 끼치나요?

언어의 특성에 따라 특정 언어를 사용하면 풀기가 더 쉬운 문제도 있습니다. 예를 들어 파이썬은 오버플로우 개념이 없어 C나 C++와 달리 오버플로우를 걱정하지 않고 프로그래밍할 수 있습니다. 또 각 언어의 표준 라이브러리가 제공하는 기능이 달라 제공하는 라이브러리에 따라 문제 해결이 용이한 언어도 있습니다. 하지만 문제를 해결하는 알고리즘 자체는 언어가 바뀐다고 변하는 것은 아닙니다. 새 언어를 학습해야 할 때 기존에 해결한 문제들을 새 언어로 해결해 보면서 새 언어의 기능을 익히는 것도 언어 비교가 직관적으로 되기 때문에 제 개인적으로는 좋은 방법이라고 생각합니다.

빅O

알고리즘 개요 빅O 슬라이드에서 29페이지를 보면 $T(n)$에 대한 간단한 문제가 있습니다. 그 문제를 해설해 주실 때 3번째와 4번째가 성립한다고 하셨는데 빅O와 빅Ω는 각각 해당 식보다 빠르다와 느리다를 의미하는 것으로 알고 있어서 n^2식은 n식에 대해 둘 중 한 개는 성립해야 된다고 생각했는데 혹시 2번이 성립하지 않는 다른 이유가 있을까요?

제가 설명을 잘 못 했군요. 2, 3, 4번이 모두 성립합니다. $\Omega(f(n))$이면 $f(n)$을 포함하여 $f(n)$보다 느린 것은 모두 포함됩니다. $O(f(n))$이면 $f(n)$을 포함하여 $f(n)$보다 빠른 것은 모두 포함됩니다. 즉, 어떤 알고리즘의 시간복잡도가 $\Omega(f(n))$이면 $f(n)$과 같거나 그보다 느리다는 것을 말합니다. 어떤 알고리즘의 시간복잡도가 $O(f(n))$이면 $f(n)$과 같거나 그보다 빠르다는 것을 말합니다.

알고리즘 분석에서 입력 크기

소인수 분해 알고리즘에서 최악의 경우 $O(2^k)$가 되는 이유가 이해되지 않아서 질문드립니다.

노트 01. 슬라이드 33에 제시된 알고리즘과 관련하여 최악의 경우는 N이 소수일 때 while 문의 반복횟수는 N-1입니다. (예: N이 7이면 d가 2부터 7까지 변하고 종료됩니다.) 그런데 배열에 대한 선형 검색과 같은 알고리즘과 비교하여 생각해보면 이 알고리즘은 항상 입력의 크기가 1이라고 생각할 수 있습니다. 하지만 N의 값이 커질수록 해야 하는 일은 증가한다는 것은 쉽게 알 수 있습니다. 정수가 하나 들어오지만, 정수의 값에 따라 하는 일이 다르다는 것입니다. 배열에 대한 선형 검색에서 입력 크기는 배열의 크기가 되며, 이 크기는 배열이 차지하는 메모리 공간에 비례합니다. 즉, 우리는 배열의 크기를 입력 크기라고 했지만, 더 정확한 크기는 그 배열이 차지하는 메모리 크기가 입력의 크기입니다. 우리가 필요한 것은 같은 알고리즘이 입력 크기가 변함에 따라 어떤 성능 변화를 보이는지 알고 싶은 것입니다. 따라서 이 경우 입력 크기를 메모리 크기로 고려하나 배열의 크기로 고려하나 차이가 없습니다. 소인수 분해 알고리즘은 이와는 다릅니다. 정수도 그 값이 커질수록 그 값을 나타내기 위한 공간이 커집니다. 즉, 정확하게 입력의 크기를 고려해야 분석이 가능합니다. 10이면 1010 4비트만 필요하지만 1000이면 1 1110 1000 9비트가 필요합니다. 주어진 N을 나타내기 위해 필요한 비트 수가 k비트이면 k비트로 표현할 수 있는 가장 큰 수는 $2^k - 1$입니다. 따라서 입력의 크기가 k비트이면 최악의 경우 $2^k - 2$번 while

문이 반복됩니다. 따라서 이 알고리즘의 빅O는 $O(2^n)$입니다. 즉, 지수시간 알고리즘입니다. k 비트로 표현된 수와 $(k+1)$ 비트로 표현된 수의 최악의 경우를 비교하면 2배 더 큰 비용이 필요합니다.

공간 복잡도

알고리즘을 풀 때 작년에 C++과제로 주셨던 first unique char 문제처럼 1개의 문자를 보고 전체에 다 있는지 확인하면 $O(n^2)$이지만 이를 크기가 26인 배열을 만들어서 알파벳마다 알파벳 위치에 1씩 더하면서 다시 처음 문자부터 중복이 있는지 배열을 이용하여 확인하면 $O(n)$인 알고리즘으로 바꿀 수 있습니다. 위의 경우 처럼 작은 공간을 잡고 이를 이용하는 경우 훨씬 효과적인 것은 알고 있는데 만약 공간을 잡아야 하는경우가 많이 커진다면 예를 들어 위의 문제에서 문자가 아닌 int형 정수의 첫 번째 중복이 없는 숫자를 찾는 것이라면 시간복잡도는 똑같이 $O(n)$이겠지만 공간 복잡도가 엄청나게 커지게 될 것 같습니다. 이런 경우라면 어느것이 더 효율적인가요? 그리고 위처럼 공간복잡도와 시간복잡도에 우선순위를 매기기 애매한 경우 어떤 기준을 통해서 순위를 정해야 할까요

그와 같은 문제를 해결해야 하는 응용의 성격과 응용이 실행되는 컴퓨팅 환경에 의해 공간복잡도와 시간복잡도의 우선순위가 결정될 것 같습니다. 다만, 예로 제시된 first unique char를 int형으로 바꾼 문제의 경우 공간복잡도가 문제 크기와 상관없이 항상 2^{32} 크기의 배열이 있어야 하므로 절대 사용할 수 없는 해결책인 것 같습니다. 보통 추가로 필요한 공간은 문제 크기와 무관하게 상수 크기가 필요하면 언제든지 사용 가능한 알고리즘이 될 것 같습니다. 알파벳의 경우에는 항상 26개 공간만 필요하니 공간복잡도가 $O(1)$이므로 훌륭한 해결책입니다. 그리고 우리가 항상 추구하는 것은 시간복잡도, 공간복잡도 모두 문제 크기에 비례하거나 상수 비용입니다.

합병 정렬

강의 영상과 다른 페이지들을 찾아봤는데 배열 수가 항상 짝수인 경우만 예시로 듭니다. 홀수일 때 정렬 과정과 수행 과정이 궁금합니다.

배열의 수가 홀수이어도 과정에는 차이가 없습니다. 예를 들어 최초 9개가 있으면 4/5 또는 5/4으로 나누어지고, 5로 나눈 부분은 다시 2/3 또는 3/2로 나누어지면 됩니다. 합병은 한쪽이 더 많아도 문제가 없습니다. 물론 9개를 3/6, 2/7로 나누어도 되지만 호출 수가 많아지므로 그렇게는 하지 않을 것입니다.

전수조사 방법

전수조사 방법을 설명하시는 슬라이드에서 전수조사 방법 이외에 다른 방법이 더 효율적이라면 다른 방법을 사용한다고 설명하셨는데, 보통 코드를 작성할 때, for문이나 while 문을 통해서 모든 경우의 수를 조사하는 방법이 전수조사 방법이라고 하는 것으로 알고 있습니다. 그 다른 방법들에는 어떠한 것들이 있는지 궁금합니다. 제가 여태까지 과제나 코딩 공부를 하면서는 전수조사 이외의 다른 방법을 보지 못한 것 같습니다.

알고리즘을 설계하는 기법이 다양하지 않지만, 전수조사가 유일한 방법은 아닙니다. 분할정복, 탐욕적 방법, 동적 프로그래밍 등 다양한 방법이 있고, 이 방법들을 알고리즘 수업을 통해 배우게 될 것입니다. 간단하게 선형검색은 전수조사하는 방법이라고 할 수 있습니다. 이진 검색은 전수조사하지 않고 검색하는 방법입니다. 물론 문제에 따라 전수조사 외에 특별히 다른 방법이 생각나지 않을 수도 있습니다. 그래도 이해되지 않으시면 탐욕적 알고리즘에서 살펴볼 거스름 문제를 생각해보세요. 주어진 금액에 맞게 최소의 동전을 주는 방법을 찾는 것입니다. 이 문제는 동전 액면가에 따라 전수조사하지 않고 탐욕적 방법으로 빠르게 찾을 수 있습니다. 여기서 탐욕적이라는 것은 가장 큰 액면가의 동전부터 이용하여 거스름을 만드는 것입니다.

전수조사 방법

알고리즘의 특성상 입력과 수행과정이 명확해야 하고 이에 따른 출력도 정확해야하는데, 그렇다면 수행과정 중의 경우의 수 또한 무한하지 않을 것 같습니다. 이론적으로 무한한 시간과 메모리가 존재한다면 전수 조사로 알고리즘에 한해서 모든 문제가 해결 가능할 거 같은데 전수조사로 해결할 수 없는 문제가 존재하나요?

해결이 가능한 모든 문제는 무한한 시간과 메모리가 존재한다면 전수조사 방법으로 해결할 수 있습니다. 하지만 해결이 가능하지 않은 문제도 존재합니다. 그런데 너무 불합리하게 많은 시간이 요구되면 해결할 수 있다는 것이 의미가 없습니다.

전수조사 방법

전수조사 방법을 사용할 수 있는 경우는 모든 경우의 수가 적은 경우에만 효과적이고, 이 모든 경우가 적다는 기준은 컴퓨터 기준이라고 하셨는데, 이 적다는 기준은 특정 숫자의 척도가 있는 것인지 궁금합니다. 아니면 케이스 별로 다를 경우 어떤 식으로 기준을 정하는지 궁금합니다

알고리즘 문제해결전략 책(pp. 115)에 보면 내가 만든 알고리즘의 시간복잡도에 최대 입력 크기를 대입하였을 때 그 값이 10^8이 넘어가면 보통 시간제한에 걸릴 가능성이 높다고 합니다. 여기서 시간제한이란 보통 1초를 말합니다. 예를 들어 입력의 최대 크기가 10,000이고 내가 만든 알고리즘의 시간복잡도가 $O(n^2)$이면 $n^2 = 100,000,000$이므로 10^8을 초과하지는 않았지만 빅O 과정에 생략된 요소나 입력 처리 시간에 따라 시간이 초과할 수도 있고 초과하지 않을 수도 있습니다. 이것은 경시대회 문제와 관련된 이야기이지만 이 내용이 질문에 대한 답을 설명할 때도 유용한 것 같습니다. 어떤 문제가 주어져 있고, 이 문제를 전수조사 방법으로 해결하고 있다고 합시다. 그러면 실제 응용에서 고려하는 문제의 크기가 정해져 있기 때문에 통상 어느 정도 시간이 걸리는지 예측할 수 있습니다. 이 시간이 우리가 수용할 수 있는 시간인지 아닌지가 결정하는 기준이 될 것 같습니다. 예를 들어 내비게이션 응용에서 최단 거리를 찾을 때 전수조사 방법을 사용하는데, 보통 출발지와 목적지를 입력하면 답을 주는데 5분 이상이 걸리면 아무도 사용하지 않을 것입니다. 제가 말한 컴퓨터 기준이란 인간은 경우의 수가 몇백 개만 되더라도 그것을 다 해볼 수 없거나 중간에 포기하겠지만 컴퓨터는 경우의 수가 10^8정도면 불만을 하지 않고 빠른 시간 이내에 모든 경우의 수를 다 해보고 답을 줄 수 있다는 것입니다.

피크닉 문제 - 전수조사

순열 계산식 방법으로 계산한 후에는 중복이 없으니 친구가 아닌 경우의 수를 빼면 더 빠른 알고리즘으로 만들 수 있지 않을까요?

친구가 아닌 경우를 빼나 친구인 경우를 찾나 모두 검사해야 하니 시간에는 차이가 없을 것 같습니다. 전체 조합을 만들 때 친구가 아닌 조합을 효과적으로 배제할 수 있다면 시간을 조금 단축할 수는 있을 것 같습니다.

분할 정복

강의 자료에는 입력 크기가 거의 n인 2개 이상의 문제로 분리될 때 분할정복이 적합하지 않다고 나와 있는데, 거이 n에 근접하지 않더라도, 만약 문제를 b개의 문제로 분리할 때 입력 크기가 n/b보다 상당히 클 경우에도 분할정복이 적합한가요? 예를 들어 $f(n) = f(n/2) + f(n/2+1) + f(n/2+2)$처럼 입력 크기가 n/b보다 1.5배 이상 클 때도 분할 정복을 사용하기에 적합한가요?

기본 생각은 빠르게 기저 사례로 도달해야 한다는 것과 너무 많은 소문제로 분할되지 않아야 한다는 것입니다. 분할 정복 알고리즘 성능에 있어 비재귀적 과정의 비용도 매우 중요합니다. 문제가 b개의 문제로 분리할 때 입력 크기가 n/b보다 크면 소문제가 비교적 많은 경우로 보입니다. 실제 성능은 분석해 보아야 할 것 같습니다. 4장의 도사 정리를 배우면 이와 같은 알고리즘을 비교적 쉽게 분석할 수 있습니다. 단, 소문제의 크기가 모두 같아야 도사 정리에 적용할 수 있습니다.

역쌍 구하기 응용

1차시 강의 중에 역쌍 구하기 응용문제에서 영화에 대한 우선순위 옆에 숫자를 구하는 방법을 간단히만 짚어주셨는데 잘 이해가 안 됩니다.

A의 과일에 대한 선호가 있습니다.
1. 딸기, 2. 바나나, 3. 사과, 4. 블루베리, 5. 수박
같은 다섯 개 과일에 대한 B의 선호
1. 바나나, 2. 블루베리, 3. 사과, 4. 수박, 5. 딸기
같은 다섯 개 과일에 대한 C의 선호
1. 바나나, 2. 사과, 3. 딸기, 4. 수박, 5. 블루베리
A의 선호와 B의 선호가 얼마나 가까운지 비교하는 것입니다. 이때 A의 선호를 기준으로 B의 선호와 C의 선호를 표현하면 다음과 같습니다.
B: 2 4 3 5 1
C: 2 3 1 5 4
B의 역쌍의 개수는 4개, C의 역쌍의 개수는 3개입니다. 따라서 C의 선호가 A의 선호와 조금 더 가깝다고 하는 것입니다.

가장 가까운 쌍 찾기

단계 2. (base case <= 3: brute-force 방법)

- $(l_1, l_2) =$ closestPair(L_x, L_y)
- $(r_1, r_2) =$ closestPair(R_x, R_y)
- $(s_1, s_2) =$ closestSplitPair$(P_x, P_y) \rightarrow O(n)$

A, B, C면 A와 B 비교, A와 C 비교, B와 C 비교라고 하셨는데 l_1, l_2 그리고 L_x, L_y는 무엇인지 도통 모르겠습니다. l_1, l_2 가 A고 L_x, L_y가 B가 되는 것인가요?

단계 2 아래에 있는 closestPair부터는 좌표들의 수가 3보다 클 때 사용하는 부분이고, 단계 2 옆에 있는 base case <= 3은 좌표가 3개보다 작을 때는 3개를 직접 비교하여 가장 가까운 두 개의 좌표를 찾아야 한다는 것입니다. 이 알고리즘은 전체 좌표 집합 Q가 있을 때, 이를 x 좌표를 기준으로 정렬한 P_x, y 좌표를 기준으로 정렬한 P_y를 생성해야 하고, P_x와 P_y를 이용하여 왼쪽 절반에 해당하는 L_x, L_y를 만들고, 오른쪽 절반에 해당하는 R_x, R_y를 만들어야 합니다. 그리고 L_x, L_y와 R_x, R_y는 재귀 호출로 해결합니다.

가장 가까운 쌍 찾기

3장 강의 슬라이드 18번에서 8개의 박스가 그려졌는데 그 박스들이 그 모양대로만 고정인지 아니면 다른 모양으로 8개가 존재할 수 있는 것인지 궁금합니다. 슬라이드에서는 $0 < y < \delta$의 범위에만 박스들이 그려졌는데 수학적으로 그런 모양만 가능한가요? 아니면 점 q가 속해 있는 박스 범위에 다른 점이 없다는 의미로만 그려져

. 있는 건가요?

박스 모양은 고정입니다. split inversion을 찾을 때 x 중앙값 $\bar{x}$와 δ 값은 이미 계산된 값입니다. 따라서 지금 비교할 좌표 q의 y값을 기준으로 8개의 박스를 그릴 수 있습니다. 물론 q는 그림에 있는 그 위치는 아닐 수 있지만 아래 4개의 박스 중 맨 아래 선에 위치하게 되고, 이 q와 비교할 좌표들이 있을 경우 q가 있는 박스를 제외한 나머지 박스에 위치할 수 있습니다. 따라서 비교할 것이 최대 7개가 존재하게 됩니다. 신기한 면이 있지만, 꼼꼼히 생각하면 이해되실 것입니다.

가장 가까운 쌍 찾기

만약 쌍집합이 (1,1), (2,4), (4,4), (4,6), (6,3)이 있다면 중심축은 3, 감마값이 3, y축은 3/2인 1.5씩 총 4개로 나눠지는게 맞나요? (이미지 첨부합니다) 이 경우 한 칸 안에 두 점이 들어간 케이스가 없는것 같은데 이 알고리즘이 항상 유효한 것인가요? 제가 잘못 이해한 것이라면 지적 부탁드립니다.

쌍집합 (1,1), (2,4), (4,4), (4,6), (6,3)을 x값을 기준으로 정렬하여 반으로 나눈 결과가 (1,1),(2,4),(4,4)와 (4,6),(6,3)이라 합시다. 왼쪽은 (2,4), (4,4)가 가장 가까운 쌍이고 그 거리는 2입니다. 오른쪽은 (4,6), (6,3)이 가장 가까운 쌍이고 그 거리는 3.6입니다. 따라서 split pair을 찾을 때 입력되는 δ 값은 2입니다. x 중앙값을 4로 하여 (4,4)와 거리 2 이내에 있는 값은 y축으로 정렬하면 (4,4) (4,6)밖에 없습니다. 다시 한번 생각해 보시고 이해가 되지 않으면 다시 질문해 주세요.

도사정리

도사정리 증명을 이해했는지 확인해보기 위해 문제들을 풀던 중 이해가 되지 않는 부분이 있어서 질문드립니다. 첨부된 이미지 중 '도사정리.PNG'를 보시면, 노란색 줄이 쳐진 부분이 있는데 이 부분이 이해되지 않습니다. 혹시 이게 반복대치인가요? 일반적인 반복대치랑은 좀 다른 것 같아서 헷갈리네요.

점화식 $T(n) <= 3T(n/2) + n^2$이면 $a = 3, b = 2, d = 2$입니다. 그러면 도사정리에 의해 $a = 3 < b^d = 2^2 = 4$이므로 경우 2에 해당됩니다. 따라서 시간복잡도는 $O(n^2)$입니다.

빠른 정렬

pivot을 중간값으로 찾아 설정하는 비용이 $O(n)$로 찾아 설정하더라도 $2n \log n$의 비용으로 $0(n \log n)$의 비용이 들것 같은데 pivot을 랜덤값으로 사용하는 이유가 있나요?

빠른정렬을 배우고 pivot 자료를 살펴보다가 linear selection 알고리즘을 알게 되었는데 원소 5개의 중앙값을 찾기위해 6번 비교를 통해 얻을 수 있다는 것을 근거로 재귀를 사용하여 중앙값을 찾는 알고리즘이었습니다.

시간복잡도가 같으면 생략된 것들에 대한 고려가 필요하고, 프로그램 코드의 형태도 고려합니다. 중간값을 선형비용으로 찾을 수 있다고 하더라도 그 비용이 랜덤으로 pivot을 찾는 비용보다 클 수밖에 없습니다. 코드도 랜덤으로 pivot을 결정하는 것이 간결합니다.

빠른 정렬

이번 5장 과제인 빠른 정렬에서 제가 작성한 것에서 pivot을 설정할 때 안에 있는 랜덤한 값을 고르는 것이 아니라 랜덤값 3개를 구하고 그 3개의 평균을 이용해서(소수점은 버림 연산) pivot을 설정했습니다. 그러다 보니 정렬을 하려는 배열 안에 pivot의 값이 있는지 확신할 수 없기 때문에 앞에서부터 비교하며 pivot값보다 큰 값이 나오면 inplace 방식으로 넘기는 방식을 그대로 사용하는 것은 문제가 안됐습니다. 그런데 중복값이 여러 개 나오고 이 값이 pivot값과 같은 경우에 문제가 발생했습니다. 맨 처음에는 pivot보다 크다고 가정하고 작성했더니 무한 루프에 빠져서 탈출하지 못했습니다. 그런데 이 값을 pivot보다 작은경우로 처리하는 경우에는 정상작동이 되는데 왜 이런 차이가 발생하는 것인가요?

피봇값은 원래 배열에 존재하는 값이어야 합니다. 물론 배열에 존재하지 않는 값을 피봇으로 사용하여 알고리즘을 만들 수 있지만 원래 알고리즘은 그렇게 하는 것이 아닙니다. 즉, 3개의 평균을 구하지 않고, 3개의 중간값을 구하는 방법은 가능합니다. 랜덤값은 주어진 범위에서 위치를 랜덤으로 선택하면 됩니다.

BFS, DFS

1. 가중치 그래프가 간선에 가중치가 할당되어 있다고 설명해주셨는데 어떠한 작업에 대한 가중치인지 궁금합니다.
2. 이 그래프(사진)에서 네모안에 있는 숫자들이 간선의 개수를 간략하게 표현했다고 이해했는데 맞는지 궁금합니다.

가중치 값은 응용에 따라 다릅니다. 도로를 그래프로 모델링하였으면 평균 교통량일 수 있고, 통행 차량의 평균 속도일 수 있고, 거리일 수 있습니다. 네모 안에 있는 숫자가 그 간선의 가중치 값입니다.

BFS, DFS

슬라이드에서 BFS는 무방향 그래프에 쓰이고, DFS는 방향 그래프에 쓰는 이유를 알고 싶습니다. 또한 그래프의 특성에 따라 BFS와 DFS를 선택하여 사용하는 것인지 알고 싶습니다.

BFS는 인접 노드를 모두 처리하는 방식이고, DFS는 한쪽으로 계속 전진하는 방식입니다. 이 특성 때문에 BFS가 무방향 그래프에서 해결해야 하는 문제의 특성에 잘 부합합니다. 예를 들어 무방향 비가중치 그래프에서 최단 경로는 인접 노드부터 차례로 전진하는 것이 효과적이라고 생각합니다. 이와 달리 위상 정렬과 같은 경우에는 한쪽으로 끝까지 가는 것이 필요한 문제이기 때문에 DFS가 문제의 특성과 부합하는 알고리즘입니다. 즉, 보통 무방향 그래프에서 해결해야 하는 문제의 성격과 BFS가 잘 맞고, 방향 그래프에서 해결해야 하는 문제의 성격이 DFS와 잘 맞는 것 같아 그렇게 사용하는 것 같습니다.

두 방식의 성능은 같고, 두 방식 모두 그래프의 방향성 여부와 무관하게 어떤 그래프에 대해서도 적용할 수 있는 일반 그래프 탐색 알고리즘입니다. 따라서 문제의 성격에 더 적합한 것을 선택하여 활용하면 됩니다. 두 방법을 사용할 때 차이가 없으면 보통 BFS를 사용하는 경향이 있는 것 같습니다.

위상 정렬

위상정렬(2/4) 슬라이드에서 위상정렬의 문제점을 2가지 소개해 주셨는데, 문제점 1과 같은 경우는 인접 행렬과 인접 리스트에 한정해서 생기는 문제인가요? 위상정렬(3/4), 위상정렬(4/4)에서는 소스 노드를 찾지 않아도 되는 방법들인데 2/4슬라이드에서 문제점을 소개하셨는지 이해가 잘 안 됩니다. 또 위상정렬에서 가중치 방향 그래프일 경우 가중치에 대한 고려는 어떻게 하는지 궁금합니다

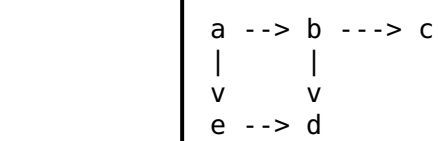
위상 정렬(2/4)에 소개한 알고리즘을 이용하여 구현하기 위해서는 소스 노드를 찾을 수 있어야 한다는 것입니다. 인접 행렬에서는 소스 노드를 찾는 것이 어렵지 않고, 인접 리스트에서는 역 인접 리스트가 없으면 불편하다는 것입니다. 그리고 3/4에서 소개한 알고리즘은 소스 노드를 찾을 필요가 없는 알고리즘이고, 그래프에 대한 수정도 필요 없습니다. 2/4에 소개된 알고리즘이 직관적이고 구현하기 쉽다는 이점이 있지만 이와 같은 문제점들 때문에 3/4에 제시된 알고리즘을 주로 사용한다는 것을 설명해 드리기 위해 2/4에 문제점을 제시하였습니다. 제가 이해한 바로는 위상 정렬은 간선의 가중치는 고려하지 않습니다. 방향 그래프에 대해 위상 정렬을 한 후에 위상 정렬의 순서를 이용하여 가중치를 고려하면서 최단 경로를 찾는 알고리즘도 있는 것 같습니다.

위상 정렬

1. 위상정렬을 구하는 방식이 한 점을 기준으로 이어진 간선들을 기억해서 큐랑 스택을 이용한 2가지 방법이 존재하는데 큐를 이용하면 BFS, 스택을 사용하면 DFS라고 생각하는데 제가 올바르게 이해한 것인가요?
2. 비재귀적으로 DFS를 통해 위상정렬을 구할 때 탐색을 진행하면서 시작노드부터 지나온 노드들을 차례대로 기억한다면(중복된 값 제외) 위상정렬를 따로 구하는 공식이 필요가 없다고 생각하는데 이 방식은 잘못된 방식인가요?

```
check_num=0
def DFS(start): #한 출발점을 기준으로 모든 노드 탐색
    visited = []
    stack = [start] #앞으로 탐색해야할 노드들
    while stack:
        n = stack.pop(0) #리스트에 하나씩 꺼내 노드를 탐색한다
        if n not in visited:
            visited.append(n)
            # 현재 탐색중인 노드에서 새로운 위치의 노드가 발견되면
            # 그 노드와 연결되는 노드들을 조사한다
            for x in graph[n]:
                stack.insert(0,x)
        # 현재 중복된 노드의 값들이 순환되서 오는 것인지 아니면 다른 경로를 통해 오는
           것인지 확인
        else:
            check_num=check1(n,n)
            if check_num==2:
                return print("사이클이 있는 그래프입니다")
    return visited
```

위상정렬은 방향 그래프를 대상으로 하는 문제이고, 기본적으로 DFS를 활용합니다. 슬라이드에 3가지 방법을 소개하고 있는데, 첫 번째 방법은 소스 노드를 찾을 수 있어야 하는 방법이고, 두 번째, 세 번째 방법은 소스 노드를 찾을 필요가 없는 방법입니다. 두 번째, 세 번째 방법은 모두 DFS를 이용하지만 하나는 재귀적 구현방법이고, 다른 하나는 비재귀적 구현 방법입니다. DFS는 스택을 사용하여 구현되며, 큐를 이용하여 구현할 수 없습니다. 큐를 이용하게 되면 BFS가 됩니다. 슬라이드에 소개한 방법은 시작노드가 어떤 노드인지 중요하지 않습니다. 코드로 제시된 것은 모든 그래프에서 올바르게 위상 정렬을 하지 못합니다. 시작 노드를 소스 노드로 생각하더라도 다음 그래프에서 dfs를 진행하면



방문하는 순서는 [a b c d e]입니다. 이 순서는 위상정렬 순서는 아닙니다. dfs로 c까지 이동하여 더 이상 갈 곳이 없으면 5를 부여하고, b로 되돌아 올 때 갈 곳이 있어 d로 이동하게 되고, d에서 더 이상 갈 곳이 없어 4를 부여하고, b로 되돌아 오면 b에 3을 그 다음 a로 되돌아 오면 e로 갈 수 있어 e로 이동하고 e에서는 갈 곳이 없어 2를

부여하는 방식으로 위상정렬을 합니다. 이 방식은 b에서 시작해도 상관이 없습니다. b에서 시작하면 c: 5, d: 4, b: 3 그다음 e에서 시작했다고 가정하면 e: 2, 남은 a에 1을 할당합니다.

다익스트라 알고리즘

다익스트라를 설명하실 때 입력에 방향 가중치 그래프라고 하셨는데, 무방향 가중치 그래프도 되지 않나요? 왜 방향 가중치 그래프에 한정하신 것인지 궁금합니다.

방향 가중치 그래프가 되면 무방향 가중치 그래프에서도 동작합니다. (무방향 가중치 그래프는 양방향으로 같은 가중치를 갖는 방향 그래프로 바꿀 수 있습니다.) 다익스트라 알고리즘을 적용할 때 제한 사항은 가중치가 양수이어야 합니다. 다익스트라는 방문한 노드와 방문하지 않은 노드를 기준으로 경계를 넘어가는 것 중 가장 최소 가중치를 선택하게 됩니다. 예를 들어 출발노드가 s이고, 이번에 선택한 간선이 (v, w)이면 이 간선에 의해 s에서 w까지 가장 짧은 경로를 찾은 것이 되어야 합니다. 이것이 성립하기 위해서는 간선의 가중치가 양수이어야 합니다. 음수이면 이것이 성립하지 않습니다. (27 28 슬라이드 증명 참조)
예) $s \to v \to w (s \to v : 1), (v \to w : 2)$
그런데 $v \to z : 3, z \to w : -4$이면 $s \to v \to z \to w$가 s에서 w로 가는 더 짧은 경로가 됩니다.

다익스트라 알고리즘

delete를 할 때 시간복잡도가 $\log(n)$이 어떻게 나오는지 모르겠습니다. 이유가 힙에 들어간 데이터가 노드의 이름, 가중치(거리값) 이렇게 2가지로 나뉘어 있을 것입니다. 맨 처음 힙에 들어있는 데이터는 우리가 갖는 그래프의 데이터가 다른 상태입니다. (초기값 inf를 넣어야 하므로) (슬라이드 32 그림에서도 간선 적힌 숫자와 뒤의 배열(힙)안의 숫자가 다릅니다) 이때 우리가 특정 노드를 delete를 하려면 노드의 이름을 가지고 추적을 해서 삭제를 해야 하는데 문제는 이때 힙은 가중치로 정렬이 되어있지 이름으로 정렬이 되어있지 않습니다. 그러면 가중치로는 찾을수 없으니 이름으로 특정 노드를 찾기위해 필요한 비용은 $O(n)$으로 보이는데 어떻게 $O(\log(n))$이 나올수 있나요?

이진 힙은 내부적으로 보통 배열로 구현됩니다. 따라서 특정 요소가 저장된 위치를 알면 그 위치와 트리의 루트 값을 교체하여 $O(\log n)$에 삭제할 수 있습니다. 이 부분은 숙제이기 때문에 자세한 답은 숙제 피드백을 통해 설명해 드리겠습니다.

캐시 문제

8번째 강의자료에서 9페이지에 있는 내용 중에 [a b c d]의 캐시에서 [c d e f a b]를 요청하게 되면 4개의 page fault가 생긴다고 하셨는데 e를 요청하면서 생기는 것, f를 요청하면서 생기는 것 외에 다른 두개의 page fault는 무엇인지 궁금합니다.

캐시의 슬롯이 4개입니다. [a b c d]가 캐시에 들어 있을 때 c d 요청은 캐시에 있기 때문에 그대로 진행됩니다. e f 요청은 캐시에 없기 때문에 기존과 교체해야 합니다. 이때 a b와 교체되어 c d e f를 처리한 후 캐시 모습이 [e f c d]이면 그다음 a b를 처리하기 위해 2개 fault가 또 발생합니다. 어떤 것을 교체하느냐에 따라 fault의 수가 달라질 수 있습니다.

캐시 문제

강의 마지막 부분에 LRU 예를 들어 주셨는데 캐시 사용하는 순서를 보면 굉장히 비효율적으로 사용하는 것 같습니다. Furthest-in-future에 대비되는 알고리즘으로 설명해주시는 건가요?

Furthest-in-future는 미래를 예측할 수 있어야 하므로 실제 사용할 수 있는 알고리즘이 아닙니다. 다른 캐시 알고리즘의 성능을 비교할 때 사용할 뿐입니다. LRU는 사용되고 있는 캐시 알고리즘 중 가장 효과적인 알고리즘 중 하나입니다.

union-find

모든 노드가 연결되어 있는 경우는 root가 하나인 것인가요? 예를 들어 첨부(1)과 같은 경우 각각의 노드는 특정 노드까지 갈 수 있는 길이 있는 것이니까 다 연결되어 있는 것 아닙니까? 그럼 union-find 시 첨부(2)와 같은 형태의 하나의 루트로 수렴되는 것인가요?

예, 하나의 루트로 수렴될 수 있습니다. 그런데 MST를 찾는 과정에서는 각 노드가 독립 분할에서 시작하여 점진적으로 하나의 리더로 수렴해 가게 됩니다.

union-find

학기 초에 질문드렸던 것과 비슷한 경우입니다. edge를 뽑아내야 하는 상황인데 이때 2가지 방법이 있다고 생각합니다.

- edge가 add될 때마다 리스트에 add하여 저장하고 요청시 리스트를 반환한다.
- 요청을 할때 edge를 다 찾아서 리스트로 만들어 반환한다.

이때 1번의 경우 add 할 때 시간복잡도 $O(1)$ (ArrayList에 add인경우라 가정) 그리고 요청할 때 시간복잡도 $O(1)$이 될것입니다. (이미 다 값을 알고있으니 가져오기만 하면 됩니다.) 하지만 이때 공간복잡도는 $O(n)$, 즉 edge의 크기만큼 계속 가지고 있어야 합니다. 그리고 2번의 경우 요청을 했다면 시간복잡도가 $O(n)$이 될 것입니다. ($O(2n)$, edge를 불러올때 무방향이기에 노드 양쪽에 저장하므로 2배가 들어갈 것입니다.) 하지만 메모리는 그 순간에만 필요할것 입니다. 위 같은 경우에는 시간복잡도가 줄어든 것에서 많은 차이 없이 공간복잡도가 늘어납니다. 그래서 1번 경우를 사용하여 edge를 빠르게 찾는 것이 더 좋은 경우처럼 보이는데 맞게 판단한 것인지 궁금합니다. 그리고 만약 시간복잡도 $O(n)$을 줄이기 위해 공간복잡도가 $O(n^2)$정도로 많이 늘어난다면 위처럼 메모리를 소모하여 속도를 높이는 게 이득인지 아니면 메모리를 아끼는 것이 이득인지에 대하여 판단하는 기준이 있는지 궁금합니다.

edge를 뽑아내야 하는 상황이 뭔지 모호합니다. 또 그래프가 인접 행렬로 표현된 경우와 인접 리스트로 표현된 경우가 다를 것 같습니다. 시간복잡도 vs. 공간복잡도 측면에서 설명해 드리면 같은 시간복잡도이면 공간복잡도도 줄이는 것이 좋겠지요. (너무 상식적??) 공간복잡도를 증가하여 시간복잡도를 줄일 수 있는 경우 (그 반대는 무의미) 시간복잡도를 줄이기 위해 공간복잡도를 증가시키는 것은 많이 사용하는 기법입니다. 특히 메모리가 여유가 많다면 더욱 그렇습니다. 더욱이 그 연산을 자주 호출한다면 공간복잡도를 증가하여 시간복잡도를 줄이는 것이 좋습니다. 질문하신 edge 관련하여 리스트를 유지하는 비용은 추가하는 비용 외에 삭제하는 비용도 고려하여야 합니다. 예를 들어 인접행렬로 그래프를 표현하고 별도 edge 리스트를 유지할 때 추가는 $O(1)$에 할 수 있는데 edge가 제거되면 비용이 edge 리스트에 대한 순차검색이면 $O(n)$이 소요됩니다. edge만 제거되는 것도 아니고 노드가 제거되는 경우도 있겠지요.