<<알고리즘>>

1. 알고리즘: 문제를 해결하기 위한 절차
2. 컴퓨터로 문제를 푸는 과정
3. Problem: 문제 파악
4. Strategy: 전략 구성
5. Algorithm: 알고리즘 설계

Input, output에 따라 step별로 어떻게 할 것인지 설계

1. Analysis: 알고리즘 분석

기준: correctness(정확성), time, space, optimality(최적화된 방법인가)

1. Implementation: 수행
2. Verification: 확인
3. Analysis of the Algorithm
4. Correctness

Input이 output이 되는 과정을 증명으로 알고리즘이 정확하게 동작함을 확인.

1. Time & Space used
2. Basic operation

필요한 basic operation의 수.

1. Worst-case analysis

최악의 경우에 대한 분석

1. Average-behavior analysis

평균 case에 대한 분석

A(n) = P(succ)A(succ;n) + P(fail)A(fail;n)

: input size가 n인 문제에서 연산 수 = 문제를 성공할 확률\*성공했을 때 필요한 연산 수 + 문제를 실패할 확률\*실패했을 때 필요한 연산 수

Pr(n): 인풋 사이즈 n이 발생할 확률

평균 연산 수 = [0,N]의 범위의 n에 대하여 Pr(n)\*A(n)의 합.

!) 모든 case의 발생 확률이 같을 수 없기 때문에 모든 인풋의 발생 확률이 같다는 가정이 필요 함. (Pr(n)은 모두 같음)

1. Optimality (Simplicity)

Complexity와 실제 필요한 일의 양을 비교함으로써 효율성을 확인.

Complexity: 문제를 풀기 위한 최소한의 일의 양(연산 수)

예) n개의 수 중에서 특정 값을 찾는 문제. Complexity = n

Optimal: 가장 최적의 알고리즘. 이보다 더 뛰어난 것은 존재할 수 없음.

문제의 complexity와 알고리즘의 worst-case complexity가 같을 때 optimal algorithm.

Input과 output의 특성이 다르면 같은 문제여도 다른 알고리즘이 최적일 수 있음.

!) worst-case complexity가 더 커도 optimal일 수도 있음.

1. Asymptotic Growth Rate (점근 분석 시 사용)

 값으로 비교.

< ∞ 이면 *f* ∈*O*(*g*): big-O

> 0 이면 *f* ∈ Ω(*g*): big-Omega

= *c* and 0 < *c* < ∞ 이면 *f* ∈ Θ(*g*): big-Theta

= 0 이면 *f* ∈ *o*(*g*): little oh

= ∞ 이면 *f* ∈ *ω*(*g*): little omega

(limit 문제들 풀 수 있어야함.)

1. Abstract data type
2. Binary Tree
3. Stack
4. Queue
5. Priority Queue
6. Dictionary
7. Union-Find (Disjoint Sets)

교집합이 없는 집합들을 표현하는 자료구조. Union, find 2개의 연산만 가지고 있으면 union으로 집합을 합치고 find로 원소가 속한 집합의 root를 찾는다.

1. Sorting
2. Insertion Sort
3. 동작 방법

앞의 정렬 되어있는 부분에서 자신의 위치를 찾아 삽입하는 방식.

[0,n-1]에서의 i에 대하여 정렬 되어 있는 [0,i]사이에 i+1번째 원소의 자리를 찾는 과정을 반복.

1. Worst-Case Complexity: O(n^2)
2. Average Behavior: O(n^2)
3. 특징: 구현이 간단하고 n이 클수록 효율이 떨어짐. Selection sort나 bubble sort보다 안정적이고 빠름.
4. Selection Sort
5. 동작 방법

최솟값을 앞으로 맨 앞의 원소와 바꾸는 방식

[0,n]에서의 i에 대하여 [i,n]구간에서의 최솟값과 i번째 원소의 자리를 바꾸는 과정을 반복.

1. Worst-Case Complexity: O(n^2)
2. Average Behavior: O(n^2)
3. 특징: 제자리 정렬 알고리즘으로 별도의 추가 메모리를 필요로 하지 않음.
4. Bubble Sort
5. 동작 방법

인접한 두 개의 원소를 비교하여 자리를 교환하는 방식.

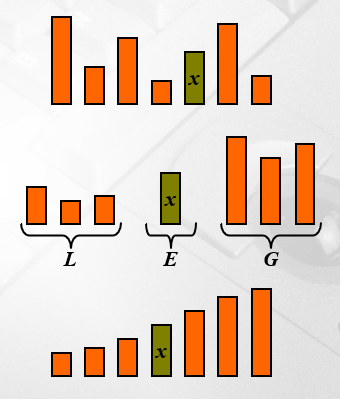
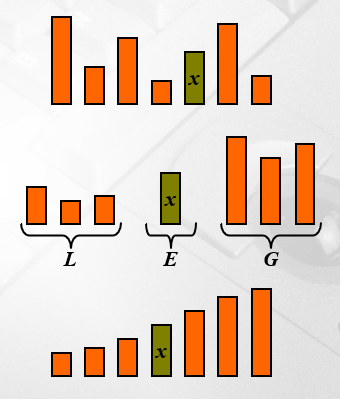
[0,n]에서의 i에 대하여 i, i+1을 비교하고 i+1과 i+2를 비교하고,… n-1과 n을 비교하는 과정을 반복.

1. Worst-Case Complexity: O(n^2)
2. Average Behavior: O(n^2)
3. Quick Sort
4. 동작 방법

pivot이라는 원소를 임의로 정하여 pivot을 기준으로 같은 원소들, 작은 원소들, 큰 원소들에 대하여 다시 알고리즘을 실행하는 **divide and conquer**전략을 사용하는 알고리즘.

이렇게 크기별로 나눠진 원소 묶음을 partition이라고 부름. (아래그림의 L, E, G)

Pivot을 정하는 방법, partition 구현 방법에 따라 다양한 버전의 quick sort가 존재. 보통은 그냥 맨 앞의 원소를 pivot으로 결정.

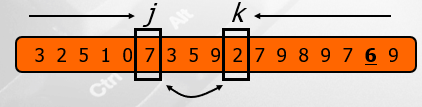
1. Worst-Case Complexity: O(n^2)
2. Average Behavior: O(n log n)
3. 특징: 배열로 구현하면 저장공간이 많이 필요하고 원소 접근시간이 줄어 듦. 반대로 linked-list로 구현하면 저장공간은 줄어들고 원소 접근시간이 많이 듦. Worst-case의 발생 확률이 매우 적다고 보이기 때문에 average case로 성능을 많이 얘기함.

Divide and Conquer 전략이지만 conquer 과정이 없는 알고리즘.

1. In-Place Quick-Sort

: quick sort의 한 버전. 주어진 배열 외에 상수 개만큼만 추가 메모리를 요구함.

오름차순으로 정렬한다고 할 때 배열의 양끝에서부터 안쪽으로 pivot과 비교를 하여 앞의 원소가 pivot보다 큰 원소와 뒤의 pivot보다 작은 원소를 swap하는 과정을 반복하여 파티션을 나눌 수 있게 함.

1. Merge Sort
2. 동작 방법

{0} 크기가 1이 하면 정렬된 것으로 본다. 그렇지 않으면 아래의 순서를 따른다.

{1} divide: 크기가 비슷한 두 부분으로 나눈다

{2} conquer: 각 부분에 대하여 재귀로 merger sort를 진행한다.

{3} combine: 정렬된 두 부분을 하나로 합병(merge)한다.

1. Worst-Case Complexity: O(n log n)
2. Average Behavior: O(n log n)
3. 특징: 점근 분석 복잡도 측면에서는 가장 optimal한 알고리즘.
4. Heap Sort
5. 동작 방법

Partial order tree property를 만족하는 complete binary tree인 max heap을 구성하여 값을 하나씩 뽑아내어 정렬하는 방법. (Partial order tree property: 부모와 자식 사이의 관계가 항상 성립. (부모>자식))

heap에서 n번의 insert후 n번의 delete와 fix-heap이 이루어진다.

{1} insert: 트리의 구조에 따라 채워져야 할 부분 (가장 아래 오른 쪽 자리)에 값을 넣고 부모와 값을 비교하여 부모보다 크면 swap하는 과정을 반복.

{2} delete: root자리에 있는 수가 가장 큰 수이므로 이 값을 지운다.

{3} fix-heap: 가장 아래 오른 쪽 노드의 값을 root자리로 옮긴 후 자식 노드와 비교하여 자식보다 클 때까지 swap하면서 내려간다.

1. Worst-Case Complexity: O(n log n)
2. Average Behavior: O(n log n)
3. Heap Implementation using an Array

Heap을 배열로 구현하여 수행할 수 있음. 부모의 인덱스가 i일 때 자식은 2i, 2i+1.

1. Radix Sort
2. 동작 방법

가장 낮은 자리 수(LSD: least significant digits) 부터 정렬해가는 알고리즘.

[0,k]에서의 i에 대해서 i번째 자리 수에 대하여 정렬하는 과정을 반복.

1. Worst-Case Complexity: O(w\*n) (w: 자릿수)
2. Average Behavior: O(w\*n)
3. 특징: 이 알고리즘은 correctness를 증명하기 위해서는 이 알고리즘이 stable(input의 sorting이 output에서 유지됨)함을 증명해야 함. 이 알고리즘은 input에 대한 다른 특성(자릿수 고정, 안정성이 보임)이 보일 때 사용.
4. Array Doubling

: 배열의 크기를 늘려야 할 때 원래 배열의 2배 크기를 할당 받아 원래의 배열에 있던 값들을 옮기는 작업. 값을 하나 옮길 때 t시간이 덜린다고 하면 size n의 배열에 n+1번째 값이 들어올 때 2n개의 새로운 메모리를 할당 받아 n개의 값들을 옮기고 n+1번째의 원소를 입력해야 한다. 이 작업에는 기존 n개의 원소를 받을 때 t\*n의 시간이, n개의 원소를 옮길 때 t\*n의 시간이 소모되어 총 2t\*n의 시간이 걸린다.

Array doubling이 중요한 이유는 1개읜 원소를 입력할 때 그 원소가 몇 번째로 입력된 것인가에 따라 t의 시간이 걸릴 수도 있고 ktn+1의 시간이 걸릴 수도 있기 때문이다. 이런 (발생할 수 있는 최악의 경우)측면을 고려한 분석방법이 amortized analysis이다.

1. Amortized Analysis

최악의 경우를 고려한 평균을 분석하는 것.

1. Amortized cost = actual cost + accounting cost

Stack에 값을 입력하는 것으로 예를 들어보자. 값을 옮기는 비용(transferring cost 사실상 stack이라 push, pop과 같음)는 t로 표기하다. Stack의 초기 메모리는 n이라고 가정하자. doubling없이 pop과 push는 1의 시간이 걸린다고 가정한다.

Stack의 push에 대한 amortized cost를 구해보자.

1. Doubling 없을 때 actual cost: 1, accounting cost: 2t
2. Doubling이 있을 때 actual cost: t\*n + 1, accounting cost: -t\*n + 2t
3. Amortized cost = 1+2t = t\*n + 1 – t\*n + 2t = 1+2t

Actual cost는 실제로 필요한 비용이고 accounting cost는 최악의 경우(이 예시에서는 doubling이 일어나는 경우)에 지불하게 될 비용이다. 이 예시에서는 값을 옮기기 옮기는 비용(transferring cost)을 말한다.

1. Red-Black Tree

자료구조에서 언급된 AVL Tree의 한 종류. Balanced Binary Tree이다. 이 tree의 특징으로는 각 노드는 red or black으로 색이 정해져 있다는 점이다. 왼쪽 자식<부모<오른쪽 자식 순으로 원소의 크기를 갖는다.

* AVL Tree: 모든 부모 노드에 대하여 모든 자식 tree의 높이의 차이가 1이하인 tree

1. Property
2. Root Property: root는 black이다.
3. External Property: leaf는 black이다.
4. Internal Property: red node의 자식은 모두 black이다.
5. Depth Property: 모든 lead node들은 모두 같은 black depth를 가진다.

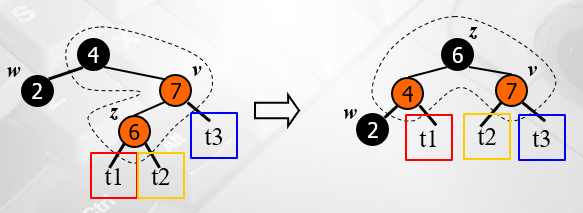
(black depth: 현재 node부터 root까지의 경로에서 존재하는 black node의 수)

* 가장 깊은 leaf node의 depth는 가장 얕은 leaf 노드의 depth의 2배 이하이다.
* n개의 node가 있을 때 tree의 높이는 2log n 이하로 O(log n)이 된다.
* 이 tree에서 search를 하게 된다면 O(log n) 시간이 걸린다.

1. Insert

Tree의 목적과 특징에 맞게 원소를 자리에 red node를 삽입한 뒤 값을 입력한다. 이때 삽입한 node의 부모가 red node이면 위의 property에 위배되므로 tree의 구조나 node들의 색을 변경시키는 작업이 필요하다.

1. Restructuring: 삽입한 부모의 sibling node가 black일 때 (z가 삽입한 node)



삽입한 node와 부모 node, 부모의 부모 node의 크기를 비교하여 위 그림처럼 배치하고 크기에 맞게 자식 tree들을 배치한다.

1. Recoloring: 삽입한 부모의 sibling node가 red일 때



위 그림처럼 부모와 부모의 sibling node와 부모의 부모 node의 색을 바꾼다.

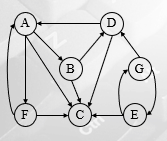
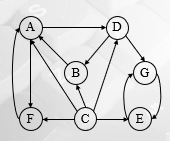
삽입된 node뿐만 아니라 tree의 변경 후에도 구조나 색이 변경된 node들에 대해서 다시 property를 위해 하지 않는지에 따라 restructuring이나 recoloring작업이 필요할 수 있다.

값을 삽입하는 과정은 node를 추가하는 데에 O(1), restructuring이나 recoloring이 최대 tree의 높이 O(log n)만큼 이뤄질 수 있으므로 총 O(log n)의 시간이 걸린다.

1. Strongly Connected Components (SCC)

: digraph의 subgraph로 component에 속한 모든 정점은 다른 모든 정점으로 갈 수 있음.

1. 찾는 방법
2. 그래프에서 DFS를 통해 방문한 정점들을 순서대로 stack에 쌓는다.
3. Stack에 쌓인 원소 순으로 역방향 그래프에서 DFS를 시작한다. 탐색에서 갈 수 있는 원소는 같은 component에 속한 원소이다.

[순방향 그래프] [탐색 순서를 담은 스택] [역방향 그래프]

위 그림에서는 (E, G), (A, B, C, D, F), (C) 가 SCC로 존재한다.

1. Greedy Algorithm

항상 최고로 보이는 것을 선택함으로써 비용을 줄이거나 이득을 최대화하는 optimization 기법

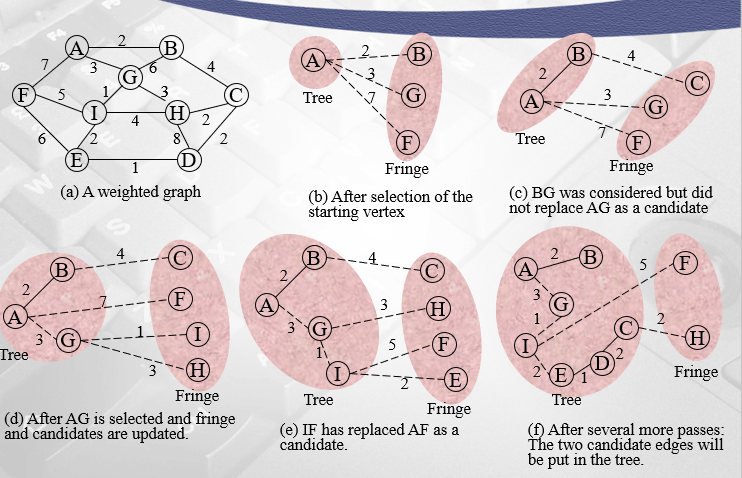
1. Minimum Spanning Tree using Greedy Algorithm
2. Minimum Spanning Tree: weighted graph에서 어느 node에서든 모든 node로 가는 경로가 존재하되 존재하는 경로의 가중치 합이 가장 작은 subgraph.
3. Prim’s Algorithm (자료구조에서도 나와있음. 같은 내용에 부가적인 내용 추가.)

{1} 시작 단계에서는 시작 정점만이 MST(최소 비용 신장 트리) 집합에 포함된다.

{2} MST에 속한 정점들이 갈 수 있는 정점을 Fringe라고 한다. Fringe에는 정점과 함께 MST에서의 거리를 저장해 놓는다. Fringe에 속한 정점들 중에 MST에서 가장 가까운 정점을 MST에 추가하고 Fringe를 갱신한다.

(가장 가까운 정점을 고른다는 것이 greedy한 것.)

{3} 앞의 과정을 트리가 모든 정점을 가질 때까지 반복한다.



Tree에 들어올 원소의 수가 n개이고 각 step에서 Fringe의 크기는 최대 n이므로 O(n^2)의 시간이 걸린다.

1. Kruskal’s Algorithm (자료구조에도 나와있음)

{1} 그래프의 간선들을 가중치의 오름차순으로 정렬한다.

{2} 정렬된 간선 리스트에서 순서대로 사이클을 형성하지 않는 간선을 선택한다. 사이클을 형성하는 간선은 제외한다.

(가중치가 낮은 간선을 우선적으로 선택하게 되는 점이 greedy.)

{3} 해당 간선의 두 정점이 연결되어 있지 않다면 현재의 MST의 집합에 추가한다.

(Union-Find(disjoint-set)을 사용하면 구현에 용이함.)

O((n+m)log n)의 시간이 걸린다.

1. Shortest path using Greedy Algorithm
2. Dijkstra’s Algorithm (알고리즘 설명은 생략)

Dijkstra에서도 priority queue나 heap을 통해 가중치가 낮은 간선을 먼저 탐색하는 greedy optimization을 사용한다.

1. Floyd-Warshall

간선에 가중치가 있는 graph에서 모든 정점에서 모든 정점으로 가는 최단 경로의 가중치를 구하는 알고리즘. (알고리즘의 설명은 생략함. 스터디에서 했음.)

1. Dynamic Programing

반복되는 작업에 대한 값을 미리 저장하여 비용을 줄이는 알고리즘 전략. (자세한 설명은 스터디에서 했기 때문에 생략함)

1. String match (Pattern match)

: 주어진 string인 text T에 pattern P의 존재여부를 matching해보는 문제. T의 길이를 n, P의 길이를 m이라고 하자. (간략하게 알고리즘 설명을 하긴 하지만 이해하기 어려울 수도 있음. KMP도 그렇고 Boyer-Moore도 한번에 이해하기 쉬운 알고리즘은 아님.)

1. Brute-force algorithm

T의 모든 원소에서 P를 비교하는 알고리즘. O(n\*m)의 시간이 걸린다.

1. Knuth-Morris-Pratt algorithm (KMP)

T의 어느 지점에서 P가 매칭되지 않을 때 다시 탐색을 시작할 부분을 찾아 효율적으로 문제를 풀 수 있는 알고리즘.

1. Failure function: prefix(접두사)와 suffix(접미사)가 같을 때 최고 길이.

F(j): P의 prefix of P[0, j]와 suffix of P[1, j]가 같을 때의 제일 긴 길이.

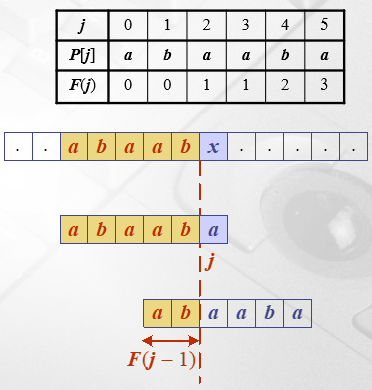
{0} F(0) = 0, i = 1, j = 0

{1} P[i]와 P[j]가 같으면 F(j) = j+1이 되고 i와 j에 1씩 더해준다.

{2} P[i]와 P[j]가 같지 않고 j>0이라면 j를 F(j-1)로 바꿔준다.

{3} {1,2}에 모두 해당하지 않는다면 F(i) = 0이고 i는 1더해준다.

{4} i<m일 동안 {1,2,3}과정을 반복한다.

<= failure 값을 채우는 예시.

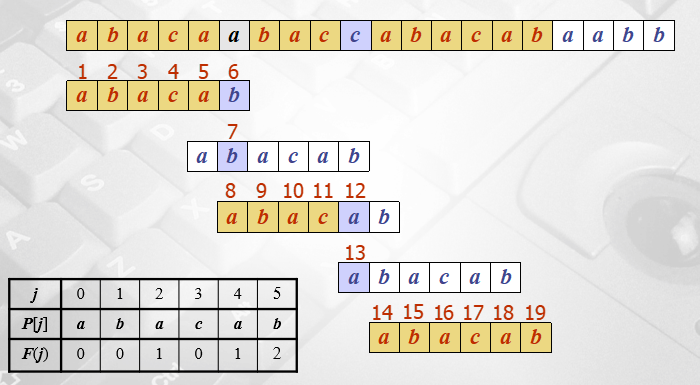
1. 알고리즘 동작

{0} P의 failure 값을 모두 구한다. i =0, j= 0

{1} T[i]와 P[j]가 같을 때 j = m-1이라면 match된 것이고 j가 m-1이 아니라면 더 봐야하기 때문에 i와 j를 1씩 늘려준다.

{2} T[i]와 P[i]가 같지 않을 때는 j>0이라면 j = F(j-1)로 바꿔주고 j가 0이라면 i를 1 더해준다.

{3} i<n일 동안 {1,2}과정을 반복한다. 반복후에 match된 것이라는 결과가 나오지 않는다면 match되지 않은 것이다.



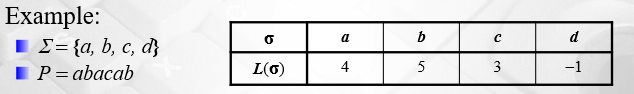
1. 알고리즘 분석

KMP알고리즘은 문제의 optimal time인 O(n+m)에 동작한다. P를 미루면서 T와 비교하는 것이기 때문이다. Failure를 구할 때 O(m)이 들고 matching에서는 O(n)이 드는 것이다.

1. Boyer-Moore Algorithm

(!) 이 알고리즘은 문자열을 뒤에서부터 매칭한다.

1. L function: 해당 문자가 마지막에 나타나는 인덱스를 찾는 함수. 모두 구하기 위해서는 문자의 종류가 s개라고 할 때 O(s)만큼 걸린다. (L(a) = 4 이런 식.)



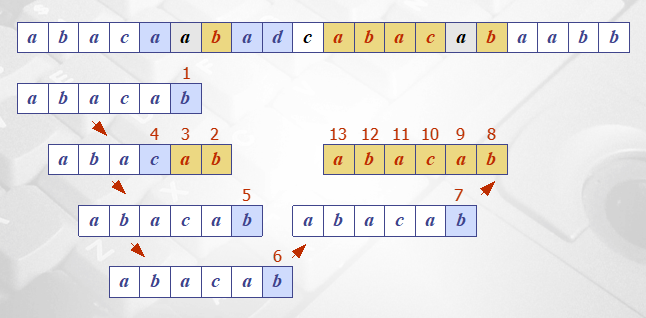
1. 알고리즘 동작

{0} P에 대하여 L function값을 모두 구한다. i = m-1, j = m-1

{1} T[i]와 P[j]가 같을 때 j =0이면 매칭된 것이고 j가 0이 아니면 더 봐야하기 때문에 i와 j를 모두 1씩 뺀다.

{2} T[i]와 P[j]가 같지 않다면 i = i+m-min(j, L(T[i]) + 1)로 j = m-1로 바꾼다.

{3} i>n-1일 동안 {2,3}을 반복한다. 반복 중에 match됨이 나오지 않는다면 match된 것이 아니다.



1. 알고리즘 분석

최악의 경우 O(n\*m + s)이지만 경우에 따라 글자들을 더 많이 건너뛰면서 KMP보다 빠른 평균 속도를 보여준다. 문자의 종류가 적은 DNA매칭과 같은 문제에서 비효율적이다.

1. NP-Complete Problems
2. Complexity of a Problem: 문제의 복잡도란 문제를 풀기위해 필요한 최소 연산 수(복잡도)를 말한다. Optimal(best) algorithm의 복잡도와 같다. 정렬되지 않은 배열에서 최대값 찾기에서는 *Ω*(*n*), 배열을 정렬하기에서는 *Ω* (*n* log *n*)이다.
3. P class: Polynomial Time

O(n^k)(k는 상수)에 해결 가능한 문제를 P problem이라고 한다.

1. NP class: Non-Deterministic Polynomial Time

O(n^k)에 verifiable한 문제를 NP class에 속한다고 말한다.

1. Verifiable: 말그대로 검증이다. 답이 문제에 적합한지 판단하는 것이다.

예를 들어 해밀턴 경로 문제를 보자

해밀턴 경로 문제는 주어진 그래프에서 모든 node를 지나가는 경로를 찾는 문제다. 이때 verify하는 문제는 입력으로 주어진 경로가 해밀턴 경로인지 확인하는 작업이다.

1. NP-hard

NP문제보다 어려우며 모든 NP문제를 polynomial time에 reduction(환원)가능한 문제를 NP-hard problem이라고 한다.

1. Reduction: 쉬운 문제를 어려운 문제로 바꿀 수 있다는 것을 의미한다.

예를 들어 A라는 문제는 n개의 원소를 정렬하는 문제라 하고 B라는 문제는 n개의 원소에서 가장 큰 값을 찾는 문제라고 하자. 이 예시에서 A가 B보다 난이도가 어렵다. 그 이유는 A문제를 푸는 알고리즘으로 B를 풀 수 있기 때문이다. 이럴 때 문제 B를 문제 A로 reduction(환원) 가능하다고 한다.

1. NP-Complete class

NP-hard에도 포함되고 NP 문제에도 포함되는 문제를 NP-complete라고 한다. 즉 NP문제들 중에서 가장 어려운 문제이다.

NP-complete 문제들 중에 단 하나라도 polynomial time(다항 시간)안에 풀어낼 수 있다면 P=NP임을 증명할 수 있다.

해밀턴 경로 문제, SAT, 외판원 문제, 그래프 색칠 문제, knapsack 문제 등이 이 class에 속한다.

(!) 검색해보면 이런 문제들을 다항시간에 풀어낸 것 같은 알고리즘들을 찾을 수 있으나 그런 시간 복잡도를 pseudo-polynomial time이라고 부른다. 항상 이런 문제를 풀 수 있는 알고리즘이 아니라 주어진 입력의 표면상의 값에 대해서만 다항시간에 가능하다는 의미이다. (Polynomial time은 주어지는 입력의 비트 길이에 대해 다항 시간이다.) 이런 pseudo-polynomial time에 가능한 문제를 weakly NP-Complete라고도 한다.

