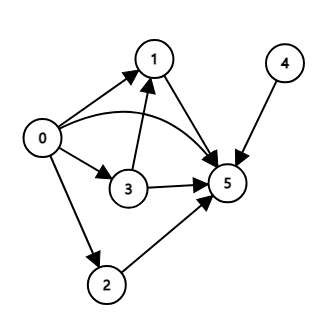
**고급소프트웨어실습I 보고서 1**

**20211584 장준영**

**A) Finding Celebrities**

유명인(Celebrity)는 그가 포함된 집단의 모두가 그를 알지만 본인은 아무도 모르는 사람을 일컫는다. N명의 사람이 포함된 집단에 대해, 집단의 사람들이 누구를 아는지에 대한 정보를 통해 유명인이 있는지, 있다면 누구인지 찾아내는 문제이다. 문제를 직관적으로 이해하기 위해, 이 문제 상황을 유향 그래프 자료구조로 표현할 수 있다.



*(※ 그래프는* [*https://csacademy.com/app/graph\_editor/*](https://csacademy.com/app/graph_editor/)*를 이용하여 그렸음.)*

위 그래프는 0부터 5까지의 고유 번호를 갖는 6명의 사람이 각각 누구를 알고 있는지 표현한 그래프이다. 그래프에 따르면, 0번 사람은 1번 사람을 알고, 4번 사람은 5번 사람을 알고 있다. 5번 사람은 나머지 모든 사람이 알고 있지만 본인은 아무도 모르기 때문에 유명인이다. 따라서, 문제 해결을 위해 해당 그래프로 표현된 관계에서 유명인을 찾아내는 방법을 떠올려야 한다. 이는 주어지는 입력에 따라 효율적인 해결 방법을 결정할 수 있다.

**[1] Edge가 입력으로 주어지는 경우**

위의 그래프를 기반으로, 사람의 수와 연결 정보(directed edge)가 입력으로 들어오는 경우를 생각할 수 있다. 입력은 다음과 같다.

6

0 1

0 2

0 3

3 1

4 5

1 5

2 5

3 5

0 5

(※ 스스로가 스스로를 가리키는 self-directed edge는 없다고 가정하고, 있다고 해도 입력 과정에서 손쉽게 removal이 가능하므로 제외한다.)

사람의 수 N과 사람의 고유번호에 따른 연결 정보(src node -> dst node)가 위와 같이 주어지면, 안으로 들어오는 indegree edge의 수 배열과 밖으로 나가는 outdegree edge의 수 배열을 각각 사용해 우리에게 필요한 연결 정보만 저장할 수 있다. 예를 들어, 3 5라는 edge 정보를 받으면 3의 outdegree를 1 증가시키고, 5의 indegree를 1 증가시키는 방식이다. 입력을 통해 만들어진 두 배열은 다음과 같다.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| idx | 0 | 1 | 2 | 3 | 4 | 5 |
| indegree | 0 | 2 | 1 | 1 | 0 | 5 |
| outdegree | 4 | 1 | 1 | 2 | 1 | 0 |

위 배열을 통해서 어떻게 유명인을 찾아낼 수 있을까? 유명인은 모두가 알지만 스스로는 아무도 모르는 사람이므로, 본인을 제외한 N-1개의 indegree edge가 있지만 0개의 outdegree edge를 갖는 노드가 될 것이다. 해당 방식을 사용하면 대략적인 시간 복잡도가 다음과 같다.

1) 입력 및 indegree, outdegree 배열 설정 : O(E)

입력되는 edge의 개수만큼만 연산하면 되므로 선형적인 시간복잡도이다.

2) 특정한 사람이 유명인인지 판정 : O(1)

완성된 두 배열을 통해 hashing 및 단순 값 판정만 하면 된다.

3) 유명인이 있는지, 있다면 누구인지 판정 : O(N)

한 집단에서 유명인이 두 명 이상이 될 순 없다. 따라서 outdegree가 0인 한 사람을 찾아 그 사람의 indegree가 N-1인지만 판정하면 된다. 이는 선형 탐색을 통해서 O(N)의 시간 복잡도로 해결할 수 있다.

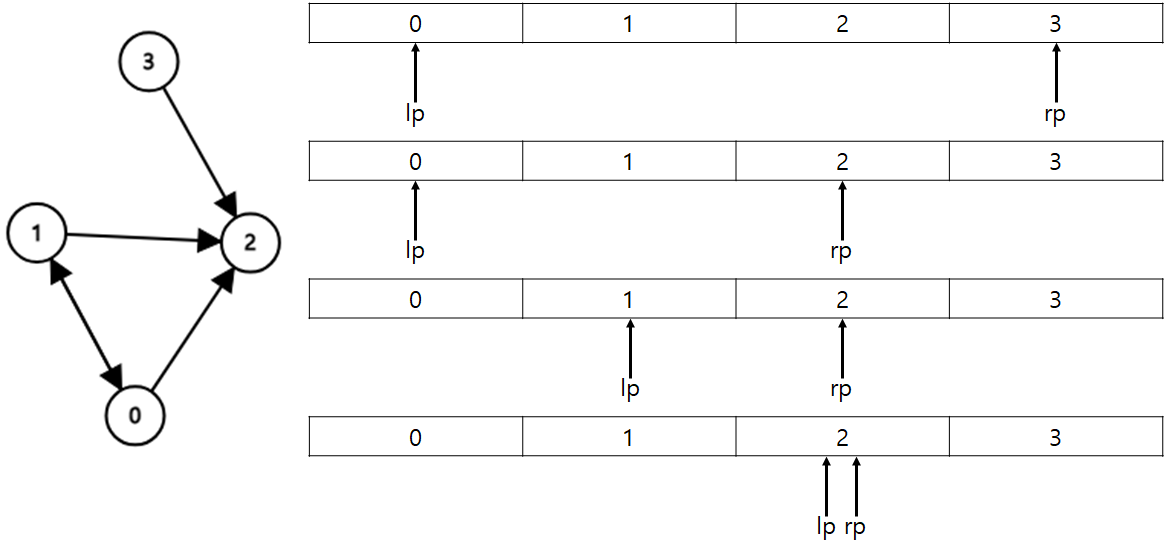
**>> 총 시간 복잡도 : O(N+E) ( ≒ O(E) ∵ Usually, N < E )**

Linear time complexity에 입력부를 제외한다면 O(logN)의 시간 복잡도로 충분히 해결 가능한, 입력부가 가장 큰 시간 복잡도를 갖는 최적의 알고리즘이 된다. 특히 특정한 사람이 유명인인지 아닌지 판정하는 것이 constant time complexity라는 장점이 있다. 또, 원리 자체는 brute-force 방식과 같기 때문에 가능한 모든 경우에 정해를 도출하는 reliable한 알고리즘이다. 하지만 입력이 위와 같이 상세하게 주어져야 한다는 constraint가 필요하다.

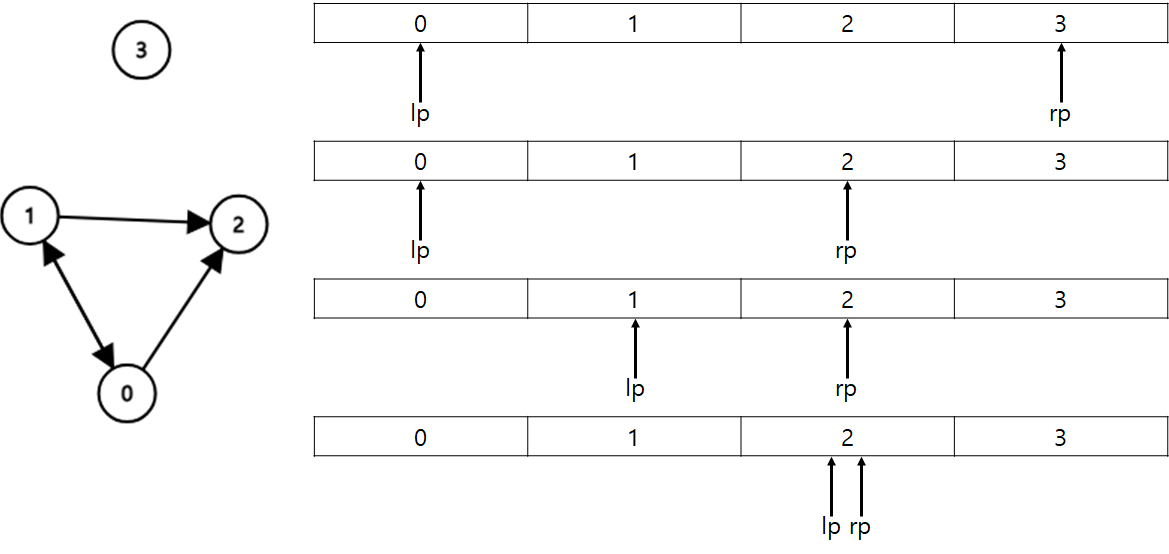
**[2] 그래프만 알 수 있는 경우**

Edge를 모두 제시하지 않고, adj list 또는 adj matrix의 형태로 그래프를 입력하는 경우 그래프에서 스스로 edge 정보를 도출해내야 한다. 특히, adj matrix 형태라면 N\*N 행렬을 모두 탐색해야 하므로 edge 도출의 시간 복잡도가 O(N^2)가 될 수 있다. 그 이후엔 **[1]**과 동일한 과정을 수행하면 된다.

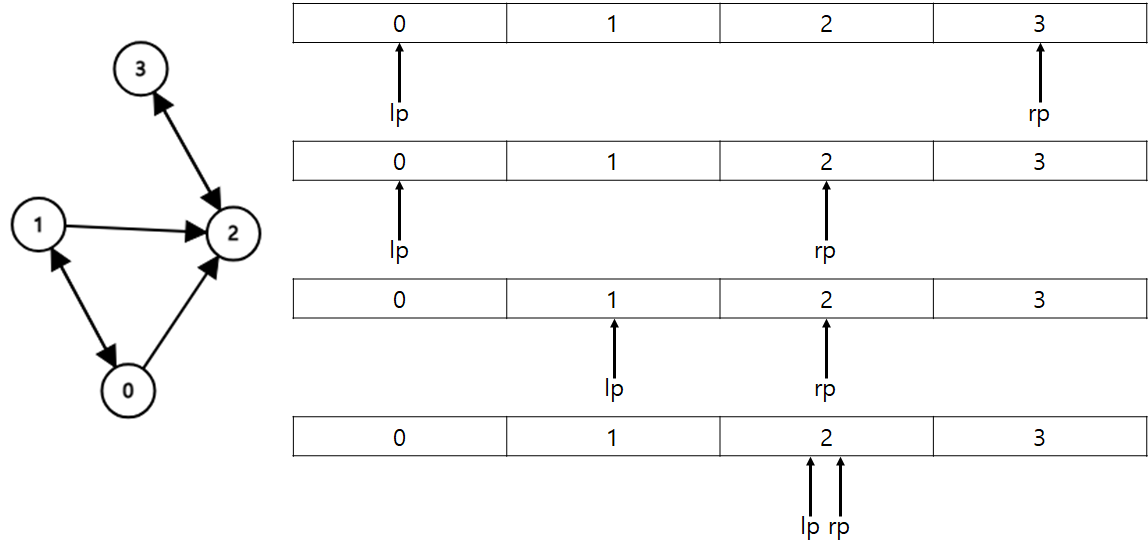
하지만, 이런 경우 굳이 indegree 배열과 outdegree 배열을 두어 O(N^2)의 시간 복잡도를 갖지 않고 two pointer를 응용하여 해결할 수 있을 거라고 생각하였다. i가 j를 알 경우와 모를 경우 두 가지가 경우의 수의 전체 집합인데, i가 j를 알 경우 i가 유명인 후보에서 제외되고, 모를 경우 j가 유명인 후보에서 제외된다. 따라서 모든 사람에 대해 유명인 후보가 아니라면 제외하고, 맞다면 다음 분기에서 검증하는 과정이 정해를 도출하므로 two pointer 알고리즘을 사용할 수 있을 것이다. 유명인이 존재하는 경우 two pointer는 다음과 같은 과정으로 움직인다.



lp가 rp를 알면 lp가 후보에서 제외되기 때문에 1 증가하고, 모르면 rp가 후보에서 제외되기 때문에 1 감소한다. 후보군을 줄여가면서, 최종적으로 lp와 rp가 같아지면 그 사람이 유명인이다. 하지만 two pointer를 사용하여 항상 정해를 도출할 수 있다는 것은 나의 추측이므로, 확인을 위해 유명인이 존재하지 않는 두 가지 경우에 대해서도 알고리즘을 동작시켜 보았다.

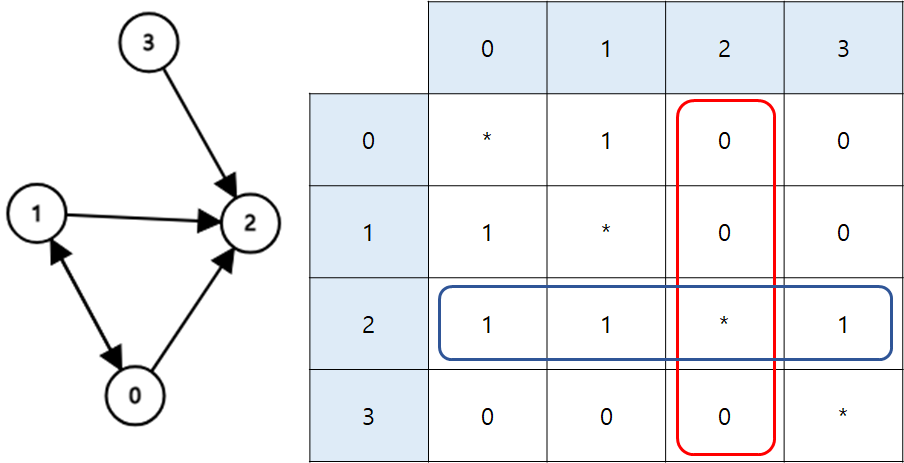


먼저, 유명인 후보를 모르는 사람이 있는 경우이다. 여전히 lp와 rp가 2로 수렴하여 2가 유명인이라고 판단할 수 있지만, 3의 경우 lp가 0인 경우에서 제외되었기 때문에 2에 대해서는 체크를 할 수가 없다. 따라서 유명인이 아닌 사람을 유명인이라고 하는, 오답을 도출한다.



다음으로 유명인 후보가 아는 사람이 있는 경우이다. 동일하게 2가 3을 알지만, 3은 0이 lp일 때 후보에서 제외되어 2가 체크할 수단이 없다. 오답을 도출한다.

이런 문제를 해결하기 위해 떠올린 수단으로는, 마지막까지 후보로 남은 인물이 있다면 그 인물에 대해서만 indegree edge와 outdegree edge를 조사하는 것이다. two pointer 방식은 adj matrix에서의 최적화를 위한 것이므로 adj matrix를 생각해보면, adjMatrix[candidateIdx][0~N-1]가 모두 0이 되는지와 adjMatrix[0~N-1][candidateIdx]가 모두 1이 되는지를 조사하면 된다. 만약 남은 후보가 이를 만족하면 유명인이고, 그렇지 않으면 그 집단은 유명인이 없는 집단이다.



이 방식을 통해 정해를 도출해내려면, lp와 rp가 수렴한 지점이 집단 내 유일한 후보가 확실하다는 것이 보장되어야 한다. two pointer를 움직이면서 절대 후보가 될 수 없는 사람들은 제외되고, 마지막까지 남은 후보는 일련의 과정에서 결격 사유를 단 한 번도 갖지 않은 사람이기 때문에 그 사람만이 후보가 될 수 있다. 따라서, two pointer를 위에서 설명한 방식으로 이용한다면 reliable한 알고리즘이다.

이 방식의 시간 복잡도는 다음과 같다. (입력부는 그래프를 입력받는 방식에 따라 다르므로 생략한다.)

1) two pointer를 통해 하나의 후보를 추려내는 과정 : O(N)

양 끝에서 시작해 한 점으로 수렴하므로, 선형 탐색을 한 번 하는 것과 같은 시간 복잡도를 가진다.

2) 후보에게 indegree edge, outdegree edge가 있는지 확인 : O(N)

본인을 제외한 다른 모든 노드에 대해 한 번만 선형 탐색하면 되므로 선형적이다.

**>> 총 시간 복잡도 : O(N)**

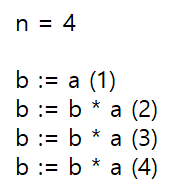
또 다시 linear time complexity가 나왔다. 충분히 짧은 시간에 해결이 가능하고, 위의 반례와 증명을 통해 reliable하다는 것도 밝혀졌다. 또, edge를 알지 못하고 adj matrix만 아는 경우에도 linear하게 해결할 수 있으므로 ‘일반적으로’ 좋은 알고리즘이다. 다만 degree array를 만드는 방식보다 구현이 조금 어렵고, 작동 방식이 떠올리기 직관적이지 않다는 단점이 있다. **[1]**의 방식과 다르게 그래프 정보를 저장해두어야 하기 때문에 공간 복잡도를 더 필요로 하는 단점도 있다.

**[3] 그래프를 알지 못하고 ‘물어보며’ 찾아야 하는 경우**

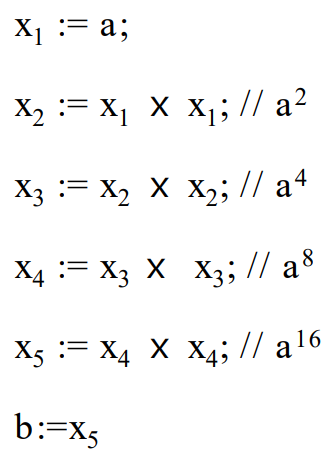
문제에서 설명한 그대로, 사람들에게 물어보며 유명인을 찾아야 하는 경우이다. 정확히 설명할 수 있을지는 모르겠지만, C++ 클래스의 private 인자로 그래프의 정보가 있고, public의 know(i, j) 함수를 호출하면 true 혹은 false를 return하는 느낌이라고 이해하였다. **[1]**의 방식만 떠올렸을 때는 모든 i, j의 조합을 brute-force(O(N^2)) 하여 degree array를 만들어야 한다고 생각하였는데, **[2]**를 고안하고 나니 그럴 필요가 없어졌다. i와 j가 연결되어 있는지 확인할 때 그래프를 직접 access 하지 않고, 물어보았을 때의 결과만 확인하면 되기 때문이다. 즉, **[2]**의 방식과 완전히 동일하게 작동할 수 있다. 따라서 시간 복잡도(O(N)) 및 장단점 또한 동일하다.

**B) Minimum Calculation**

할당 연산(:=)과 곱셉 연산(\*)만을 이용하여, 곱셈 연산을 최소화한 거듭 제곱 연산을 하는 것이 문제의 목적이다. 두 연산만 가능하기 때문에, math header의 pow 등의 함수나 행렬 등을 활용할 수 없고, 오로지 수를 여러 번 곱하여 거듭 제곱의 값을 구해야 한다. 기본적으로 떠올릴 수 있는 거듭 제곱 방법은 다음과 같다.

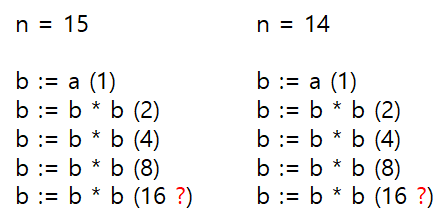


n이 4일 때의 예시이다. 거듭 제곱이라는 단어 그대로, 거듭해서 a를 곱해나가는 방식이다. 이 방식대로면 n-1번의 곱셈 연산만으로 거듭제곱을 마칠 수 있다. O(n-1)=O(n)의 시간 복잡도를 갖기 때문에 얼핏 보면 아무 문제 없이 이 방식을 사용해도 될 것 같지만, 거듭 제곱은 아주 기본적인 연산이기 때문에 코드 내에서 자주 반복되어 나올 수 있고, 반복문 안에 들어갔을 때 시간 복잡도가 n의 제곱 꼴이 되기 때문에 빠른 연산에 지장을 줄 수 있다. 이를 해결하기 위해 고안된 방법이 강의자료에 수록되어 있는 방법이다.



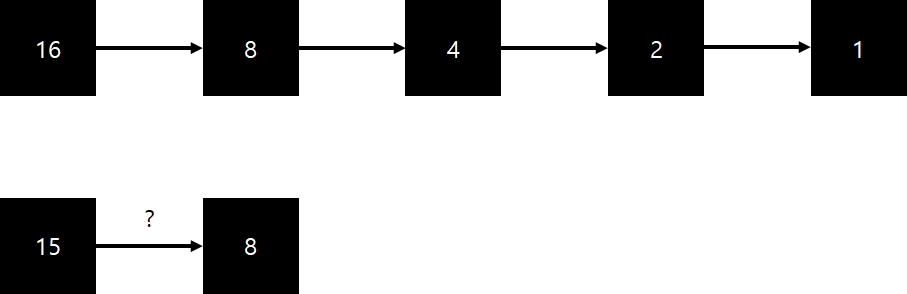
*(강의자료 4p)*

n이 16일 때의 예시이다. 우선 x1에 a를 할당하고, x2는 x1을 두 번 곱한 값을 할당한다. 이렇게 하면 x2는 a의 제곱이 된다. 그리고 x3에는 x2를 두 번 곱한 값을 할당한다. 이렇게 하면 x3은 a의 네제곱이 된다. … 이 방식을 활용하면 지수의 증가값이 산술적이 아닌 기하적으로 변하게 된다. 당연히 필요한 곱연산의 수도 로그 스케일로 줄어든다. 하지만 n이 16인 경우만 보아서는 이 방법을 일반화 할 수 없다.

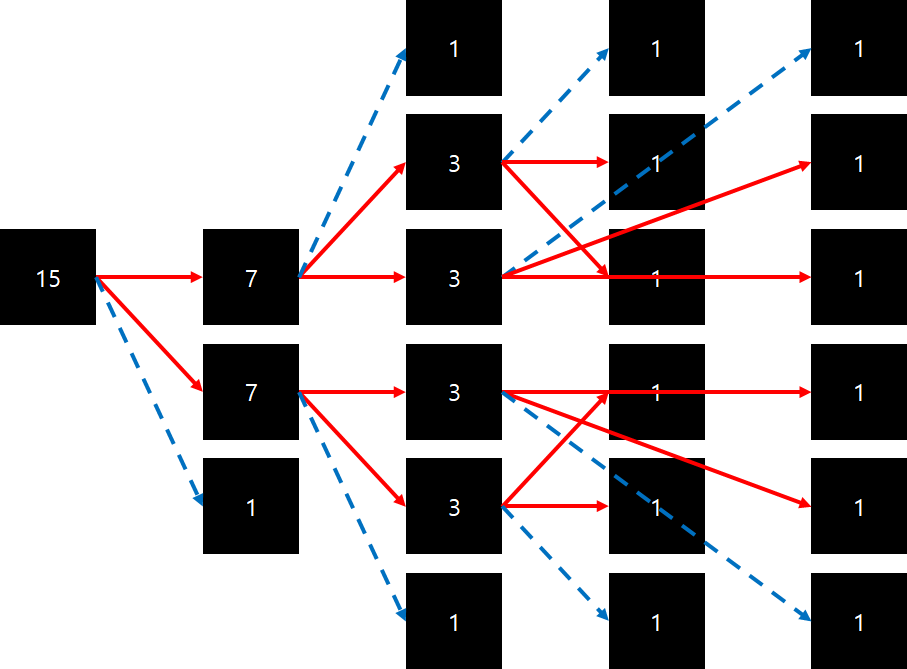


n이 15인 경우와 14인 경우를 보자. 15는 홀수를, 14는 2의 제곱꼴이 아닌 숫자를 통해 반례를 확인하기 위한 가정이다. 두 경우 모두 기하적으로 증가하는 8과 16 사이의 지수값을 갖길 원하는데, 막상 이 방식대로 곱하기만 하면 원하는 값을 얻을 수 없다. 그렇다고 몇 번 나누어서 값을 찾을 수도 없고, n을 소인수분해 할 수도 없다. 따라서, 곱셈만을 이용해 2의 제곱꼴이 아닌 숫자의 지수를 얻을 방법을 찾아야 한다.

핵심적인 아이디어는, ‘해결해야 되는 문제의 크기’인 ‘지수’를 절반으로 나누어 log scale의 시간단축을 이루는 데에 있다. 하지만 지수가 홀수가 되면 절반으로 나뉘지 않고, 목표 지수가 2의 제곱꼴이 아니라면 분명히 역연산 과정에서 한 번은 홀수의 지수를 만나게 된다. 이를 해결하기 위해 홀수를 짝수로 만들어 절반으로 나누는 과정이 필요하다는 아이디어를 떠올렸다.

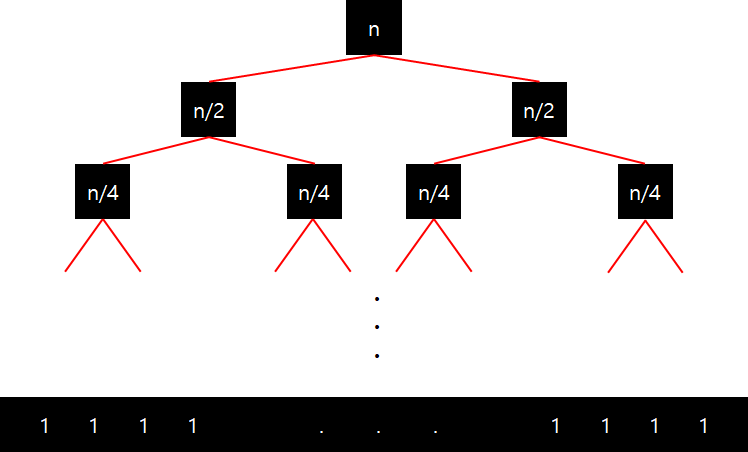


우리가 어떤 수를 거쳐가는지 알기 위해선 다음과 같은 역연산이 필요하다. 목표 지수로부터 절반을 나눠가면서 확인하는 것이다. 하지만, 15의 경우 1까지 도달할 수 있는 루트를 설정하기가 어렵다. 그렇기 때문에 원래의 수부터 시작해 제곱해나가는 과정에서 15에 도달할 수 없는 것이다.



하지만 역연산 과정에서 다음과 같은 아이디어를 얻을 수 있다. 크기를 절반으로 줄이기 위해서 1을 미리 빼두고 곱하는 것이다. 1은 원래의 숫자를 계산값에 한 번만 더 곱하면 얻을 수 있기 때문에, 문제의 스케일을 줄이는 데에는 지장이 없다. 역연산을 다시 역연산하여 원래 연산을 찾아보면, 목표 지수까지 전부 1에서 얻을 수 있는 값이 된다. 이처럼 홀수 지수 항을 만났을 때는 지수를 1 빼서 계산하면 모든 n에 대하여 일반화가 가능해진다. 이 역연산을 점화식으로 표현하면 다음과 같다.

점화식을 유심히 보면, 분할 정복하여 시간 복잡도를 낮출 수 있다는 점을 발견할 수 있다. 일반화된 재귀 트리를 그려보면, 트리의 깊이가 logn이 되어 총 연산의 수 또한 logn이 된다. 따라서, 시간 복잡도 O(logn)이면 O(logn)개의 곱연산만을 통해 거듭 제곱을 계산할 수 있다.



구현 방법으로는 재귀, 반복문, 바텀 업 DP 등이 있다.