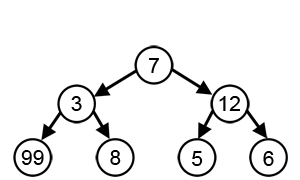
1. **그리디 알고리즘** (욕심쟁이 알고리즘, Greedy Algorithm)

매 선택에서 **지금 이 순간 당장 최적인 답**을 선택하여 적합한 결과를 도출하는 [알고리즘](https://namu.wiki/w/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98) 설계 기법

2. 특징 https://gamedevlog.tistory.com/60

가. 순간 최적 선택한 결과가 전체 문제의 최적이라는 보장이 없다.

나. 현재의 선택이 앞으로 남은 선택에 어떤 영향을 끼칠지는 고려하지 않는다.

**\* 아니, 단점이 더 큰데 왜 써요? 장점은요?**

다. **빠르다**, 탐욕 알고리즘을 사용해도 항상 최적해를 구할 수 있는 경우,

동적 계획법보다 수행 시간이 훨씬 빠르다.

라. **적당히 괜찮은 답을 구할 수 있다**, 시간이나 공간적 제약으로 인해 최적해를 찾기

어렵다면 적당히 괜찮은 답을 찾는 것으로 타협할 수 있다.

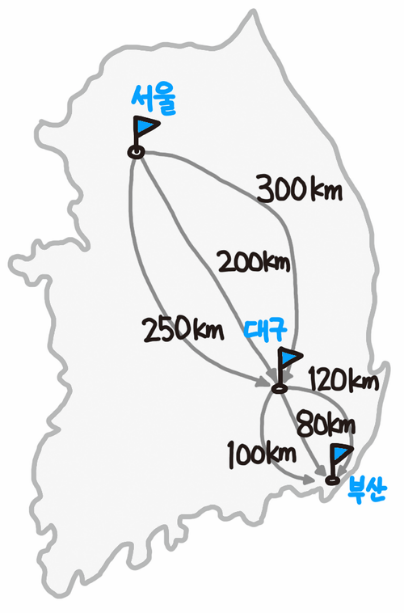
이때, 탐욕법을 사용하면 임의의 답 보다는 더 괜찮은 답으로 사용될 수 있다.

라. 항목 참고 : **휴리스틱(heuristic)**  **'경험에 의거한' 문제 풀이 기법으로, '대충 어림 짐작해서 풀기'**

항상 최적의 답을 찾아내지는 못하지만, 근사값 등을 통해 어느 정도 현실에 가까운 답을 빨리 찾기 위한 용도로 사용된다. (최적화 용도로 사용) **A\* 알고리즘 (에이스타)**

**심심풀이 미로찾기 방법 3가지** <https://www.youtube.com/watch?v=YS4ng_vKr7o>

**심심풀이 파이썬 코드 잘 짜는 법** [**https://www.youtube.com/watch?v=fJeGAx27-vU**](https://www.youtube.com/watch?v=fJeGAx27-vU)

3. 동작에 Fit한 경우

가.  **탐욕 선택 속성** **(greedy choice property)**

1) **한 번의 선택이 다음 선택에 전혀 무관한 값이어야 한다.**

2) DP와 가장 큰 차이점, 이전으로 돌아가 재고하는 과정이 없다.

나.  **최적 부분 구조** **(optimal substructure)**

1) **매 순간의 최적해가 문제에 대한 최적해여야 한다.**

4. 그리디 알고리즘 문제 풀이 방식

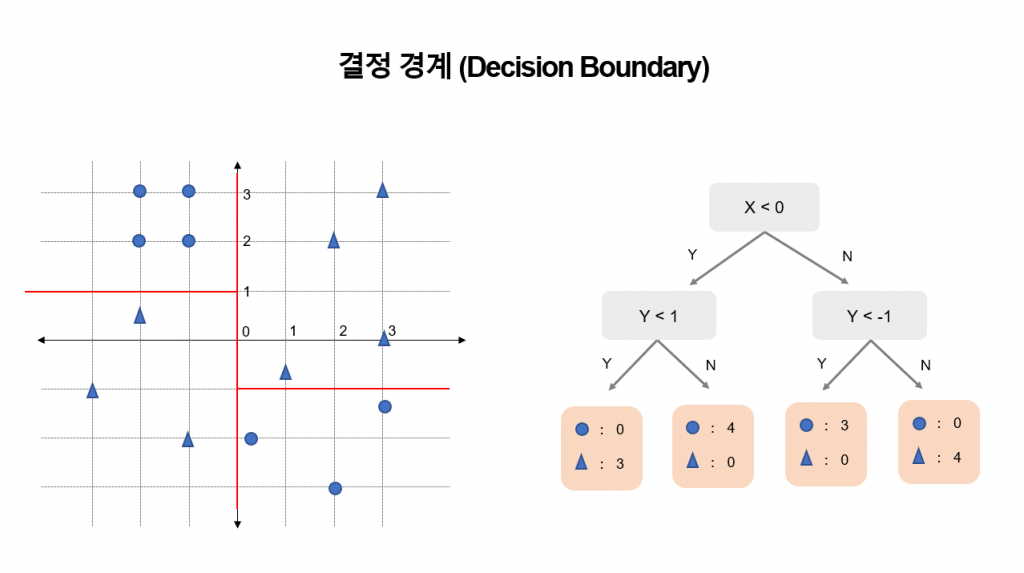
가. 문제의 답을 만드는 과정을 여러 조각으로 나눈다.

나. 각 조각마다 어떤 우선순위로 선택할지 결정 (작은 풀이로 insight 획득)

다. 두 가지 속성 증명 (greedy choice property, optimal substructure)

5. 어디에 쓸까?

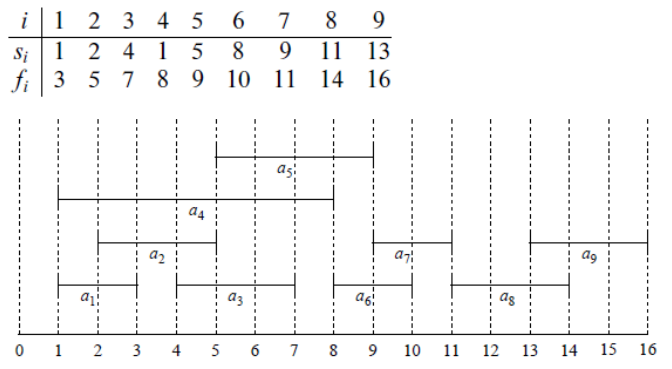
가. 결정 트리 학습법 (Decision Tree – 머신러닝)



나. 활동 선택 문제 (Activity selection problem)

한 번에 하나의 활동만 처리할 수 있는 하나의 강의실에서 제안된 활동들 중 가장 많은

활동을 처리할 수 있는 스케줄을 짜는 문제

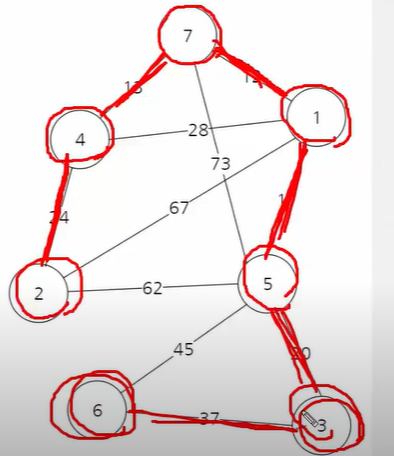


다. 거스름돈 문제

500원, 100원, 50원, 10원

라. 최소 신장 트리 (Minimum spanning tree) – 크루스칼 알고리즘

간선 수 = 노드 수 – 1 <https://www.youtube.com/watch?v=LQ3JHknGy8c> 4분 50초부터



최소 비용 신장 [트리](https://namu.wiki/w/%ED%8A%B8%EB%A6%AC(%EA%B7%B8%EB%9E%98%ED%94%84))를 O(ElogV) 만에 구하는 [알고리즘](https://namu.wiki/w/%EC%95%8C%EA%B3%A0%EB%A6%AC%EC%A6%98)

(= 크러스컬 이라고도 부른다)

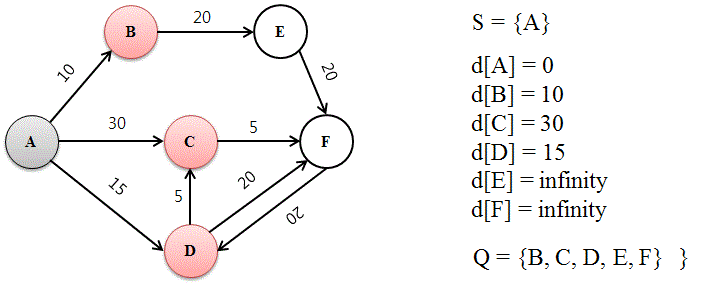
전체 그래프에서 노드가 연결되지 않은 얘들의 가장 최소 비용의 간선 선택

마. 제약 조건이 많은 대부분 문제

항상 그런 것은 아니지만, 프로그래밍 문제를 풀 때 제약조건이 많다면 대부분 그리디로

풀리는 경우가 많다. 다만 그리디인 줄 알고 풀었다가 피보는 경우도 있다.

바. 다익스트라 알고리즘 <https://www.youtube.com/watch?v=tZu4x5825LI&t=249s>



**1. 다익스트라 알고리즘은 아직 확인되지 않은 거리는**

**전부 초기값을 무한으로 잡는다.**

**2. 루프를 돌려 이웃 노드를 방문하고 거리를 계산한다.**

**3. 첫 루프 이후의 그래프의 상태가 업데이트되는 방식**

**4. 더 빠른 경로를 발견한 경우 : 값 업데이트**

사. 허프만 코드 <https://www.youtube.com/watch?v=haXz9MEOGbo> 4분 28초부터

허프만 알고리즘에 의해 생성된 최적 이진코드를 말한다.

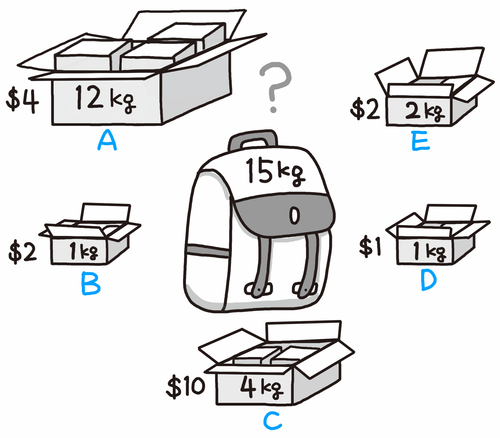
허프만 알고리즘(Huffman's Algorithm)은 허프만 코드에 해당하는 이진트리를 구축한다.

5-2. 최적값을 구하는데 실패하는 문제

가. 외판원 순회 문제 (TSP, Traveling Salesperson problem)



나. 배낭 문제 (Knapsack Problem)



6. 그리디 vs 동적 계획법(DP) vs 분할정복 (지난 시간에 배운 기출문제)

가. 그리디 알고리즘

1) 경우의 수가 무한 (재귀적 점화식 불가)

2) 한 번 선택하면 번복하지 않음 (그 순간 최적 선택)

3) 동적 계획법보다 빠른 성능

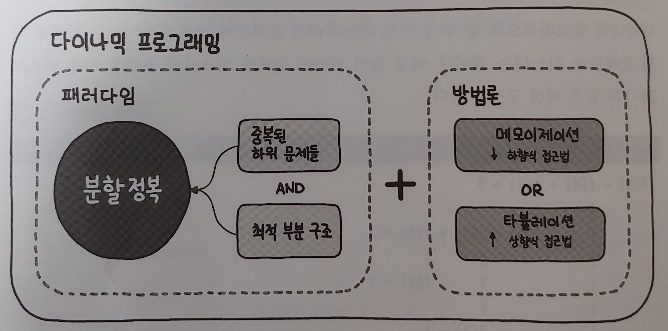
나. 동적 계획법 : 계산 정보를 DP 테이블에 담아두고, 다음 층 계산시 활용하는 방법

1) 재귀적인 점화식 먼저 설계, 조건을 만족하면 캐싱

2) 재귀적인 점화식의 기본적인 조건? 재귀가 종료되는 case가 존재

3) 모든 경우의 수를 따져본다

4) **문제의 해가 중복되는 하위문제 해를 사용하여 풀 수 있음**



def fibo(x):

if x == 1 or x == 2:

return 1

return fibo(x-1) + fibo(x-2)

**DP 메모제이션(캐싱) 기법**

**d = [0] \* 50 # 답 저장할 공간**

def fibo(x):

if x == 1 or x == 2:

return 1

**if d[x] != 0: # 이미 답이 있다면**

**return d[x] # 저장된 값 불러옴**

d[x] = fibo(x-1) + fibo(x-2)

return d[x]

다. 분할 정복 : 직접 해결할 수 있을 정도로 간단한 문제가 될 때까지 문제를 재귀적으로 나누고,

그 하위 문제의 결과를 조합하여 원래 문제의 결과로 만들어 낸다.

**그리디**

탐욕 선택 속성

최적 부분 구조

다익스트라

허프만 코딩 알고리즘

ID3 알고리즘 (머신러닝)

**동적 계획법**

점화식 (중복된 하위 문제)

최적 부분 구조

DP 테이블, 캐싱 기법

배낭 문제

동전 바꾸기 문제

가장 큰 합

피보나치

다익스트라

**분할 정복**

최적 부분 구조

동일한 부분 문제 반복 계산X

병합정렬, 퀵정렬