

Algorithms with Python

2024

Last updated: August 31 2022

Algorithms with Python¹



[컴퓨터공학부](#)

¹ 한국외대 컴퓨터공학부의 <알고리즘설계와해석> 등의 강의 내용을 바탕으로 고급 기법을 추가해 정리한 것임.

² 신찬수, chansu@gmail.com - 이 교재는 한국외국어대학교 컴퓨터공학부 신찬수 교수의 강의교재임. 저자의 허락을 받고, 출처를 밝히고, 수정 하지 않고, 상업적 목적이 없는 개인 학습용으로만 사용해야 함.

³ 일부 내용은 신찬수의 유튜브 채널 ([Chan-Su Shin](#))에 동영상으로 올려져 있음.

* 이 자료에서 인용한 이미지와 일부 내용은 wikipedia 등에 공개된 것임. 본 교재의 2019년 1차 버전은 **Algorithms in Python**이라는 이름의 책 (ISBN: 979-11-968591-3-8)으로 (비매품으로) 출판되었음.

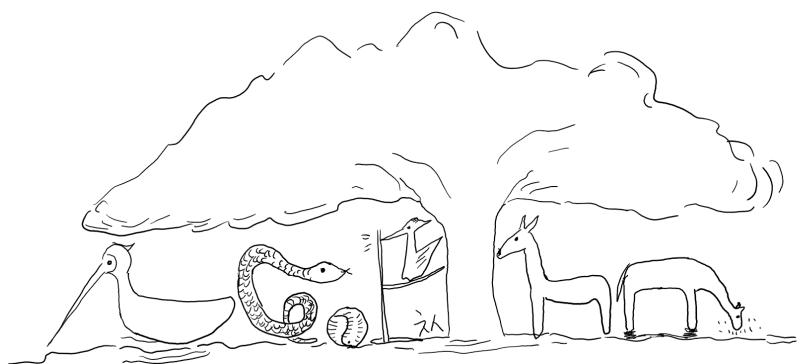
[저자 생각] 정식 출판을 염두에 두지 않고 정리한 내용으로 대충 그린 그림과 불친절한 설명에 다양한 형태의 오탏과 오류가 범벅된 책입니다. 그럼에도 시간과 정성을 들여 3년 넘게 숙성해 얻은 결과물로 알고리즘에 조금이라도 다가오고 싶은 분들에게 조금이나마 도움이 되었으면 합니다.

algorithm

noun

Word used by programmers when they do not want to explain what they did.

somewhere on the internet



알고리즘 목차

- 시작하기: 파이썬, 알고리즘 교재 소개

1. 알고리즘 (Algorithm), 자료구조 (Data Structures), 복잡도 (complexity)

- a. 가상 컴퓨터, 가상 언어, 가상 코드 (virtual computer/language/code)
- b. 알고리즘 (시간) 복잡도(complexity), Big-O 표기법

2. 재귀 (Recursion) 맛보기

- a. 재귀란? (예제 중심)
- b. 재귀 알고리즘의 수행시간 - 점화식 (recurrence relation)

3. 선택 알고리즘 (Selection Algorithm)

- a. 최소/최대값 동시 찾기, 가장 큰 두 수 찾기
- b. k번째로 작은 수 (큰 수) 찾기:
 - i. Quick select algorithm
 - ii. Median of Medians (MoM) algorithm
- c. [고급] 선택 문제의 비교횟수 하한 증명 방법 - Adversary Argument

4. 분할정복 알고리즘 (Divide-and-Conquer Algorithm)

- a. 피보나치 수 계산, 이진탐색
- b. 큰 수 곱셈(Karatsuba algorithm)
- c. 점화식(Recurrence relation) 풀이
- d. 인터뷰 문제와 응용 문제

5. 정렬 알고리즘 (Sorting Algorithm)

- a. 기본 정렬 알고리즘: selection, insertion, bubble 정렬 알고리즘
- b. 빠른 정렬 알고리즘: quick, heap, merge 정렬 알고리즘
- c. 비교 기반 정렬 알고리즘의 하한(lower bound)
- d. radix 정렬 알고리즘
- e. 정렬과 선형 탐색을 이용한 문제 풀이

6. 동적계획법 알고리즘 (Dynamic Programming Algorithm)

- a. 피보나치 수
- b. 최대구간합 찾기: 3가지 방법 - 반복, 분할정복, 동적계획법
- c. zig-zag 수열 찾기
- d. 최장 공통 부수열 (LCS: Longest Common Subsequence) 찾기
- e. 괄호 치기 (Matrix Multiplication)
- f. 최장 증가 부수열 (LIS: Longest Increasing Subsequence) 찾기

7. 그리디 알고리즘 (Greedy Algorithm)

- a. 동전교환 (Coin exchange)
- b. 강의실배정 (Activity selection)
- c. 허프만 코드 (Huffman code)

8. 백트래킹 알고리즘(Backtracking Algorithm)

- a. 미로탈출, N-queens 문제
- b. Subset sum 문제
- c. 0/1-knapsack 문제 (+ fractional knapsack 문제)

9. 그래프 알고리즘 (Graph Algorithm)

- a. 그래프 정의와 표현법, 탐색법(DFS vs. BFS)
- b. DAG(Directed Acyclic Graph)와 위상 정렬(topological sort)
- c. 최단경로 알고리즘 (shortest path algorithm)
 - i. One-to-All: Bellman-Ford algorithm, Dijkstra algorithm
 - ii. All-to-All: Floyd-Warshall algorithm
- d. 최소신장트리 알고리즘 (minimum spanning tree algorithm)
 - i. Kruskal 알고리즘
 - ii. Prim 알고리즘
- e. 네트워크 플로우 알고리즘 (network flow algorithm)
 - i. Ford-Fulkerson algorithm
 - ii. Edmonds-Karp algorithm

10. 문자열 알고리즘 (String Algorithm)

- a. Knuth-Morris-Pratt (KMP) 알고리즘
- b. Rabin-Karp 알고리즘

11. [*] 기하 알고리즘 (Geometric Algorithm)

- a. 세 점이 이루는 방향 검사
- b. 가장 가까운 두 점 찾기
- c. 볼록 헬 (convex hull)
- d. 보로노이 다이그램 (Voronoi diagram)

12. [*] FFT (Fast Fourier Transform)를 이용한 다항식 곱셈 (polynomial multiplication)

- a. 다항식 표현법 세 가지
- b. DFT (Discrete Fourier Transform), Inverse DFT
- c. 응용 문제 5가지:
 - i. 다항식 곱셈, 큰 정수 곱셈, X+Y, 3SUM, 문자열 매칭 등

13. [*] 계산 모델, 복잡도, P, NP, NP-hard, NP-complete 소개

- a. 계산 모델 (computation model)과 복잡도 (computational complexity)
- b. 상한 (upper bound)과 하한 (lower bound)
- c. P vs. NP
- d. 변환 (reduction)
- e. NP-hard 문제와 NP-complete 문제
- f. NP-complete 문제 공략 방법 - Randomization, Approximation

14. [*] 알아두면 유용한 기법

- a. 비트 연산 활용 (bit manipulation)

시작하기

1. 왜 파이썬으로 알고리즘을 코딩하면 좋을까?

- a. 직관적이라 알고리즘의 pseudo 코드와 매우 유사해 생산성이 높다
- b. 내부, 외부 모듈이 다양하고 확장성이 좋아 응용 분야의 프로그래밍에 유리하다
 - 데이터베이스 응용, 데이터 분석/처리/시각화, 머신러닝, 웹 프로그래밍
- c. 학습에 도움이 되는 (무료/유료) 동영상 자료가 넘쳐난다 (아래 자료들 참고)

2. 단점은 없나? (당연히 있다. 세상에 공짜 점심은 없다)

- a. 인터프리터 언어이기에 출 단위로 컴파일하고 실행하기에 수행 시간이 느리다
 - C, C++: source code (xxx.c, xxx.cpp) → (compiler) → 실행 파일 (xxx.exe)
 - 1. CPU가 exe 파일을 직접 실행하기에 상대적으로 빠름
 - Python: source code (xxx.py) → (interpreter) → 출 단위로 interpreter가 실행
 - 1. CPU는 interpreter를 실행하고 interpreter가 출 단위로 실행하는 형식으로 상대적으로 느림
 - java: compiler 방식과 interpreter 방식을 혼용해 동작하기에 둘 사이에 위치
- b. 비교 예: 2048 x 2048 크기의 2차원 행렬 두 개를 곱하는 3중 for 루프를 C, java, Python으로 구현해 CPU 시간을 비교한 내용
 - [medium.com에 게시된 글에서 인용](https://medium.com/@geekydev/running-time-comparison-between-c-java-and-python-for-matrix-multiplication-10333a2a2a2) (C 코드는 -o3 옵션 사용)

Language	Running time	Absolute Speed Up
Python	2821.108	1
Java	84.612	33.34
C	49.958	56.47

c. 그럼 빠르게 실행할 수 있는 파이썬 코드를 작성하는 건 어렵나?

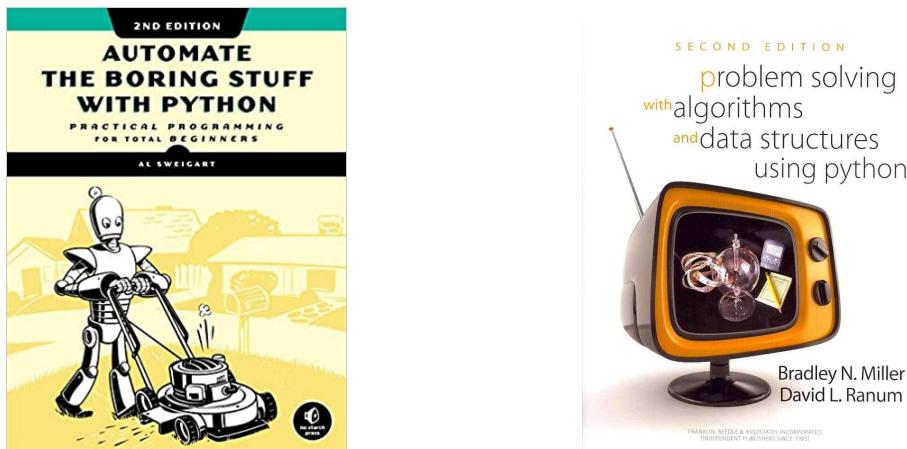
- Python3를 주로 사용하는데, 이 버전의 interpreter는 CPython인데, 새로운 interpreter인 PyPy3를 사용하면 많은 경우에 더 빠른 실행 시간을 얻을 수 있다
- 그 이유는 자주 사용되는 코드 조각을 cache에 저장해서 다시 interpreter를 사용하지 않고 재사용하기 때문이다. (대신 메모리는 더 많이 사용) 따라서 간단한 코드는 Python3로, 반복되는 부분이 많은 복잡한 알고리즘 코드는 PyPy3로 작성하면 된다
- PyPy3: <https://www.pypy.org/> 참조
- 교재에 나오는 모든 코드는 Python3.6x를 기준으로 작성되었다

3. 충분히(!) 참고할 가치가 있는 자료들

- a. 공식 파이썬 홈페이지: www.python.org
- b. 동영상 무료 강의 사이트(기초):
 - 생활코딩 - Python/Ruby 강의: <https://opentutorials.org/course/1750>
 - programmers - Python 강의: <https://programmers.co.kr/learn/courses/2>
- c. 영어 튜토리얼 사이트:
 - tutorialspoint: <https://www.tutorialspoint.com/python3/index.htm>
 - diveintopython3: <http://www.diveintopython3.net/>

4. 참고용 교재 (Python + Data Structures + Algorithms)

- a. [Python] 점프 투 파이썬, 박응용 지음, 이지스퍼블리싱 (다음 링크에서 책의 내용을 모두 읽을 수 있음: <https://wikidocs.net/4321>)
- b. [Python] [Automate The Boring Stuff with Python](#), Al Sweigart (가장 평가가 좋은 Python 책 중 하나)



- c. [알고리즘+Python] [Problem Solving with Algorithms and Data Structures Using Python](#), Bradley N. Miller and David L. Ranum (2013 version is available online [here](#), 웹 사이트 [here](#))
- d. [자료구조+알고리즘의 바이블] [Introduction to Algorithms](#), T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, MIT Press (3rd Edition) (기초부터 높은 수준까지 친절하게 정리된 알고리즘의 바이블! 국내 번역본도 출간되었지만 비쌈)
- e. [알고리즘의 강의 노트, paperback] Algorithms, Jeff Erickson (University of Illinois at Urbana-Champaign) 교수의 (거의 완벽한) 강의 노트. 종이 책으로도 출간되어 아마존에서 구입 가능하지만, PDF 파일로도 자유롭게 다운로드해서 공부할 수 있도록 허용 ([다운로드](https://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf): <https://jeffe.cs.illinois.edu/teaching/algorithms/book/Algorithms-JeffE.pdf>)

f. [코딩 인터뷰 문제 모음]

- [Cracking the Coding Interview: 189 Programming Questions and Solutions](#)
(6th Edition), Gayle Laakmann McDowell
- [Elements of Programming Interviews in Python: The Insiders' Guide](#) 2016,
Adnan Aziz, Tsung-Hsien Lee, Amit Prakash
- 구글, 아마존 등을 포함한 유명 해외 기업체 코딩 문제를 모아 단순한 알고리즘부터
가장 빠른 알고리즘까지 다양한 방법으로 해결하는 과정을 친절하게 설명해 놓은 책.
코드는 java로 되어 있음. 한글 번역본도 출간되어 있지만, 비쌈
- 이와 유사한 문제+정답 모음집 (무료) PDF 파일도 있음
<https://www.programcreek.com/wp-content/uploads/2012/11/coding-interview-in-java.pdf>

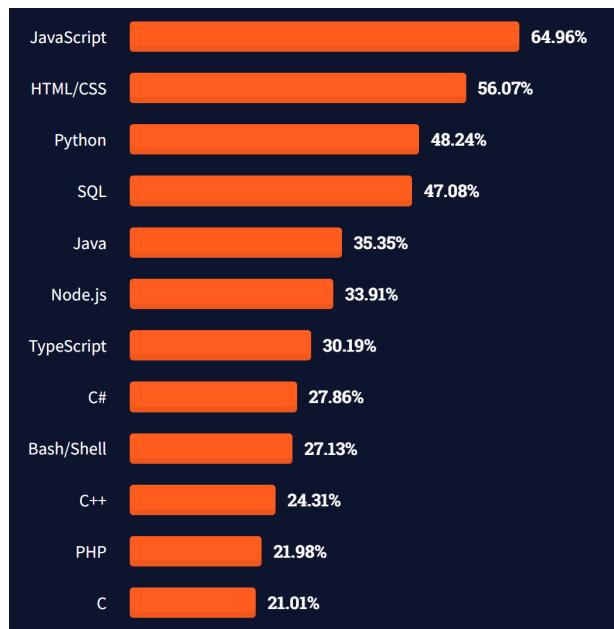
5. Python은?

- a. 1989년에 [Guido van Rossum](#)에 의해 설계, 제안
- b. 고수준 -- 범용성 -- 대화형 -- 객체지향 -- 동적타입을 지원하는 언어
- c. 어느 정도 인기가 있을까?
 - TOIBE 프로그래밍 언어 랭킹, IEEE Spectrum 프로그래밍 언어 랭킹

TOIBE Index for Programming Languages (2020-2021)

Dec 2021	Dec 2020	Change	Programming Language	Ratings	Change
1	3	▲	 Python	12.90%	+0.69%
2	1	▼	 C	11.80%	-4.69%
3	2	▼	 Java	10.12%	-2.41%
4	4		 C++	7.73%	+0.82%
5	5		 C#	6.40%	+2.21%
6	6		 Visual Basic	5.40%	+1.48%
7	7		 JavaScript	2.30%	-0.06%
8	12	▲	 Assembly language	2.25%	+0.91%
9	10	▲	 SQL	1.79%	+0.26%
10	13	▲	 Swift	1.76%	+0.54%
11	9	▼	 R	1.58%	-0.01%
12	8	▼	 PHP	1.50%	-0.62%

Stackoverflow 2021 survey (전 세계 개발자 5만여 명을 포함한 8만여명의 조사 결과)



IEEE Spectrum 2018 조사

Language Rank	Types	Spectrum Ranking
1. Python	🌐💻📱	100.0
2. C++	📱💻📱	98.4
3. C	📱💻📱	98.2
4. Java	🌐📱💻	97.5
5. C#	🌐📱💻	89.8
6. PHP	🌐	85.4
7. R	💻	83.3
8. JavaScript	🌐📱	82.8
9. Go	🌐💻	76.7
10. Assembly	📱	74.5

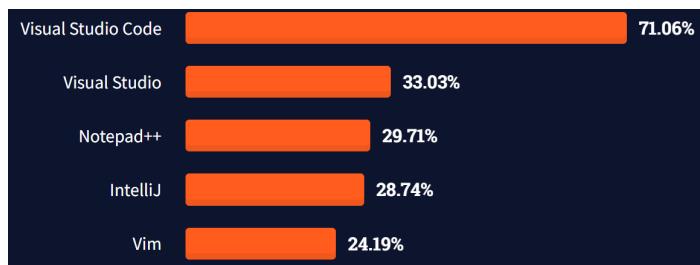
6. 교재에 나오는 파이썬 코드는

- a. **version 3.6.x** 이상이면 실행 가능
- b. Window 운영체제 기준으로 설명함
- c. [python download site](#) - 2022년 7월 기준 최신 버전은 3.10.5임
- d. 다운 받아 설치하는 데, Install Now 화면의 가장 아래 두 개의 체크박스 모두 체크함
 - Install launchers for all users (recommended)

- Add Python 3.x to PATH

7. 코딩 환경

- a. 에디터와 터미널을 따로 사용하는 방법
 - 코드 작성을 위한 에디터: notepad++ (기본 설치된 메모판도 가능) [[download](#)]
 - 코드 실행을 위한 터미널: Powershell, cmd shell (둘 다 Windows 기본 내장)
- b. 에디터와 터미널이 함께 제공되는 IDE - 통합 개발 환경을 사용하는 방법
 - IDLE: python 패키지에서 제공하는 내장된 IDE (파이썬 설치할 때 함께 설치됨)
 - **vscord** (Visual Studio Code ) [[download](#)] (**강력 추천!**)
 - atom [[download](#)]
 - 참고: Stackoverflow 2021 survey에서 조사한 IDE 사용률



8. 강의노트 기호

- a. [👍] - 꽤 중요한 설명, [?] - 점검용 질문, [💪] - 강의 내용에 따른 실습 또는 구현문제
 - b. [⌚] - 알고리즘 관련 (매우 유용한) 퍼즐
 - c. [🎤 인터뷰 문제] - IT 기업 사원 채용 인터뷰에서 등장하는 알고리즘 문제
 - d. [💡 코딩 대회 문제] - 프로그래밍 대회에서 등장하는 꽤 난이도 있는 문제
9. [💪] [해보기 1]
- a. 윈도우 하단 왼쪽의 에 Powershell 또는 cmd를 입력해서 명령 창 띄우기
 - b. 명령 창에서 python 입력하고 엔터
 - c. print("hello") 입력 후 엔터 (결과 확인)
 - d. exit() 또는 Ctrl+C 입력 후 python 종료
10. [💪] [해보기 2]
- a. 적절한 장소에 **Algorithm**이라는 과목용 디렉토리 생성
 - b. 디렉토리 창의 파일 메뉴에서 Windows Powershell 열기 클릭 (powershell이 열림. 또는 윈도우 왼쪽 아래 검색 창에 cmd 입력후 명령 프롬프트 선택해 실행)
 - c. 명령 창에서 notepad 또는 (설치했다면) notepad++ 입력 후 실행
 - d. 새 파일에 print("hello") 한 줄을 입력 후 hello.py 파일 이름으로 저장
 - e. 명령 창에서 python hello.py 입력 후 결과 확인

11. 만약, powershell에서 명령을 인식하지 못하는 경우는 해당 프로그램의 위치를 윈도우가 모르기 때문이다. 예를 들어, notepad++를 제대로 인식하지 못하는 경우:

- a. 내 PC 우클릭 → 속성 → 고급 시스템 설정 → 환경변수 → 사용자 변수 리스트에서 path 더블 클릭 또는 새로 만들기 → notepad++.exe가 있는 디렉토리 path 입력

12. IDE: **vscode** (강력 추천:  [Download Visual Studio Code - Mac, Linux, Windows](#) 다운)

- a. extension에서 Python extension을 검색해서 install해도 되고, vscode가 필요한 extension을 추천할 때 설치해도 됨
- b. File → New File 선택해 hello.py 파일 새로 만든 후, 첫 줄에 print("hello") 입력
- c. Terminal → New Terminal 선택하면 cmd 또는 Powershell 터미널 생성됨. 터미널에 명령 python hello.py 입력하여 실행해 봄
- d. 장점: 직관적이며 다양한 extension을 설치해서 사용할 수 있는 인터페이스, 거의 모든 언어 지원, git 명령 지원, Jupyter 에디터도 지원

13. IDE: Anaconda (<https://www.anaconda.com/download/>에서 다운)

- a. Jupyter Notebook 이란 웹 코드 에디터도 함께 설치됨

14. IDE: atom (<https://atom.io/>에서 다운 받아 설치)

- a. File → Settings → Install
 - platformio-ide-terminal 과 script 두 패키지 검색 후 install
- b. platformio-ide-terminal 패키지:
 - 하단 왼쪽에 + 버튼을 누르면 cmd 창이 열림
 - python 코드를 에디터 창에서 작성한 후, cmd 창에서 실행
- c. script 패키지
 - Shift + Ctrl + b 를 누르면 현재 편집되는 python 코드를 실행시켜 줌

15. 기본 cmd shell 명령들

- a. **dir or ls**: 디렉토리에 있는 다른 디렉토리와 파일을 화면에 출력함
- b. **cd A**: 현재 디렉토리에서 A 디렉토리로 이동함
 - .: 현재 디렉토리, ..: 부모 디렉토리, ~: 루트 디렉토리
- c. **mkdir A**: 새로운 디렉토리 A를 현재 디렉토리 안에 새로 생성함
- d. **move A B**: 파일 A를 새로운 디렉토리 또는 새로운 파일 B로 바꿈 (rename/move)
- e. **copy A B**: 파일 또는 디렉토리 A를 파일 또는 디렉토리 B로 복사함
- f. **del A or rm A**: 파일 A를 지움 (A가 디렉토리라면 A의 내용이 없어야 함)
- g. **more A**: 파일 A의 내용(텍스트)을 화면에 출력함
- h. reading:
<http://www.cs.princeton.edu/courses/archive/spr05/cos126/cmd-prompt.html>
- i. 위/아래 화살표: 전/후 명령어 리스트 탭색, 탭 기능

16. 강의 노트에 등장하는 외부 이미지나 자료 등은 wikipedia 등의 GFDL 또는 CC-BY-SA license 조건에서 인용된 것이다

알고리즘

1. Algorithms, Data Structures, Time Complexity

- 🔥 알고리즘(algorithm)

- 알고리즘의 어원은 9세기 페르시아(이란-이라크) 수학자 Al-Khwarizmi의 라틴어 이름인 **algorismus**와 수를 나타내는 그리스어 **arithmos**가 섞여 만들어졌다는 게 정설이다[[Wiki](#)]
- 아래 사진은 사우디아라비아 Jeddah에 위치한 **KAUST** 대학의 컴퓨터과학과가 위치한 빌딩의 이름 (직접 촬영했음!)



- 알고리즘은 문제의 입력(input)을 수학적이고 논리적으로 정의된 연산과정을 거쳐 원하는 출력(output)으로 변환/계산(computation)하는 절차이고, 이 절차를 C나 Python과 같은 언어로 표현하면 프로그램(program) 또는 코드(code)가 된다
 - 입력은 배열(array(C) 또는 list(Python)), 연결리스트(linked list), 트리(tree), 해시테이블(hash table), 그래프(graph)와 같은 자료의 접근과 수정이 빠른 자료구조(data structure)에 저장된다
 - 자료구조에 저장된 입력 값을 기본적인 연산(primitive operation)을 차례로/반복적으로 적용하여 원하는 출력을 계산한다

- 인류 최초의 알고리즘

- 그리스 수학자로 기하학의 아버지로 알려진 Euclid의 유명한 저서인 “Elements”(BC. 300)에 설명된 최대공약수 (Greatest Common Divisor: GCD)를 계산하는 알고리즘이 최초라고 알려져 있다
- 과학사 연구가들은 Euclid 이전의 Pythagoras 학파의 결과나 70여년 전의 다른 수학자의 결과가 더 앞서 있을 수도 있다고 주장한다
 - https://en.wikipedia.org/wiki/Euclidean_algorithm#Historical_development

c. Pseudo code (엄밀한 코드가 아닌 설명 중심의 코드란 뜻으로 실제로 실행되지 않는 코드)

```
algorithm gcd(a, b):
    while a*b != 0 do    # 두 수 중 하나가 0이 되기 전까지 빼기 연산 반복
        if a > b
            a = a - b
        else
            b = b - a
    return a+b           # 두 수 중 0이 아닌 수가 gcd이므로 합 자체가 gcd임
```

- 큰 수에서 작은 수를 빼는 과정을 큰 수가 작은 수보다 작아질때까지 반복하고 큰 수와 작은 수의 역할이 바뀌어 이 과정을 반복해서 작은 수가 0이 될 때까지 진행된다
- 여기서 큰 수가 작은 수보다 작을 때까지 작은 수만큼을 줄이게 된다는 것은 결국 큰 수를 작은 수로 나눈 나머지를 구하는 과정과 같다. 결국, 반복해서 뺄 것이 아니라 단순히 (큰 수 % 작은 수)를 하면 된다는 것을 알 수 있다

```
algorithm gcd(a, b):
    while a*b != 0 do
        if a > b
            a = a % b      # %는 나머지 연산자
        else
            b = b % a
    return a+b
```

d. 위의 코드를 Python으로 변환해보자

- i. **algorithm** gcd(a, b) → **def** gcd(a, b):
 - ii. **while** a*b != 0 **do** → **while** a*b != 0:
 - iii. **if** a > b → **if** a > b:
 - iv. **else** → **else**:
- e. 만약 $a > b$ 라면, $\text{gcd}(a, b)$ 는 $\text{gcd}(a-b, b)$ 이 동일하다. 또한, $\text{gcd}(a, b)$ 는 $\text{gcd}(b, a \% b)$ 와 같다. (왜?) 이 점을 이용하면 gcd 함수를 재귀함수로도 작성할 수 있다. 예 : $\text{gcd}(16, 6) \rightarrow \text{gcd}(6, 4) \rightarrow \text{gcd}(4, 2) \rightarrow \text{gcd}(2, 0)$

```
algorithm gcd(a, b):
    if a*b == 0: # 바닥 조건 또는 탈출 조건
        return a+b
    if a > b: return gcd(b, a%b)
    else: return gcd(a, b%a)
```

- f. 결국, 두 수의 gcd 계산은 세 가지 방법 (알고리즘) - 뺄셈 계산 반복, 나머지 계산 반복, 재귀 계산 방법이 존재한다 → 뺄셈 방법과 나머지 방법에 대한 수행 시간의 자세한 분석은 19페이지를 참조한다

● 최초의 프로그래머는?

a. Ada Lovelace (1817 - 1852)



Ada Lovelace

Number of Operations.		Statement of Results.		Data.		Working Variables.		Result Variables.																							
Name of Operation.	Value of Operands.	Temporary working results.	Indication of change in value on any Variable.	V_1	V_2	V_3	V_4	V_5	V_6	V_7	V_8	V_9	V_{10}	V_{11}	V_{12}	V_{13}	V_{14}	V_{15}	V_{16}	V_{17}	V_{18}	V_{19}	V_{20}	V_{21}	V_{22}	V_{23}	V_{24}	V_{25}			
1. $\times V_1 \times V_2 \rightarrow V_3$	$V_1 = 1, V_2 = 2$	$V_3 = 2$		—	2	n	2n	2n	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—				
2. $- V_3 - V_4 \rightarrow V_5$	$V_3 = 2, V_4 = 1$	$V_5 = 1$		—	—	—	2n-1	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—				
3. $+ V_5 + V_6 \rightarrow V_7$	$V_5 = 1, V_6 = 2$	$V_7 = 3$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
4. $- V_7 - V_8 \rightarrow V_9$	$V_7 = 3, V_8 = 1$	$V_9 = 2$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
5. $\times V_9 \times V_{10} \rightarrow V_{11}$	$V_9 = 2, V_{10} = 2$	$V_{11} = 4$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
6. $- V_{11} - V_{12} \rightarrow V_{13}$	$V_{11} = 4, V_{12} = 1$	$V_{13} = 3$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
7. $+ V_{13} + V_{14} \rightarrow V_{15}$	$V_{13} = 3, V_{14} = 1$	$V_{15} = 4$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
8. $- V_{15} - V_{16} \rightarrow V_{17}$	$V_{15} = 4, V_{16} = 1$	$V_{17} = 3$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
9. $\times V_{17} \times V_{18} \rightarrow V_{19}$	$V_{17} = 3, V_{18} = 2$	$V_{19} = 6$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
10. $- V_{19} - V_{20} \rightarrow V_{21}$	$V_{19} = 6, V_{20} = 1$	$V_{21} = 5$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
11. $+ V_{21} + V_{22} \rightarrow V_{23}$	$V_{21} = 5, V_{22} = 1$	$V_{23} = 6$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
12. $- V_{23} - V_{24} \rightarrow V_{25}$	$V_{23} = 6, V_{24} = 1$	$V_{25} = 5$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
13. $\times V_{25} \times V_{26} \rightarrow V_{27}$	$V_{25} = 5, V_{26} = 2$	$V_{27} = 10$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
14. $- V_{27} - V_{28} \rightarrow V_{29}$	$V_{27} = 10, V_{28} = 1$	$V_{29} = 9$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
15. $+ V_{29} + V_{30} \rightarrow V_{31}$	$V_{29} = 9, V_{30} = 1$	$V_{31} = 10$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
16. $- V_{31} - V_{32} \rightarrow V_{33}$	$V_{31} = 10, V_{32} = 1$	$V_{33} = 9$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
17. $\times V_{33} \times V_{34} \rightarrow V_{35}$	$V_{33} = 9, V_{34} = 2$	$V_{35} = 18$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
18. $- V_{35} - V_{36} \rightarrow V_{37}$	$V_{35} = 18, V_{36} = 1$	$V_{37} = 17$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
19. $+ V_{37} + V_{38} \rightarrow V_{39}$	$V_{37} = 17, V_{38} = 1$	$V_{39} = 18$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
20. $- V_{39} - V_{40} \rightarrow V_{41}$	$V_{39} = 18, V_{40} = 1$	$V_{41} = 17$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
21. $\times V_{41} \times V_{42} \rightarrow V_{43}$	$V_{41} = 17, V_{42} = 2$	$V_{43} = 34$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
22. $- V_{43} - V_{44} \rightarrow V_{45}$	$V_{43} = 34, V_{44} = 1$	$V_{45} = 33$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
23. $+ V_{45} + V_{46} \rightarrow V_{47}$	$V_{45} = 33, V_{46} = 1$	$V_{47} = 34$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
24. $- V_{47} - V_{48} \rightarrow V_{49}$	$V_{47} = 34, V_{48} = 1$	$V_{49} = 33$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
25. $\times V_{49} \times V_{50} \rightarrow V_{51}$	$V_{49} = 33, V_{50} = 2$	$V_{51} = 66$		—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—			
			by Variable-end																												
			by Variable-cont																												

Bernoulli number를 계산하는 코드

● 최초의 컴퓨터 과학자는?

- Alan Mathison Turing (1912 - 1954) - [Imitation Game](#) (관련 영화)
- 컴퓨터(Turing Machine)에서의 Algorithm과 Computation의 수학적 토대를 완성한 수학자이자 컴퓨터 과학자
- 핵심 업적은 컴퓨터로 해결할 수 없는 문제도 존재함을 증명한 것!** (13장 계산 복잡도 등을 설명하는 절에서 추가 설명)

- **가상컴퓨터, 가상언어, 가상코드 (Virtual Machine, Pseudo Language, Pseudo Code)**

- 자료구조와 알고리즘의 성능(performance)은 대부분 수행 시간(시간복잡도: time complexity)으로 따지는 것이 일반적이다
 - 이를 위해, 실제 코드(C, Java, Python 등)로 구현하여 실제 컴퓨터에서 실행한 후, 수행 시간을 측정할 수도 있지만, HW/SW 환경을 하나로 통일해야 하는 어려움이 있다
 - 따라서, 가상언어로 작성된 가상코드를 가상컴퓨터에서 시뮬레이션하여 HW/SW에 독립적인 계산 환경(computational model)에서 측정해야 한다
 - 가상컴퓨터(virtual machine)**
 - 현대 컴퓨터 구조는 Turing machine에 기초한 von Neumann 구조를 따른다
 - 현재 가장 많이 사용하는 현실적인 가상 컴퓨터 모델은 **RAM**(Random Access Machine) 모델이다
 - RAM 모델은 CPU + memory + register + primitive operation으로 정의되는 가상컴퓨터 모델 중 하나이다
 - **CPU**: 연산(operation, command)을 수행
 - **메모리**: 임의의 크기의 실수 (즉, 정확한 실수 값을) 저장할 수 있는 무한한 워드(word)로 구성
 - **레지스터(register)**: CPU의 계산에 활용되는 충분한 개수의 독립된 메모리
 - **기본연산(primitive operation)**: 단위 시간에 수행할 수 있는 연산으로 정의
 - 복사연산**: $A = B$ (B 의 값을 A 레지스터에 복사 또는 B 레지스터의 값을 A 메모리에 복사)
 - 산술연산**: $+$, $-$, $*$, $/$ (나머지 $\%$ 연산은 허용 안되나, 본 강의에서는 포함한다) - 레지스터에 복사된 값에 대해 기본연산을 수행
 - 비교연산**: $>$, \geq , $<$, \leq , $=$, $!=$
 - 논리연산**: AND, OR, NOT
 - 비트연산**: bit-AND, bit-OR, bit-NOT, bit-XOR, $<<$, $>>$
- 가상언어(Pseudo Language)**
 - 영어나 한국어와 같은 실제 언어보다 간단 명료하지만, C, Python 같은 프로그래밍 언어보다 융통성있는 언어로, Python과 유사하게 정의함. (수학적/논리적으로 모호함 없이 명령어가 정의되기만 하면 됨.)
 - 기본 명령어**: (기본연산을 정의하는 명령어로 일반 프로그래밍 언어와 유사함)
 - **복사연산**: $A = B$
 - **산술연산**: $+$, $-$, $*$, $/$, $\%$

- 비교연산: `>`, `>=`, `<`, `<=`, `==`, `!=`
- 논리연산: `AND`, `OR`, `NOT`
- 비트연산: `bit-AND`, `bit-OR`, `bit-NOT`, `bit-XOR`, `<<`, `>>`
- 비교문: `if`, `if else`, `if elseif ... else` 문
- 반복문: `for` 문, `while` 문
- 함수정의, 함수호출, `return` 문

f. **가상코드**(Pseudo Code) - 가상언어로 작성된 코드

- i. 예: 배열 A의 n개의 정수 중에서 최대값을 계산하는 가상코드 (반드시 아래 형식을 따를 필요는 없음)
- ii. **algorithm arrayMax(A, n)**

```

input: n개의 정수를 저장한 배열 A ← 입력을 설명하는 주석의 역할
output: A의 수 중에서 최대값 ← 출력을 설명하는 주석의 역할
currentMax = A[0]
for i = 1 to n-1 do
    if currentMax < A[i]
        currentMax = A[i]
return currentMax

```

- iii. 위 코드에서 배정연산, 비교연산 등이 사용된다

• 알고리즘의 시간복잡도

- a. 가상컴퓨터에서 가상언어로 작성된 가상코드를 실행(시뮬레이션)한다고 가정한다
- b. 특정 입력에 대한 수행시간은 그 입력에 대해 수행되는 기본연산(primitive operation)의 횟수로 정의된다
- c. 문제는 입력의 종류가 무한하므로 모든 입력에 대해 수행시간을 측정하여 평균을 구하는 것은 현실적으로 매우 까다롭다는 점이다
- d. 따라서 최악의 경우의 입력(worst-case input)을 가정하여, 최악의 경우의 입력에 대한 알고리즘의 수행시간을 측정한다
- e. 최악의 경우의 입력에 대한 수행시간이 모든 입력에 대한 평균 수행시간보다는 정확하지는 않지만, 어떤 입력이 주어지더라도 최악의 경우의 수행시간보다 더 많은 시간이 걸리지 않는다는 점을 보장한다!

Checkpoint:

알고리즘의 수행시간 = 최악의 경우의 입력에 대한 기본연산의 수행 횟수

- f. 최악의 경우의 수행시간은 입력의 크기 n에 대한 함수 $T(n)$ 으로 표기 된다.
- g. $T(n)$ 의 수행시간을 갖는 알고리즘은 어떠한 입력에 대해서도 $T(n)$ 시간 이내에 종료됨을 보장한다

h. 실시간 제어가 필요하고 절대 안전이 요구되는 분야(항공, 교통, 위성, 원자로 제어 등)에선 실제로 최악의 경우를 가정한 알고리즘 설계가 필요하기 때문에, 유효한 수행시간 측정방법으로 볼 수 있다

i. 최악의 경우의 입력에 대해, 알고리즘의 기본연산(복사, 산술, 비교, 논리, 비트논리)의 횟수를 아래의 최대 값을 계산하는 예를 통해 자세히 세어보자

ii. 예: n개의 정수 중 최대값을 찾는 알고리즘

```
algorithm arrayMax(A, n)
    input: n개의 정수를 저장한 배열 A
    output: A의 수 중에서 최대값
    currentMax = A[0]
    for i = 1 to n-1 do
        if currentMax < A[i]
            currentMax = A[i]
    return currentMax
```

- **if** 문의 결과에 따라 *currentMax = A[i]*가 실행되든지 아니든지 결정된다
- 최악의 경우의 입력은 무조건 *currentMax = A[i]*을 실행해야 하므로 **if** 문을 계속 참(True)이 되도록 해야 한다. 이 같은 입력은 *A*의 저장된 값이 오름차순으로 정렬된 경우이다. (즉, 오름차순으로 정렬된 *n*개의 값이 저장된 배열 *A*가 최악의 경우의 입력이라는 의미이다)

ii. 최악의 입력에 대한 횟수 분석

```
algorithm arrayMax(A, n)
    input: n개의 정수를 저장한 배열 A
    output: A의 수 중에서 최대값
    currentMax = A[0] (1번)
    for i = 1 to n-1 do
        if currentMax < A[i] (n-1번)
            currentMax = A[i] (n-1번)
    return currentMax
```

[주의 1] *A[i]*도 실제로 시간이 필요하다. *i*번째 원소의 (메모리) 주소를 계산해 메모리에서 값을 읽어야 하기 때문이다. 기본 연산 횟수 분석을 간단하게 하기 위해, 이 주소 계산을 위한 시간은 걸리지 않는다고 가정한다

[주의 2] **for** 루프의 제어에 관계하는 변수 *i*에 대한 기본 연산 역시 분석을 간단히 하기 위해 무시한다. (*i* = 1 연산 1회, 루프 반복 때마다 *i* <= *n*-1의 비교 1회, *i* = *i* + 1의 +, = 연산 2회가 필요하지만 대범하게 무시한다. 무시한 기본 연산 횟수는 루프의 반복 횟수에 비례하는 값이므로 결국 반복 횟수의 몇 배만 추가하는 정도의 영향만 미치게 된다. 이는 앞으로 살펴볼 Big-O로 표기된 최종 수행시간에 영향을 주지 않기에 무시하기로 한다. 😊)

- 따라서 $T(n) = 2n - 1$ 이 된다
- $n = 10$ 이면 $T(10) = 19$ 가 되어 19번 이내의 기본연산을 수행한다는 뜻이고, $n = 200$ 이면, $T(200) = 399$ 번의 기본연산을 수행한다는 의미이다
- [?] 질문: 아래 알고리즘의 최악의 입력에 대해 수행하는 기본연산의 횟수는?

```
algorithm arraySum(A, B, n)
    sum = 0                                # 1번
    for i = 0 to n-1 do
        for j = i to n-1 do                # 이중 for 루프는 n번 + (n-1)번 + ⋯ + 1번 반복
            sum = sum + A[i]*B[j]          # 이 문장 자체는 *, +, = 3번의 기본연산
    return sum
```

$$\text{결국, } T(n) = 1 + 3*(1+2+\dots+(n-1)+n) = 3n^2/2 - 3n/2 + 1$$

- [GCD 계산 문제 - skip 가능] 앞에서 두 수 a , b 의 최대공약수(gcd)를 계산하는 세 가지 방법에 대해 살펴봤다. 이 gcd 문제에 대한 다음 질문에 답해보자. ($a > b$ 라고 가정한다)
 - 먼저, 뺄셈 연산을 하는 gcd 계산 알고리즘부터 생각해보자. 이 알고리즘의 수행시간은 **while 루프의 반복 횟수에 비례**한다
 - 질문1: $a = 100$, $b = 99$ 라고 할 때, while 루프의 반복 횟수는 얼마인가?
 - $a > b$ 이므로 $a = a - b = 1$ 이 되고, 그 이후엔 b 의 값이 1씩 감소해 두 값 모두 1이 되고, 한 번 더 루프를 수행하면 두 값 중 하나가 0이 되어 루프 반복이 끝난다. 결국, 100번의 반복이 필요하다
 - a 와 b 의 값의 차이가 1인 경우 (예: $a = b - 1$ 인 경우)에 a 번 만큼 while 루프를 반복하게 된다!
 - 질문2: 입력의 크기 n 은 얼마인가? 예를 들어,
 - 입력으로 주어진 값이 2개 뿐이므로 $n = 2$ 인가?
 - 두 수 중 큰 수의 값이 100이므로 $n = 100$ 이라고 해야 할까?
 - $n = 2$ 라고 주장하는 경우
 - $a = b - 1$ 인 경우에는 루프는 a 번 반복된다. 따라서 gcd 알고리즘의 수행시간은 a 에 비례하는 식이 된다. 그러면 수행시간을 n 에 관한 식이 아닌 a 에 관한 식이 되어 제대로 정의되지 않는다. 문제는 a 값이 n 과 전혀 관계가 없으며 **매우 매우 클 수 있다**는 것이다. 즉, 입력의 크기는 매우 작은데 실제 수행시간은 **입력의 크기와 무관하게** 매우 클 수 있다는 게 문제다
 - $n = 100$ 이라고 (**무리하게**) 주장하는 경우 (즉, $n = a$ 라고 주장하는 경우)
 - 100번의 반복을 하기에 $n (= a)$ 번의 반복을 한다고 주장할 수 있다. 즉, 입력의 크기 n 에 관한 식으로 수행시간을 표현할 수 있다. 하지만 전혀 자연스럽지 않다!
 - 다음으로 나머지를 반복적으로 계산하는 gcd 알고리즘을 생각해보자
 - $a=100$, $b=99$ 라고 하면, 나머지 연산 (%)을 몇 번 하게 되나? **2번**

- ii. 그럼 $a > b$ 인 경우에 최대 몇 번의 나머지 연산을 수행하게 될까?
 - 1. 사실1: $a > b > 1$ 에 대해, $a \% b \leq a/2$ 임이 성립한다
 - a. 증명: 각자 해보기
 - 2. 사실2: $a > b > 1$ 에 대해, 나머지 % 연산은 최대 $2\log_2 a$ 번 이하만 수행된다
 - a. 증명: 사실1을 이용하면 쉽게 증명 가능하다. 큰 수 a 가 반으로 줄어들기에, a 와 b 가 번갈아 반으로 줄어들어 0에 도달하는 것이므로 $\log_2 a + \log_2 b \leq 2\log_2 a$ 번 반복할 수 있다
- iii. 사실 2에 의해 gcd 알고리즘은 $2\log_2 a$ 에 비례하는 횟수의 기본연산을 수행한다
 - 1. $n = 2$ 라고 주장하는 경우에는 수행시간에 n 이 등장하지 않는다는 문제가 여전히 존재한다
 - 2. $n = a$ 라고 주장하는 경우에는 결국 수행시간이 $\log_2 a$ 에 비례한다는 결론에 도달한다. 뺄셈 반복 알고리즘의 수행시간 a 보다는 매우 빨라졌지만, 여전히 입력 값의 크기가 수행시간에 등장하는 부자연스러운 현상이 발생한다!
- iv. 입력으로 주어지는 수가 두 개이므로 $n = 2$ 라고 하는 게 일관성이 있지만, 실제 n 에 관한 식이 아닌 주어진 입력 값에 대한 식으로 수행시간이 표현된다는 문제가 있고, $n = a$ 로 하면 a 의 실제 값의 크기 자체가 입력의 크기가 된다는 부자연스러움이 있다. 두 가지 경우 모두 마음에 안든다는 게 결론!
- v. a 의 값이 컴퓨터에서는 $\log_2 a$ 비트로 표현된다. 산술, 비교 등의 기본 연산도 $\log_2 a$ 비트의 두 수에 대해 이루어지는 셈이다. 즉, $\log_2 a$ 비트 수가 일종의 입력의 크기라고 볼 수도 있다. a 의 값이 증가하면 그에 대한 비트 수도 증가한다는 직관에 맞다. 결국, 더 섬세하게 정의하면 gcd 문제에서의 입력의 크기 $n = \log_2 a$ 라고 정의할 수 있다. 이 경우를 해당 문제의 비트 복잡도(bit complexity)라고 부른다. (일반적인 경우는 비트 복잡도가 아닌 입력의 주어진 값의 개수를 입력의 크기 n 으로 정한다는 사실에 주의하자. 따라서 gcd 문제의 일반적인 입력은 $n = 2$ 가 된다)
- vi. gcd 문제에서는 입력을 비트 복잡도로 정의하는 것이 자연스럽기 때문에 $n = \log_2 a$ 로도 정의할 수 있다. 그러면 while 루프의 반복 횟수가 최악의 경우에 $2\log_2 a$ 를 넘지 않기에, gcd 알고리즘의 기본 연산 횟수는 $2n$ 을 넘지 않는다고 말할 수 있다. 다시 말하면, 입력의 비트 복잡도 크기 n 에 관한 식으로도 수행시간을 표현할 수 있다.
- vii. 그러나 본 교재에서는 비트 복잡도가 아닌 입력의 주어지는 값의 개수를 입력의 크기로 정의하겠다 [중요]

viii. **질문3:** 나머지 gcd 알고리즘의 수행시간이 가장 클 때의 두 수 a , b 는 어떤 값을 가질 때인가?

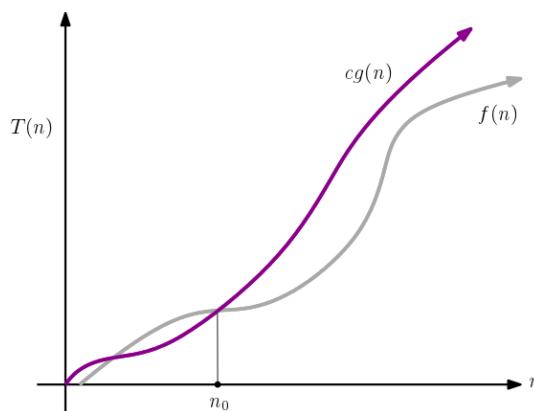
1. 뺄셈 gcd 알고리즘에서는 두 수가 인접한 수일 때 두 수 중 큰 값만큼 루프를 반복했다
2. 나머지 gcd 알고리즘에서는 두 수가 _____ 때 루프 횟수가 최대가 된다
3. [힌트] 연속한 두 Fibonacci 수 F_{k+1} , F_k 에 대해 루프 횟수를 계산해보자
4. $F_{k+1} \% F_k = (F_k + F_{k-1}) \% F_k = (F_k \% F_k + F_{k-1} \% F_k) \% F_k = (0 + F_{k-1}) \% F_k = F_{k-1}$
5. 결국 $\gcd(F_{k+1}, F_k) = \gcd(F_k, F_{k-1})$ 이 된다
6. 이 과정을 k 번 반복하면 $\gcd(F_1, F_0)$ 에 도달하게 되고, 이 경우에는 $\gcd(1, 0)$ 이므로 gcd 값이 1이 되어 종료한다
7. 즉, k 번 반복하게 된다.

ix. **질문4:** 그럼 k 값은 얼마인가?

1. 자료를 찾아보면, F_k 는 ϕ^k 에 비례한다. 여기서 ϕ 는 황금비율이라 불리며, $x^2 - x + 1 = 0$ 의 양의 실근이다
 2. 결국, $n = F_k = \phi^k$ 이므로 $k = \log_{\phi} n$ 이 된다
- x. 결론적으로 나눗셈 반복 방법으로 gcd를 계산하는 알고리즘의 최악의 입력은 인접한 두 피보나치 수이며, 이 경우에 $\log_{\phi} n$ 정도 반복하게 된다
- xi. **[hard] 질문5:** $\gcd(F_a, F_b) = F_{\gcd(a, b)}$ 이 성립함을 증명하라!

- Big-O 표기법

- 이제부터 입력의 크기 n 은 무조건 입력으로 주어지는 값의 개수로 정의한다
- 최악의 입력에 대한 기본연산의 횟수를 정확히 세는 건 일반적으로 귀찮고 까다롭다
- 정확한 횟수보다는 입력의 크기 n 이 커질 때, 수행시간 $T(n)$ 의 증가하는 정도(rate of the growth of $T(n)$ as n goes big)가 훨씬 중요하다
- 수행시간 함수 $T(n)$ 이 n 에 관한 여러 항(term)의 합으로 표현된다면, 함수 값의 증가율이 가장 큰 항 하나로 간략히 표기하는 게 시간 분석을 간단하게 하는 데 큰 도움이 된다
- 예를 들어, $T(n) = 2n + 5$ 이면, 상수항보다는 n 의 일차항이 $T(n)$ 의 값을 결정하게 되므로 상수항을 생략해도 큰 문제가 없다
- $T(n) = 3n^2 + 12n - 6$ 이면, n 값이 커짐에 따라 n^2 항이 $T(n)$ 의 값을 결정하게 되므로, 일차항과 상수항을 생략해도 큰 문제가 없다
- 이렇게 최고차 항만을 남기고 나머지는 생략하는 식으로 수행시간을 간략히 표기하는 방법을 **근사적 표기법**(Asymptotic Notation)이라고 부르고, Big-O(대문자 O)를 이용하여 다음의 예처럼 표기한다.
 - $T(n) = 2n + 5 \rightarrow T(n) = O(n)$
 - $T(n) = 3n^2 + 12n - 6 \rightarrow T(n) = O(n^2)$
 - [\[자료\]](#)
- Big-O 표기하기 위한 방법은 다음과 같다.
 - n 이 증가할 때 가장 빨리 증가하는 항(최고차 항)만 남기고 다른 항은 모두 생략한다
 - 가장 빨리 증가하는 항에 곱해진 상수 역시 생략한다
 - 남은 항을 $O(\)$ 안에 넣어 표기한다
- Big-O에 대한 엄밀한 수학적 정의
 - 예를 들어, $T(n) = 3n^2 + 12n - 6$ 이 있을 때, 최고차 항이 n^2 인 함수 클래스에 포함된다는 뜻은 $n > n_0$ 이상의 모든 값에 대해 $3n^2 + 12n - 6 \leq cn^2$ 이 성립하는 n_0 값과 c 값이 항상 존재한다는 뜻이다. (아래는 영어 정의)
 - $O(f(n)) = \{g(n) : \text{there exists constant } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq cg(n) \text{ for all } n > n_0\}$



- iii. $T(n) = 3n^2 + 12n - 6 = O(n^2)$ 이라고 말할 수 없다. 이유는 위의 정의에 의하면, $3n^2 + 12n - 6 \leq cn$ 을 만족하는 n_0 값과 c 값을 찾을 수 없기 때문이다
- n 의 값이 작으면 위 부등식이 성립하지만 이차식의 증가율이 훨씬 커서 n 이 상당히 커지면 c 의 값을 어떻게 정하더라도 위의 부등식을 만족할 수 없기 때문이다

j. 예 1: $\log_a n = O(\log_2 n)$

- $\log_a n = \log_2 n / \log_2 a = (1/\log_2 a) \log_2 n = c \log_2 n = O(\log_2 n)$ (여기서 $c \geq 1/\log_2 a$ 인 임의의 상수로 정하면 됨)
- 따라서 밑의 값의 상관없이 $\log n$ 은 Big-O 표기법으로 밑이 2인 $\log_2 n$ 이라고 통일해도 상관없다

k. 예 2: (python 또는 pseudo code로 기술한 알고리즘의 수행 시간)

- $O(1)$ 시간 알고리즘: constant time algorithm: 값을 1 증가시킨 후 리턴

```
def increment_one(a):
    return a+1
```

- $O(\log n)$ 시간 알고리즘: logarithmic time algorithm: \log 의 밑은 2라고 가정: n 을 이진수로 표현했을 때의 비트수 계산 알고리즘

```
def number_of_bits(n):
    count = 0
    while n > 0:
        n = n // 2
        count += 1
    return count
```

- $O(\sqrt{n})$ 시간 알고리즘: n 의 약수의 개수를 세는 알고리즘

```
def number_of_factors(n):
    count, k = 0, 1
    while k*k < n:
        if n%k == 0:
            count += 2
            k += 1      # need more?
    return count
```

- $O(n)$ 시간 알고리즘: linear time algorithm: n 개의 수 중에서 최대값 찾는 알고리즘

- $O(n^2)$ 시간 알고리즘: quadratic time algorithm: 두 배열 A, B의 모든 정수 쌍의 곱의 합을 계산하는 알고리즘

```
def array_sum(A, B, n)
    sum = 0
    for i = 0 to n - 1 do
        for j = 0 to n - 1 do
```

```

    sum += A[i]*B[j]
    return sum

```

- vi. $O(n^3)$ 시간 알고리즘: cubic time algorithm: $n \times n$ 인 2차원 행렬 A와 B의 곱을 계산하여 결과 행렬 C를 리턴하는 알고리즘

```

def mult_matrices(A, B, n)
    input: n x n 2d matrices A, B
    output: C = A x B
    for i = 1 to n do
        for j = 1 to n do
            C[i][j] = 0
    for i = 1 to n do
        for j = 1 to n do
            for k = 1 to n do
                C[i][j] += A[i][k] * B[k][j]
    return C

```

- vii. $O(2^n)$ 이상의 시간이 필요한 알고리즘: exponential time algorithm: k번째 피보나치 수 계산하는 재귀 알고리즘 또는 n개의 원소를 갖는 집합의 모든 부분집합을 출력하는 알고리즘

- 아래 fibonacci 알고리즘은 $O(\phi^n)$ 시간 알고리즘이다

```

def fibonacci(n):
    if n <= 1: return n
    return fibonacci(n-1) + fibonacci(n-2)

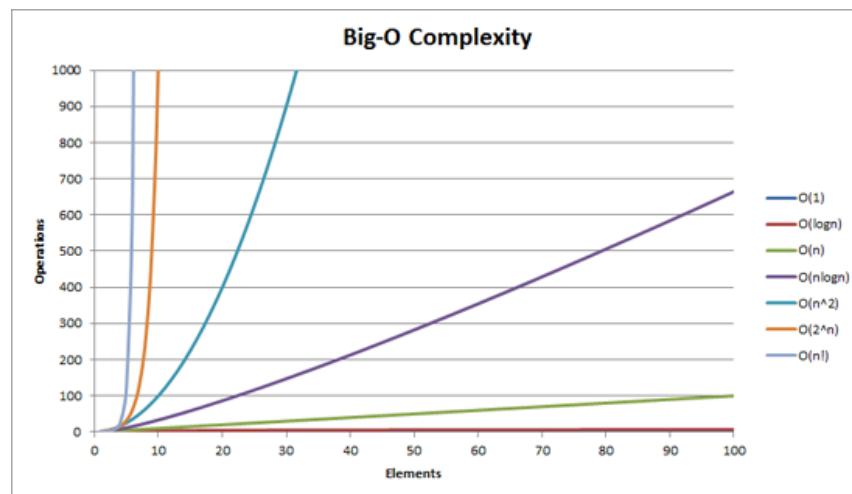
```

- viii. $O(n!)$ 시간이 필요한 알고리즘: 1부터 n까지의 자연수에 대한 순열(permutation)을 모두 출력하는 알고리즘. 순열의 개수가 $n!$ 이므로 최소한 그 정도의 시간이 필요한 알고리즘

- I. **다항 시간 (polynomial time)** 알고리즘: 수행 시간이 $O(n^k)$ 인 알고리즘 (k 는 0보다 큰 실수)

- m. **지수 시간 (exponential time)** 알고리즘: 다항 시간이 아닌 지수 시간 알고리즘

n. 비교 그림 [출처: [stackoverflow](#)]



o. 더 자세한 비교 그림 및 설명 <http://bigocheatsheet.com/>

p. 자료구조와 기본 알고리즘의 작동 원리를 코드와 애니메이션으로 보여주는 사이트 소개
i. 싱가포르 대학 visualgo: <https://visualgo.net/ko>



[⏰ 알고리즘 퍼즐 1] 독살 음모를 밝혀라!

- 정답은 왼쪽의 QR 코드로 확인

왕이 마실 와인 8병 중 한 병에 강력한 독이 들어 있다. 한 방울만이라도 치명적이다. 정상 와인을 아무리 섞어도 치명적이다. 다행히 검사장비를 구할 수 있다. 단, 검사는 1시간이 필요하다. 왕은 무조건 1시간 후에 와인을 마실테니 독이 든 병을 찾아내라는 명령을 내렸다. 이를 위해, 최소 몇 대의 검사 장비가 있으면 충분할까? (한 번 검사할 때에는 여러 병의 와인을 섞어 검사 가능하다)



[⏰ 알고리즘 퍼즐 2] 가장 빠른 말 찾기 (구글 인터뷰 질문)

- 정답은 왼쪽의 QR 코드로 확인

25마리의 말 중에서 가장 빠른 세 마리를 선택하고 싶다. 경기장엔 한 번에 다섯 마리 이하로 경주를 할 수 있고, 시계가 없어 기록을 챌 수 없고, 대신 순위만 알 수 있다. 한 경주에서 같은 순위는 없다고 가정한다. 최소 몇 번의 경주로 가장 빠른 세 마리를 알 수 있는가?

[Summary] 자료구조 연산시간 정리

[출처: <http://bigocheatsheet.com/>]

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity	
	Average				Worst					
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion		
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Stack	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Queue	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Doubly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	
Skip List	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n \log(n))$	
Hash Table	N/A	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Binary Search Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
Cartesian Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(n)$	$O(n)$	$O(n)$	$O(n)$	
B-Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
Red-Black Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
Splay Tree	N/A	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	N/A	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
AVL Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	
KD Tree	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$\Theta(\log(n))$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	

2. Recursion (재귀) 맛보기



1. 예1: 1부터 n까지의 합을 계산해보자

a. 루프를 활용한 방법:

```
def sum(n):
    s = 0
    for i in range(1, n+1):
        s += i
    return s
```

b. 재귀적으로 작성하는 법:

1. $1 + 2 + 3 + \dots + n = (1 + 2 + \dots + n-1) + n$
2. 1부터 n까지의 합은 1부터 n-1까지의 합에 n을 더한 값이다 → 1부터 n-1까지의 합을 안다면 거기에 n만 더하면 된다 → 1부터 n-1까지의 합은 다시 같은 방식으로 (재귀적으로) 계산하면 된다

$$\begin{aligned} 3. \quad \text{sum}(n) &= (1 + 2 + \dots + n-1) + n \\ &= \text{sum}(n-1) + n \\ &= (\text{sum}(n-2) + n-1) + n \\ &= ((\text{sum}(n-3) + n-2) + n-1) + n \\ &\dots \\ &= (\dots(\text{sum}(1) + 2) + 3) + \dots + n \end{aligned}$$

4. $\text{sum}(1) = 1$ 이므로 더 이상 전개할 필요 없다: 바닥 조건 (base case)

```
def sum(n):
    if n == 1: return 1
    return sum(n-1) + n
```

c. 수행시간

- i. 루프 함수: n번 반복되고, 매번 상수번의 기본연산만 수행 $O(n)$
- ii. 재귀 함수: $\text{sum}(n-1)$ 을 계산하는 시간 + (n 을 더하는 기본연산 1회)
 1. $\text{sum}(n-1)$ 을 계산하는 시간 = $\text{sum}(n-2)$ 계산 시간 + 1
 2. ...
 3. 결국, 수행 시간도 재귀적으로 정의해야 함!
4. $T(n)$: $\text{sum}(n)$ 의 수행시간을 나타내는 함수로 정의하면:

$$T(n) = T(n-1) + 1$$

단, $T(1) = 1$ (바닥조건에 대한 시간. 즉, 점화식의 첫 번째 항)

d. 점화식 $T(n) = T(n-1) + 1$, $T(1) = 1$

- i. 점화식을 전개하여 n에 관한 식으로 표현해야 함
- ii. 바닥 경우에 이를 때까지만 전개하면 됨 ($T(1)$ 의 값은 알고 있으므로)

$$\begin{aligned} T(n) &= T(n-1) + 1 = (T(n-2) + 1) + 1 \\ &= \dots \\ &= T(1) + (1 + \dots + 1) = n = O(n) \end{aligned}$$

2. 예 2: 1부터 n까지의 합을 계산해보자. 좀 다르게~

- a. 문제를 조금 수정해 a부터 b까지의 합을 구해보자
- b. $\text{sum}(a, b)$ 함수는 a부터 b까지 더한 값을 리턴 (1부터 n까지 합은 $\text{sum}(1, n)$ 호출)
- c. a와 b의 중간의 값을 m이라 하자 ($m = (a+b)/2$)
- d. $\text{sum}(a, b) = \text{sum}(a, m) + \text{sum}(m+1, b)$ 의 재귀 형식으로 정의 가능
 - i. 한 번이 아닌 두 번의 재귀 호출이 이루어짐: $\text{sum}(a, m)$ 과 $\text{sum}(m+1, b)$
 - ii. 여기서 base 경우는? $\text{sum}(a, b)$ 에서 $a == b$ 인 경우
 1. $a == b$ 인 경우엔 a를 리턴하면 됨
- e. $\text{sum}(2, 7)$ 의 함수 호출 순서를 따라가보자

$$\begin{aligned} \text{sum}(2, 7) &= (\text{sum}(2, 4) + \text{sum}(5, 7)) \\ &= ((\text{sum}(2, 3) + \text{sum}(4, 4)) + (\text{sum}(5, 6) + \text{sum}(7, 7))) \\ &= (((\text{sum}(2, 2) + \text{sum}(3, 3)) + 4) + ((\text{sum}(5, 5) + \text{sum}(6, 6)) + 7)) \\ &= (((2+3)+4) + ((5+6)+7)) \\ &= 27 \end{aligned}$$

f. 코드

```
def sum(a, b):
    if a == b:
        return a
    m = (a+b)//2
    return sum(a, m) + sum(m+1, b)
```

g. $\text{sum}(1, n)$ 을 호출한 경우의 수행시간 $T(n)$ 의 점화식

- i. $\text{sum}(1, n)$ 은 $\text{sum}(1, n//2)$ 와 $\text{sum}(n//2+1, n)$ 을 먼저 계산한 후 두 계산된 수를 더하면 되므로, $T(n)$ 은 $T(n/2)$ 의 2배의 시간과 덧셈 한번의 시간을 더한 값이 된다. 따라서 아래의 점화식을 얻을 수 있다. (여기서 c는 양의 상수)

$$T(n) = T(n/2) + T(n/2) + c = 2T(n/2) + c, \quad T(1) = 1$$

- ii. 전개해보자

1. Big-O로 표기하면 최고차 항에 곱해진 상수는 무시되기에 $n = 2^k$ 로 가정해도 상관없음에 유의!

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + c \\
 &= 2\left(2T\left(\frac{n}{2^2}\right) + c\right) + c \\
 &= 2^2T\left(\frac{n}{2^2}\right) + 2c + c \\
 &\vdots \\
 &= 2^kT\left(\frac{n}{2^k}\right) + c(1 + 2 + \dots + 2^{k-1}) \\
 &= 2^k + c\frac{2^k - 1}{2 - 1} \\
 &= O(n)
 \end{aligned}$$

3. 예3: factorial 계산

```
def factorial(n):
    if n == 1: return n
    else: return n * factorial(n-1)
```

호출과정:

```
factorial(4) = 4 * factorial(3)
              4 * (3 * factorial(2))
              4 * (3 * (2 * factorial(1)))
              4 * (3 * (2 * 1))
              4 * (3 * 2)
              4 * 6
              24
```

- a. 리턴되는 값을 가지고 계속 계산을 하며 완성해야 함 → recursion stack이라는 메모리 사용
- b. [��] [easy] list A에 있는 가장 큰 수를 찾는 재귀 함수 find_max(A, n)을 작성해보자

```
def find_max(A, n): # A[0] ~ A[n-1] 중 최대값을 찾아 리턴
    # 이를 위한 재귀 코드는?
    if n == 1:
        return A[0]
    return max(find_max(A, n-1), A[-1])
```

4. 예4: 리스트의 값을 거꾸로 배치하기

- a. A의 값을 반대방향으로 재 배치하는 함수 reverse(A) 재귀적으로 작성하기
- b. 첫 번째 방법: 이 방법의 단점은 무엇일까?

```
def reverse(A):
    if len(A) == 1:
        return A
    return reverse(A[1:]) + A[0] # A[0]를 맨 뒤로 보내고 나머지는 재귀
```

- c. 두 번째 방법: $A[\text{start} \dots \text{stop}-1]$ 까지의 값을 거꾸로 배치
 # 이 방법과 첫 번째 방법을 비교해보자!

```
def reverse(A, start, stop): # stop-1까지가 대상임에 유의
    if start < stop-1: # 2개 이상의 값이 있는 경우만 의미 있으므로
        A[start], A[stop-1] = A[stop-1], A[start]
        reverse(A, start+1, stop-1)
```

- d. $A = [1, 2, 3, 4, 5, 6]$ 에 위의 두 가지 방법을 적용해보자

- e. 수행시간을 위해 두 방법의 수행시간 점화식을 세워보자 (답은 왼쪽 QR 코드)

- i. 첫 번째 방법: $T(n) =$



* slicing의 시간은 포함하지 않았음에 유의. slicing 시간을 포함하면, slicing은 리스트를 복사하는 것이기에 점화식은 다음과 같아야 한다: $T(n) = T(n-1) + cn$

- ii. 두 번째 방법: $T(n) =$



5. 예5: $\{0, 1, \dots, n-1\}$ 집합의 모든 부분집합을 출력해보자

- a. 부분집합의 개수가 2^n 이므로 그 이상의 수행 시간이 필요 (아래 코드를 이해해보자)

```
def gen_subsets(k):
    if k == n:
        print(S)
    else:
        S.append(k)
        gen_subsets(k+1) # k가 들어가는 부분집합
        S.pop()
        gen_subsets(k+1) # k가 들어가지 않는 부분집합

S = []
n = int(input())
gen_subsets(0)
```

6. 예6: 길이가 n인 모든 순열(permuation)을 출력해보자

- a. 순열의 개수가 $n!$ 이므로 당연히 그 이상의 시간이 필요 (아래 코드를 이해해보자)

```
def gen_permutation():
    if len(P) == n:
        print(P)
    else:
        for i in range(1, n+1):
            if chosen[i] == True:
                continue
            chosen[i] = True
            P.append(i)
            gen_permutation()
            chosen[i] = False
            P.pop()

P = []
n = int(input())
chosen = [False for _ in range(n+1)]
gen_permutation()
```

7. [고급] Tail Recursion (Python에서 제공하는 기능)

- a. 예3의 factorial 재귀 함수는 $\text{factorial}(n) \rightarrow \dots \rightarrow \text{factorial}(1)$ 로 재귀호출된 후, $\text{factorial}(1)$ 이 계산이 된 후에 return 되면서 전체 계산이 완성된다. 즉, 바닥 경우에 도달할 때까지 재귀 호출이 계속하게 된다

```
def factorial(n, value = 1):
    if n == 1:
        return value
    else:
        return factorial(n-1, value*n)
```

호출과정: 예3의 factorial 호출과정과 비교해보자!

```
factorial(4)
factorial(3, 1*4)    # value = 1
factorial(2, 4*3)    # value = 4
factorial(1, 12*2)   # value = 12
factorial(1)          # value = 24, return value
24 → 24 → 24 → 24  # return four times with the same 24
```

- b. 재귀 함수의 매개변수로 현재의 중간 계산된 factorial 값을 직접 전달해주기 때문에, 바닥 경우인 $n = 1$ 인 경우에 이미 최종 값 24가 계산되었고, 이후엔 그 값을 return만 하면서 전달하게 됨. 이 성질을 이용하면 하나의 recursion stack의 내용을 overwrite하는 식으로

메모리 사용을 크게 줄일 수 있다 ⇒ TRO(Tail Recursion Optimization, 언어에 따라 지원 여부가 결정)

- c. 그러나 재귀 호출은 호출마다 현 상태를 recursion stack에 저장(push)하고 리턴되면서 복원(pop)해야 하는 부담이 발생하기에 가급적 피해야 한다. 또한 recursion stack의 크기가 제한되어 있어 깊이가 큰 재귀 알고리즘은 실패할 수도 있기 때문이다
- d. 재귀 호출 대신 비재귀 반복문을 사용하는 게 좋다. 위의 tail recursion 재귀식을 반복문으로 바꾸면 아래와 같다

```
def factorial(n, value=1):  
    while True:  
        if n == 0:  
            return value  
        else:  
            value = value * n  
            n = n - 1
```

3. Selection (선택 문제)

1. Selection problem: 교재에 따라서는 Order Statistics 문제라고 부르기도 한다



- a. 입력: 리스트에 n 개의 수와 1과 n 사이의 자연수 값 k 가 주어진다
- b. 출력: 입력으로 주어진 수 중에서 **k 번째로 작은 수**를 찾아 리턴한다
- c. 목표: **비교 횟수를 되도록 줄인다** (최소 비교 횟수가 목표)

2. Problem 1: 가장 큰 수 (최대값) 찾기 ($k = n$)

- a. 가장 단순한 selection 문제

```
currentMax = A[0]
for i = 1 to n-1 do
    if currentMax < A[i] # 두 수의 비교가 이루어지는 비교문 (*)
        currentMax = A[i]
return currentMax
```

- b. 위의 코드에서 비교 횟수는? 즉, 두 수의 비교가 이루어지는 비교문의 수행회수는? $n - 1$
- c. 질문 1: 위와 비교 횟수는 같지만 다른 순서로 비교해서 가장 큰 수를 찾을 수 있다. 어떻게 하면 될까?
- 답: 토너먼트 방식으로 비교해서 찾을 수 있다. 이 경우에도 $(n-1)$ 번의 비교가 반드시 필요함을 쉽게 확인할 수 있다
- d. 질문2: 이 횟수보다 더 적게 비교해서 최대 값을 찾을 수 있을까? 찾을 수 없다면 왜 그럴까? 증명할 수 있을까?
- 답: 더 적은 비교로 찾을 수 없음을 증명할 수 있다! 이 장의 끝에 설명하는 선택문제의 비교회수 **하한 증명법**을 참조하기 바란다
- e. Python에서는 `min`과 `max`함수를 기본적으로 제공하므로 리스트와 튜플에서의 최소/최대 값을 찾을 수 있다. 단, 이 두 함수 역시 $(n-1)$ 번의 비교가 반드시 필요하므로 $O(n)$ 시간이 걸림에 유의하자

3. Problem 2: 가장 작은 수와 가장 큰 수를 모두 찾기 ($k = 1, k = n$)

- a. 알고리즘 1:

- 가장 작은 수를 $(n-1)$ 번의 비교로 찾는다.
- 가장 작은 수를 제외하고, 나머지 $n-1$ 개의 수 중에서 다시 $(n-2)$ 번의 비교로 가장 큰 수를 찾는다
- 두 수를 찾기 위한 최대 비교 횟수는 **$2n-3$ 번**
- 이 보다 적게 비교해서 찾을 순 없을까?
 - [hint] 가장 작은 수를 찾을 때 비교한 결과를 이용해서 가장 큰 수를 찾을 때 비교를 덜 할 수 있지 않을까?

b. 알고리즘 2:

- i. n 이 짹수라고 가정 (홀수인 경우에도 유사하게 생각할 수 있다)
- ii. 가장 작은 수를 먼저 찾는데, 토너먼트식으로 $(n-1)$ 번 비교하여 찾는다
 - 1. 토너먼트의 1라운드에서 탈락한 수 중에 가장 큰 수가 존재한다!
- iii. 토너먼트 1라운드에서 탈락한 수는 $n/2$ 개 (n 이 짹수인 경우) 또는 $n/2+1$ 개 (n 이 홀수인 경우)
- iv. 이 탈락한 수 중에서 가장 큰 값이 전체에서 가장 큰 값이 된다. 이를 위해 필요한 비교 횟수는 $(n/2 - 1)$ 번
- v. 따라서 전체 비교 횟수는 $3n/2 - 2$ 번이면 충분하다
- vi. 질문: 이보다 더 적은 비교로 가능할까? (답: 불가능하다. 하한 증명은 이 장의 끝에서 확인)

4. Problem 3: 가장 작은 수와 두 번째로 작은 수 찾기 ($k = 1, k = 2$)

- a. [힌트] Problem 2에서 사용한 토너먼트 비교를 이용해보자 [[10초간 생각해보기](#)]
- b. 가장 작은 수를 찾았다면, 두 번째로 작은 수는 어떤 수들 중에 있을까? [[10초 다시 생각해보기](#)]
 - i. 가장 작은 수가 토너먼트 우승자라면, 두 번째로 작은 수는 반드시 제일 작은 수를 만나 경기를 벌여야 한다 (맞다/틀리다)
 - ii. 그럼, 결승전에서 만나 진 수가 바로 두 번째로 작은 수이다 (맞다/틀리다)
 - iii. 그럼, 어떤 수들 중에 두 번째로 작은 수가 있을까?
답: (마우스로 굽기) → 가장 작은 수와 비교해서 탈락한 수 중에 반드시 존재한다!
- c. 두 번째로 작은 수가 될 수들은 몇 개일까?
 - i. 힌트1: 토너먼트는 총 몇 라운드인가? ($n = 2, 3, 4, \dots$ 증가시키면서 따져보기)
 - ii. 힌트2: 가장 작은 수와 비교된 수는 라운드마다 ___ 개 씩 존재하는가?
 - iii. 답: (마우스로 굽기) → $\log_2 n$ 의 올림한 값
- d. 결국, $(n-1) + (\log_2 n - 1)$ 번의 비교로 가장 작은 두 수를 찾을 수 있다!

5. Problem 4: 임의의 k 번째로 작은 수 찾기 (k 가 입력으로 함께 주어지는 일반적인 selection 문제)

- a. $k = 1$ 이면 가장 작은 값 찾는 문제와 동일
 $k = n$ 이면 가장 큰 값 찾는 문제와 동일
 $k = n/2$ 이면 중간 값(median) 찾는 문제
- b. n 개의 수와 함께 k 가 입력으로 함께 주어짐
- c. 아이디어: 앞의 다른 문제의 알고리즘처럼 찾고 있는 수가 **존재할 범위를 줄여나감!**
- d. 예: 처음에는 전체 n 개 중에서 k 번째 작은 수가 있음 → 추가 작업으로 k 번째 작은 수가 있을 범위를 반으로 줄임 ($n/2$ 개 수로 후보를 줄임) → 다시 추가 작업으로 범위를 반으로 줄여 $n/4$ 개 중에 찾는 값이 있음 → 마지막 값 하나가 남을 때까지 위 과정을 반복함. 그러면 남은 값 하나가 찾는 값임!
 - i. $T(n) = n$ 개의 수 중에 k 번째 작은 수를 찾는 데 필요한 (최악의 경우의) 비교 횟수
 - ii. $T(n) =$ 추가 작업을 위해 사용한 비교 횟수 + $T(n/2)$

- iii. 만약, 추가 작업을 위해 2n번 비교했다면
 iv. $T(n) = T(n/2) + 2n$ 이라는 점화식이 성립함!

$$T(n) = T\left(\frac{n}{2}\right) + 2n \quad (T(1)=1, \underline{n=2^k})$$

비교 총 횟수

$\overline{2n} \quad 2^{k+1}$

n
 \downarrow
 $n/2$ 후보를 반으로 $\frac{n}{2}$ 로 줄임
 \downarrow
 $n/4$ 후보를 $n/2^2$ 로 줄임
 \downarrow
 $n/8$ 후보를 $n/2^3$ 로 줄임
 \vdots
 $\square \square \quad 2n$
 $\downarrow \quad 2 \cdot 2 = 4$
 $\square \quad 1 \geq n$

$n \quad 2^k \quad 2^{k+1}$

$\overline{\frac{n}{2}} \quad \frac{n}{2^2} \quad \frac{n}{2^3} \quad \vdots$

$\overline{2^2 \cdots + 2^k + 2^{k+1}}$

$\text{총 비교 횟수} = 2^2 + 2^3 + \cdots + 2^k + 2^{k+1}$
 $= \frac{2^2(2^k - 1)}{2 - 1} = 2^{k+2} - 2^2 = \boxed{4n - 4}$
 $(n=2^k)$

점화식으로 풀어보면,

$$T(n) = T\left(\frac{n}{2}\right) + 2n, T(1)=1$$

① $\alpha_m = \alpha_{\frac{n}{2}} + 2n, \alpha_1 = 1$ 처럼 2번째.

② $n = 2^k$ 라고 가정 (제거의 규칙 사용)

$$\begin{aligned}
 T(n) &= T\left(\frac{n}{2}\right) + 2n \\
 &\downarrow \\
 T\left(\frac{n}{2}\right) &= T\left(\frac{n}{2^2}\right) + 2\left(\frac{n}{2}\right) \\
 &\therefore T\left(\frac{n}{2^2}\right) + n \text{ 은 대입} \\
 &= \left(T\left(\frac{n}{2^2}\right) + n\right) + 2n \\
 &= T\left(\frac{n}{2^2}\right) + n + 2n \\
 &\vdots \\
 &= T\left(\frac{n}{2^k}\right) + \frac{n}{2^{k-2}} + \cdots + \frac{n}{2^1} + \frac{n}{2^0} + 2n \\
 &= T(1) + n \left(2 + 1 + \frac{1}{2^1} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k-2}}\right) \\
 &\leq 1 + n \left(2 + \frac{1}{1-\frac{1}{2}}\right) \\
 &\leq 4n + 1
 \end{aligned}$$

$\therefore T(n) \leq 4n + 1 = O(n)$

- v. 결국, 선형시간에 k 번째로 작은 수를 고를 수 있다. 여기서 후보를 매번 반씩 줄이는 것은 매우 효율적인 방법이다. 그런데 반드시 반씩 줄일 필요는 없다. 예를 들어, cn 번의 비교로 $2/3$ 씩 줄여도 Big-O 표기로는 모두 $O(n)$ 시간에 가능하다 (계산해보면 쉽게 알 수 있으니 직접 해볼 것!)

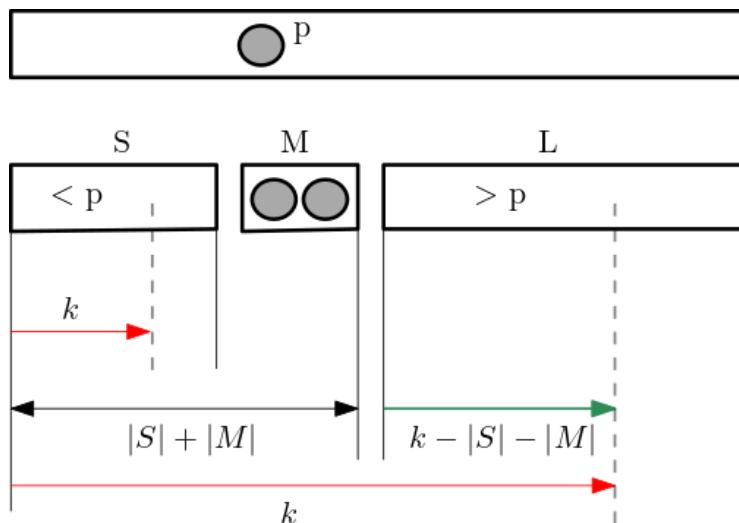
e. Quick Select 알고리즘



위의 아이디어를 아주 단순하게 구현한 알고리즘 (수행시간의 함정이 있으니 주의할 것!)

- 알고리즘 단계: (아래 그림을 함께 보면서)
 - 현재의 후보 중에서 임의로 수 하나를 선택
 - pivot이라 부르고 p 로 표기
 - 후보 수를 하나씩 p 와 비교하여 p 보다 작은 수는 집합 S 에, 같은 수도 있을 수 있으니 같은 수는 집합 M 에, p 보다 큰 수는 집합 L 로 분류
 - $|S| > k$ 라면: (즉, p 보다 작은 수가 k 개가 넘기 때문에 k 번째로 작은 수는 S 에 있음. 따라서 S 에서 k 번째로 작은 수를 찾으면 됨)
 - $|S| + |M| < k$ 라면: p 보다 작거나 같은 수가 k 개 미만이라면 k 번째로 작은 수는 당연히 L 에 있어야 함. 그러면 L 에서 k 번째로 작은 수를 찾으면 된다 (진짜?)
 - 위의 두 경우가 아니라면: 당연히 k 번째로 작은 수는 M 에 있고, 그 수는 바로 피봇 p 임. p 를 리턴함

- 그림으로 설명해보자~



iii. Pseudo code:

```

def quick_select(A, k): # L에 있는 수 중에서 k번째로 작은 수 리턴
    p = A[0] # pivot
    S = L = M = []
    M.append(p)
    for x in L: # 분류: 비교 횟수는?
        if x < p: S.append(x)
        elif x > p: L.append(x)
        else: M.append(x)

    # 재귀 호출 준비
    if |S| >= k: # 그림에서의 첫 번째 경우
        return quick_select(S, k)
    elif |S|+|M| < k: # 그림에서의 두 번째 경우
        return quick_select(L, k-|S|-|M|)
    else:
        return p

```

- iv. 원래 문제가 작은 문제로 쪼개져서 같은 함수를 호출 → 재귀 호출 → 수행 시간이 점화식으로 표현됨
- v. S 또는 L이 선택되어 재귀 호출됨. S와 L의 크기는 입력마다 피봇마다 천차만별임!

1. 가장 좋은 경우는?

- a. S 또는 L이 A의 반 정도로 계속 줄어드는 경우 → 다음 재귀 호출로 S와 L 중에서 어느 것이 선택되더라도 A의 반이 된다!
- b. 이 경우의 수행시간의 점화식은?

$$T(n) = T(n/2) + cn$$

2. 가장 안 좋은 경우는?

- a. S 또는 L에 p를 제외한 모든 수가 몰리는 경우
- b. 이 경우의 수행시간의 점화식은?

$$T(n) = T(n-1) + cn$$

f. Quick Select 알고리즘의 근본적인 약점은?

- i. S 또는 L의 크기를 A의 $1/c$ 이내로 보장하기 위해선 피봇 선택이 중요하다
- ii. 예를 들어, A의 중간값(median)을 계산해 그 값을 피봇으로 선택하면 S 또는 L의 크기가 A의 $1/2$ 이 된다
- iii. 그러나 중간값을 알아내는 과정이 꽤 복잡하고 추가 시간이 필요하다
- iv. 대신, quick_select 알고리즘은 평균적인 수행시간(average running time)은 매우 훌륭하다 → $O(n)$ 평균 수행 시간 보장!

[고급: skip 가능] 방법1: 기대값(평균값)으로 분석해보기

1. A가 1부터 n까지의 랜덤 순열 (random permutation)이라 가정
 - 피봇 $p = A[0]$ 인데, A의 어떤 값이 pivot으로 선택될 확률은 랜덤 순열을 가정했기 때문에 $1/|A|$ 으로 모두 동일하다. 즉, A의 개별 값이 $A[0]$ 에 위치할 확률이 같기에 **피봇으로 뽑힐 확률은 $1/|A|$** 이라는 뜻이다
2. **핵심 1:** 피봇의 값에 따라 S와 L의 크기가 결정되고, S와 L에 동시에 충분한 개수의 값이 포함되어야 S와 L 중 어느 것이 선택되더라도 재귀의 깊이가 작아져 수행시간이 줄어든다. 이런 피봇을 "좋은" 피봇이라고 부르고, 아래처럼 정의해보자
3. **좋은 피봇:** $|S| \geq n/3$ 이고 $|L| \geq n/3$ 이 되는 피봇
 - 반드시 $n/3$ 이상일 필요는 없다. 1보다 큰 적당한 상수 c 에 대해, n/c 이상이 되는 피봇을 **좋은 피봇**으로 정의해도 된다
4. **질문 1:** 랜덤 순열에서 첫 번째 값을 p 를 선택한 경우, p 가 좋은 피봇이 될 확률은 얼마인가?
 - **크기 순**으로 따져서, $n/3$ 번째부터 $2n/3$ 번째 사이의 값이 피봇으로 선택된다면 S와 L에는 최소 $n/3$ 개의 값은 포함하게 된다
 - 모든 값이 피봇으로 선택될 확률은 동일하기 때문에, p 가 좋은 피봇일 확률은 $(n/3)/n = 1/3$ 이다
5. $T(n) = n$ 개의 값 중에서 k 번째 작은수를 찾기 위해 필요한 비교 횟수
6. $T(n)$ 이 일종의 확률 변수라고 생각하면 된다. 이 변수의 기대 값 (즉, 평균 값)을 계산해보자
7. p 의 확률로 좋은 피봇을 선택할 수 있고, p 에 의해 두 부분으로 나뉘는데, 한 부분의 크기는 $2n/3$ 을 넘지 않는다. 찾는 수가 $2n/3$ 개 중에 있다고 하면, $pT(2n/3)$ 의 평균시간이 필요하고, $(1-p)$ 의 확률로 좋은 피봇이 아니고 이 경우에는 재귀 시간이 $T(n)$ 을 넘지 않는다. 따라서 아래와 같은 점화 부등식이 성립한다

$$\begin{aligned} T(n) &\leq cn + pT\left(\frac{2n}{3}\right) + (1-p)T(n) \\ &= \frac{cn}{p} + T\left(\frac{2n}{3}\right) \leq 3cn + T\left(\frac{2n}{3}\right) \\ &\leq 3cn\left(1 + \frac{2}{3} + \frac{4}{9} + \dots\right) \leq 3cn \sum_{i \geq 0} \left(\frac{2}{3}\right)^i \\ &\leq 9cn = O(n) \end{aligned}$$

8. 결론: 수행시간의 기대값 (평균 수행시간)은 **$O(n)$** 으로 매우 좋은편!

[고급: skip 가능] 방법2: 점화식으로 분석해보기

1. 평균 수행시간을 점화식으로 표현해보자
2. S의 크기가 i 개라고 하자. 그러면 L의 크기는 $n-i-1$ 개가 된다. 그런데 $i = 0, 1, \dots, n-1$ 가능하기 때문에 아래와 같은 점화식이 성립한다
 - a. 왜? 항상 최악의 경우의 수행시간을 고려하기에 i 와 $n-i-1$ 중 더 큰 값을 선택해야 하므로…

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} T(\max(i, n-i-1)) + cn$$

3. 위의 식을 귀납법 (induction)으로 $T(n) \leq 4cn$ 이 성립함을 증명한다

- a. Base case: $T(1) = 1$ 로 정의하면 $T(1) = 1 \leq 4c$ 가 성립하도록 1 이상의 상수 c 를 정할 수 있으므로 base case는 성립
- b. Hypothesis case: n 보다 작은 값에 대해서 부등식이 성립한다고 가정
- c. Induction case: 위의 가정을 이용해 n 에 대해 부등식이 성립함을 보임

$$\begin{aligned} T(n) &= cn + \frac{1}{n} \sum_{i=0}^{n-1} T(\max(i, n-i-1)) \\ &\leq cn + \frac{2}{n} \sum_{i=n/2}^{n-1} T(i) \\ &= cn + \frac{2}{n} (T(n/2) + \dots + T(n-1)) \\ &\leq cn + \frac{2}{n} (4c \frac{n}{2} + \dots + 4c(n-1)) \\ &\leq cn + \frac{8c}{n} (\frac{n}{2} + \dots + (n-1)) \\ &\leq cn + \frac{8c}{n} \cdot \frac{n(n-1)}{2} - \frac{8c}{n} \frac{\frac{n}{2}(n-1)}{2} \\ &\leq cn + 4c(n-1) - 2c(n/2-1) \\ &\leq 4cn - 2 \\ &\leq 4cn \end{aligned}$$

4. 점화식 분석이 기대값 분석보다는 상수 값 ($9c$ vs. $4c$)의 크기에서 좀 더 개선된 분석이다

g. Median-of-Medians (MoM) 알고리즘:



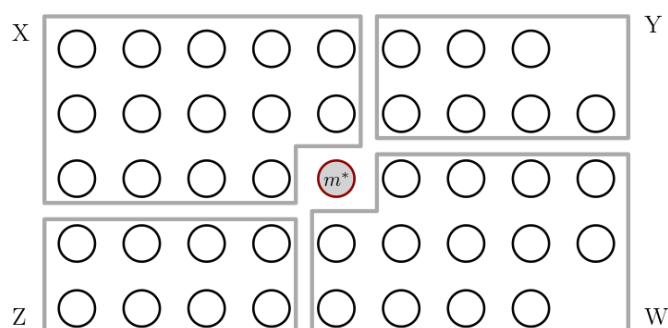
1/2(알고리즘)



2/2(수행시간분석)

- i. 강제로 S와 L의 크기를 A의 $\frac{1}{4}$ 이상, $\frac{3}{4}$ 이하가 되도록 하는 알고리즘
- ii. 아이디어는 위와 같이 분할되도록 pivot을 추가로 시간을 들여 선택하는 것!
- iii. 알고리즘 단계:

1. L의 수를 다섯 개씩 묶는다 $\rightarrow n/5$ 개의 묶음이 존재
2. 각 묶음에 대해, 다섯 개의 수들의 중간값을 찾는다 [비교: _____ 번]
 - a. 중간값보다 작은 2개의 수 \rightarrow 중간값 \rightarrow 중간값보다 큰 2개의 수 순서로 재배열한다
3. 묶음 별 중간값을 모두 모아 ($n/5$ 개), 그 값들의 중간값을 찾는다
 - a. 즉, 중간값들의 중간값(median of group medians)을 찾는다
 - b. 어떻게? \rightarrow 이 알고리즘을 $n/5$ 개에 대해 다시 재귀호출!!
 - c. 그 중간값을 m^* 라 한다
4. 각 묶음을 재배열하는데, 묶음의 중간값이 m^* 보다 작으면 왼쪽으로 모으고, 크면 오른쪽으로 모은다. 결국, m^* 가 중간값인 묶음이 가운데에 위치한다
5. m^* 를 기준으로 네 구역 X, Y, Z, W로 나눈다 (그림 참조)
 - a. X의 있는 값들은 모두 m^* 보다 작고, W에 있는 값들은 모두 m^* 보다 크다. Y, Z의 값들은 작을 수도 클 수도 있다
6. $S = X + \{Y \text{와 } Z \text{의 값 중에서 } m^* \text{보다 작은 값}\}$
 $L = W + \{Y \text{와 } Z \text{의 값 중에서 } m^* \text{보다 큰 값}\}$
 $M = m^* \text{와 같은 값}$
7. $|S| > k$ 이면 k 번째로 작은 값이 S에 있으므로 S에 대해 재귀호출
8. $|S| + |M| < k$ 이면 k 번째로 작은 값이 L에 있으므로 L에 대해 재귀호출
9. 위의 두 경우가 아니라면, k 번째로 작은 값이 m^* 라는 의미이므로 리턴



iv. 비교횟수 $T(n)$ 을 단계별로 세보자

1. 0번
2. 5개의 중간값을 찾기 위해 필요한 비교횟수는 최대 몇 번? 넉넉히 **10번**
이내의 비교로 찾을 수 있다고 하자
 - a. **$10*(n/5) = 2n$** 번으로 중간값 찾고,
 - b. 그룹내에서의 재배열은 물리적으로 하지 않음! (그래서 비교 불필요)
0번
3. **$T(n/5)$** 번 (재귀호출하므로)
4. 이 재배치도 물리적으로 수행하지 않음. 비교 불필요. 0번
5. 0번
6. X, Y, Z, W의 모든 값을 m*와 비교하여 S와 L을 계산하면 되므로 총 **n**번
비교
7. 8. 9. 재귀호출:
 - a. $|X| \geq n/4, |W| \geq n/4$
 - b. $|S| \leq n - |W| \leq 3n/4, |L| \leq n - |X| \leq 3n/4$
 - c. **$T(3n/4)$** 번의 비교가 (재귀호출에 의해) 최악의 경우에 필요함

$$\begin{aligned} T(n) &\leq T(3n/4) + T(n/5) + 2n + n \\ &= T(3n/4) + T(n/5) + 3n \end{aligned}$$

v. 이 점화식을 계산해보자

1. **$T(n) \leq 60n$** 임을 induction 방법으로 증명해보자!
 - a. $n \leq 5$ 일 때 (base-case step) 성립함을 보인다. (왜 $n = 1$ 인 경우를 생각하지 않을까? 이유는 induction step에서 설명한다)
 - i. (최대) 5개의 값에서 selection하는 것으로 5개의 값을 정렬할 때 필요한 비교 횟수면 충분하다. 최대 값을 차례로 찾아 정렬하는 경우에는 $4 + 3 + 2 + 1 = 10$ 번의 비교로 충분하다. 10번의 비교 횟수는 $T(5) = 300$ 을 넘지 않으므로 점화식이 성립!
 - b. $< n$ 일 때 점화식이 성립한다고 가정한다 (hypothesis step)
 - c. $= n$ 일 때 점화식이 성립함을 보인다 (induction step)
 - i. $T(n) \leq T(3n/4) + T(n/5) + 3n$
 - ii. $T(3n/4)$ 와 $T(n/5)$ 은 $n \leq 5$ 인 n 에 대해서 $3n/4$ 와 $n/5$ 는 모두 n 보다 작은 수이므로 점화식 가정을 적용할 수 있다. 따라서 $T(3n/4) \leq 60(3n/4) = 45n$ 이고, $T(n/5) \leq 60(n/5) = 12n$ 이 성립한다
 - iii. 따라서, $T(n) \leq 45n + 12n + 3n = 60n$ 으로 부등식이 성립한다 (증명 끝)

vi. 앞에서 설명한 알고리즘은 이해를 돋기 위해, 실제 구현에서는 필요하지 않은 단계를 추가한 것이다. 실제 구현에서는 위의 단계를 반드시 따를 필요는 없다

vii. 아래 python-like pseudo 코드를 이해해보고, 빈 칸에 들어갈 내용을 채워보자!

```
def MoM(A, k):
    if |A| == 1: return A[0] # no more recursion
    S, L, M, medians = [], [], [], []

    i = 0
    while i+4 < |A]:
        mediants.append(find_median_five(L[i:i+5]))
        -----
        if i < |A| and i+4 >= |A]:
        -----
        mom = MoM(_____, _____)
        for v in A:
            if v < mom: _____
            elif v > mom: _____
            else: M.append(v)
            if _____: return MoM(_____, _____)
            elif _____: return MoM(_____, _____)
            else: return mom
```

viii. 점화식을 세우고 계산하는 연습이 반드시 필요하다!

ix. 질문: MoM은 왜 5개씩 묶어 뒤집음별 중간값의 중간값을 구하는 식으로 진행했을까?
3개씩 묶거나 7개씩 묶으면 안될까?

1. 7개씩 묶는다면, 중간값의 중간값을 계산하는 데 $T(n/7)$ 시간이 필요하고 재귀 호출에는 5개의 경우와 같이 $T(3n/4)$ 시간이 필요하다. 따라서 $T(n) = T(n/7) + T(3n/4) + c_1n$ 이 되고 점화식을 풀어보면 여전히 $O(n)$ 이 된다.
5개씩 묶는 경우와 다른 점은 Big-O 표기에 생략된 상수가 더 커진다는 점 뿐이다
2. 3개씩 묶으면, Big-O 시간의 상수가 줄어들지 않을까? 3개씩 묶으면 $T(n) = T(n/3) + T(3n/4) + c_2n$ 이 된다. 그런데 이 점화식을 풀면 $O(n)$ 보다 커진다. 왜 그럴까? 크기가 n 인 문제를 $n/3$ 크기와 $3n/4$ 크기의 작은 두 문제로 분할한다. 그런데 $n < n/3+3n/4 = 13n/12$ 가 되어 분할 후 문제의 총 크기가 커져 오히려 분할의 이점이 없게 된다. 반면에 7개씩 묶는 경우엔 $n/7 + 3n/4 = 25n/28 < n$ 이 되어 분할이 시간을 줄일 수 있게 되어 $O(n)$ 이 그대로 유지된다

6. [고급 - skip 가능] 선택 문제의 비교 횟수의 하한 (lower bound) - 악당과의 게임



선택 문제의 비교 횟수의 하한 (lower bound)를 증명하는 대표적인 방법인 adversary argument (악당과의 게임)이라는 증명 기법을 소개한다. 어려운 내용은 아니지만 일반적인 강의 범위를 벗어나므로 건너 뛰어도 된다

- a. 예 1: 서로 다른 n 개의 값 중에서 최대 값을 구하는 문제는 최소 $n-1$ 번의 비교 필요
 - i. 1장의 arrayMax 알고리즘에서는 왼쪽 값 ($A[1]$)부터 차례대로 값을 보면서 현재까지의 최대 값을 유지하는 방법이다. 이 경우 총 $n-1$ 번의 비교를 수행한다. 이 비교횟수는 상한 (upper bound)이다. 즉, 어떤 입력에 대해서도 $n-1$ 번의 비교면 항상 최대 값을 찾을 수 있다는 의미이다
 - ii. 하한 역시 $n-1$ 번일 것이라고 쉽게 추측할 수 있다. 그보다 비교횟수가 작다면 최소한 하나 이상의 값이 비교에 참여하지 못할 것 같기 때문이다. 그런데 두 개씩 짹을 지어 토너먼트 방식처럼 비교하면 $n/2$ 번의 비교로 모든 값이 최소한 한 번씩 비교에 참여하게 된다. 따라서 하한이 $n-1$ 이라는 게 당연하게 증명되지는 않는다
 - iii. 악당과의 게임 증명 방법은 알고리즘(나)과 악당(adversary)이 서로 게임을 하는 것이다. 알고리즘은 두 수를 선택해 두 수의 비교 결과를 악당에게 물어보고, 악당은 그 결과를 알려주는 방식으로 진행된다. 알고리즘은 비교횟수를 최소로 하려 하고, 악당은 비교횟수를 늘리기 위해 비교 결과를 결정해 알려주는 게임이다
 - iv. 알고리즘 입장에서는 실제 값이 중요한 것이 아니라 두 수의 비교 결과가 중요하다. 악당의 입장에서는 비교 결과를 알려줘야 하는데 n 개의 값을 미리 결정하지 않고 알고리즘이 비교를 많이 하도록 최대한 나중에 결정한다 (그러나 비교 결과는 최종적으로 정해지는 값의 비교 결과와 일치해야 한다)
 - v. 각 값의 상태를 N, W, L 세 가지로 구분한다
 - 1. N: 비교에 전혀 참여하지 않은 상태 (초기에 모든 값은 상태 N이 됨)
 - 2. W: 한 번 이상의 비교에 참여했고 모두 이긴 상태 (비교 값보다 모두 컸다는 의미)
 - 3. L: 한 번 이상의 비교에 참여했고 최소한 한 번은 패배한 상태
 - vi. 초기에는 n 개의 값의 상태가 모두 N \rightarrow 알고리즘의 마지막에는 한 개의 값의 상태만 W이고 나머지 $n-1$ 개의 상태는 L이 되어야 한다 (모든 값이 서로 다르다고 가정했기 때문에, 최대 값의 상태는 W이고 나머지는 한 번 이상은 비교에서 패배해야 하기에 상태가 L이 되어야 한다.)
 - 1. 어떤 알고리즘도 최대 값을 찾기 위해서는 L 상태의 값이 $n-1$ 개는 되어야 한다. L 상태를 하나의 정보라고 한다면 어떤 알고리즘도 $n-1$ 개의 정보를 반드시 얻어야 한다
 - vii. 이제 알고리즘이 두 수를 선택해 비교 결과를 악당에게 요구하면, 악당은 알고리즘에게 **정보를 되도록 주지 않는 비교 결과**를 정해 알려줘야 한다. 아래 표와 같이 대답을 해보자 (악당의 입장에서 생각해보기)

x, y 의 비교 전 상태	악당의 답	x, y 의 비교 후 새로운 상태	알고리즘이 얻는 새로운 L의 개수
N N	$x > y$	W L	1
N W	$x < y$	L W	1
N L	$x > y$	W L	0
W W	$x > y$ 또는 $x < y$ L이 되는 수의 값을 재조정	W L 또는 L W	1
W L	$x > y$	W L	0
L L	두 수의 배정할 값에 따라 모순없이	L L	0

1. W N, L N 등 대칭적인 경우는 모두 생략했다
 2. 위의 표처럼 모순 없이 실제 값을 배정하려면 어떻게 해야 할까?
 - a. L 상태의 값은 $-n, -(n-1), \dots, -1$ 순서로 차례대로 배정하고, W 상태의 값은 $1, 2, \dots, n$ 순서로 차례대로 배정해보자
 - b. $N \rightarrow L$ 이 되거나 $W \rightarrow L$ 이 되는 경우엔 현재 $[-n, \dots, -1]$ 의 배정된 가장 큰 값보다 1이 더 큰 값으로 배정한다. $N \rightarrow W$ 가 되는 경우엔 $[1, \dots, n]$ 의 배정된 (가장 큰 값 + 1)인 값으로 배정한다
 - c. 악당이 이 전략으로 실제 값을 배정하면 악당의 비교 답이 모순이 발생하지 않음을 보장할 수 있나? (직접 따져 보기)
- viii. 이 표에 따르면 한 번 비교할 때 알고리즘이 얻을 수 있는 정보의 양은 최대 하나뿐이다. 따라서 **n-1 개의 L 상태를 얻기 위해선 최소 n-1번은 비교**를 해야만 한다. (당연히, 어떤 알고리즘이라도 그래야 한다.)
- ix. 결론: n개의 서로 다른 값 중에서 최대 값은 어떤 알고리즘도 n-1번의 비교를 반드시 수행해야 한다 → 최대 값을 찾는 문제의 비교 횟수의 하한은 n-1이다

b. 예 2: 서로 다른 n 개의 값 중에서 최대 값과 최소 값을 찾기 위한 비교 횟수의 하한

- i. 먼저 상한은 얼마인가? (몇 번의 비교면 항상 충분한가?) 상한: _____
- ii. 최대 값은 비교에서 모두 승리한 값이고 최소 값은 모두 패한 값이 된다. 나머지 $n-2$ 개의 값은 최소 한 번 이상 승리하고 동시에 최소 한 번 이상 패배해야 한다
- iii. 값의 상태를 N, W (항상 승리한 상태), L (항상 패배한 상태), WL (한 번 이상 승리하고 한 번 이상 패배한 상태) 네 개의 상태로 정의한다
- iv. 알고리즘은 최종적으로 $(n-2)$ 개의 WL을 얻어야 한다. 그런데 WL은 N 상태에서 바로 WL 상태가 되는 것이 아니라 $N \rightarrow W \rightarrow WL$ 또는 $N \rightarrow L \rightarrow WL$ 상태가 되므로 상태 변화가 2번 있어야 한다. 즉, $2(n-2)$ 번의 상태 변화가 있어야 한다. 나머지 두 수는 $N \rightarrow W$ 와 $N \rightarrow L$ 이 되어야 하므로 추가로 2번의 상태 변화가 있어야 한다. 따라서 **최소 $2n-2$ 번의 상태 변화** (비교로부터 얻어야 하는 정보의 양)가 있어야 한다
- v. 예처럼 비교에 따른 상태 변화 표를 만들어보자 (빈 칸을 직접 채워보자. 힌트는 비교 결과로부터 알고리즘이 얻어 가는 정보의 양이 되도록 작아야 한다는 점이다)

1. 마우스로 빈 칸을 긁으면 답을 볼 수 있음

비교 전 상태 x y	악당의 답	비교 후 상태 x y	새로운 상태 변화
N N	x > y	W L	2
N W	x < y	L W	1
N L	x > y	W L	1
N WL	x > y	W WL	1
W W	모순 없이	W WL or WL W	1
W L	x > y	x > y	0
W WL	x > y	W WL	0
L L	모순 없이	WL L or L WL	1
L WL	x < y	L WL	0
WL WL	모순 없이	WL WL	0

- vi. 위의 표의 각 비교를 통해 얻을 수 있는 정보의 양이 2, 1, 0 세 가지 뿐이다.

1. 알고리즘이 얻어야 하는 정보의 양은 **$2n-2$** 이므로, 최대 정보를 얻을 수 있는 $(N \quad N)$ 비교를 최대한 많이 해야 한다 $\rightarrow n/2$ 번 비교로 **n** 개의 정보 획득 가능

2. 이제 N 상태인 값이 하나도 없다. 이 후의 모든 비교는 최대 1의 정보만 제공하므로 남은 정보 $n-2$ 를 위해선 최소 $n-2$ 번의 추가 비교가 필요하다
3. 따라서 총 $3n/2-2$ 번의 비교가 필요하다 (상한과 일치한다!)

c. 예 3: 서로 다른 n개의 값 중에서 가장 큰 값과 두 번째 큰 값을 찾기 위한 비교 횟수의 하한

i. 상한: _____

ii. 가장 큰 값을 M, 두 번째 큰 값을 m이라 하자

사실 1: m 보다 큰 값이 정확히 하나 뿐이고 작은 값은 $n-2$ 개여야 한다. 이것은 m을 알기 위해서는 최대 값 M도 알아야 한다는 뜻이다

사실 2: M을 제외한 다른 값에 패한 값은 m이 될 수 없다

iii. M을 알아야 하기에 최소 $(n-1)$ 번 비교가 필요하다

iv. M에 패한 값의 개수가 x개라면 사실 2에 의해 x개 중 하나를 제외한 $(x-1)$ 개는 M이 아닌 다른 값과 한 번 더 비교해서 쳐야 한다. 따라서 총 $(n-1)+(x-1)$ 번의 비교가 반드시 필요하다

v. 결국, M과 비교하는 값의 최소 개수 x를 구하면 된다

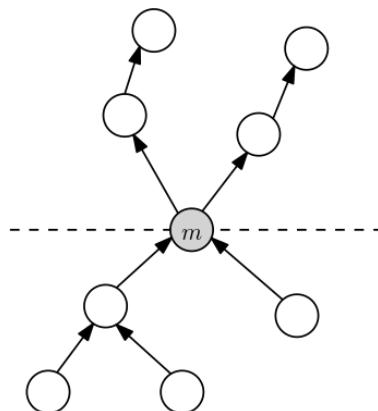
vi. 예 1, 예 2와 다르게, 값 x에 가중치를 $w(x)$ 를 부여한 후, 악당과의 게임을 한다. 초기 값은 $w(x) = 1$ 로 정한다. 악당과의 게임은 아래 표처럼 진행한다

경우	악당의 답	가중치 수정
$w(x) > w(y)$	$x > y$	$w(x) = w(x)+w(y)$ $w(y) = 0$
$w(x) = w(y) > 0$	$x > y$	$w(x) = w(x)+w(y)$ $w(y) = 0$
$w(x) = w(y) = 0$	보순이 없도록	수정 없음

- vii. 우선, 비교를 한 후에는 패자의 가중치가 승자의 가중치에 더해지므로 가중치의 총 합은 n으로 유지된다. 그리고 한 번만 지더라도 가중치가 0이 된다. 그 이후엔 계속 0을 유지한다
- viii. 최대 값 M은 항상 이기고, 이길 때마다 상대방의 가중치 값을 가져오기에 가중치가 최대 두 배씩 증가한다. 전체 가중치의 합이 n으로 유지되고 M을 제외한 다른 값은 0이 되어야 한다. 따라서 M은 최소 $\log n$ 번 이상의 비교에 참여하게 된다. 즉, $x \geq \log n$
- ix. 결론: 가장 큰 값과 두 번째로 큰 값을 찾기 위해선 최소 $n + \log n - 2$ 번의 비교가 필요하다

d. 예 4: 서로 다른 n 개의 값 중에서 중간 값 (median)을 찾기 위한 비교 횟수의 하한

- i. 상한: 교재에서 설명한 MoM 알고리즘의 비교 횟수는 $60n$ 이하의 비교면 충분하다. 현재까지 알려진 가장 작은 비교 횟수 상한은 $2.95n$ 이다
- ii. n 이 홀수고 모든 값이 다르다고 가정한다. 그러면 중간 값 m 은 m 보다 큰 값이 $(n-1)/2$ 개이고, 작은 값은 $(n-1)/2$ 개이어야 한다
 1. 비교에서 진 값에서 이긴 값으로 화살표를 그리면 아래와 같은 비교 트리가 정의된다 ($n = 9$ 일 때 예제>)



- iii. 결국 위 그림과 같은 비교 트리가 정의되기 때문에 $n-1$ 번의 비교는 피할 수 없다
- iv. 이 트리의 에지에 해당하는 비교를 "중요"하다고 정의하자. 악당은 중요하지 않은 비교를 더하도록 비교 결과를 결정할 것이다
- v. 값의 상태를 N, L, S로 나누고, L은 중간 값보다 큰 값을 할당한 상태이고, S는 중간 값보다 작은 값을 할당한 상태라고 하자
- vi. 다음처럼 질의-응답을 해보자
 1. N N : 값 하나는 m 보다 크고, 다른 하나는 m 보다 작은 값 할당 후 응답
 2. L N : N 값에는 m 보다 작은 값 할당 후 응답
 3. S N : N 값에는 m 보다 큰 값 할당 후 응답
 4. L L, S S, L S : 할당된 값에 따라 모순이 없도록 응답
- vii. 위의 4가지 형태의 비교 중에서 1, 2, 3의 비교는 m 보다 큰 값과 작은 값 사이의 "중요하지 않은" 비교가 된다. 중요하지 않은 비교를 통해 N \rightarrow L 또는 S로 변화시키는 가장 빠른 방법은 1번 비교인 N N을 $(n-1)/2$ 번 하는 것이다. 즉, 중요하지 않은 비교 횟수는 최소 $(n-1)/2$ 번이다
- viii. 중요한 비교 $n-1$ 번과 중요하지 않은 비교 $(n-1)/2$ 번을 더한 $3n/2 - 3/2$ 번이 하한이다
- ix. 현재 상한은 $2.95n$ 이고 하한은 $1.5n$ 정도로 차이가 존재한다. 이는 최소 횟수의 비교를 통해 중간 값을 구하는 문제가 완전히 해결되지 않았다는 것을 뜻한다!

4. Divide and Conquer (분할정복법)



1. 원래 문제를 풀 수 있을 정도의 작은 크기의 부분 문제로 분할해 각각 해결한 후, 부분 문제의 해답을 잘 모아 원래 문제의 해답을 얻는 방법
 - a. 크기가 작아질 뿐 문제 자체는 변하지 않기 때문에, 분할은 **재귀적인 분할**이 된다
 - b. 재귀적인 분할이므로 분할정복 방법의 알고리즘 수행시간 $T(n)$ 은 **점화식**으로 표현되는 것이 일반적이다 (앞의 재귀 편의 설명 참조)
 - c. Selection 편에서 배운 quick_select 알고리즘과 median_of_medians 알고리즘 모두 분할정복 알고리즘들이다!
2. 예1: n개의 수 중에서 최대 값 구하기
 - a.

```
def find_max1(A):
    if len(A) == 1: return A[0]
    else: return max(A[0], find_max1(A[1:]))
```
 - b.

```
def find_max2(A):
    if len(A) < 1: return -infinity # -infinity = 매우 작은 값
    elif len(A) == 1: return A[0]
    else:
        return max(find_max2(A[:len(A)//2]), find_max2(A[len(A)//2:])))
```
 - c. 두 함수 모두 slicing 복사를 하는데 이 복사 시간은 포함하지 않고 수행시간을 분석해 보자
 - d. **find_max1:**
 - i. 함수 호출 순서 + 리턴 값 반환 순서를 그려보자
 - ii. 함수의 수행시간 $T(n)$ 을 점화식으로 나타내고, 전개한 후, Big-O로 표기하자
 - iii. $T(n) = T(n-1) + c, T(1) = c \rightarrow T(n) = cn = O(n)$
 - e. **find_max2:**
 - i. 함수 호출 순서 + 리턴 값 반환 순서를 그려보자
 - ii. 함수의 수행시간 $T(n)$ 을 점화식으로 나타내고, 전개한 후, Big-O로 표기하자
 - iii. $T(n) = 2T(n/2) + c, T(1) = c \rightarrow T(n) = c(1 + 2 + \dots + 2^k) \leq 2cn = O(n)$
 - f. 두 함수의 차이점은 무엇인가? 선형 재귀 호출 vs. 이진 재귀 호출

3. 예2: a^n 을 계산하기 (n 을 양의 정수로 가정)

```
a. def power1(a, n): # 선형 재귀 호출로 작성하기
    if n == 1:
        return a
    return a * power1(a, n-1)
```

수행시간 점화식: $T(n) = T(n-1) + c = \dots = O(n)$

```
b. def power2(a, n): # 이중 재귀 호출로 작성하기
    if n == 0: return 1
    if n%2 == 1: # n 홀수
        return power2(a, n//2)*power(a, n//2)*a
    else: # n 짝수
        return power2(a, n//2)*power(a, n//2)
```

수행시간 점화식: $T(n) = 2T(n/2) + c = \dots = O(n)$

```
c. def power3(a, n): # 선형 재귀 호출이지만, 빠른 방법
    if n == 0: return 1
    p = power2(a, n//2)
    if n%2 == 1: # n 홀수
        return p * p * a
    else: # n 짝수
        return p * p
```

수행시간 점화식: $T(n) = T(n/2) + c = \dots = O(\log n)$

d. x^y 를 계산하는 함수 $\text{pow}(x, y)$ 작성 (y 를 임의의 정수로 가정)

i. 위의 함수 power3 을 활용해서 작성해보자. (y 가 음수일 수 있다!)

```
def pow(x, y):
    if y < 0:
        return pow(1.0/x, -y)
    나머지는 위의 power3의 재귀 코드와 동일
```

ii. 아래 함수는 어떤 결과를 낼까?

- 힌트: $3^{11} = 3^{1011} = 3^{1000} * 3^{0010} * 3^{0001}$

```
def pow(x, y):
    result, power = 1.0, y
    if y < 0:
        power, x = -power, 1.0/x
    while power:
        if power & 1:
            result *= x
        x, power = x * x, power >> 1
    return result
```

4. [🎤 인터뷰 문제] 두 정수 $a * b$ 계산하기

- 단, * 연산을 사용하지 말고, +, >>, % 연산만을 (최대한 적게) 사용해 계산해야 한다면?
- 위의 a^n 계산에 사용한 power3 방법을 사용해보자.
 - >> 연산은 / 대신 사용하는 연산임
 - +, >>, % 연산을 각각 몇 번씩 사용하게 되는가?

5. 예3: 피보나치수(Fibonacci number) 구하는 5가지 방법

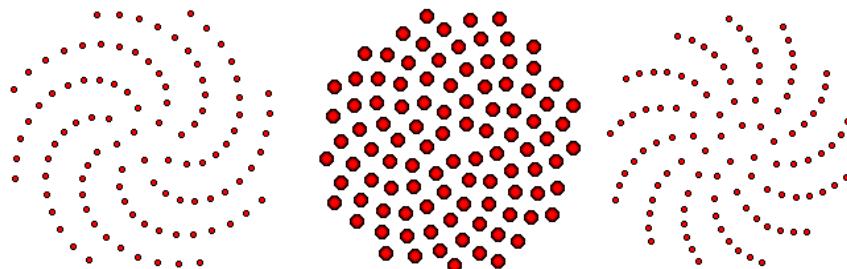
피보나치 수: 0, 1, 1, 2, 3, 5, 8, ...

- $F_0 = 0, F_1 = 1$ 로 시작
- $F_n = n$ 번째 피보나치 수 = $F_{n-1} + F_{n-2}$

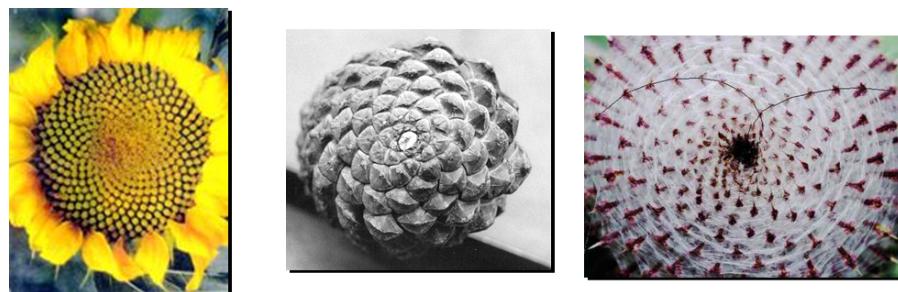
- 피보나치와 자연 [이언 슈트어트, 생명의 수학(The Mathematics of Life) 추천]



- 꽃잎의 수 : (3: 백합, 5, 들장미, 8, 기생초, 13: 금잔화, 21: 치커리, 34: 데이지)
- 이웃한 두 잎의 바퀴 수: ($\frac{1}{2}$: 잔디, $\frac{1}{3}$: 개암나무, $\frac{5}{8}$: 살구나무, $\frac{5}{13}$: 배나무, 5/13: 베드나무) Why?



- 꽃메미가 종류에 따라, 7년 또는 14년 주기로 등장해 짹짓기 (Why?)
- 꽃 또는 열매의 나선 방향에 따른 나선 수



- Golden Ratio(황금률)

$$\frac{a+b}{b} = ab = 1.618\dots = \phi$$

- 황금률 ϕ 는 두 인접한 피보나치 수의 비율과 유사: $\phi = F_{n+1}/F_n$
- 실제 n 번째 피보나치 수 F_n 의 정확한 값은 아래와 같다. 이는 $O(\phi^n)$ 정도의 값으로 매우 큰 값이다

$$F_n = \frac{\phi^n - (1 - \phi^n)}{\sqrt{5}}$$

c. **분할 정복 방법 1: [slow]**

```
def fibonacci1(n): # n번째 피보나치 수를 리턴
    if n <= 1: return n
    return fibonacci1(n-1) + fibonacci1(n-2)
```

i. 수행시간 $T(n)$ 의 점화식과 풀이는?

$T(n) = T(n-1) + T(n-2) + 1$ # 재귀적으로 두 수 구한 후 덧셈 1번
 $T(n) + 1 = T(n-1) + 1$ $T(n-2) + 1$ # 양변에 1 더한 후, $S(n)$ 으로 치환
 $S(n) = S(n-1) + S(n-2)$, $S(0) = T(0)+1 = 1$, $S(1) = T(1)+1 = 2$

$S(n) = 1, 2, 3, 5, 8, \dots$ # 결국 $S(n) = F_{n+1}$ 이 됨!
 $T(n) = S(n) - 1 = F_{n+1} - 1 = O(\phi^n)$

ii. fibonacci1 방법은 결국 **지수시간**이 걸리는 매우 느린 알고리즘이다

d. **분할 정복 방법 2: [fast]**

$$\begin{pmatrix} F_{k+2} \\ F_{k+1} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_{k+1} \\ F_k \end{pmatrix}, \begin{pmatrix} F_{n+1} \\ F_n \end{pmatrix} = A^n \begin{pmatrix} F_1 \\ F_0 \end{pmatrix}$$

결국, 이차원 행렬 A 의 n 승, A^n 을 구하면 된다. power3(A , n)의 방식을 응용하면 $O(\log n)$ 시간이면 충분하다

```
def power(A, n):
    if n == 1: return A
    B = power(A, n//2)
    if n%2 == 0:
        return mult_matrix(B, B) # mult_matrix는 2x2 두 행렬 곱셈 함수
    else:
        return mult_matrix(A, mult_matrix(B, B))

def fibonacci2(n): # power3(A, n)의 분할정복방법으로...
    if n <= 1: return n
    An = power([[1,1], [1,0]], n)
    return An[1][0]
```

i. 수행시간 $T(n)$ 의 점화식과 전개한 결과는?

- 실수 a 대신 2x2 행렬 A 가 사용되는 것으로 $T(n) = T(n/2) + c = O(\log n)$ 이면 충분하다

e. **분할 정복 방법 3:**

i. 다음의 증명된 사실을 이용해보자

- $F_{2k} = F_k^2 + 2F_kF_{k-1}$
- $F_{2k+1} = F_{k+1}^2 + F_k^2$

```
def fibonacci3(n):
    if n == 0 or n == 1: return n
    k = n//2
    Fk = fibonacci3(k)
    Fk_minus_1 = fibonacci3(k-1)      # T(n/2) 시간 필요
    Fk_plus_1 = Fk_minus_1 + Fk       # T(n/2-1) 시간 필요

    if n % 2 == 0: # even
        return Fk*Fk + 2*Fk*Fk_minus_1
    else: # odd
        return Fk_plus_1*Fk_plus_1 + Fk*Fk
```

ii. 수행시간 $T(n)$ 의 점화식을 세우고 전개해 Big-O로 표기해보자

- $T(n) = T(n/2) + T(n/2 - 1) + c$
 $\leq 2T(n/2) + c = \dots = O(n)$

iii. 방법 2와 비교해 어떤 차이가 있는가?

f. **선형 방법 4:** 리스트를 이용한 방법 (모든 피보나치 수를 계산)

- i. n 개의 피보나치 수를 리스트에 담아 모두 계산하는 방법 (재귀적인 방법이 아님!)
- ii. 리스트 $F[0] = F_0 = 0$, $F[1] = F_1 = 1$ 이라 정하고, $F[i] = F[i-1] + F[i-2]$ 을 통해 i 번째 피보나치 수를 계산함!

```
def fibonacci4(n):
    F = [0, 1]
    for i in range(2, n+1):
        F.append(F[i-1] + F[i-2])
    return F[n] # return F
```

iii. 위 함수의 수행시간은? $O(n)$

반복문을 $n-1$ 번 반복하고 상수 시간 연산을 수행하므로

g. **선형 방법 5:** 두 변수만 사용한 방법

- i. n 번째 피보나치 수를 리스트를 이용하지 않고 변수 두 개만으로 계산하는 방법 (모든 피보나치 수를 저장할 필요 없으니 리스트를 사용할 이유 없음)

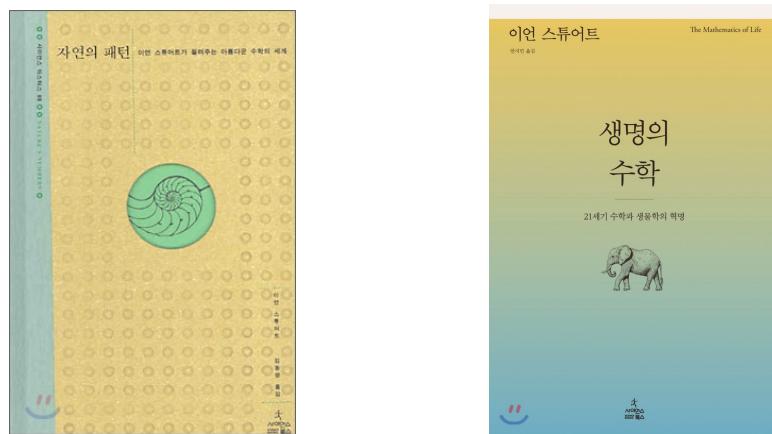
```
def fibonacci5(n):
    a, b = 0, 1
    for i in range(2, n+1):
```

```
a, b = b, a+b
return b
```

- ii. 위 함수의 수행시간은? 당연히 $O(n)$

6. [책 소개] 자연에서 나타나는 수의 오묘함

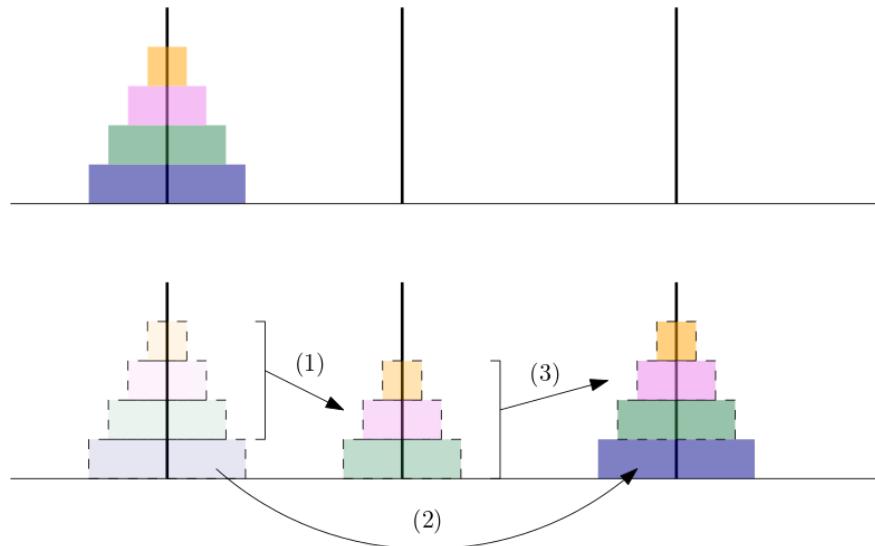
- a. 영국의 수학자이자 과학 저술가 **이언 스튜어트** (Ian Stewart)의 두 책을 소개합니다
- b. 자연이 품고 있는 패턴을 수로 표현하고 수의 아름다움에 관한 책으로 피보나치 수 등의 흥미로운 내용으로 가득차 있다!
 - i. **자연의 패턴**: 이언 스튜어트가 들려주는 아름다운 수학의 세계
 - ii. **생명의 수학**: 21세기 수학과 생물학의 혁명



[이미지 출처: yes24.com에 제공하는 책 이미지]

7. 예4: 하노이 탑(Hanoi Tower)

- a. 한 막대에 원반의 크기 순서(큰 원반이 밑에, 작은 원반은 위에)로 쌓여 있을 때, 큰 원반이 작은 원반위에 놓이면 안되는 규칙을 만족하면서 다른 막대로 옮기고 싶다. 이 때 필요한 원반의 최소 이동 횟수는 얼마인가?



- b. **분할**: $(n-1)$ 개와 가장 큰 1개로 분할 → **정복**: $(n-1)$ 개를 옮기는 문제를 2번 재귀적으로 해결함
- c. $(n-1)$ 개를 먼저 가운데 막대로 (재귀적으로) 옮김: $T(n-1)$ 번 이동
 - i. 왜 재귀적으로 가능한가? 가장 큰 원반 위에는 나머지 원반이 자유롭게 놓일 수 있기 때문에
- d. 가장 큰 1개를 오른쪽 막대로 옮김 : 1번 이동
- e. 가운데 막대의 $(n-1)$ 개를 오른쪽 막대로 옮김: $T(n-1)$ 번 이동
- f. $T(n) = n$ 개 원반을 다른 막대로 옮기는 데 필요한 횟수 = $2T(n-1) + 1$
- g. $T(n) = 2T(n-1) + 1$, (단, $T(1) = 1$,)

$$= 2^2T(n-2) + (2 + 1) = 2^{n-1} + \dots + 2 + 1 = 2^n - 1 = O(2^n)$$
- h. 하노이 탑 문제의 이동 횟수는 정확히 $2^n - 1$ 으로 지수 시간이 필요하다
 - i. 예를 들어, 원반 하나를 막대에서 빼 다른 막대로 옮기는 과정을 1초에 할 수 있다고 하자
 - ii. 원반 개수가 $n = 20$ 이라면, 정확히 $2^{20}-1$ 초가 걸리고 이는 약 12일이 된다!

8. 예5: 이진 탐색(Binary Search)



[이진탐색]



[첨화식]

- a. [워밍업 퀴즈] Oxford English Dictionary 2판은 1989년 3월에 출간되었는데, 21,730 페이지에 이르는 방대한 영어 사전이다
 - i. 여기서 Prometheus 단어를 찾아보자. 단, P로 시작하는 단어의 첫 페이지로 가지 않고, 사전을 펼치는 작업을 반복해서 찾아보자. 물론 펼치는 횟수가 적을수록 좋다
 - ii. 가장 좋은 방법은 21,730 페이지의 반이 되는 10,865 페이지를 펼쳐 Prometheus가 있는지 살펴본다. 없다면 그 페이지를 기준으로 사전을 두 부분으로 찢는다. (그냥 상상하는 거니 아깝다고 생각할 필요는 없다)
 - iii. 원하는 단어는 두 부분 중 하나에 들어 있을테고, 나머지 반은 버린다 (이렇게 버려도 되는 근본적인 이유가 무엇인가?)
 - iv. 이 과정을 반복하면 Prometheus가 있는 1장만 남게 된다 (만약, 이 단어가 사전에 없다면, 어떤 페이지도 남지 않게 되겠지만…)
 - v. 이 과정이 이진탐색이다. 21,730 페이지의 사전을 최대 몇 번 펼쳐보면 원하는 단어를 항상 찾을 수 있을까?

- b. 리스트 A에 n개의 수가 저장되어 있다. 어떤 수 K가 A에 저장되어 있는지 알고 싶다. 최소 몇 번의 비교로 찾을 수 있을까? (항상 최악의 시나리오를 가정해야 한다.)

- c. K가 A에 없는 경우에는 A의 모든 수를 살펴봐야 하므로 n번의 비교는 반드시 필요하고, 이 경우가 최악의 경우이다. 어떤 경우에도 n번의 비교는 피할 수는 없다

- d. 만약, A의 수들이 오름차순으로 정렬된 상태라면 더 적은 비교로 가능할까? (당연히, 오름차순 순서를 이용해야 한다!)

- e. **IDEA:** 한 번의 비교로 탐색 범위를 반을 줄일 수 있다!
 - i. if $A[n/2] < K$: 범위가 $A[0:n] \rightarrow A[n/2+1, n]$ 으로 줄고
 - ii. if $A[n/2] > K$: 범위가 $A[0:n] \rightarrow A[0:n/2]$ 로 준다
 - iii. if $A[n/2] == K$: $A[n/2]$ 가 우리가 찾던 수이다

- f. Slicing을 이용한 재귀적으로 이진 탐색하기

```
def binary_search(A, K): # K 존재하면 인덱스 리턴, 존재안하면 -1 리턴
    if len(A) < 1:
        return -1 # K가 A에 존재하지 않으므로
    m = len(A)//2
    if A[m] < K:
        return binary_search(A[m+1:], K)
    elif A[m] > K:
        return binary_search(A[:m], K)
```

```

else: # A[m] == k
    return m

```

- i. 수행시간 $T(n)$ 의 점화식을 세워보고 전개한 후 Big-O로 표기해 보자
- 주의: slicing 연산은 실제 리스트의 일 부분을 복사하는 것이므로 수행시간이 늘어나는 부작용이 있다

$$\begin{aligned}
 T(n) &= \text{한 번의 비교 시간} + \text{slicing 복사시간} + \text{줄어든 범위에서 재귀 탐색} \\
 &= 1 + n + T(n/2) \\
 &= T(n/2) + cn \\
 &= \mathbf{O(n)} \quad (\leftarrow \text{앞에서부터 차례로 찾는 선형탐색 시간과 동일})
 \end{aligned}$$

- g. Slicing을 이용하지 않고 재귀적으로 이진 탐색하기

```

def binary_search(A, l, r, K): # A[l], ..., A[r]에서 K 탐색
    if l > r:
        return -1
    m = (l+r)//2
    if A[m] > K:
        return binary_search(A, l, m-1, K)
    elif A[m] < K:
        return binary_search(A, m+1, r, K)
    else:
        return m

```

- h. Slicing을 이용하지 않고 비재귀적으로 이진 탐색하기

- i. Slicing을 이용하지 않아 메모리 낭비가 적고, 비재귀적이라 재귀호출의 부담이 없는 가장 바람직한 이진 탐색 코드

```

def binary_search(A, K):
    l, r = 0, len(A)
    while l - r >= 0:
        m = (l+r)//2
        if A[m] > K: # K is in A[1] ... A[m-1]
            r = m - 1
        elif A[m] < K: # K is in A[m+1] ... A[r]
            l = m + 1
        else:
            return m

```

- i. [マイク インタビュー問題 1] 입력 값 n의 제곱근(square root)의 근사 값을 이진탐색과 유사한 방식으로 어떻게 계산할 수 있을까?

- i. n의 제곱근을 정확히 구하기는 매우 어렵다. (무한소수이라 사실상 불가능하다)
그래서 어떤 근사 오차 범위를 벗어나지 않는 제곱근을 구하면 출력하는 것으로 한다
- ii. eps를 허용 가능한 오차(absolute error)라고 하면 현재 계산된 제곱근 x의 값이 $n - \text{eps} \leq x^2 \leq n + \text{eps}$ 를 만족할 때까지 반복한다
- iii. x가 존재하는 범위가 처음에 $[0, n]$ 이므로, left = 0, right = n으로 지정한 후에, 이진 탐색 방법으로 이 범위를 $\frac{1}{2}$ 쪽 줄여 나간다

```

x = (left+right)//2
if abs(n - x**2) <= eps: # 오차범위에 들어왔으므로 리턴
    return x
elif n - eps > x**2:      # 제곱근이 너무 작음: [x, right]으로 조정
    left = x
else:                      # 제곱근이 너무 큼: [left, x]로 범위 조절
    right = x

```

- iv. 새로운 방법: Babylonian method

- 수열 x_k 를 다음과 같이 정의해보자

$$x_{k+1} = (x_k + n/x_k)/2$$

- 여기서 x_0 는 초기 값으로 우리가 정해야 하고, x_k 은 k 값이 클수록 n의 제곱근에 가까운 값을 갖게 된다. 왜 정확한 제곱근에 가까워 질까?
- $x_k < \sqrt{n} < n/x_k$ 또는 $n/x_k < \sqrt{n} < x_k$ 가 성립함을 보일 수 있다.
그러면 x_{k+1} 이 \sqrt{n} 이 존재하는 범위의 중간 값으로 정의되어 계속 제곱근에 가깝게 된다
 - a. $x_k < \sqrt{n}$ 이라면 n/x_k 는 당연히 \sqrt{n} 보다 크게 되고,
 - b. $x_k > \sqrt{n}$ 이라면 n/x_k 는 당연히 \sqrt{n} 보다 작게 되므로 성립한다
- 마지막으로 결정할 것은 x_0 를 어떻게 정하는 게 좋을까 하는 것이다.
 \sqrt{n} 은 당연히 $0 < \sqrt{n} < n$ 이므로 0과 n 사이의 임의의 값으로 정해도 상관없다. 예를 들어, $n/2$ 으로 정해도 된다
 - a. 좀 더 현명한 방법은 {2, 7, 20, 70, 200, 700, ...} 중에서 선택하는 것이다. 이상해 보이지만, 꽤 일리가 있다 (Rule of Twos and Sevens이라고 불림)
 - b. 만약 n이 d자리 수라고 하면 \sqrt{n} 은 대략 $d/2$ 자리 수가 된다. n = 250이라고 하면 \sqrt{n} 은 2 자리 수를 넘지 않는다. 그러면 위의 후보들 중에서 2자리 수 20과 70 중에서 20의 제곱이 250에 더

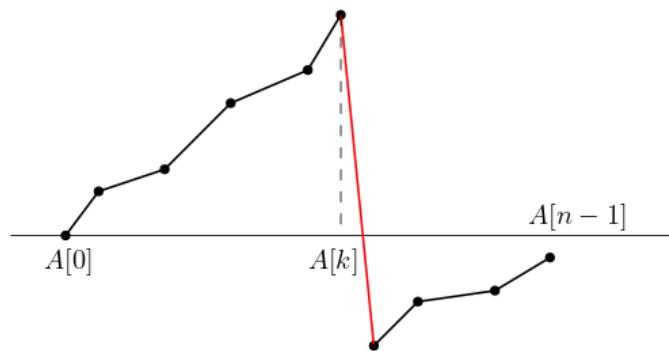
가깝기에 20을 x_0 로 결정한다. 만약, $n = 800$ 이라면 70이 더 가까운 2자리 수이므로 70으로 결정하면 된다

c. 이 방법이 정답이 아니고 제안된 방법 중 하나일 뿐임에 유의하자

j. [マイク インタビュー 문제 2] 리스트 A의 값을 오름차순으로 정렬한 후, 실수로 왼쪽 방향으로 몇 번 rotation 이동을 했다. 예를 들어, $A = [5, 8, 9, 15, 18, 20, 31]$ 이었는데, 왼쪽으로 3번의 rotation을 한 후에 $A = [15, 18, 20, 31, 5, 8, 9]$ 처럼 된 경우이다. 입력으로 주어지는 리스트는 몇 번의 rotation을 한 것인지 알 수 없다. 이 경우에 특정 값 K를 탐색하는 방법은 무엇일까? 예를 들어, 위의 예에서 K = 8을 찾고 싶다면, 어떤 순서로 비교를 하는 것이 비교 회수를 줄일 수 있는 것일까?

i. 모든 값이 다른 경우부터 생각해보자

- **힌트1:** 이진탐색 적용. 하지만 리스트가 증가 부수열 두 개로 구성된다는 점이 다르다
- **힌트2:** 그림을 그려보면 이진탐색을 어떻게 적용해야 하는지 쉽게 알 수 있다



- 먼저, rotation 이동이 몇 번 이루어졌는지를 계산해보자. 이 횟수는 최대 값의 인덱스를 찾으면 된다. 위 그림처럼 $A[k]$ 가 최대 값이라면, $(n-k-1)$ 번 rotation을 했다는 의미이다
- 최대 값 역시 이진탐색으로 어렵지 않게 $O(\log n)$ 시간에 찾을 수 있다. 모든 값이 다르기 때문에, $A[0] < A[n-1]$ 이라면 rotation이 전혀 이루어지지 않은 것이므로 고민할 필요가 없다. 따라서, $A[0] > A[n-1]$ 이 경우만 생각하면 된다. $A[\text{mid}] > A[0]$ 라면, $A[\text{mid}] \sim A[n-1]$ 사이에 존재해야 하고, $A[\text{mid}] < A[0]$ 이라면, $A[\text{mid}] \sim A[n-1]$ 사이에 존재해야 한다. 이 과정을 반복하면서 범위를 반씩 줄여나가면 된다
- rotation 이동 횟수를 알게 되면, 특정 값을 탐색하는 것 역시 이진탐색으로 $O(\log n)$ 시간에 쉽게 찾을 수 있다
- 실제 코드는 직접 작성해보자

ii. 같은 값이 존재하는 경우에는?

- **힌트:** $A = [2, 2, 2, 3, 4, 2]$ 인 경우에 이진탐색을 적용해 보면 어떤 일이 발생할까? 모든 다른 값의 경우와 어떤 점이 다를까?

- k. [🎤 인터뷰 문제 3] 리스트 A의 값이 오름차순으로 정렬되어 있다. 그러나 A의 값이 몇 개인지 모른다고 가정하자. 그러면 입력으로 주어진 값 K를 되도록 적은 비교만으로 찾으려면 어떤 전략을 써야 할까?

- i. 상당히 유명한 인터뷰 문제 중 하나이다
- ii. 이 문제의 핵심은 A의 전체 크기를 알 수 없다는 것이다. 그렇기 때문에 `at(A, i)` 함수를 제공한다. `at(A, i)`는 i가 범위 안의 인덱스면 `A[i]`를 리턴하고, 범위를 벗어난 인덱스라면 `None`을 리턴한다고 가정한다
- iii. 목표는 `at` 함수를 되도록 적게 호출해야 한다. 여러분의 전략은 무엇이고, 그 전략에서 함수의 호출 횟수는 최대 몇 번인가? Big-O로 표기하면?
 - 아래 그림처럼 크기가 8인 리스트인 경우를 예로 생각해보자



- iv. 이 문제는 제한된 정보만 알고 있는 상태에서 탐색을 하는 문제 부류에 포함된다. 유사한 문제가 꽤 많다. 트리에서 LCA(Least Common Ancestor)를 찾는 알고리즘이 이 전략을 사용한다
- v. 이 주제와 관련된 유튜브 세 편의 동영상을 참조하기 바란다



9. [💪][응용 1] n개의 정수가 저장된 리스트 A이 주어지면, 연속된 정수들의 합 중 최대가 되는 구간을 찾아보자!

- a. 이 "최대 구간 합" 문제는 다양한 방식으로 해결된다
 - i. $O(n^3)$: 모든 인덱스 쌍 (i, j)에 대해, $A[i] + \dots + A[j]$ 를 $O(n)$ 시간에 계산해 최대 합이 되는 구간 선택
 - ii. $O(n^2)$: prefix 합 $P[i] (= A[0]+\dots+A[i])$ 를 $O(n)$ 시간에 계산 후, 모든 인덱스 쌍 (i, j)에 대해 $A[i]+\dots+A[j] = P[j] - P[i-1]$ 를 계산해서 그 중에서 최대 합이 되는 구간 선택
 - iii. $O(n\log n)$: 분할정복 방법으로 해결 (아래에 자세히 설명)
 - iv. $O(n)$: 동적계획법 (Dynamic Programming) 방법으로 해결
- b. 왼쪽의 $n/2$ 개의 수 L과 오른쪽 $n/2$ 개의 수 R로 나눠 생각해보자
- c. 최대 구간이 존재하는 위치에 따라 3가지 경우 뿐이다.
 - i. L에 있는 경우, R에 있는 경우, L과 R에 걸쳐 있는 경우

- ii. L 또는 R에만 있는 경우는 재귀적으로 구할 수 있다
- iii. 걸쳐 있는 경우는 어떻게 구하면 될까? (걸쳐 있다는 의미는 L의 가장 끝 수와 R의 첫 수를 포함한다는 것이고, 이 두 수를 포함하는 최대 구간을 찾아야 한다)
 - L의 가장 끝 수부터 왼쪽으로 prefix 합을 구하면서 가장 큰 구간을 찾고, R의 첫 수부터 오른쪽으로 prefix 합을 구하면서 가장 큰 구간을 찾아 두 구간을 연결하는 것이 L과 R에 걸쳐있는 합이 가장 큰 구간이 된다. 이 구간을 M이라 하자. M은 두 번의 prefix 합을 계산하는 것과 같기에 $O(n)$ 시간이면 충분하다
- iv. 정답은 세 경우 중 합이 최대인 것이 된다
- v. $T(n) = L\text{과 }R\text{에 대한 최대 구간 합 계산} + M\text{ 계산}$
 $= 2T(n/2) + cn$
 $= O(n \log n)$

10. 예6: 봉우리 찾기 (peak finding)

- a. [1차원 버전] 리스트 A에 n개의 서로 다른 양의 정수가 있다고 하자. 값 $A[i]$ 는 지점 i의 높이 (해발 고도)를 의미한다. 높이 $A[i]$ 가 양 옆의 높이 ($A[i-1]$, $A[i+1]$)보다 높다면 봉우리가 된다. $A[0]$ 는 왼쪽 위치가 없으므로 $A[1]$ 보다 높으면 봉우리가 되고, $A[n-1]$ 은 오른쪽 위치가 없으므로 $A[n-2]$ 보다 높으면 봉우리가 된다. 문제는 주어진 리스트 A에 봉우리가 있다면 봉우리 하나를 찾아 출력하는 것이다
 - i. A에 봉우리가 없을 수도 있나? [힌트: 최대값은 당연히 봉우리 아닌가?]
 - ii. 가장 간단한 방법은 $A[0]$ 부터 봉우리인지 차례대로 검사하는 것이다. $O(n)$ 시간이 필요하다
 - iii. 더 빠른 방법은 없을까? [힌트: 이진탐색이 가능할까?]
 - $A[k-1] > A[k]$ 라면, 최소한 $A[0], \dots, A[k-1]$ 중에 하나 이상의 봉우리가 존재한다. 왜?
 - $A[k-1] < A[k]$ 이면, 반대로 $A[k], \dots, A[n-1]$ 중에 봉우리가 하나 이상 있다
 - iv. 수행시간 $T(n)$ 을 점화식으로 표현하고 전개해서 Big-O로 정리해보자
 - 이진탐색이 가능하기에 $T(n) = T(n/2) + c$ 이고, $T(n) = O(n)$ 이다
- b. [2차원 버전] $n \times m$ 2차원 리스트 A에 서로 다른 양의 정수가 저장되어 있다. $A[i][j]$ 가 봉우리이기 위해선 $A[i][j]$ 의 왼쪽, 오른쪽, 위쪽, 아래쪽의 이웃 원소보다 모두 커야 한다. 물론, 1차원 버전처럼 인덱스가 0이거나 $n-1$ 인 경우엔 이웃 원소가 4개보다 작을 수 있다. 문제는 2차원 봉우리가 존재한다면 하나를 찾아 출력하는 것이다
 - i. A에는 하나 이상의 봉우리가 항상 존재하는가?
 - ii. 값을 하나 씩 살펴보는 방법은 $O(nm)$ 시간이 필요

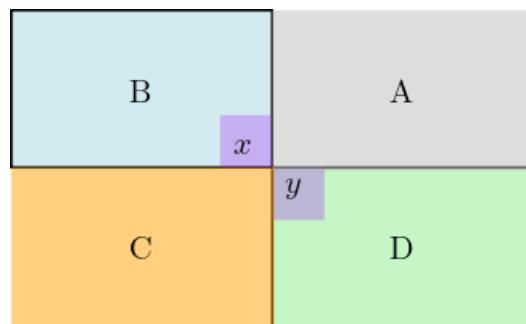
- iii. $m/2$ 번째 열 (중간 열)에 대해서 1차원 봉우리 찾기 알고리즘으로 해당 열(column)에 대한 1차원 봉우리를 찾은 다음, 1차원 봉우리가 속한 행(row)에서 다시 봉우리를 찾으면 되나?
 - 이 방법은 올바르지 않다. 반례(counter example)을 찾아보자
 - iv. $m/2$ 번째 열 (중간 열)에 대해서 최대값(maximum)을 찾아보자 (최대값은 그 열에서 1차원 봉우리임을 명심) 이 최대값이 $A[i][m/2]$ 라 하자. 그러면 $A[i][m/2-1]$ 과 $A[i][m/2+1]$ 의 값과 각각 크기를 비교해보자. 이 비교를 통해 2차원 봉우리를 찾거나 아니면 2차원 봉우리 중 하나가 반드시 $m/2$ 번째 열의 왼쪽에 있거나 오른쪽에 있음을 확신할 수 있다. 왜 그럴까?
 - 1차원 봉우리의 이진탐색이 가능했던 이유 + $A[i][m/2]$ 가 $(m/2)$ -번째 열에서 최대 값이라는 사실에 의해 확신 가능하다 (곰곰히 생각해보길)
- c. 이 알고리즘의 수행시간 $T(n, m)$ 에 대한 점화식을 세워보고 전개해보자
- i. 점화식으로는 $n \times m$ 배열에 대한 문제가 $n \times (m/2)$ 배열 문제로 작아진다. 이를 위해 $m/2$ 번째 열에 대한 최대값을 구해야 한다
 - ii. 열의 개수가 반씩 줄어 결국 한 열만 남겨지면 $n \times 1$ 배열까지 분할된다. $n \times 1$ 에서의 최대값은 $O(n)$ 시간에 찾을 수 있으므로 $T(n, 1) = cn$ 으로 정의하면 된다
 - iii. 따라서, $T(n, m) = T(n, m/2) + cn$, $T(n, 1) = cn$ 이 된다
 - iv. 풀어서 설명하면, 열에 대한 이진탐색을 하므로 $O(\log m)$ 이진탐색 단계가 필요. 매 단계마다 해당 열의 최대값을 찾아야 하므로 $O(n)$ 시간 필요. 결국 $O(n \log m)$ 시간이면 충분하다

11. [💪][**응용 2**] $n \times m$ 2차원 리스트 A 의 각 행의 값들과 각 열의 값들은 모두 오름차순으로 정렬되어 있다고 하자. 입력으로 주어진 값 K 를 빠르게 찾을 수 있는 방법을 생각해보자!

- a. **[힌트]** 행과 열이 각각 오름차순으로 정렬되어 있다는 사실을 이용해 이진 탐색에 적용해 찾을 수 있는 방법이 없을까? 즉, 이진탐색의 2차원 버전?
- b. 예: 아래와 같이 A 가 주어지고, $K = 16$ 를 찾아야 한다고 하자
 - i. $A[2][2] = 20 > K$ 이다. 그러면, 20을 기준으로 4사분면에 있는 값들은 모두 20보다 크기 때문에 그 곳에는 K 값이 있을 수 없다 → 4사분면은 탐색 범위에서 제외하면 된다!
 - ii. $A[1][1] = 8 < K$ 이다. 그러면, 8을 기준으로 2사분면에 있는 값들은 모두 8보다 작기 때문에 그 곳에는 K 값이 있을 수 없다 → 2사분면은 탐색 범위에서 제외하면 된다!
 - iii. $A[1][1] < K < A[2][2]$ 이므로 $A[1][1]$ 의 2사분면과 $A[2][2]$ 의 4사분면의 값들은 탐색 범위에서 제거하면 되고, 나머지 범위 (1사분면과 3사분면 범위)에 대해 재귀적으로 탐색을 계속하면 된다

2	5	10	19
3	8	16	19
7	20	20	32
13	25	37	44

- c. 단순히 모든 값을 차례로 비교한다면, $O(nm)$ 의 수행시간이 필요하지만, 이진탐색을 적용하면 실제 수행시간 (또는 점화식)은 어떻게 될까?



- d. 위의 그림에서 가운데 두 칸의 값 x , y 와 찾고 싶은 값 K 를 비교한다고 하자. 이 비교 결과에 따라, 다음에 탐색할 영역이 어디인지 살펴보자

- i. $K == x$ 또는 $K == y$: 빙고!
- ii. $K < x$ 경우: D에 없다고 확신! A, B, C에 존재 가능 \rightarrow A, B, C에서 재귀 탐색
- iii. $y < K$ 경우: B에 없다고 확신! A, C, D에 존재 가능 \rightarrow A, C, D에서 재귀 탐색
- iv. $x < K < y$ 경우: B와 D에 없다고 확신! A와 C에 존재 가능 \rightarrow A, C에서 재귀 탐색
- v. $n \times n$ 행렬이라고 단순하게 가정하면, $T(n) = n \times n$ 행렬에 대한 탐색 시간

$$T(n) \leq 3 T(n/2) + c$$

vi. 이 점화식을 풀면? $O(n^{\log_2 3}) = O(n^{1.584})$

- e. 더 빠른 탐색 방법이 있을까?

12. 예7: 큰 수 곱셈하기(Karatsuba algorithm)

a. 매우 큰 두 정수를 곱하고 싶다면?

- i. 일반적인 컴퓨터 언어(C/C++/Java 등)에서는 표현 가능한 정수의 범위가 정해져 있다
- ii. C 언어에서의 `int` 타입은 4byte에 정수를 저장한다
 - $-2,147,483,647 \sim 2,147,483,647$ 사이의 수만 표현 가능!



b. 학교에서 배운 곱셈 방법 (school method):

$$\begin{aligned}
 & a_{n-1} \dots a_1 a_0 \times b_{n-1} \dots b_1 b_0 \\
 = & a_{n-1} \dots a_1 a_0 \times b_0 && \leftarrow b_0 \text{와 } n \text{번 곱셈 수행} \\
 + & a_{n-1} \dots a_1 a_0 \times (b_1 \times 10^1) && \leftarrow b_1 \text{과 } n \text{번 곱셈 수행, 마지막 } 0\text{ 추가} \\
 + & \dots \\
 + & a_{n-1} \dots a_1 a_0 \times (b_{n-1} \times 10^{n-1}) && \leftarrow b_{n-1} \text{과 } n \text{번 곱셈 수행, 마지막 } 0 \text{ } n-1 \text{개 추가}
 \end{aligned}$$

- i. $T(n) = n$ 자리수의 두 수를 곱셈하는 데 필요한 기본 연산(+, \times)의 (어림잡은) 횟수
= $O(n^2)$
- ii. 이 방법은 한 자리의 두 수 곱셈을 총 n^2 번 수행하고 0을 추가하는 작업을 $1+2+\dots+n-1$ 번 수행한다 → 총 $O(n^2)$ 번의 기본 연산 수행

c. 위 방법보다 기본 연산 횟수를 줄일 수 있는 방법은 없을까?

- i. 일반적인 분할정복 알고리즘을 생각해보자! (편의상 $n = 2^k$ 라 가정)

$$\begin{aligned}
 u &= x \times 10^{n/2} + y \\
 v &= w \times 10^{n/2} + z
 \end{aligned}$$

$$u \times v = xw \times 10^n + (xz + yw) \times 10^{n/2} + yz$$

- ii. $T(n) = n$ 자리수의 두 수를 곱셈하는 데 필요한 기본 연산 (+, \times) 횟수 (점화식)

$$T(n) = \dots$$

d. Anatoly Karatsuba 의해 빠른 방법이 1962년에 제시 [[Wikipedia](#)]

- i. 위의 방법에서는 n -자리수 곱셈 문제 1개 = $(n/2)$ -자리수 곱셈 문제 4개로 분할
- ii. Karatsuba 알고리즘은 $(n/2)$ -자리수 곱셈 문제 3개로 분할!

$$\begin{aligned}
 u \times v &= xw \times 10^n + (xz + yw) \times 10^{n/2} + yz \\
 &= \boxed{xw} \times 10^n + (\boxed{(x+y)(w+z)} - \boxed{xw} - \boxed{yz}) \times 10^{n/2} + \boxed{yz}
 \end{aligned}$$

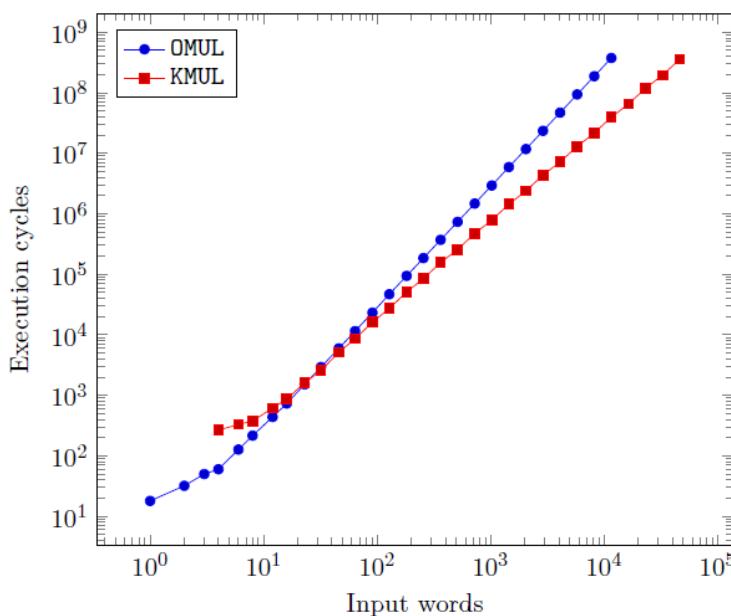
- iii. $T(n) = 2T(n/2) + T(n/2+1) + cn$ (왜 $T(n/2 + 1)$ 이 포함?)
- 간단히 $T(n) = 3T(n/2) + cn$ 으로 계산해도 Big-O 표기는 불변

- iv. $T(n) = 3T(n/2) + cn \Rightarrow$ 전개해서 Big-O로 표기하면:

$$\begin{aligned}
 T(n) &= 3T(n/2) + cn \\
 &= 3^2T(n/2^2) + cn((3/2) + 1) \\
 &= \dots \\
 &= 3^kT(n/2^k) + cn(1 + (3/2) + \dots + (3/2)^{k-1}) \\
 &= 3^k + 2c(3^k - 1) \\
 &= O(3^k) = O(3^{\log_2 n}) = O(n^{\log_2 3}) = O(n^{1.5849})
 \end{aligned}$$

e. 더 빠른 방법은 없을까? FFT (Fast Fourier Transform) 방법을 이용하면 $O(n \log n)$ 시간에 계산할 수 있다 (교재 후반부의 FFT의 다항식 곱셈 파트에 자세히 설명되어 있다.)

f. 학교곱셈 방법(OMUL) vs. Karatsuba 방법(KMUL)의 수행시간 비교



출처: Fast Multiplication of Large Integers: Implementation and Analysis of the DKSS Algorithm, April 2014, Thesis, Christoph Lüders

13. [💪][한 걸음 더] 이차원 행렬 곱셈 문제

- 위의 두 큰 수 곱셈을 1차원 행렬의 곱셈 문제라고 볼 수 있음. 이 문제를 2차원으로 확장하면 자연스럽게 2차원 행렬의 곱셈 문제가 됨
- $n \times n$ 행렬 A, B에 대해, $A \times B = C$ 를 단순한 방법 (3중 **for** 루프를 사용한) 방법으로 구현하면 두 수의 덧셈과 곱셈 연산을 $O(n^3)$ 번 해야 함
- 이 보다 빠르게 하기 위해, 위의 Karatsuba 알고리즘처럼 A와 B를 각각 4개의 $n/2 \times n/2$ 부행렬로 나눠 중복 곱셈을 최대한 피하면서 분할정복 방법으로 계산할 수 있다: 이 방법은 1969년에 Volker Strassen에 의해 제시되었다
 - https://en.wikipedia.org/wiki/Strassen_algorithm

- ii. $n \times n$ 두 행렬의 곱셈을 (8개가 아닌) 7개의 $n/2 \times n/2$ 행렬 곱셈 문제로 나누어 계산:
 $T(n) = 7T(n/2) + cn^2$

d. 현재 알려진 알고리즘들의 수행시간 변천사~ [처절함이 느껴진다]

- i. Strassen (1969): $O(n^{\log 7}) = O(n^{2.80735492})$
- ii. Coppersmith-Winograd (1990): $O(n^{2.375477})$
- iii. Andrew Stothers (2010): $O(n^{2.374})$
- iv. Virginia Vassilevska Williams (2011): $O(n^{2.3728642})$
- v. Francois Le Gall (2012): $O(n^{2.3728639})$ ← 현재 가장 빠른 곱셈 알고리즘!

14. [💪][💡] π (원주율) 최대한 정확하게 계산하기

a. C 언어와 같이 제한된 메모리에 정수 값이 저장되는 환경에서 매우 큰 수를 표현하고 싶다면? (Python에서는 큰 수에는 더 많은 byte를 자동 할당하여 표현함에 유의!)

- i. 배열(리스트)에 자리수를 나눠 저장하는 방법이 일반적이다

ii. $A = [32, 1415, 9201, 2120, 187, 6091]$

32	1415	9201	2120	0187	6091
----	------	------	------	------	------

- iii. 3,214,159,201,212,001,876,091이라는 큰 수를 $A[i]$ 에 4자리씩 나눠 저장한 예. 물론 $A[0]$ 에는 4자리보다 작은 수가 저장될 수도 있다. 물론 4자리보다 더 많은 자리수를 저장해도 된다

b. 리스트에 저장된 큰 두 수를 더하거나 빼고 싶다면?

- i. $[254, 3266, 7901, 1301] + [45, 6789, 2280, 0302] = ?$

- ii. 파이썬 리스트의 덧셈은 두 리스트를 이어 붙이는 연산이므로 실제 덧셈을 수행하는 함수를 직접 구현해야 함

c. 리스트 원소에 몇 자리를 저장할지 결정: K자리 저장한다고 가정

d. 덧셈: 리스트 A와 B에 저장된 두 큰 수를 더해 리스트 C에 저장해 리턴

```
i. def long_add(A, B):      # pseudo code
    d = len(A)
    C = [0] * d
    M = 10**K                # 리스트 한 셀에 K자리씩 저장
    carry = 0
    for i in range(d-1, -1, -1):
        C[i] = A[i] + B[i] + carry
        if C[i] >= M: # M을 넘기면 옆자리로 M을 넘김 (carry 발생)
            C[i] = C[i] - M
            carry = 1
        else: carry = 0
```

e. 뺄셈: 리스트 A의 큰 수에 리스트 B의 큰 수를 빼 리스트 C에 저장하는 연산

```
def long_sub(A, B):
    d = len(A)
    C = [0] * d
    M = 10**K           # 리스트 한 셀에 K자리씩 저장
    carry = 0
    for i in range(d-1, -1, -1):
        C[i] = A[i] - B[i] - carry
        if C[i] < 0: # 음수가 되면 옆자리에서 M을 빌려옴 (carry 발생)
            C[i] = C[i] + M
            carry = 1
        else: carry = 0
```

f. 곱셈: 리스트 A의 큰 수에 정수 b를 곱해 리스트 C에 저장하는 연산 (정수 b에 유의!)

i. 리스트 A에 다른 리스트 B를 곱하고 싶다면 Karatsuba 알고리즘을 사용하면 됨

```
def long_mult(A, b, C):
    # 가장 작은 자리 A[d-1]에 b를 곱해 M을 넘기면 그 차이만
    # 남기고 carry = 1로 지정한 후 다음 자리로 넘어가는 방식
    # 직접 해보자~
```

g. 나눗셈: 리스트 A의 큰 수를 정수 b로 나눈 결과를 리스트 C에 저장하는 연산

```
def long_div(A, b, C):
```

h. 클래스 LongNumber로 선언해보기 (연산자 overloading 이용)

```
class LongNumber:
    def __init__(self, size, digit_per_slot):
        self.L = [0]*size
        self.K = digit_per_slot
    def __getitem__(self, k):
```

```

        return self.L[k]
    def __setitem__(self, k, value):
        self.L[k] = value
    def __str__(self):
        return str(self.L)
    def __len__(self):
        return len(self.L)
    def __add__(self, B):
        C = LongNumber(len(self), self.K)
        ...
    return C

```

i. π 값을 근사적으로 계산하는 [Machin 공식](#)

- i. 위의 리스트로 매우 큰 값을 표현하는 방식은 당연히 소수점 이하 자리수가 많은 정밀한 값에도 적용됨! 따라서 π 같은 무한소수를 저장할 때에도 유용!

$$\begin{aligned}
 \arctan x &= x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \dots \\
 \pi &= 16 \arctan \frac{1}{5} - 4 \arctan \frac{1}{239} \\
 &= 16\left(\frac{1}{5} - \left(\frac{1}{5}\right)^3 \frac{1}{3} - \left(\frac{1}{5}\right)^5 \frac{1}{5} + \dots\right) - 4\left(\frac{1}{239} - \left(\frac{1}{239}\right)^3 \frac{1}{3} + \dots\right) \\
 &= \sum_{i=1}^T (-1)^{i+1} \left(\frac{16}{5^{2i-1}} - \frac{4}{239^{2i-1}} \right) \frac{1}{2i-1}
 \end{aligned}$$

$w_i \qquad v_i$

ii. T : 항의 개수. 를 수록 실제 π 값에 가까워진다

- 보통 $T = 1 + \text{precision}/1.39894$ 로 정한다. 예를 들어, 소수점이하 100자리까지 정확하게 계산하고 싶다면, $T = 1 + 100/1.39894 = 73$ 정도로 결정하면 된다
- iii. 항 w_i 은 $16/5$ 을 $1/5^5$ 로 계속 나누는 형식이라 매우 작은 수를 25로 계속 나누는 모양이고, 항 v_i 는 $4/239$ 를 239^2 로 계속 나누는 모양이다. 결국 `long_div` 함수를 호출하면 된다
- iv. $w_i - v_i$ 결과를 다시 $2n-1$ 로 나눠야 하므로 다시 `long_div` 함수 호출
- v. 이 결과를 i 값의 훌수면 더하고, 짝수면 빼게 되므로 `long_add`, `long_sub`를 번갈아가며 호출하면 된다
- vi. 결국, π 값을 저장할 리스트 P , w_i , v_i 를 저장할 리스트 W , V 는 기본적으로 정의되어야 한다

5. Sorting (정렬)

1. 리스트에 저장된 값들을 크기 순서에 따라 재배열하는 알고리즘의 고전 문제
 - a. 되도록 **비교 횟수**와 **자리 바꿈 횟수를 최소로 하는 것이** 목표
 - b. 매우 기본적이고 중요한 알고리즘 문제 중 하나



2. Python에서는 정렬 함수를 이미 제공하고 있다
 - a. Python에서는 Tim sort 알고리즘을 사용한다. 이 장의 후반부에 Tim 정렬 알고리즘에 대한 설명이 나오니 참고 바람

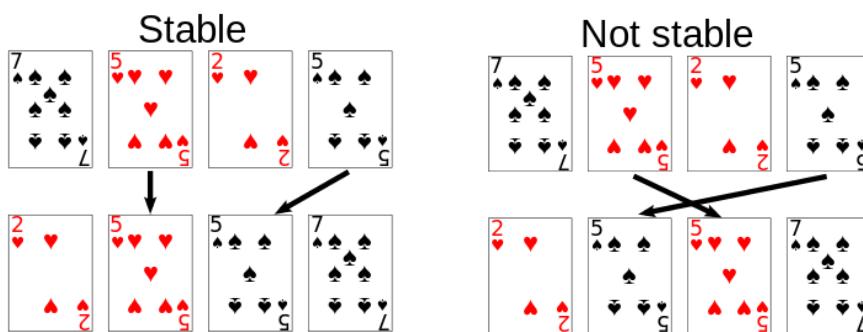
```
a = [3, 1, 2, -5]
a.sort()
print(a)                      # a = [-5, 1, 2, 3] 오름차순으로 정렬됨
a.sort(reverse = True)
print(a)                      # a = [3, 2, 1, -5] 내림차순으로 정렬됨
b = sorted(a)                 # a의 값의 순서는 변하지 않고, 정렬되어 리스트 b로 복사
```

3. 단순 분류:

- a. 기본 정렬 알고리즘: 느리지만 단순한 정렬 알고리즘
 - i. insertion, selection, bubble 정렬 알고리즘 등
- b. 빠른 정렬 알고리즘: 빠르지만 복잡할 수도 있는 정렬 알고리즘
 - i. quick, heap, merge 정렬 알고리즘 등
- c. 특별한 상황에 맞는 정렬 알고리즘
 - i. count, radix, bucket 정렬 알고리즘 등

4. [중요] 정렬 알고리즘의 성질 2가지

- a. **stable** vs. **unstable**
 - i. 크기가 같은 숫자인 경우, 원래 입력 순서를 정렬 후에도 유지하는 알고리즘을 stable하다고 함
 - ii. 이미지 출처: https://en.wikipedia.org/wiki/Sorting_algorithm



- b. **in-place** vs **not-in-place**
 - i. 상수 개의 추가 변수(메모리)만 사용한 정렬 알고리즘을 in-place하다고 함

5. 기본정렬 알고리즘:

- a. 느리지만 간단한 정렬 알고리즘: insertion, selection, bubble
- b. 세 알고리즘은 모두 $(n-1)$ 번의 round를 거치며 매 round마다 하나의 값을 정렬하는 방식이다. $(n-1)$ 번의 라운드 후에는 나머지 하나의 수는 자동적으로 정렬된다
 - i. insertion sort: 원쪽부터 점진적으로 정렬하는 방식 (오른쪽부터 정렬할 수도)
 - ii. selection sort: 가장 큰 값부터 차례대로 찾는 방식 (작은 수부터 찾을 수도 있음)
 - iii. bubble sort: 가장 큰 값부터 차례대로 찾는 방식 (작은 수부터 찾을 수도 있음)

c. Insertion sort 알고리즘

```
def insertion_sort(A, n): # A[0] ~ A[n-1]까지 insertion sort
    for i in range(1, n): # round i에 i번째 작은 수를 찾음
        j = i-1
        while j >= 0 and A[j] > A[j+1]:
            A[j], A[j+1] = A[j+1], A[j]
            j = j - 1
```

d. Selection sort 알고리즘

```
def selection_sort(A, n): # A[0] ~ A[n-1]까지 selection sort
    for i in range(n-1, 0, -1): # 큰 수를 차례대로 찾음
        m = get_max_index(A, i) # A[0]~A[i] 중 최대값 인덱스 구함
        A[i], A[m] = A[m], A[i] # A[m]이 최대값이므로 A[i]에 배치
        # 자리바꿈(swap)이 발생
```

e. Bubble sort 알고리즘

```
def bubble_sort(A, n): # A[0] ~ A[n-1]까지 bubble sort
    for i in range(n):
        # 큰 수를 차례대로 찾음
        for j in range(1, n):
            if A[j-1] > A[j]: # A[n-1]~A[i] 인접한 수 비교 후 swap
                A[j-1], A[j] = A[j], A[j-1]
```

f. 아래 리스트의 값을 위의 세 가지 알고리즘으로 정렬해보자. 어떻게 값들이 재배치되는지 round 별로 알아보자

A = [12, 4, 9, 10, 21, 3, 8, 0, 7, 9, 6]

- Insertion
- Selection
- Bubble

g. [한 발 더] 이진탐색을 이용한 insertion 정렬 알고리즘

- i. insertion 정렬에서 $A[i]$ 를 왼쪽의 정렬된 부분 ($A[0], \dots, A[i-1]$)에서 자신의 위치를 찾아 "insert"하는 과정을 이진탐색을 이용하면 어떨까? 즉, $A[j] \leq A[i] < A[j+1]$ 이라면, $A[j]$ 를 이진탐색으로 찾을 수 있다. (왜?) 다음엔 $A[j+1]$ 부터 $A[i-1]$ 까지의 값을 오른쪽으로 한 칸씩 이동하고 $A[i]$ 를 $A[j+1]$ 로 이동한다
 - 이 방법의 총 **비교 횟수**를 Big-O로 표기하면?
 - $A[i]$ 를 insert하기 위한 이진탐색에서의 비교회수는 $O(\log i)$ 이므로 모든 i 에 대해 더하면 $O(n\log n)$ 이 된다
 - 이 방법의 총 **이동 횟수**를 Big-O로 표기하면?
 - $A[i]$ 가 삽입될 위치를 찾으면 그 위치 오른쪽에 있는 값들은 모두 한 칸씩 오른쪽으로 이동해야 하므로, 삽입될 위치가 가장 왼쪽이라면 매번 $n-1$ 개의 값이 이동해야 하므로 $O(n^2)$ 번의 이동이 필요할 수도 있다
 - 이 방법의 (최악의 경우의) 수행시간을 Big-O로 표기하면?
 - 비교횟수 $O(n\log n) +$ 이동횟수 $O(n^2) = O(n^2)$
- ii. 이진탐색을 사용하지 않는 insertion 정렬과 비교횟수와 이동횟수를 비교해보자

h. 수행시간 (최악의 경우) - stable - in-place 여부를 따져보자!

- i. insertion:
- ii. selection:
- iii. bubble:

6. [매우 중요!] Quick sort 알고리즘

- a. 실제로 구현해서 실행해 본 여러 결과에 따르면 가장 빠른 정렬 알고리즘 중 하나임
- b. 전형적인 divide-and-conquer 알고리즘의 예 중 하나
- c. Quick select 알고리즘과 매우 유사. 정렬 알고리즘이므로, 피봇(pivot)보다 작은 값들을 리스트의 왼쪽으로, 큰 값들을 리스트의 오른쪽으로 재배열 한 후, 양 쪽을 재귀적으로 다시 정렬하는 식으로 진행됨
- d. **Pseudo code:** quick_sort(A, first, last) # A[first] ~ A[last]를 퀵 정렬
 - i. `first >= last`인 경우는 정렬할 값이 1개 이하이므로 그냥 리턴
 - ii. `first < last`인 경우는 quick selection과 유사하게 pivot을 정해 나눔
 - `pivot = A[first]` (pivot을 가장 왼쪽의 값으로 정함. 사실 어떻게 정해도 최악의 경우의 수행시간에는 영향이 없음)
 - A의 값들을 pivot보다 작은 값은 왼쪽에 큰 값들은 오른쪽에 오도록 재배치
 - pivot보다 작은 값들에 대해 quickSort 재귀호출 + 큰 값들에 대해 quickSort 재귀호출
- e. 구현 방법 1: 리스트 A에서 pivot을 기준으로 나누는 첫 번째 방법 (*in-place/unstable*)

```
def quick_sort(A, first, last):
    if first >= last: return
    left, right = first+1, last
    pivot = A[first]      # 첫 번째 값을 pivot으로 정함
    while left <= right: # pivot보다 작은 값은 왼쪽에, 큰 값은 오른쪽으로 재배치
        | while left <= last and A[left] < pivot:
        |     left += 1
        | while right > first and A[right] > pivot:
        |     right -= 1
        | if left <= right: # swap A[left] and A[right]
        |     A[left], A[right] = A[right], A[left]
        |     left += 1
        |_
    right -= 1
```

A[first], A[right] = A[right], A[first] # pivot을 제 위치로 보냄

```
quick_sort(A, first, right-1)      # pivot보다 작은 값을 재귀 정렬
quick_sort(A, right+1, last)       # pivot보다 큰 값을 재귀 정렬
```



f. 구현방법 2: 리스트 A에서 pivot을 기준으로 나누는 두 번째 방법 (*in-place/unstable*)

```
def quick_sort(A, first, last):
    if first >= last: return

    # new simple partition
    smaller, equal, larger = first, first, last+1
    pivot = A[first]

    # < pivot: A[first:smaller]에 배치되도록
    # == pivot: A[smaller:equal]에 배치되도록
    # unclassified: A[equal:larger]에 배치되도록
    # > pivot: A[larger:last+1]에 배치되도록

    while equal < larger:
        # A[equal] is the value we are now looking at
        if A[equal] < pivot:
            A[smaller], A[equal] = A[equal], A[smaller]
            smaller, equal = smaller+1, equal+1
        elif A[equal] == pivot:
            equal += 1
        else: # A[equal] > pivot
            larger -= 1
            A[equal], A[larger] = A[larger], A[equal]

    quick_sort(A, first, right-1)
    quick_sort(A, right+1, last)
```

g. 구현 방법 3: 새로운 리스트를 추가로 사용하여 나누는 방법 (*not-in-place/stable*)

```
def quick_sort(A): # first, last가 필요없음
    if len(A) <= 1: return A
    pivot = A[0]
    S, M, L = [], [], []
    for x in A:
        if x < pivot: S.append(x)
        elif x > pivot: L.append(x)
        else: M.append(x)
    return quick_sort(S) + M + quick_sort(L)
```

h. A = [4, 2, 5, 8, 6, 2, 3, 7, 10]인 경우, 위의 세 가지 구현방법에 따라 시뮬레이션 해보자

- i. 위 세 가지 퀵 정렬 코드가 각각 stable한지, in-place한지 따져보자
- j. **수행시간:** 전적으로 pivot보다 작은 값들의 수와 큰 값들의 수, 즉 어떤 비율로 분할되는 가에 따라 결정된다! → quick_select 알고리즘의 수행시간과 분석과 유사
 - i. 가장 좋은 시나리오: 대략 절반씩, $n/2$ 개 - $n/2$ 개로 계속 분할되는 경우
 - $T(n) = \underline{\hspace{10em}}$
 - ii. 가장 나쁜 시나리오: 0개 - $(n-1)$ 개 (또는 $(n-1)-0$)으로 계속 분할되는 경우
 - $T(n) = \underline{\hspace{10em}}$
- k. [고급 - skip 가능] 확률적으로 평균 수행 시간(average running time)으로 계산해보기

- i. 매우 전형적인 증명 방법이므로 최대한 이해해보길 바란다~
- ii. 수행시간은 두 수의 비교 횟수로 정의한다
- iii. 1부터 n 까지의 랜덤 순열 (random permutation)이 A 에 저장되어 있다고 가정 → 모든 수가 pivot으로 선택될 확률은 동일함을 의미
- iv. X_{ij} = 퀵 정렬 동안에 A 의 두 수 i 와 j 가 비교되면 1, 비교되지 않으면 0으로 정의되는 랜덤 변수 (random variable)
- v. 총 비교 횟수 X 는 당연히 모든 $i < j$ 에 대해 X_{ij} 를 더한 값으로 정의된다
- vi. 계산하고 싶은 값은 X 에 대한 기대값 $E[X]$ 이다

$$E[X] = E[\sum_{i=1}^n \sum_{i < j} X_{ij}] = \sum_{i=1}^n \sum_{i < j} E[X_{ij}] = \sum_{i=1}^n \sum_{j > i} \text{Prob}(X_{ij} = 1)$$

- vii. $\text{Prob}(X_{ij} = 1)$ 은 두 수 i 와 j 가 퀵 정렬 과정에서 비교될 확률이다. 이 확률을 구하는 게 핵심이다
- viii. 퀵 정렬이 진행되면서 두 그룹으로 나뉘어져 재귀적으로 호출된다. i 와 j 가 비교되었다면 현재 상태에서 같은 그룹에 포함되어야 하고, 그 중 하나는 반드시 pivot으로 선택되어야 한다
- ix. 사실 1: $[i, i+1, \dots, j-1, j]$ 에 속하는 값들은 모두 같은 그룹에 속해야 한다
 - 만약에 $[i+1, \dots, j-1]$ 의 어떤 수가 다른 그룹에 속했다면 그 수가 pivot으로 선택되었다는 의미이다. 그러면 i 와 j 가 다른 그룹으로 나눠지게 된다

x. i와 j가 속한 그룹에는 i와 j를 포함해 최소 $(j-i+1)$ 개의 값들이 있음을 사실 1을 통해 알 수 있다

xi. 사실 2: $\text{Prob}(X_{ij} = 1) \leq 2/(j-i+1)$

- i와 j가 중에서 하나는 pivot으로 선택되어야 하는데, 해당 그룹의 모든 값이 pivot으로 선택될 확률이 같기에 특정 값이 pivot이 될 확률은 1/그룹의 크기이 된다. 그런데 i와 j가 둘 중 하나만 pivot이 되면 되기에 2/그룹의 크기이 된다. 그룹에는 최소 $(j-i+1)$ 개의 값이 포함되므로 이 확률은 최대 $2/(j-i+1)$ 이 된다

$$\sum_{i=1}^n \sum_{j=i+1}^n \frac{2}{j-i+1} \leq 2 \sum_{i=1}^n \sum_{k=1}^n \frac{1}{k} \leq \sum_{i=1}^n H_n = 2cn \log_2 n = O(n \log n)$$

- $H_n = 1 + 1/2 + 1/3 + \dots + 1/n = c \log n$

- 하모닉 넘버라고 부른다. 함수 $1/x$ 를 1부터 n까지 정적분 해보면 쉽게 증명할 수 있다

xii. 결론: 쿼 정렬의 평균적인 수행시간은 **O(n log n)**이다

I. [고급] 점화식으로 평균 수행시간을 계산해 보기

i. 멱급수 전개 방식을 사용한다

$$\begin{aligned}
 T(n) &= cn + \frac{1}{n} \sum_{k=1}^n T(k-1) + T(n-k) \\
 &= cn + \frac{2}{n} T(0) + \frac{2}{n} \sum_{k=1}^{n-1} T(k) \\
 nT(n) &= cn^2 + 2T(0) + 2 \sum_{k=1}^{n-1} T(k) \\
 (n-1)T(n-1) &= c(n-1)^2 + 2T(0) + 2 \sum_{k=1}^{n-2} T(k) \\
 \hline
 nT(n) - (n-1)T(n-1) &= c(2n-1) + 2T(n-1) \\
 nT(n) - (n+1)T(n-1) &= c(2n-1) \text{ divided by } n(n+1) \\
 \frac{1}{n+1} T(n) - \frac{1}{n} T(n-1) &= c \frac{2n-1}{n(n+1)}
 \end{aligned}$$

Let $S(n) = T(n)/(n+1)$.

$$\begin{aligned}
 S(n) - S(n-1) &\leq 2c/(n+1) \\
 S(n) - S(1) &\leq 2c \left(\frac{1}{n+1} + \dots + \frac{1}{3} \right) \\
 S(n) &\leq 2c \left(\frac{1}{n+1} + \dots + \frac{1}{2} + 1 \right) \\
 &= 2cH_{n+1} \\
 T(n) &= 2cH_{n+1}(n+1) = O(n \log n)
 \end{aligned}$$

7. Merge sort 알고리즘

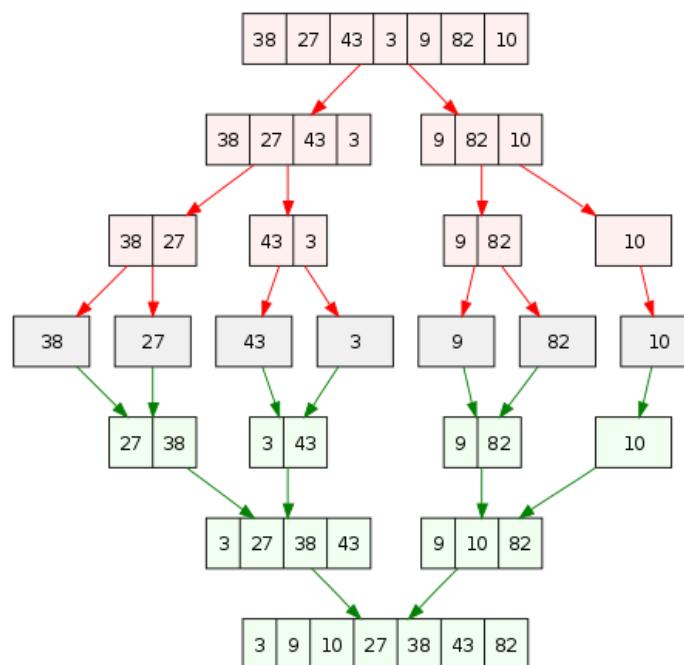
- a. 전형적인 divide-and-conquer 알고리즘 중 하나
- b. 최악의 경우에도 $O(n \log n)$ 에 동작하는 **최적** 정렬 알고리즘 중 하나
 - i. quick_sort의 가장 큰 문제는 분할된 크기가 비슷하지 않아 재귀를 많이 하게 되어 수행시간이 커진다는 것!
 - ii. 애초에 강제로 균등하게 분할하면 되지 않을까 하는 아이디어에서 착안



- c. Pseudo code:

```
merge_sort(A, first, last): # A[first] ... A[last] merge sort 하기
    i. if first >= last: return # 정렬 불필요
    ii. merge_sort(A, first, (first+last)//2) # 앞 부분 반 재귀 정렬
    iii. merge_sort(A, (first+last)//2+1, last) # 뒷 부분 반 재귀정렬
    iv. merge_two_sorted_lists(A, first, last) # 정렬된 두 부분을 합병!

        # 현재 A는 앞 쪽 반과 뒷 쪽 빈이 정렬된 상태
        # 두 정렬된 부분을 합병하는 함수
```



- d. 예: 위의 그림
 - i. **분할**: (빨간색 부분) 숫자 하나씩 분할될 때까지 반씩 분할해 간다
 - ii. **정복 (또는 합병)**: (초록색 부분) 다시 분할 순서의 반대로 정렬된 두 리스트를 합병해 올라간다

e. Pseudo code [merge_two_sorted_lists]

```

def merge_two_sorted_lists(A, first, last):
    m = (first + last)//2
    i, j = first, m+1
    B = []
    while i <= m and j <= last:
        if A[i] <= A[j]:
            B.append(A[i])
            i += 1
        else:
            B.append(A[j])
            j += 1
    for k in range(i, m+1):           # why?
        B.append(A[k])
    for k in range(j, last+1):       # why?
        B.append(A[k])
    for i in range(first, last+1):   # A ← B
        A[i] = B[i - first]

```

f. stable?

g. in-place?

h. 수행시간 $T(n)$ 을 위한 점화식을 세워보고 전개한 후 Big-O로 표기해보자

i. $T(n) = \underline{\hspace{10em}}$, $T(1) = c$

- i. [한 발 더] 합병 정렬에서는 $n/2$ 개의 정렬된 값과 $n/2$ 개의 정렬된 값을 n 개의 값을 저장하는 임시 리스트를 이용해 합병한다. 즉, $n/2$ 개, $n/2$ 개 합병에 필요한 추가 메모리는 n 이다.

- i. 추가 메모리를 $n/2$ 만 사용하고도 가능하다. 어떻게 하면 될까? ^^

예: $A = [1, 3, 5, 7, 9, 2, 4, 6, 8, 10]$ 를 통합하기 위해 크기가 5인 임시 리스트 B 를 사용해 합병해보자

- $B \leftarrow A[0:5] \ # B = [1, 3, 5, 7, 9]$
- B 와 $A[5:10]$ 을 합병해 A 에 저장하는 과정으로 처리하면 된다 (왜?)

- ii. 리스트에 a개의 정렬된 값과 b개의 정렬된 값이 차례대로 있을 때, $\min(a, b)$ 정도의 추가 메모리를 사용해서 합병해보자. 어떻게 하면 될까?

1. $a < b$ 인 경우와 $a > b$ 인 경우를 다르게 다루어야 한다!

예1: $A = [1, 3, 5, 2, 4, 6, 8, 10]$ # $a = 3 < b = 5$

예2: $A = [1, 3, 5, 7, 9, 2, 4, 6]$ # $a = 5 < b = 3$

j. [merge 정렬 방식 응용 1] 입력 리스트의 숫자들에 대한 inversion 개수 세기 문제

- i. 예: $A = [2, 4, 1, 3, 5]$ 라면 (2, 1), (4, 1), (4, 3) 세 쌍의 숫자가 서로 역전되어 있으므로 inversion의 개수는 3개이다. 즉, $\text{inversion}(A) = 3$ 이 된다

- ii. 방법 1: 가장 단순한 알고리즘은?

1. 모든 쌍 (a, b)를 보면서 $a > b$ 인지 검사해본다
2. 이 방법의 수행시간은 $O(n^2)$ - 너무 느려~

- iii. 방법 2: 보다 빠른 알고리즘은?

1. [힌트] merge 정렬을 하면서 동시에 inversion 개수도 추가로 셀 수 있다!

2. merge_sort처럼 반으로 나눠 분할한다

3. 왼쪽 반의 inversion 개수 L 을 재귀적으로 계산해 알고 있고, 오른쪽 반의 inversion 개수 R 을 알고 있다고 하자

4. $L+R$ 이 전체 inversion의 개수인가? 아니면 어떤 inversion이 빠졌나?

5. 결국, 왼쪽 반에 있는 수와 오른쪽 반에 있는 수의 쌍이 inversion 쌍인지 고려하지 않았다. 이 쌍의 개수를 M 이라 하면, 전체 inversion 갯수는 $L+M+R$ 이 된다

6. 그러면, M 을 어떻게 계산할까? 바로 `merge_two_sorted_lists`와 유사한 방식으로 합병을 하면서, 동시에 inversion 개수도 세면 된다. 어떻게?

- a. 현재 왼쪽 그룹의 값 a 와 오른쪽 그룹의 값 b 가 비교된다고 하자. 만약, $a > b$ 라면, (a, b)는 inversion이다. 동시에 b 의 오른쪽에 있는 값들도 역시 a 와 inversion이 된다. 따라서, b 와 b 의 오른쪽에 있는 값의 개수만큼 inversion 개수를 더하면 된다. $a < b$ 이면, inversion이 발생하지 않으므로 더할 필요 없다

- b. 이 과정은 합병하는 시간에 비례하므로 $O(n)$ 이면 충분하다

7. 따라서 전체 알고리즘의 수행시간은 합병 정렬의 시간과 같은 $O(n \log n)$ 이다

k. [merge 정렬 방식 응용 2] 두 리스트 A와 B에 저장된 값이 각각 오름차순으로 정렬되어 있을 때, A와 B를 오름차순으로 합병(merge)하여 A에 저장하고 싶다. [주의: 합병한 값들이 A에 저장되어야 함을 유의하자] 가능하면 최소의 이동(move) 횟수가 되도록 합병하려면 어떻게 해야 할까?

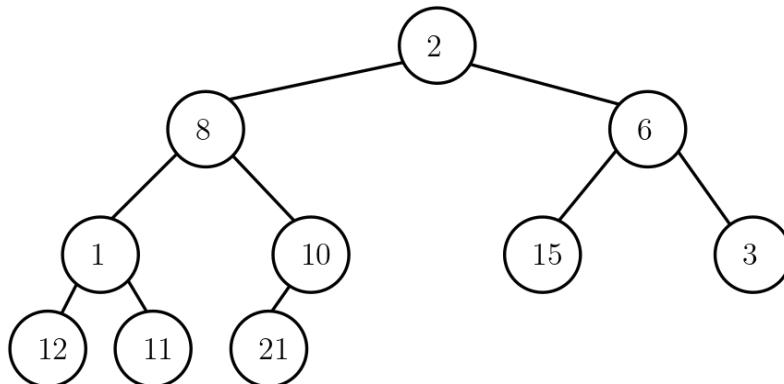
- i. A의 크기는 A와 B의 모든 값을 저장할 수 있을 정도로 크다고 가정하자
- ii. [힌트] 합병 정렬의 merge 단계처럼 A와 B의 왼쪽 끝부터 값을 차례대로 비교하면서 합병해도 된다. 그런데 이 방법은 오른쪽 끝의 값부터 비교하는 방법보다는 더 많은 이동이 필요하지 않을까?

8. Heap sort 알고리즘: 자료구조 수업에서 이미 배운 정렬 알고리즘

[1/2]



[2/2]



2	8	6	1	10	15	3	12	11	21
---	---	---	---	----	----	---	----	----	----

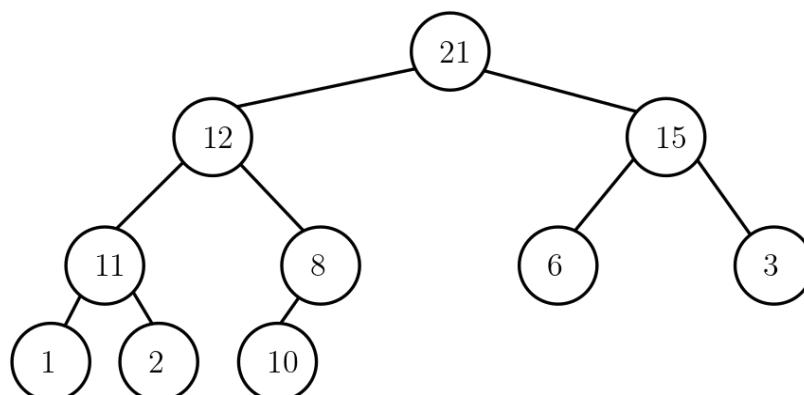
a. 힙(heap): 다음의 모양과 힙 성질을 만족하는 리스트에 저장된 값의 시퀀스

i. (모양 성질) 다음과 같은 이진트리여야 한다:

1. 마지막 레벨을 제외한 각 레벨의 노드는 모두 채워져 있어야 한다
2. 마지막 레벨에선 노드들이 왼쪽부터 채워져야 한다

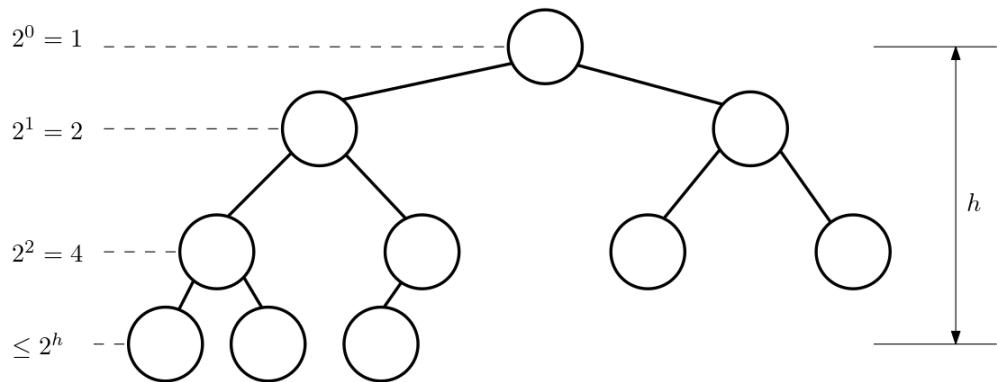
ii. (힙 성질) 루트 노드를 제외한 모든 노드에 저장된 값(key)은 자신의 부모 노드의 값보다 크면 안된다

1. 위의 그림의 이진트리는 모양성질은 만족하지만 힙 성질은 만족하지 않는다.
(8의 부모가 2가 되어 성질 위배)
2. 반면에 아래 그림은 모양과 힙 성질 모두 만족한다



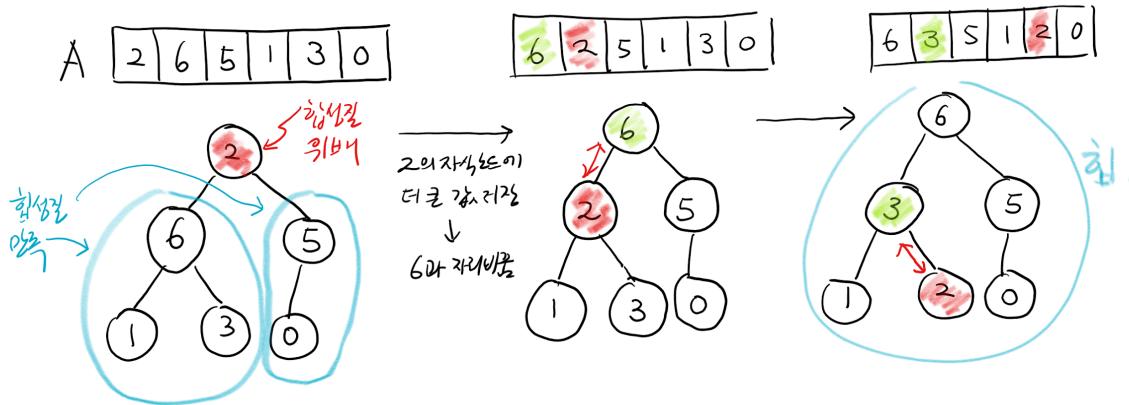
21	12	15	11	8	6	3	1	2	10
----	----	----	----	---	---	---	---	---	----

- iii. [매우 중요] 힙 성질에 따라 루트 노드에는 가장 큰 값이 저장되게 된다
- iv. 여기서 주의할 건, 실제 데이터 값은 리스트에 저장되어 있고 리스트의 값이 표현하는 (가상의) 이진트리가 모양 성질을 만족한다는 의미이다
- v. 힙의 높이 h :
 - 1. n 개의 값으로 구성된 힙의 높이 h 는 최대 어느 정도일까?
 - 2. 모양 성질에 따르면, 레벨 0부터 $h-1$ 에 있는 노드는 모두 채워져 있어야 하므로, 노드 개수가 총 $1 + 2 + 2^2 + \dots + 2^{h-1} = 2^h - 1$ 이다
 - 3. 레벨 h 에는 하나 이상의 노드가 존재하므로 전체 노드 수 $n \geq 2^h$ 이 성립한다
 - 4. 양변에 \log_2 를 취하면 $h \leq \log n$ 이 성립한다. $h = O(\log n)$ 이다



- vi. Heap 클래스:


```
class Heap:
    def __init__(self, L=[]): # default: 빈 리스트
        self.A = L
        self.make_heap() # A의 값을 힙성질이 만족되도록
    def __str__(self):
        return str(self.A)
```
- vii. 리스트에 저장된 값을 이진트리로 해석하면 자동으로 모양 성질을 만족한다. 그러나 힙 성질을 만족하지 않을 수도 있다. 위의 두 번째 경우가 그런 예이다
- viii. 힙 성질을 만족하지 않으면, 값을 재배열해서 힙을 만들 수 있다 → make_heap 함수 → 이를 위해 heapify_down 함수가 필요
- ix. heapify_down(k) 함수:
 1. $A[k]$ 의 자손 노드들은 모두 힙 성질을 만족한다고 할 때, $A[k]$ 값을 (필요하다면) 아래로 내려가면서 힙 성질을 만족하는 위치로 이동시키는 함수.



```

def heapify_down(self, k, n):
    # n = 힙의 전체 노드수 [heap_sort를 위해 필요함]
    # A[k]가 힙 성질을 위배한다면, 밑으로
    # 내려가면서 힙 성질을 만족하는 위치를 찾는다
    while 2*k+1 < n: # [?] 조건문이 어떤 뜻인가?
        L, R = 2*k + 1, 2*k + 2 # [?] L, R은 어떤 값?
        if L < n and self.A[L] > self.A[k]:
            m = L
        else:
            m = k
        if R < n and self.A[R] > self.A[m]:
            m = R
        # m = A[k], A[L], A[R] 중 최대값의 인덱스

        if m != k: # A[k]가 최대값이 아니라면 힙 성질 위배
            self.A[k], self.A[m] = self.A[m], self.A[k]
            k = m
        else: break # [?] 왜 break 할까?
    
```

2. **heapify_down의 수행시간**을 알아보자

- 수행시간은 $A[k]$ 가 내려온 레벨 수에 비례한다
- 가장 많이 내려오려면 루트인 $A[0]$ 가 힙의 높이(가장 깊은 곳의 리프)까지 내려오는 것이다.
- 한 레벨 내려올 때의 연산은 상수번의 비교를 하고 1번의 자리바꿈을 하는 것이므로 $O(1)$ 시간이면 충분하다.
- 따라서, 최악의 경우에 힙의 높이에 비례하는 시간이 필요하다. 즉, **$O(\log n)$ 시간**이 필요하다

x. **make_heap:** 현재 리스트의 값들을 힙 성질을 만족하도록 재배열하는 함수

1. 리스트 A의 각 값에 대해 heapify_down을 호출해 재배치한다
2. 여기서 어떤 값부터 차례로 heapify_down을 호출해야 할까?

```

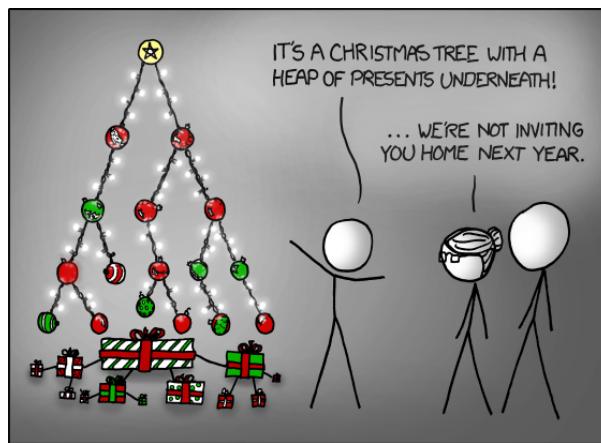
def make_heap(self):
    n = len(self.A)
    for k in range(n-1, -1, -1): # A[n-1]→...→A[0]
        self.heapify_down(k, n)
    
```

3. `range(n-1, -1, -1) → range(n//2-1, -1, -1)`로 변경해도 문제 없음 (왜 그럴까? [?])
 4. `make_heap`의 수행시간은?
 5. **분석 1:** for 반복문은 n번 반복되고, 반복할 때마다 `heapify_down`이 한번씩 호출된다. `heapify_down`의 수행시간이 $O(\log n)$ 이므로 $O(n \log n)$ 시간이면 충분하다
 6. **분석 2:** 엄밀하게 분석하면 $O(n)$ 시간이면 충분하다 (아래 분석 참조. 건너 뛰어도 됨)
 - level i에 있는 노드 수는 2^i 개 또는 그 이하이다
 - level i에 있는 노드가 `heapify_down`을 통해서 움직이는 레벨의 수는 $h-i$ 임. 즉 $O(h-i)$ 시간이 필요함
 - 결국 level i에 있는 모든 노드들의 시간의 합은 $O((h-i)2^i)$
 - 모든 레벨 $i = 0, \dots, h$ 에 대해 이 시간을 모두 더하면,

$$S = \sum_{i=0}^h c(h-i)2^i = c \sum_{i=1}^h 2^{h-i}$$
 - 역급수 형태이므로 $2S$ 를 계산한 후, $2S$ 에서 S 를 빼면 다음 식을 얻는다. 단, $n \leq 2^{h+1}$ 이다

$$S = c(-2^{h-1} - h + \sum_{i=1}^h h2^i) = c(2^{h-1} - h - 2) = O(n)$$
 - 따라서 `make_heap`의 수행시간은 $O(n)$ 이다
- xi. 힙 정렬 `heap_sort()`
1. `make_heap`과 `heapify_down` 함수를 반복적용하여 값들을 오름차순으로 재배치하는 함수 → 힙 정렬(heap sort) 알고리즘이라 불림
 2. [핵심] 가장 큰 값이 힙의 루트에 저장되어 있다. 이 값은 정렬 순서에 따라 리스트의 가장 마지막에 위치해야 한다
 - $A[0], A[n-1] = A[n-1], A[0]$ (두 값을 교환 - swap)
 - $A[0]$ 의 값은 힙 성질을 만족하지 않을 수 있으므로 `heapify_down(n-1, 0)`를 호출해서 성질을 만족하는 자리로 이동
 - 이제 힙은 $(n-1)$ 개의 값($A[0] \dots A[n-2]$)으로 구성되므로 $n = n - 1$ 로 변경 후, 위의 두 과정을 다시 반복

```
def heap_sort(self):
    n = len(self.A)
    for k in range(len(self.A)-1, -1, -1):
        self.A[0], self.A[k] = \
            self.A[k], self.A[0]
    n = n - 1      # A[n-1]은 정렬되었으므로
    self.heapify_down(0, n)
```



3. 춤으로 표현한 힙 동영상: <https://youtu.be/Xw2D9aJRBY4>
4. [👉] [해보기] 위에서 설명한 Heap 클래스 구현한 후 다음을 수행해보자

```
heap = Heap([2,8,6,1,10,15,3,12,11])
# 이미 Heap 객체 생성시 make_heap이 호출되어 힙이 됨
print(heap)          # [15, 12, 6, 11, 10, 2, 3, 1, 8]
heap.heap_sort()
print(heap)          # [1, 2, 3, 6, 8, 10, 11, 12, 15]
```
- xii. 삽입 연산 `insert(key)`: 기준의 힙에 새로운 key 값을 삽입하는 연산
 1. 힙 리스트 A의 가장 오른쪽에 새로운 값 x를 저장하고, 이 값을 힙 성질이 만족하도록 위치를 재조정해야 한다
 2. 이 경우엔 x가 힙의 리프에 위치하므로, 루트 노드 방향으로 올라가면서 자신의 위치를 조정하면 된다 → `heapify_up` 함수 구현

```
def heapify_up(self, k): # 올라가면서 A[k]를 재배치
    while k>0 and self.A[(k-1)//2] < self.A[k] :
        self.A[k], self.A[(k-1)//2] = \
            self.A[(k-1)//2], self.A[k]
        k = (k-1)//2

def insert(self, key):
    self.A.append(key)
    self.heapify_up(len(self.A)-1)
```
 3. **수행시간**: 가장 마지막 레벨의 가장 오른쪽 빈 칸에 삽입되어 루트 노드까지 올라갈 수 있으므로 `heapify_up`과 `insert` 모두 힙의 높이만큼의 시간이 필요 → **$O(\log n)$**
- xiii. 삭제 연산 `delete_max()`: 임의의 값을 삭제하는 것이 아닌 루트 노드에 저장된 가장 큰 값을 삭제후 리턴하는 함수
 1. 루트 노드의 값을 삭제 후 리턴해야 하므로, `A[0]`를 `A[n-1]`의 값으로 대체하고, `heapify_down(0, n-1)`를 호출해 `A[0]`를 재배치한다

```

def delete_max(self):
    if len(self.A) == 0: return None
    key = self.A[0]
    self.A[0], self.A[len(self.A)-1] = \
        self.A[len(self.A)-1], self.A[0]
    self.A.pop() # 실제로 리스트에서 delete!
    heapify_down(0, len(self.A)) # len(A) = n-1
    return key

```

2. **수행시간**: heapify_down이 1번 호출되고, 이 함수의 수행시간이 전체 시간을 결정하므로 $O(\log n)$ 시간에 수행

xiv. 연산 및 수행시간 정리 (n개의 값을 저장한 리스트와 힙에 대해)

1. heapify_up, heapify_down: $O(\log n)$
2. make_heap = n times x heapify_down = $O(n \log n) \rightarrow O(n)$
3. insert = 1 x heapify_down: $O(\log n)$
4. delete_max = 1 x heapify_down: $O(\log n)$
5. heap sort = make_heap + n x heapify up = $O(n \log n)$

xv. [Python] **heapq** 모듈에서 heap 관련 함수를 그대로 제공한다

1. 단, 부모노드에 자식노드보다 더 크지 않은 값이 저장되는 min-heap임에 유의하자 (앞에서 다른 것은 max-heap임!)
2. **import** heapq
3. 힙 자체는 리스트를 사용한다: $h = []$
4. 지원연산:
 - o heappush(h , key): 힙 h 에 key 값을 삽입 (= insert와 동일)
 - i. heappush(h , (key , $value$))처럼 튜플 삽입 가능
 - o heappop(h): 최소값을 지우고 리턴 (delete_min의 역할)
 - o heapify(A): 리스트 A 를 힙 성질이 만족되도록 변환
 - i. make_heap()과 동일 (단, min-heap으로 변환)
 - o $h[0]$: 힙의 최소값을 알고 싶다면

9. 정렬을 위해 반드시 필요한 최소 비교 횟수는 얼마인가? (하한, lower bound)

- a. n개의 값을 비교해서 정렬하는 데 필요한 최소 비교횟수는 얼마일까?
- b. 이를 정렬 문제의 비교횟수에 대한 하한(lower bound)이라 부른다
- i. 어떤 정렬 알고리즘도 이 최소 비교횟수보다 더 작은 횟수로 정렬하기란 불가능함을 의미한다
- c. 예를 들어, 세 개의 서로 다른 값 a, b, c가 있다고 하자. a, b, c 값이 어떻게 주어지더라도(즉, 최악의 경우) 최소 몇 번의 비교는 해야할까?
- i. a < b이고 b < c라면 (2번의 비교 결과) a < b < c로 결론 (2번 비교 충분)
 - ii. a < b이고 b > c라면 b가 가장 크다는 건 알지만, a와 c의 크기 순서는 모른다.
그래서 a ? c 비교를 한 번 더 해야 한다 (3번 비교 필요)
 - a < c 이면, a < c < b로 결론
 - a > c 이면, c < a < b로 결론
 - iii. a > b인 경우에도 유사하게 세 번의 비교가 필요한 경우가 있다
 - b < c < a인 경우, c < b < a인 경우, b < a < c인 경우 세 가지 중 하나의 결론에 도달
 - iv. 결국, 6가지 결론($3! = 6$) 중 하나에 도달하는 것이 정렬 알고리즘의 목표다
 - v. 이를 위해, 최소 3번의 비교가 필요한 경우가 존재하므로, 세 개의 값을 비교 정렬하기 위해서 필요한 비교횟수의 하한은 3이다
- d. 이 논리를 n개의 값 A[0] ~ A[n-1]에 대한 최소 비교횟수를 증명하는 데 이용할 수 있을까?
답은 YES!
- i. 두 수를 한 번 비교하면 그 결과에 따라 왼쪽과 오른쪽으로 나뉘고, 다시 같은 비교 과정을 반복하면 트리(tree) 모양이 된다.
 - ii. 각 노드는 비교이고, 각 에지는 비교의 결과(True/False)를 나타낸다
 - iii. 리프 노드는 정렬의 결과(경우) 중 하나이다
 - iv. 이러한 증명법을 **결정 트리 (algebraic decision tree)**를 이용한 하한 증명법이라 부른다
 - v. 정렬 하한을 증명하기 위해서 사용되는 결정 트리 모델은 각 노드가 두 수의 크기를 비교하는 연산, 즉 두 수를 뺀 값이 0보다 큰지 작은지 (True or False)를 결정하는 것이기에 특별히 **비교 결정 트리 (comparison decision tree)** 모델이라 부른다
- e. 이 트리의 리프 노드 개수는? [힌트: n = 3인 경우를 그대로 확장해서…]
- i. _____
- f. 어떤 비교를 통해 n개의 값을 정렬하는 알고리즘도 여러 번의 비교를 통해 리프 노드에 도달하게 된다. 이 알고리즘이 수행한 비교는 루트 노드에서 리프 노드까지 방문한 노드(비교)의 개수와 같다. 가장 깊은 곳의 리프 노드까지의 경로의 길이가 트리의 높이(height)이므로 최소한 결정 트리의 높이만큼의 비교는 반드시 필요하게 된다!
- g. n! 개의 리프 노드를 갖는 이진 트리의 **최소 높이** h는 얼마일까?



i. [?] 왜 최소 높이를 생각할까?

- (긁어보세요) 비교횟수의 하한을 증명해야하기 때문이다. $n!$ 개의 리프를 갖는 트리의 높이는 매우 다양할 수 있다. 높이가 큰 트리는 해당 높이만큼 비교를 해야 하지만, 더 작은 높이의 트리도 있으므로 하한을 증명해야 하는 입장에서는 트리의 높이를 최대한 줄여야 한다. 더 줄일 수 없는 최소 높이만큼 (어떤 알고리즘도) 비교는 해야 하기 때문이다

ii. [?] h 는 얼마일까?

- (긁어보세요) 최소 높이를 갖는 트리는 힙 자료구조처럼 좌우 부트리의 균형이 최대한 맞추도록 구성하면 된다. 결국, 이러한 트리는 최소 $\log n$ 이다

iii. [질문] $\log(n!) \geq cn\log n$ (c 는 적절한 실수 상수) 증명?

- $n! = n \times (n-1) \times \dots \times 2 \times 1 \geq n \times (n-1) \times \dots \times n/2$
 $\geq n/2 \times n/2 \times \dots \times n/2 = (n/2)^{n/2}$

$$\log(n!) \geq \log(n/2)^{n/2} = (n/2)(\log n - 1) \geq cn\log n$$

iv. 결론: 비교를 통해 정렬하는 어떤 알고리즘도 최소 $cn \log n$ 번 이상의 비교가 반드시 필요하다

- merge_sort는 더 이상 잘 할 수 없는 최적(optimal) 알고리즘임!!

10. [유일성 문제 하한: skip 가능] 선형 결정 트리를 이용한 하한 증명의 다른 예

- 유일성 문제 (uniqueness problem): n 개의 입력 값이 모두 다르면 YES, 같은 값들이 있으면 NO를 출력하는 문제
- 우선 이 문제에 대한 알고리즘을 생각해보자. (리스트 A에 n 개의 값이 저장)
 - 알고리즘 1: 이중 for 루프를 반복하면서 $A[i]$ 에 대해서 $A[i]$ 와 같은 $A[j]$ 가 있는지 일일이 비교하는 방법
 - $O(n^2)$ 비교 필요
 - 알고리즘 2: 오름차순으로 정렬한 후, 작은 수부터 차례대로 인접한 두 수가 같은지 비교하는 방법
 - $O(n\log n)$ 비교 필요
 - 알고리즘 3: 해시 테이블을 이용해 같은 슬롯에 저장되는 수를 비교하는 방법 (단, 테이블의 슬롯 수가 충분히 많아야 하고, universal 해시 함수를 사용. 자세한 내용은 자료구조 교재의 해시 테이블 편 참고)
 - 평균 $O(n)$ 비교 필요 (최악의 경우의 수행시간이 아니고 평균 수행시간임!)
- 비교 횟수의 하한은 얼마일까?

- i. 정렬처럼 결정트리를 정의해서 증명해보자. 내부 노드에서는 두 수를 비교하고 결과에 따라 자식 노드로 이동하고 이를 반복해 리프 노드에 도착한다. 리프 노드는 문제의 최종 답을 의미하는데, 이 문제에서는 YES, NO 둘 중 하나이다
- ii. 정렬 문제의 결정트리에서는 리프 노드가 $n!$ 개나 존재한다. 즉, 문제의 답 자체가 $n!$ 개 중 하나가 된다. $n!$ 개의 리프 노드를 갖는 이진트리의 높이가 최소 $n \log n$ 이상이 되어야 하고 바로 이 높이가 최소 비교 횟수를 의미하게 된다. 그런데 이 문제의 답은 두 가지뿐이므로 극단적으로 YES를 나타내는 리프 노드 하나와 NO를 나타내는 리프 노드, 총 2개의 리프 노드만 갖는 결정트리가 될 수도 있다. 그러면 트리의 높이가 1이 되어 하한의 의미가 없다
- iii. 실제 결정트리의 리프 노드가 2개 아니라 최소 $n!$ 개임을 증명할 수 있다. 정확히 말하면, $n!$ 개의 YES 리프 노드가 존재함을 증명할 수 있다
- iv. 증명 아이디어:
 - 입력은 n 개의 값으로 구성된 순열 (permutation)이라고 가정한다. 단순하게 1부터 n 까지의 수의 순열이 주어진다고 가정한다. 입력으로 주어지는 순열의 모든 종류가 $n!$ 개이다
 - $n!$ 개의 YES 리프 노드가 존재함을 보이기 위해서는 정확히 하나의 입력 순열이 하나의 리프 노드에 도달함을 보이면 된다. 이를 위해서는 서로 다른 순열은 서로 다른 리프 노드에 도달한다는 것을 보이는 것과 같다
 - 구체적인 증명은
<https://sites.cs.ucsb.edu/~suri/cs235/lowerBounds.pdf> 참조
- v. 결론: 비교를 통해 모든 값이 다른지를 판별하는 문제에 대한 어떤 알고리즘도 최소 $\lceil n \log n \rceil$ 번 이상의 비교가 반드시 필요하다
- vi. 이 하한은 다른 문제들의 하한을 증명하는 데 이용할 수 있다. 이를 위해, 13장에서 언급한 변환(reduction) 방법을 사용하는데, 아래 문제 모두 유일성 문제로부터 변환되어 최소 $\lceil n \log n \rceil$ 번 이상의 연산이 반드시 필요하다고 증명되었다
 - **Min Gap 문제**: 실수 축 위에 주어진 n 개의 값들 중에 가장 가까운 두 인접한 수를 계산하는 문제
 - **Point Set Diameter 문제**: 2차원 평면에 n 개의 점들이 주어질 때, 가장 먼 두 점 사이의 거리를 계산하는 문제

11. Radix sort 알고리즘

- a. 실제 정렬하는 값의 범위가 정렬 전에 이미 알려져 있는 게 일반적이다
- b. 예를 들어, 시험 점수를 정렬한다면, 0부터 100 사이의 정수인 경우가 많다
- c. 이렇게 정렬할 값의 범위가 정해진 경우, 이 **범위 정보를 이용**하면 더 빠르게 정렬할 수 있다
→ radix 정렬, count 정렬 등
- d. radix 정렬 알고리즘은 두 값을 비교해서 정렬하는 **비교기반** (comparison-based) 정렬 알고리즘이 아니므로, $c n \log n$ 번의 비교가 반드시 필요한 것이 아님에 유의하자! 다시 말하면, 이 비교 횟수의 하한의 적용을 받지는 않는다
- e. 자리수가 최대 d개인 정수 n개를 radix 정렬한다고 가정하자
- f. 예: $A = [10, 1234, 9, 7234, 67, 9181, 733, 197, 7, 3]$ $n = 10$, $d = 4$
 - i. $A[0]$ 부터 차례대로 1의 자리 값만 보고, 아래 해당되는 큐(queue)에 enqueue한다
 $[[10], [9181], [], [733, 3], [1234, 7234], [], [], [67, 197, 7], [], [9]]$
 - ii. 이 값들을 0번째 큐부터 차례대로 dequeue해서 다시 모은다. 그러면 값들이 1의 자리 수에 의해 오름차순으로 정렬된 상태가 된다
 $A = [10, 9181, 733, 3, 1234, 7234, 67, 197, 7, 9]$
 - iii. 이제는 10의 자리 값만 보고 해당되는 큐에 enqueue한다
 $[[3, 7, 9], [10], [], [733, 1234, 7234], [], [], [67], [], [9181], [197]]$
 - iv. 다시 dequeue를 해서 모은다. 그러면 값들이 10의 자리와 1의 자리 값에 따라 오름차순으로 정렬된 상태가 됨을 확인 할 수 있다
 $A = [3, 7, 9, 10, 733, 1234, 7234, 67, 9181, 197]$
 - v. 이 과정을 100의 자리와 1000의 자리에 대해 반복하면 최종적으로 오름차순 정렬 순서를 얻을 수 있게 된다
 - vi. 아래 10개의 큐를 이용해서 직접 쓰면서 알고리즘을 시뮬레이션 해보자

0	1	2	3	4	5	6	7	8	9	

g. Pseudo code:

```

B = 10                                # B는 진수를 의미. 십진수이므로 10으로 정함
for i in range(d):
    slots = [[], ..., []]      # 큐를 B개의 빈 리스트로 구성
    for a in A:
        slots[a%(B**i)].append(a)
A = []
for i in range(B):          # 큐에 있는 값을 다시 차례로 모음
    A += slots[i]
del slots

```

h. 수행시간을 계산해보자

- i. 자리수가 d개라면, d번 값을 큐에 enqueue, dequeue를 한 번씩 한다. 큐 연산은 $O(1)$ 시간이 필요하기에, $d \times O(n) = O(dn)$ 시간이면 된다
- i. [?] 수행시간과 비교횟수 하한을 비교해보자
 - i. 두 수를 비교해서 정렬하는 경우에는 $c n \log n$ 번 이상의 비교가 반드시 필요하지만, radix 정렬처럼 두 수를 비교하지 않고, 입력 값의 자리 값만 보고 정렬을 하는 **비-비교기반 (non-comparison-based)** 정렬 알고리즘의 경우에는 두 수의 비교가 필요가 없다. 따라서 비교기반 정렬 문제의 하한을 적용할 수 없다
 - ii. radix 정렬 알고리즘의 $O(dn)$ 시간이 걸리고, d가 상수 값이라면 $O(n)$ 시간에 수행된다. 따라서 자리수가 크지 않는 경우에 매우 빠른 정렬 알고리즘이다
- j. 위 알고리즘이 사용하는 메모리의 양은? 슬롯에 n개의 수가 나눠 저장된다. 따라서 $O(n)$ 이면 된다. 그런데 슬롯에 수를 직접 저장하지 않고, count라는 배열의 $count[i]$ 에는 해당 자릿수가 i인 수의 개수를 저장하는 방식으로 **메모리의 양을 줄일 수 있다**
 - i. 첫 자리에 대해서, $count=[1, 1, 0, 2, 2, 0, 0, 3, 0, 1]$
 - ii. $count$ 의 prefix sum을 구한다: $count=[1, 2, 2, 4, 6, 6, 6, 9, 9, 10]$
 - iii. A의 가장 마지막 수부터 보면서 해당 자리수를 기준으로 정렬된 위치(index)를 찾아 임시 배열 B에 저장한다 (예: 3은 첫 자리가 3이므로 $count[3] = 4$. 의미는 첫 자리 순서로 4번째라는 의미로 $B[4-1] = B[3] = 3$ 처럼 저장한다. 그리고 $count[3]$ 의 값을 1 뺀다. 이 방식으로 각 자리수에 대해 반복한다

12. Python에서 제공하는 정렬 관련 기능을 적극 활용해보자

```
>>> A = [3, 1, 2, -5]                                     # A의 값을 오름차순으로 정렬
>>> A.sort()                                              # A의 값을 내림차순으로 정렬
>>> A.sort(reverse=True)                                    # A의 값을 복사한 후, 오름차순으로 정렬해 리턴
>>> B = sorted(A)                                         # A의 값을 복사한 후, 오름차순으로 정렬해 리턴
>>> P = ['8', '3', '12']                                    # str 순서(사전식 순서)에 따라 오름차순 정렬
>>> P.sort()
>>> Q = ['8', '3', '12']
>>> Q.sort(key=int)                                       # key에 저장된 함수를 불러 그 값으로 비교 수행
>>> S = [(1, 2), (4, 5), (4, -3)]
>>> S.sort(key=lambda x: x[1])                             # 어떻게 정렬? _____
>>> S.sort(key=lambda x: x[0]+x[1])                         # 어떻게 정렬? _____
>>> S.sort()                                              # 정렬 결과는?
[(1, 2), (4, -3), (4, 5)]                                # x를 기준으로 정렬
# x[0] 값이 같다면 x[1]의 값으로
>>> S.sort(key=lambda x: (x[0], -x[1]))                  # 직접 실행해보자
# 어떻게 정렬? _____
```

- a. lambda 함수는 한 줄 함수 (파이썬 교재 참고)
- b. 일반 함수를 정의해서 key 값으로 넘겨도 됨



13. [고급: skip 가능] Python의 sort 함수는 어떤 정렬 알고리즘을 사용할까?

- a. 2002년에 Tim Peters가 제안한 Tim sort 알고리즘을 Tim Peters가 직접 sort 함수로 구현
 - i. Java, JavaScript, Android, Swift에서 제공되는 정렬 함수는 모두 Tim sort 알고리즘으로 구현되어 사용하게 되면서, 여러 언어에서 사용되는 표준 정렬 알고리즘으로 자리매김 함
 - ii. 실제 환경에서 다루는 값들은 "랜덤"한 데이터가 아니라 부분적으로 정렬된 데이터가 많다는 점을 정렬 알고리즘에 반영함. 따라서 실제 데이터 정렬에 매우 좋은 성능을 보여준다는 평가
 - iii. 요약하면, insertion 정렬과 merge 정렬을 결합한 hybrid 정렬 알고리즘
 - iv. [참조: Timsort](#) (Wikipedia 설명)
 - b. 수행 시간과 특징
 - i. 최악의 경우: $O(n \log n)$ [2014년에야 이론적으로 증명됨]
 - ii. (거의) 정렬된 입력의 경우: $O(n)$
 - iii. 사용하는 메모리의 양: $O(n)$
 - iv. Stable 알고리즘이지만 in-place 알고리즘은 아님
 - c. Tim sort 알고리즘 개요
 - i. run이라는 연속된 값들의 덩어리로 나눈다
 - run의 최소 크기는 $2^5 - 2^6$ 사이의 값으로 주로 2^5 사용
 - run은 이진탐색을 이용한 insertion 정렬 알고리즘을 사용해 오름차순이나 내림차순으로 정렬함
 - run의 첫 두 수 a와 b가 $a \leq b$ 면 오름차순으로, $a > b$ 면 내림차순으로 정렬
 - 정렬된 run 들이 만들어지는데, 내림차순으로 정렬된 run은 순서를 바꿔 오름차순으로 재배열 함 → 결국, 오름차순으로 정렬된 run 들이 연속적으로 리스트에 저장됨 (run에 속한 값들이 정렬된 것이지, 전체 run들이 정렬된 것이 아님을 유의하자)
 - ii. run 들을 stack에 차례대로 push하면서 (조건에 맞으면) 병합(merge)함
 - stack에 run을 물리적으로 copy해서 push하는 것이 아니라 run의 (start index, end index) 쌍만 push 함
 - run A를 push하는 경우, 가장 위에는 A, 그 밑에는 차례로 B, C 세 개의 run이 있다고 하자. 여기서 A, B, C는 실제 리스트에서도 연속적으로 위치함에 유의.
 - (1) $|A| < |B|$ 이고 (2) $|A| + |B| < |C|$ 라는 두 조건 중 하나라도 만족하지 않는다면,
- B를 A와 C 중에서 더 짧은 run과 merge한다. 이 과정을 위의 두 조건을 모두 만족할 때까지 스택의 가장 위 세 개의 run에 대해 반복한다

- 위 병합과정을 거친 후, 스택에 최종적으로 저장되는 run이 가장 위부터 A, B, C, D, ...라고 하면, 그 크기는 $|A| < |B| < |C| < |D| < \dots$ 이고, $|C| > |A|+|B|$, $|D| > |B|+|C| \dots$ 의 관계가 성립

- 이 부등식을 통해 스택에 저장된 run이 최대 몇 개인지 알 수 있을까?
(즉, 스택의 최대 크기?)

[힌트] 부등식이 Fibonacci 수와 유사?

run의 최소 길이를 32로 정했다고 가정하면:

$$32 \leq |A| < |B|, 32+32 < |C|, 32 + 64 < |D|, \dots$$

스택의 가장 바닥에 있는 run의 크기는 k번째 Fibonacci 수의 크기 이상을 갖게 된다 (처음 시작 수는 32, 32) 이 값은 $O(1.618^k)$ 정도라서, $k = O(\log n)$ 이라고 해도 된다. 즉, 스택의 크기는 $O(\log n)$ 정도면 충분하다는 사실!

iii. 합병 과정에서 여러 최적화 기법이 동시에 적용됨

- A와 B를 합병할 때 사용되는 임시 메모리는 $\min(|A|, |B|)$ 만 사용하는 합병 방법을 사용해 메모리 절약 (합병 정렬의 [한 발 더] 부분을 읽어 볼 것!)
- B의 제일 작은 값보다 작은 A의 초반의 값들을 찾고, A의 마지막 값보다 큰 B의 후반의 값들을 찾으면, 그 값들은 합병할 필요가 없게 된다. 나머지 값들에 대해서 합병을 시도함

예: A = [1, 2, 5, 7, 9, 10, 13, 15, 16] B = [8, 9, 10, 14, 19, 21, 28] 이면 B의 처음 값 8보다 작은 A의 값이 1, 2, 3, 5, 7인데, 이 값들은 합병된 후에 가장 먼저 등장할 값들이 되어 굳이 합병 알고리즘으로 처리할 필요가 없다. 그리고 A의 마지막 값인 16보다 큰 B의 후반의 19, 21, 28 역시 마찬가지다. 결국, 붉은 색으로 강조한 값들만 합병하면 된다!

- 질주 모드 (galloping mode): 예를 들어, A = [15, 22, ...], B = [1, 3, 5, 6, 8, 10, 12, 14, 16, ...] 라면, A의 15값보다 작은 B의 값들을 비교를 통해 찾아 15보다 먼저 합병해야 한다. 예에서는 그런 값이 8개이므로 8번 (차례대로) 비교하면 된다. 처음 3개의 값 1, 3, 5가 연속해서 15보다 작다면 질주 모드로 바뀌는데, $2^1, 2^2, 2^3, 2^4$ 칸씩 건너뛰면서 비교하는 것을 의미한다 (값의 개수가 충분히 많다면 비교횟수를 얼마나 줄일 수 있을까?)

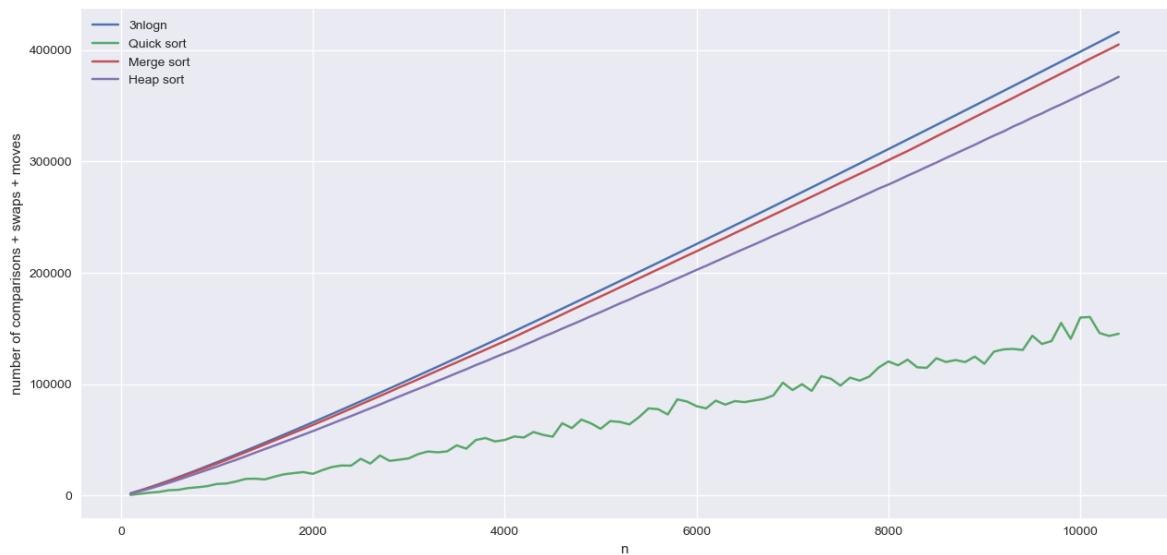
iv. 알고리즘 설명이 친절하게 되어 있는 한글 사이트:

<https://d2.naver.com/helloworld/0315536>

14. 정렬 알고리즘 비교횟수 정리 표 [wikipedia] (Big-O 기호는 생략함)

	Best	Average	Worst	추가 Memory	Stable	In- place	Note
Insertion	n	n^2	n^2	1	Yes	Yes	
Selection	n^2	n^2	n^2	1	No	Yes	
Bubble	n^2	n^2	n^2	1	Yes	Yes	tiny code
Quick	$n \log n$	$n \log n$	n^2	$\log n \sim n$	No	Yes	practically quickest
Merge	$n \log n$	$n \log n$	$n \log n$	n	Yes	No	easy to parallelize
Heap	$n \log n$	$n \log n$	$n \log n$	1	No	Yes	
Tim	n	$n \log n$	$n \log n$	<n	Yes	No	Python standard sort function
Radix	-	dn	dn	n	Yes	No	Non-comparison sort

- [실험 예] 다양한 n 에 대해 자연수 n 개를 랜덤하게 생성하여 각각 quick, merge, heap 정렬 알고리즘을 수행하고, 세 알고리즘의 비교 횟수 + 이동 횟수를 얻어 `matplotlib` 패키지의 `pyplot` 모듈로 그린 결과임. 가장 위의 함수는 $3n\log n$
 - quick 정렬이 압도적으로 좋고, heap 정렬, merge 정렬 순서
 - 세 정렬 알고리즘 모두 비교횟수와 이동횟수의 합이 $3n\log n$ 을 넘지 않음



15. 정렬과 선형 탐색 (linear scan)을 이용한 문제 풀이

- a. 상당히 많은 종류의 문제를 최적의 시간에 풀 수 있는 알고리즘 기법으로 기업 알고리즘/코딩 인터뷰에 자주 등장한다
- b. [문제 1: pair-sum] n 개의 정수 값이 저장된 리스트 A 와 정수 값 K 가 입력으로 주어진다. 더해 K 가 되는 A 의 두 수를 찾아라
 - i. 가장 단순한 방법은 모든 $A[i]$ 와 $A[j]$ 쌍에 대해, 합이 K 가 되는지 검사하면 된다. 이중 for 루프를 사용하면 $O(n^2)$ 시간에 가능
 - ii. 더 빠르게 할 수 없을까?
 - [힌트] 오름차순으로 먼저 정렬을 하자. $i = 0, j = n-1$ 에서부터 i 는 증가하고 j 는 감소하면서 $A[i] + A[j] == K$ 인지 검사하는 식으로 선형 탐색을 해보자
 - 만약, $A[i] + A[j] < K$ 인 경우엔 i 를 늘려야 할까, j 를 줄여야 할까? (답: i 를 늘려야 $A[i]$ 값이 커져 합이 K 에 더 가까워진다) $A[i] + A[j] > K$ 인 경우에는? (답: j 를 줄여야 $A[j]$ 값이 작아져 합이 K 에 더 가까워진다)
 - 이 알고리즘의 수행시간은?
 - $O(n \log n)$ 시간 정렬 후, 양 쪽 끝부터 한 번의 비교 후 한 번씩 움직이는 과정을 반복하는 $O(n)$ 시간 선형 스캔으로 구성되어 전체 수행 시간은 $O(n \log n)$
 - iii. 이 보다 더 빠르게 가능할까?
 - [힌트] 평균 $O(n)$ 시간이라면 가능하다 → 해시 테이블 자료구조를 사용하자
 - 각 i 에 대해, $A[i]$ 값을 모두 해시 테이블에 삽입한다 → 평균 $O(n)$ 시간
 - 각 j 에 대해, $K - A[j]$ 값이 해시 테이블에 있는지 탐색한다 → 탐색 결과가 True면 $A[i] + A[j] = K$ 인 두 수가 존재한다는 뜻이다 → 평균 $O(n)$ 시간
- c. [문제 2: min-difference] 두 리스트 A 와 B 에 각각 n 개와 m 개의 정수 값이 저장되어 있다. A 의 값과 B 의 값의 차이가 최소가 되는 두 수를 선택하고 싶다. 어떻게 해야 할까?
 - i. 역시, 가장 단순한 방법은 이중 루프를 돌면서 모든 쌍 ($A[i], B[j]$)에 대해 차를 구하고 최소 차를 유지하면 된다 → $O(n^2)$ 시간 필요. 더 빠른 방법은?
 - ii. [힌트] 위의 pair-sum 문제와 유사
 - iii. 두 리스트의 값을 먼저 오름차순으로 정렬한다. $i = 0, j = 0$ 으로 초기화 한 후, $A[i]$ 와 $B[j]$ 의 차를 현재까지의 최소 차 min_diff 값과 비교해 업데이트한다
 - $A[i] > B[j]$ 이면, $i += 1$ or $j += 1$?
 - $A[i] < B[j]$ 이면, $i += 1$ or $j += 1$?
 - iv. 예를 들어, $A = [2, 9, 3, 1, 0, 2, 8]$, $B = [3, 1, -5, 8, 10, 2]$ 에 적용~
 - v. 이 알고리즘의 수행시간은?

d. [문제 3: swap-sum] 두 리스트 A와 B에 각각 n개와 m개의 정수 값이 저장되어 있다. A의 값 하나와 B의 값 하나를 서로 교환해서 A의 값의 합과 B의 값의 합이 같도록 만들고 싶다. 어떤 두 수를 선택해 교환해야 할까?

- i. 역시, 가장 단순한 방법은 모든 쌍을 고려해 교환해보는 방법이지만, 더 빠른 방법이 존재한다.
- ii. A의 수 a와 B의 수 b를 서로 교환한다고 하자. 그러면 교환 후 합은 각각 $\text{sum}(A) - a + b$, $\text{sum}(B) - b + a$ 가 된다. 두 합이 같아야 하므로 $\text{sum}(A) - \text{sum}(B) = 2(a - b)$ 가 된다
- iii. $c = \text{sum}(A) - \text{sum}(B)$ 라고 하면, c는 상수이다. 즉, $(a-b)$ 의 2배가 상수 c인 쌍 (a, b)를 찾으면 된다! pair-sum 문제와 본질적으로 동일하지 않은가?
- iv. 알고리즘을 완성해보자. 이 알고리즘의 수행시간은 당연히 $O(n \log n)$

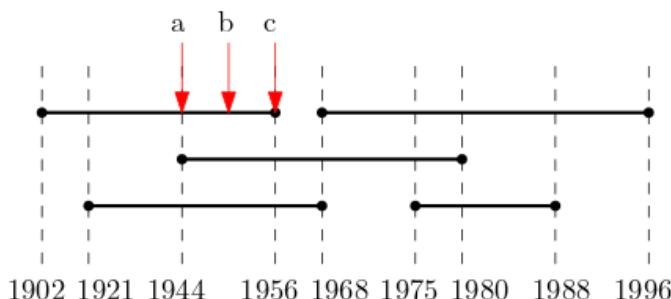
e. [문제 4: majority 선택] 리스트 A에 양의 정수 n개가 저장되어 있다. 전체 개수의 절반 넘게 등장하는 수를 majority 수라고 정의한다. 예를 들어, $A = [1, 4, 1, 2, 1]$ 이라면, 전체 5개의 수 중에서 1이 3개로 $5/2 = 2.5$ 개보다 많기에 majority 수이다. 그러나 $A = [1, 3, 1, 2]$ 라면, 1이 2번 등장하지만, $4/2 = 2$ 개를 넘지 않기 때문에 1은 majority 수는 아니고 2와 3 역시 majority 수가 아니기에 이 리스트에는 majority 수가 없다. 여러분들은 주어진 A에서 majority 수가 있다면 출력하고 없다면 -1을 출력하는 코드를 작성해야 한다

- i. [느린 방법] 모든 $A[i]$ 가 몇 번씩 나타나는지 $O(n)$ 시간에 검사한 후, $n/2$ 번 보다 많이 등장하는 수가 있는지 본다 - $O(n^2)$ 시간
- ii. [약간 빠른 방법] $O(n \log n)$ 시간에 오름 차순으로 정렬한다. 정렬된 값을 차례로 보면서 (linear scan) 각 수가 몇 번씩 나타나는지 세는 건 쉽다. 결국 $O(n \ log n)$ 시간이면 충분 (조금만 더 생각해보면 각 수를 셀 필요도 없다!)
- iii. [평균적으로 빠른 방법] 삽입, 탐색에 평균 $O(1)$ 시간이 걸리는 해시 테이블을 사용한다. 해시 테이블 아이템은 $(A[i], C_i)$ 로 C_i 는 $A[i]$ 의 등장 횟수로 정의한다. 모든 $A[i]$ 를 삽입하는 데, 이미 테이블에 있다면 C_i 를 1 증가시킨다. 평균 $O(n)$ 시간에 해결 가능하다
- iv. [최악의 경우에도 빠른 방법] 아래 예를 보자. A의 값을 왼쪽부터 하나씩 보면서 판단을 해보자.
 - $A = [1, 2, 3, 1, 4, 1, 1, 1, 2, 3, 1]$
 - $A[0] = 1, A[1] = 2$ 에서 알 수 있는 사실은? 현재까지 본 2개의 값만을 한정하면, 1은 majority 수가 아님을 알 수 있다. (물론, 나중에 1이 많이 등장해 majority 수로 판명될 수도 있다)
 - 같은 이유로 2 역시 현재까지는 majority 수는 아니다
 - $A[2] = 3, A[3] = 1$ 로 서로 다르므로, 역시 두 수도 현재까지 고려한 수들만 고려할 때 majority 수가 아니고, 역시 $A[4] = 4, A[5] = 10$ 으로 두 수 모두 majority 수가 아니다.

- $A[6] = 1$, $A[7] = 1$ 이 되어 처음으로 같은 수가 등장했다. 1이 현재까지 중에 majority 수가 될 수 있는 가능성이 있다. $\text{majority} = 1$, $\text{count} = 2$ 로 기록한다
- $A[8] = 2$ 이고, majority 수인 1과 다르므로 count 를 1 감소
- $A[9] = 3$ 이고 majority 수인 1과 다르므로 count 를 1 감소하면, 결국 $\text{count} = 0$ 이 된다. 그러면 1은 더 이상 majority 수가 아니다 (즉, 현재까지 1이 등장한 횟수만큼 1이 아닌 수가 등장했다는 의미이므로)
- $A[10] = 1$ 이 되었고, 현재 $\text{count} = 0$ 이므로 $\text{majority} = 1$ 로 수정한다. 더 이상 수가 없으므로 $\text{majority} = 1$ 이 된다!
- 이 방법이 항상 올바른 답을 출력하는가? 더 필요한 과정은 없을까?
- 최종 알고리즘의 수행시간이 $O(n)$ 임을 확인해보자

v. 위의 네 가지 방법을 모두 코드로 옮겨보자

- f. [문제 5: most-people] n 명의 태어난 해와 사망한 해의 값이 입력으로 주어진다. $\text{birth}[i]$ 와 $\text{death}[i]$ 이 각각 i 번의 태어난 해와 사망한 해이다. 여러분은 가장 많은 사람들이 생존한 연도를 알아내야 한다. 단, 모든 사람은 1900년 이후 태어났고, 2000년 이전에 사망했다고 가정한다
- 가장 단순한 방법은 무엇일까? 1900년부터 2000년까지 모든 연도에 대해 해당 연도를 포함하는 사람 수를 세서 최대 수를 유지하면 된다. 그럼 $O(2000n)$ 정도의 시간이면 충분
 - 그런데, 1900년부터 2000년까지 101개의 연도를 모두 고려해야 할까? $\text{birth}[i]$ 와 $\text{death}[i]$ 를 하나의 구간(interval)로 표현할 수 있기 때문에, 아래와 같은 예를 생각해볼 수 있다. 그럼 a, b, c에 해당하는 연도에 생존하는 사람들은 일치한다. 이 관찰로부터 알 수 있는 사실은?



- 결국, 모든 연도를 고려할 필요없이 $\text{birth}[i]$ 와 $\text{death}[i]$ 에서 몇 명이 생존하는지만 세면 된다. 이런 연도는 모두 $2n$ 개이다. 각각에 대해 $O(n)$ 시간으로 세면 $O(n^2)$ 이 필요하다 (더 느려짐!)
- 이 시간을 $O(n)$ 시간으로 줄여보자. [힌트: 태어난 해와 사망한 해를 한꺼번에 오름차순으로 정렬해서 차례대로 살펴보자]

g. [문제 6: longest equal number] A와 B가 섞인 문자열이 s가 주어진다. 그러면 A와 B가 같은 개수로 등장하는 $s[i] \dots s[j]$ 까지의 부문자열(substring) 중에서 가장 긴 길이의 부문자열을 찾아보자

i. 예: $S = AABABBAAABABBAAB$

$a = 1223334566777899$ # $a[i] = s[0] \dots s[i]$ 의 A의 개수

$b = 0011233334456667$ # $b[i] = s[0] \dots s[i]$ 의 B의 개수

$d = 1212101232321232$ # $d[i] = a[i] - b[i]$

ii. 리스트 d의 값은 무엇을 의미하나?

- $d[0] = d[2] = 1$ 로 두 값은 서로 같다. 무슨 의미일까?

- d의 값이 같은 $d[i]$ 와 $d[j]$ 에 대해, $s[i+1] \dots s[j]$ 는 두 문자의 개수가 같은가? $s[i] \dots s[j-1]$ 의 두 문자 개수는?

- 그러면 d의 값이 같은 가장 멀리 떨어진 인덱스 쌍 (i, j)를 구하면 되지 않나?

- 위의 예에서 가장 멀리 떨어진 쌍 (i, j)는?

- 이 인덱스 쌍은 어떻게 구해야 할까?

- 해시 테이블을 이용하는 방법?

i. $d[0] = 1$ 이다. 값 1이 해시 테이블에 없기에, $[1, 0, 0]$ 을 삽입한다. 첫 번째 값 1은 현재 d 값을, 두 번째 값 0은 1의 인덱스를 의미한다. 세 번째 값은 가장 멀리 떨어진 같은 d 값의 인덱스이다. 현재는 값 하나만 등장했기에 두 번째 값과 같은 값으로 초기화 한다

ii. $i = 1, \dots, n-1$ 순서로, $d[i]$ 값이 해시 테이블이 있는지 검사한다. 있다면, 이미 같은 값이 등장했다는 것이므로, 세 번째 값(인덱스)을 i 로 업데이트 하면 된다. 없다면, $d[i]$ 값이 처음 등장했다는 의미이므로 $[d[i], i, i]$ 를 삽입한다

iii. 결국, 모든 d의 값을 보면, 등장하는 값의 가장 먼 인덱스 쌍이 해시 테이블에 저장된다. 그 중 가장 먼 쌍을 찾아 출력하면 된다

iv. 각 $d[i]$ 값에 대해, 한 번의 탐색과 (최대) 한 번의 삽입 연산만 수행하면 되기에, 전체적으로 $O(n)$ 시간이면 충분하다

- 제 3의 리스트 c를 이용하는 방법?

i. [힌트] 차이 값 $d[i]$ 값의 범위는?

ii. 차이 값 $d[i]$ 범위는 0부터 n 사이의 정수이다. 따라서 크기가 $n+1$ 인 리스트 c를 준비하는 데, c의 인덱스를 $d[i]$ 값을 의미하도록 한다. 즉, $c[j]$ 는 차이 값이 j 가 첫 번째 등장한 인덱스와 현재까지 가장 먼 인덱스 쌍이 저장된다. 그러면 $O(n)$ 시간에 $d[i]$ 값을 차례로 살펴보면서 c를 채울 수 있다 (대신, c를 위해 $O(n)$ 의 메모리를 추가 사용했다)

6. Dynamic Programming (동적계획법)

1. 가장 강력하고 널리 쓰이는 알고리즘 기법 중의 하나

- a. 프로그래밍 경진대회 출제 문제 중 최소 20% 이상은 동적계획법(DP)으로 해결 할 수 있는 문제가 출제될 정도로 비중이 높음
- b. 다양한 DP 문제를 풀어보는 경험을 통해 DP의 원리를 이해하는 것이 가장 확실한 학습 방법임
- c. 9개의 DP 문제를 차례대로 꼼꼼히 풀어보고 이해하기!



2. 예1: Fibonacci 수 계산하기

a. $F_0 = 0, F_1 = 1, F_n = F_{n-1} + F_{n-2}$

b. 재귀 알고리즘:

```
def fibo(n): # n번째 피보나치 수 리턴
    if n <= 1: return n
    return fibo(n-1) + fibo(n-2)
```

- i. 문제점: $fibo(5) \rightarrow fibo(4) + fibo(3) \rightarrow fibo(3)+fibo(2), fibo(2)+fibo(1) \rightarrow fibo(2)+fibo(1), fibo(1)+fibo(0), fibo(1)+fibo(0)$ 처럼 $fibo(3)$ 이 2번 호출, $fibo(2)$ 는 3번 호출, $fibo(1)$ 은 4번 호출되어 같은 값을 여러번 (재귀적으로) 중복 호출되어 매우 느리게 실행됨
- ii. $O(F_n) = O(\phi^n)$ 정도의 지수 시간이 필요함 (n 이 조금만 커져도 수행시간이 지수적으로 증가해 현실적으로 의미 없는 시간)
- iii. IDEA: 피보나치 수를 (1) 계산한 후 기록해 놓고 (2) 필요할 때 사용(재사용)하자

1. [기억] 기록 + 재사용

c. 기록+재사용 재귀 알고리즘 1: (memoization 방법이라 부른다)

```
def fibo(n):
    # memo = {} → memo는 dict로 global 선언되어 있다고 가정
    # memo[k] = F_k 형식으로 저장되는 dict

    if n in memo.keys():      # F_n이 이미 memo에 기록이 있어 재사용
        return memo[n]

    # 전에 계산된 값이 아니라면:
    if n <= 1: f = n
    else: f = fibo(n-1) + fibo(n-2)
    memo[n] = f
    return f
```

- i. $fibo(5)$ 를 호출했다고 가정해보자.

1. $\text{fibo}(5)_1 \rightarrow \text{fibo}(4)_1 + \text{fibo}(3)_1$
 2. $\text{fibo}(4)_1 \rightarrow \text{fibo}(3)_2 + \text{fibo}(2)_1$
 3. $\text{fibo}(3)_2 \rightarrow \text{fibo}(2)_2 + \text{fibo}(1)_1$
 4. $\text{fibo}(2)_2 \rightarrow \text{fibo}(1)_2 + \text{fibo}(0)_1$ # memo[1]=1, memo[0]=0
 5. $\text{fibo}(2)_2$: memo[2] = 2 기록
 6. fibo(3)₂는 memo[2] + memo[1] 계산 후, memo[3]에 기록
 7. fibo(4)₁ 역시 memo[3] + memo[2] 계산 후, memo[4]에 기록
 8. fibo(5)₁ 역시 memo[4] + memo[3] 계산 후, memo[5]에 기록
- ii. 결국 fibo(k) 값은 처음 계산이 되었을 때 memo에 저장되고 그 이후엔 memo의 값이 참조되어 처음부터 계산하지 않고 재사용된다
- iii. 즉, fibo(0), fibo(1), ..., fibo(n)이 한 번씩 계산되어 기록되고 그 이후엔 재사용된다
- iv. [?] 총 수행시간은? $O(n)$

d. 기록+재사용 비재귀 알고리즘 2:

- i. 위의 재귀 알고리즘 1에서는 재귀호출하면서 memo에 값을 처음에만 기록하고 그 이후엔 재사용한다. 이 방식은 기록한 후 재사용하는 재귀호출 방법이지만 재귀호출하지 않고도 리스트를 사용하여 직관적이고 간단한 방법으로 변경할 수 있다

```
def fibo(n):
    F = [0, 1]
    if k in range(2, n+1):
        F.append(F[k-1] + F[k])
    return F[n]
```

- ii. 알고리즘 1과 재귀/비재귀의 차이일 뿐 본질적으로 같은 방법이다
- iii. [?] 수행시간 역시 $O(n)$

3. Dynamic Programming, DP, 동적계획법

- a. 원래 문제를 (반복적으로) 작은 문제로 (중복될 수도 있지만) 분할하여 분할된 문제의 해답을 기록해 놓은 후 더 큰 문제의 해답을 계산할 때 재사용하는 방법을 의미한다
- i. 문제의 해답의 성질을 분석해, 적절하게 작은 문제로 분할하고, ($F_n = F_{n-1} + F_{n-2}$ 이므로 F_n 문제가 더 작은 두 문제 F_{n-1} 과 F_{n-2} 를 계산하는 부문제로 분할됨)
 - ii. 분할된 문제의 해답이 표에 기록되어 있다면, 그 해답들을 조합해 더 큰 문제의 해답을 계산하여 다시 표에 기록하는 식으로 진행된다 ($F[n-1]$ 과 $F[n-2]$ 가 표 - 리스트에 저장되어 있다면 두 값을 더 해서 F_n 을 계산하고, $F[n]$ 에 저장하여 나중에 다시 사용할 수 있도록 함)

4. 예2: 계단 오르는 경우의 수

- a. 바닥을 1층이라 하고, n개의 계단이 있다고 하자. 한 계단 또는 두 계단씩 오를 수 있는 경우 1층에서 n층까지 갈 수 있는 서로 다른 경우의 수는 총 몇 가지인가?
 - i. 고등학교 수학의 경우의 수에서 많이 보던 문제?
- b. 1층에서 n층까지 가는 경우 하나를 생각해보자
 - i. 1층에서 시작해서 계단을 오르는 경우를 상상하기 시작하면 꼬이기 시작한다. 어떤 부문제로 분할할 수 있는지를 생각할 때에는 거꾸로 따져보는 게 좋은 경우가 많다
n층에 도달하는 마지막 순간부터 거꾸로 따져보는 것이다!
 - ii. n층에 도달하기 전에는 n-1 층에서 한 칸 올라오는 경우와 n-2 층에서 두 칸 올라오는 두 가지 경우뿐이다

(1) 해답의 구조를 생각하는 단계

- iii. 이는 n-1 층까지 올라와서 한 칸 올라오는 경우의 수와 n-2 층까지 올라와서 두 칸 올라오는 경우의 수를 더하면 된다는 것이다. 즉,
**1층에서 n층까지 올라오는 경우의 수 = 1층에서 n-1 층까지 올라오는 경우의 수 +
1층에서 n-2층까지 올라오는 경우의 수**
- iv. $A[n] = 1\text{층에서 } n\text{층까지 올라오는 경우의 수}$ 로 정의하면,

$$A[n] = A[n-1] + A[n-2]$$
이라는 점화식이 정의된다. 즉, 부문제 답의 식으로 표현됨
 - 큰 문제의 답이 작은 부문제의 답의 (점화)식으로 표현할 수 있는 문제만이 DP를 적용할 수 있다는 의미도 된다

(2) 부문제로 분할하여 목표 답을 부문제의 답으로 표현하는 단계

- v. 결국, 피보나치 수와 동일하다 (단, $A[1] = 0, A[2] = 1$)
- vi. 리스트로 A를 표현하면 A가 부문제의 답을 기록한 테이블이 되고, $k = 3, \dots, n$ 까지 차례대로 식을 계산하여 테이블을 채우면 된다

(3) 순서에 따라 테이블에 기록하면서 채우는 단계

- vii. 원래 문제의 해답은 $A[n]$ 이 저장되어 있으므로 그 값을 리턴한다

(4) 알고리즘이 항상 올바른 답을 계산함을 보이고 출력하는 단계

4. DP 알고리즘의 4 단계 정리:

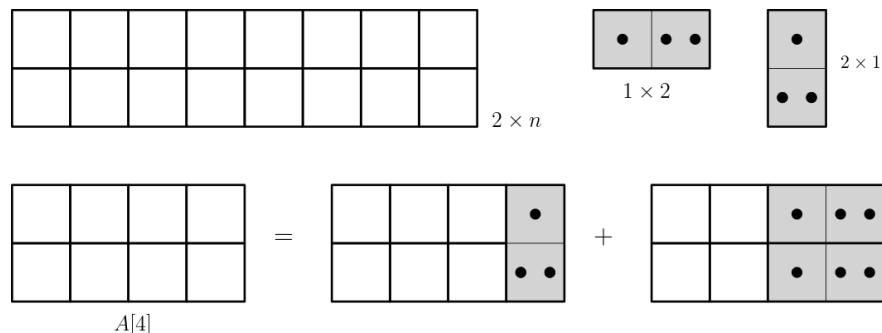
- 해답의 구조를 파악해 어떻게 부문제(subproblem)로 나눌 지 결정한다
- 부문제의 해답을 어떻게 조합해 더 큰 문제의 해답을 만들 수 있는지 식을 마련한다
 - 보통 점화식으로 표현된다
- 작은 문제에서 큰 문제로 적당한 순서에 따라 해답을 계산하여 테이블에 저장하고 재사용하여 답을 계산한다
- 위의 단계들이 원래 문제의 답을 항상 출력함을 증명하고 답을 리턴한다

5. 수행시간의 계산 요령

- 부문제의 갯수 \times 한 부문제를 푸는 데 걸리는 시간 + 기타 필요한 부가 시간

6. 예3: 도미노 타일링 (Tiling with Dominoes)

- 2×1 (또는 1×2) 도미노 여러개를 이용해 $2 \times n$ 타일을 빈틈 없이 메우는 경우의 수는?



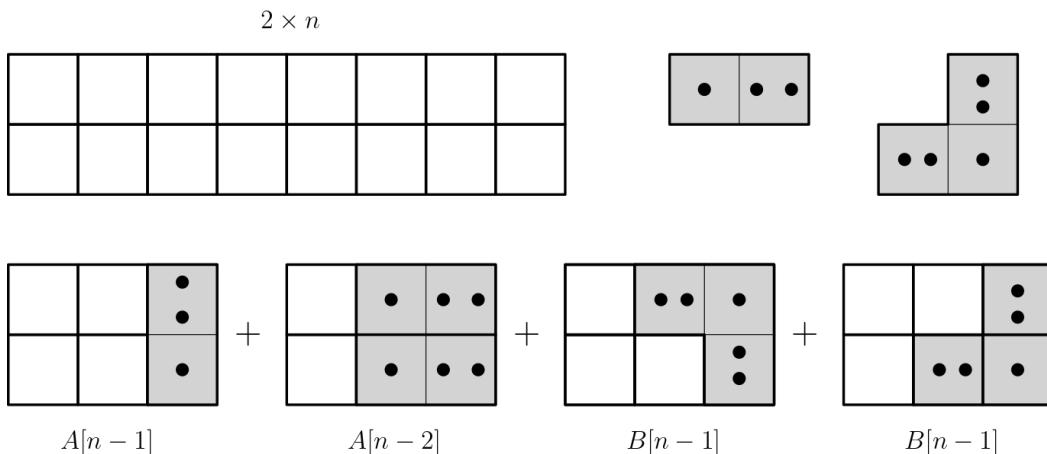
- $A[n] = 2 \times 1$ 도미노 여러개를 이용해 $2 \times n$ 타일을 빈틈 없이 메우는 경우의 수로 정의
- 위의 그림을 이용해, $A[n]$ 을 더 작은 문제의 타일링 경우의 수의 점화식 유도해보자!
 - 도미노를 두 부분으로 나눠 생각하면 쉽다. 가장 오른쪽 열 (column)을 채우는 경우와 나머지 부분을 채우는 경우를 나눈다. 나머지 부분을 채우는 경우의 수를 부문제의 답으로 표현해본다
 - 가장 오른쪽 열을 채우는 방법은 위의 그림에서 보듯 세로 방향으로 채우는 방법과 옆으로 두 개의 타일을 이용하는 방법 두 가지이다. (이 두 경우는 다르기 때문에 당연히 따로 세어야 한다.)
 - 남은 부분은 $2 \times (n-1)$ 도미노와 $2 \times (n-3)$ 도미노가 되고, 이 부분을 채우는 경우의 수는 당연히 부문제의 답이다. 즉, $A[n-1]$ 과 $A[n-2]$ 에 부문제의 답이 저장되어 있다
 - 따라서 $A[n] = (\text{오른쪽 열을 타일 하나로 채우는 경우}) + (\text{오른쪽 열을 타일 두개로 채우는 경우}) = A[n-1] + A[n-2]$
 - $A[1] = 2 \times 1$ 도미노를 채우는 경우 = 1

$A[2] = 2 \times 2$ 도미노를 채우는 경우 = 2

6. 결국, 점화식은 $A[n] = A[n-1] + A[n-2]$, $A[1] = 1$, $A[2] = 2$ 이 된다 → 여기서도 Fibonacci 수가 됨을 알 수 있다

- b. [한단계 더!] 모양이 다른 두 도미노를 여러 개를 이용해 $2 \times n$ 타일을 빈 틈 없이 메우는 경우의 수 $A[n]$ 은?

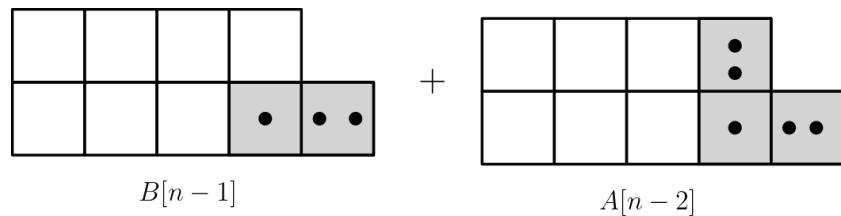
- i. 두 가지 모양의 도미노를 회전하여 사용할 수 있고, 충분히 많은 도미노가 제공된다고 가정하자



- ii. 위의 그림에서 2×3 타일을 메우는 방법은 총 5가지이다. 즉, $A[3]=5$. 이를 분석해 DP 점화식을 만들어보자. [힌트] DP 테이블이 2개가 필요!

- iii. 가장 오른쪽 열을 채우는 경우를 생각해보자

1. 세로 방향으로 긴 타일 1개를 채우는 경우, 나머지 부분을 채우는 경우의 수는 $\rightarrow A[n-1]$
2. 긴 타일 2개를 가로 방향으로 채우는 경우, 나머지 부분을 채우는 경우의 수는 $\rightarrow A[n-2]$
3. L 모양 타일 1개를 ↗자 모양으로 채우는 경우 \rightarrow 남은 부분이 첫 행은 $n-2$ 개의 빈 칸, 둘째 행은 $n-1$ 개의 빈 칸으로 구성 \rightarrow 이 남은 부분을 채우는 경우는 $A[k]$ 로 표현할 수 없다 \rightarrow 이런 모양에 대한 경우의 수를 저장하는 두 번째 DP 테이블 $B[n]$ 을 정의한다. $B[n]$ 은 한 행은 n 개의 칸, 다른 행은 $n-1$ 개의 칸으로 구성된 영역을 채우는 경우의 수가 저장된다 \rightarrow 그러면 ↗자 모양으로 채우는 경우는 $B[n-1]$ 이 된다 \rightarrow ↗자 모양을 90도 회전해서 채우는 경우도 있는데, 이 경우도 결국 $B[n-1]$ 이 된다
4. 따라서 $A[n] = A[n-1] + A[n-2] + 2*B[n-1]$ 의 점화식이 만들어진다
5. 주의할 점은 $B[n]$ 의 값도 계산해야 하기에 $B[n]$ 에 대한 점화식도 만들어야 한다



6. 위의 그림처럼 다시 가장 오른쪽 돌출된 한 칸을 채우는 (두 가지) 경우로 나눠 식으로 정의할 수 있다 $\rightarrow B[n] = B[n-1] + A[n-2]$

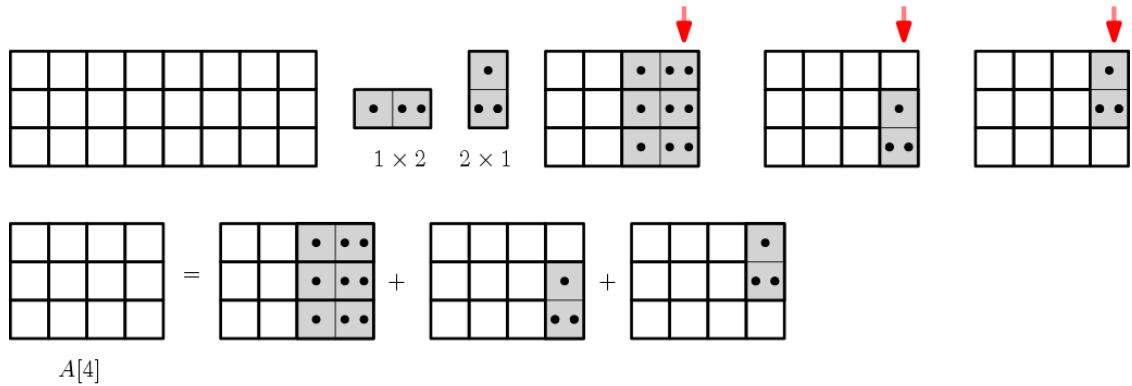
iv. 정리하면 두 종류의 DP 테이블 A, B에 대해, 다음의 점화식이 정의된다

$$1. A[n] = A[n-1] + A[n-2] + 2*B[n-1]$$

$$2. B[n] = B[n-1] + A[n-2]$$

3. 초기 값 $A[1], A[2], B[1], B[2]$ 는 각자 노트에 그려보면 계산해보자

c. [다시 한단계 더!] 2×1 도미노 여러개를 이용해 $3 \times n$ 타일을 빈 틈 없이 메우는 경우의 수는?



- i. 여기서는 테이블이 두 개 또는 그 이상 등장함에 유의하자! $\rightarrow A, B, C (?)$

7. 예4: 최대 합 계산하기

설명 동영상 3편:



1/3



2/3



3/3

- a. **문제:** 정수 n개의 수열이 $A[0], \dots, A[n-1]$ 처럼 주어질 때, 이 수열에서 $A[i] + A[i+1] + \dots + A[j]$ 의 합이 최대가 되는 인덱스 i, j ($i \leq j$)를 계산해 값 출력
 - i. $A = [1, -1, 3, -4, 5, -4, 6, -2]$ 라면, $5+(-4)+6 = 7$ 이 되어, $A[4]+A[5]+A[6]$ 이 최대합이 된다
- b. 이 문제는 다양한 알고리즘으로 풀린다. (1) prefix 합을 이용한 $O(n^2)$ 시간 알고리즘, (2) 분할정복법을 이용한 $O(n\log n)$ 알고리즘 (3) DP를 이용한 $O(n)$ 시간 알고리즘이 가능하다
- c. 4단계 분석:

i. 부문제(subproblem)로 분할하기:

1. 최대합이 $A[i] + \dots + A[j]$ 이라고 할 때, $A[i] + \dots + A[j-1]$ 은 $A[j-1]$ 로 끝나는 구간 중에 합이 최대인 구간이 되어야 한다 (**중요한 관찰**)
 - a. 왜?
 - b. 합이 최대가 아니라면, 최대인 구간에 $A[j]$ 를 더하면 $A[i] + \dots + A[j]$ 보다 더 큰 합을 만들 수 있기 때문에
2. 그런데 최대합이 $A[j]$ 단 하나의 숫자로만 구성될 수도 있다는 사실도 반영해야 한다
3. 결국, $A[j]$ 로 끝나는 구간 중에 최대합은 $A[j]$ 자체일 수도, $A[j-1]$ 로 끝나는 최대 구간 합에 $A[j]$ 를 더 한 것일 수도 있다. 즉, 두 가지 경우 중 하나이다
4. 이 두 가지 중 어느 것인지 모르니 둘 모두 계산해서 그 중 최대값을 취하면 된다
5. 그런데 최대합이 어떤 $A[j]$ 에서 끝날지 모르기 때문에, 모든 $j = 0, \dots, n-1$ 까지 모두 계산해보고 그 중 최대합을 출력하면 된다
6. 사실, DP는 (원칙적으로) 모든 경우를 고려하는 것이다. 대신 모든 경우를 부문제들의 해의 수식으로 표현하고, 부문제 해를 기록한 후 필요할 때 재사용하여 빠르게 계산하는 것이 다른 점이다

ii. 부분문제의 답을 큰 문제의 답으로 표현하기:

1. $A[0], \dots, A[i]$ 에 대해서, $A[i]$ 로 끝나는 최대합을 $S[i]$ 라 하면, $S[i]$ 는 $S[i-1] + A[i]$ 또는 $A[i]$ 둘 중 큰 값이 된다

$$S[i] = \max(S[i-1]+A[i], A[i])$$

iii. 적당한 순서로 테이블 채우기:

1. 여기서는 S 를 리스트로 표현하고, $S[i]$ 를 계산하기 위해선 $S[i-1]$ 을 알아야 하므로, i 가 $0, 1, \dots, n-1$ 까지 커지는 순서로 계산한다
2. 단, 리스트의 처음 값 $S[0]$ 를 미리 정의해야 하는데, 이 의미는 $A[0]$ 에 대해서 최대합을 구하는 것이므로 $S[0] = A[0]$ 로 정하면 된다

iv. 항상 옳은 답을 내는지 확인하고 답 출력하기:

1. 항상 옳은 답을 구하는지는 거의 당연 ^^
2. 답은 S 의 가장 큰 값이므로 최대값을 리턴한다

```
def find_max_sum(A):
    S = [0] * len(A)
    S[0] = A[0]
    for i in range(1, n):
        S[i] = max(S[i-1]+A[i], A[i])
    return max(S)
```

- v. 수행시간 = S 의 엔트리 수(원소 개수) \times 엔트리 하나 채우는 데 필요 시간 = $O(n) \times O(1) = O(n)$

8. 예5: zig-zag 수열 계산하기

- a. 문제: 정수 n 개의 수열이 $A[0], \dots, A[n-1]$ 처럼 주어질 때, 이 수열에서 증가와 감소가 교대로 나타나는 zig-zag 수열 중 가장 긴 부분수열을 찾아, 그 길이를 출력하라



- i. 1, 7, 4, 9, 2, 5는 1에서 7로 증가, 7에서 4로 감소, 4에서 9로 증가 … 증가와 감소가 교대로 나타나는 zig-zag 수열이다
- ii. 1, 7, 9, 4, 5, 2는 처음에 두 번 증가를 하기 때문에 zig-zag 수열이 아니다
- iii. 4는 숫자 하나로 구성된 zig-zag 수열이다
- iv. 1, 7, 9, 4, 5, 2는 zig-zag 수열이 아니지만, 1, 7, 4, 5, 2처럼 부수열 중에 zig-zag 수열이 있다. 가장 긴 zig-zag 부수열은 1, 7, 4, 5, 2 또는 1, 9, 4, 5, 2로 길이가 5이다

- b. 지겹지만, 다시 4단계 분석:

i. 부문제로 분할하기:

1. zig-zag 수열을 생각해보자. 이 수열의 가장 마지막 수를 $A[k]$ 라 하고 그 전 수를 $A[j]$ 라 표기하자 (당연히 $j < k$ 이다)
2. $A[k]$ 는 zig-zag 수열이기 때문에 $A[j] < A[k]$ 일 수도, $A[j] > A[k]$ 일 수도 있다
 - a. $A[j] < A[k]$ 이라면, $A[k]$ 를 high라고 부르고, 아니면 low라고 부른다
 - b. 결국 zig-zag 수열은 … high-low-high 식이든지, … low-high-low 식으로 끝나야 한다

3. $A[k]$ 가 **high**로 끝나는 경우와 **low**로 끝나는 경우, 두 가지를 모두 기억해야 한다. [중요!] 왜? 가장 긴 zig-zag 수열 (답)이 **high**로 끝날지, **low**로 끝날지 모르기 때문에 두 가지 경우를 모두 구해서 그 중 긴 수열로 답을 출력해야 하기 때문이다
 4. $A[k]$ 가 **high**로 끝나는 경우: $A[j]$ 는 (zig-zag 수열이므로) 당연히 **low**여야 한다. 따라서 $A[j]$ 가 **low**로 끝나는 가장 긴 zig-zag 수열에 $A[k]$ 를 추가하면 되므로 길이가 1 증가한다
 5. $A[k]$ 가 **low**로 끝나는 경우: $A[j]$ 는 (zig-zag 수열이므로) 당연히 **high**여야 한다. 따라서 $A[j]$ 가 **high**로 끝나는 가장 긴 zig-zag 수열에 $A[k]$ 를 추가하면 되므로 길이가 1 증가한다
 6. **부분문제**: $A[k]$ 가 **high**로 끝날 때의 가장 긴 zig-zag 수열의 길이와 **low**로 끝날 때 가장 긴 zig-zag 수열의 길이를 계산하는 문제
- ii. 부문제의 답으로 큰 문제의 답 표현하기:
1. $A[k]$ 가 **high**로 끝날 때의 가장 긴 zig-zag 수열의 길이를 **high[k]**에 저장하고, $A[k]$ 가 **low**로 끝날 때의 가장 긴 zig-zag 수열의 길이를 **low[k]**에 저장하자 (즉, 두 리스트 **high**와 **low**를 준비함)
 2. $A[k]$ 가 **high**인 경우에 $A[j]$ 는 $A[0], \dots, A[k-1]$ 중에서 $A[k]$ 보다 작은 어떤 수도 후보가 될 수 있다. 그 수는 반드시 **low**로 끝나야 하므로 $low[j]+1$ 을 계산해서 이 값이 가장 큰 $A[j]$ 를 선택해야 한다 ($A[k]$ 가 **low**인 경우는 반대로 생각하면 됨)
 3. 즉, $high[k] = \max(low[j]+1) \text{ for } j \text{ in range}(k) \text{ if } A[j] < A[k]$
 $low[k] = \max(high[j]+1) \text{ for } j \text{ in range}(k) \text{ if } A[j] > A[k]$
- iii. 적당한 순서로 테이블을 채우기:
1. $high[k]$ 와 $low[k]$ 는 모두 k 보다 작은 인덱스 j 에 대한 $low[j]$ 값과 $high[j]$ 값을 필요로 하므로 j 를 $0, \dots, k-1$ 까지 차례로 증가시키면서 값을 채우면 된다
- iv. 항상 옳은 답을 내는지 확인하고 답 출력하기:
1. 답이 옳다는 건 당연하므로 생략
 2. 답은 $\max(\max(high[k], low[k])) \text{ for } k \text{ in range}(n)$ 이 된다 ([?] 왜?)
- v. Pseudo code: 스스로 구현해볼 것! (위의 언급한 내용을 차례대로 나열하면 끝!)
- vi. 알고리즘의 수행시간 = **high**, **low**의 엔트리 수 \times 엔트리 하나 계산 시간 = $O(n) \times O(n) = O(n^2)$

c. [고급 - skip 가능] 더 빠르게 할 순 없을까? 놀랍게도 $O(n \log n)$ 시간에 가능하다!

- i. 앞의 DP 식은 k 번째 $A[k]$ 값으로 끝나는 zig-zag 수열의 가장 긴 길이를 두 DP 테이블에 저장했다. 이제 완전히 다른 방식으로 DP 식을 정의해보자
- ii. 현재 입력 값들에 대해 길이가 k ($k = 2, 3, \dots$)인 zig-zag 수열은 역시 두 종류이다. low로 끝나거나 high로 끝나거나
- iii. DP 테이블 $low[k]$ 에는 low로 끝나는 길이가 k 인 zig-zag 수열 중에서 수열의 가장 작은 마지막 값의 인덱스를 저장한다고 정의하자 (정의가 복잡해 보이니 여러 번 읽어보고 아래 설명을 따라가 보자)
 1. 예: $A = [1, 0, 5, 8, 3, 5, 6, 4, \dots]$ 이라고 하고 앞의 8개의 값만을 고려할 때, $k = 4$ 라고 하면, 길이가 4이면서 low로 끝나는 수열은 $(1, 0, 5, 3)$, $(1, 0, 5, 4)$, $(1, 0, 8, 3)$, $(1, 0, 8, 5)$, $(1, 0, 8, 6)$, $(1, 0, 8, 4)$, $(1, 0, 6, 4)$ 등이 존재한다. 그러면 길이가 5인 수열을 위해 이 길이가 4인 수열을 모두 기록해 놓아야 할까?
 2. 이 수열 뒤에 새로운 high 값을 붙여 길이가 5인 수열을 만든다면, 어떤 수열이 가장 유리할까? 당연히 가능하면 작은 값으로 끝나는 수열이 유리하다. 위의 예에서는 3으로 끝나는 길이 4인 수열이 더 긴 수열을 만들기에 가장 유리하다. 그래서 이 수열만 저장해 놓으면 어떨까? 즉, $low[k]$ 에 길이가 k 인 low로 끝나는 수열 중에서 가장 작은 마지막 값의 인덱스를 저장한다
- iv. 이제, high와 low를 업데이트하는 DP 식을 만들어보자. $low[k]$ 의 DP 식은 길이가 $k-1$ 인 high로 끝나는 값보다 작은 값을 찾아 붙이면 된다. 문제는 그런 값이 여러 개라면 그 중 가장 작은 값을 찾아야 한다. 어떻게 찾을 수 있을까?
 1. $j = high[k-1]+1, \dots, n-1$ 까지 보면서 $A[j] < A[high[k-1]]$ 인 $A[j]$ 중에서 가장 작은 값을 찾아야 한다. 그런데 이런 방식은 $O(n)$ 시간이 걸리기 때문에 전체적으로 $O(n^2)$ 시간을 피할 수 없다. (단순한 $O(n^2)$ 시간 알고리즘과 시간에서 차이가 없다)
- v. 다른 방식으로 접근해보자. $A[0], A[1], \dots, A[n-1]$ 를 차례대로 하나씩 보면서 두 리스트 high와 low를 업데이트해보자. 현재 $A[i]$ 를 고려하는 단계라고 하면, $A[0], \dots, A[i-1]$ 에 대한 high, low 리스트가 이미 계산되어 있다. 여기에 $A[i]$ 가 high로 끝나는 경우와 low로 끝나는 경우를 각각 계산해 리스트 high, low의 값을 업데이트하는 식이다
 1. 리스트 low를 업데이트 하는 경우를 고려해보자. $A[i]$ 가 $A[high[1]]$ 뒤에 붙어 길이가 2인 low로 끝나는 zig-zag 수열이 되거나 $A[high[2]]$ 뒤에 붙어 길이가 3인 low로 끝나는 수열이 되거나 등등. 단, $A[i]$ 가 low로 끝나야 하기에 $A[high[1]]$ 이나 $A[high[2]]$ 보다는 작아야 자격이 된다. 여기서 중요한 성질은 $A[high[1]] < A[high[2]] < \dots < A[high[k]] < \dots$ 처럼 증가한다는 것이다. 왜 그럴까?

2. 증가 성질 때문에, 우리는 $A[\text{high}[k-1]] < A[i] < A[\text{high}[k]]$ 를 만족하는 인덱스 k 를 **이진탐색!**으로 찾을 수 있다. 그러면 $A[i]$ 는 $A[\text{high}[k]]$ 뒤에 붙어 길이가 $k+1$ 인 low 로 끝나는 zig-zag 수열을 이룰 수 있다. 당연히 $\text{low}[k+1]$ 의 값(인덱스)이 변경될 수 있다. 즉, $\text{low}[k+1] = i$ 로 조정한다
- vi. 각 $A[i]$ 에 대해, 이진탐색으로 자신이 끝에 붙어 수열의 길이를 증가시키는 경우를 계산해 $O(\log n)$ 시간에 high 와 low 리스트를 업데이트할 수 있다. 따라서 전체 수행시간은 $O(n \log n)$ 이다. 이와 유사한 방식으로 LIS (Longest Increasing Subsequence) 문제를 해결할 수 있다. (이 문제는 예제 9에서 자세히 설명)
 - vii. 이 DP가 다른 DP와 구별되는 핵심 3가지: (1) DP 테이블 high 와 low 를 수열의 길이에 대해 정의한 것. (2) $A[\text{high}[i]]$ 값과 $A[\text{low}[j]]$ 값이 각각 i 와 j 에 대해 증가하는 것. (3) (2)번 사실을 이용해 $A[i]$ 에 대해 이진탐색으로 $A[i]$ 가 연결될 zig-zag 수열의 위치를 찾는 것

9. 예6: LCS(Longest Common Subsequence) 계산하기

a. 대표적인 DP 문제임: Wagner, Fischer, 1974년 논문에 등장

b. 문제: 두 문자열 X, Y의 가장 긴 공통 부문자열을 찾는 문제

- i. $X = ABCBDAB$
- ii. $Y = BDCABA$
- iii. 부문자열(subsequence)은 문자열에서 몇 개를 지우고 남은 문자의 열을 의미한다
- iv. X의 부문자열은 ACB, ABDA, CBDAB 등 매우 많다. BCAD는 부문자열이 아니다.
- v. X, Y의 공통 부문자열은 두 문자열에 모두 포함된 부문자열을 의미한다
- vi. ABA, BCBA 등이 공통 부문자열이다
- vii. 공통 부문자열에서 가장 긴 길이의 부문자열을 찾는 문제이다
- viii. 이 예제에서는 LCS가 무엇인가? 하나 이상일 수도 있다
 $BCAB, BDAB, BCBA, BCAB$ (총 4가지로 길이가 4이다)



c. 어떤 경우에 유용한가?

- i. DNA 시퀀스 $S_1 = ACCGGTCGAGTGCGCGAAGCCGGCCGAA$ 과 $S_2 = GTCGTTGGAATGCCGTTGCTCTGTAAA$ 이 서로 얼마나 가까운 유전자인지를, LCS의 길이를 구해 예측할 수 있다. LCS의 길이가 클수록 유전적으로 가깝다고 추측할 수 있다
- ii. 이 경우 $LCS = GTCGTCGGAAGCCGGCCGAA$ 이 된다

d. 기본 표기

- i. 두 입력 문자열: $X = x_1 \dots x_n, Y = y_1 \dots y_m$
- ii. 프리픽스(prefix) $X_i = x_1 \dots x_i, Y_j = y_1 \dots y_j$
- iii. $LCS(i, j) = X_i$ 와 Y_j 의 LCS
- iv. $\text{len}(i, j) = |LCS(i, j)| = X_i$ 와 Y_j 의 LCS 길이
 - 1. 최종적으로 구하고 싶은 것은 $LCS(n, m)$ 이고, 길이 $\text{len}(n, m)$

e. 4단계 분석

- i. 부문제로 분할하기
 - 1. $LCS(i, j)$ 를 계산하는 문제가 자연스럽게 부문제가 됨
- ii. 부문제의 답으로 큰 문제의 답 표현하기:
 - 1. 핵심은 $X_i = x_1 \dots x_i, Y_j = y_1 \dots y_j$ 의 마지막 두 문자이다.
 - 2. $x_i == y_j$ 라면, 이 문자는 LCS에 포함될까? (포함하는 게 절대 손해는 아니다!)
 - a. 왜? 포함하지 않고 더 긴 LCS를 만들 수 있다고 가정해보자. 그러면 $\text{len}(i, j) = \max(\text{len}(i-1, j), \text{len}(i, j-1))$ 일 것이다. 포함하는 경우에는 $\text{len}(i, j) = \text{len}(i-1, j-1) + 1$ 이 된다. 그러면, $\max(\text{len}(i-1, j), \text{len}(i, j-1)) > \text{len}(i-1, j-1) + 1$ 이 성립해야 한다. 그러나 이 부등식은 성립하지 않는다. 이유를 생각해보자. 따라서 포함하는 게 절대 손해는 아니다!
 - b. 따라서, $LCS(i, j) = LCS(i, j) + x_i$ (or y_j)이고 $\text{len}(i, j) = \text{len}(i-1, j-1) + 1$ 이 된다

3. $x_i \neq y_j$ 라면, 둘 중 최소 하나는 LCS에 포함될 수 없다. 그런데 어느 문자가 포함되지 않는지 모르겠지만, 포함되지 않았을 때, 더 긴 LCS를 주는 문자를 골라내면 된다
- x_i 가 포함되지 않는 경우라면, $\text{LCS}(i, j) = \text{LCS}(i-1, j)$ 가 됨(왜?) 따라서 $\text{len}(i, j) = \text{len}(i-1, j)$ 임
 - y_j 가 포함되지 않는 경우라면, $\text{LCS}(i, j) = \text{LCS}(i, j-1)$ 가 됨(왜?) 따라서 $\text{len}(i, j) = \text{len}(i, j-1)$ 임
 - 위의 두 경우 중 더 긴 값을 택해야 하므로:
 $\text{len}(i, j) = \max(\text{len}(i-1, j), \text{len}(i, j-1))$

iii. 적당한 순서로 테이블을 채우기:

- len 이라는 이차원 리스트를 준비한다
- $\text{len}[i][j]$ 를 계산하기 위해선, $x_i == y_j$ 와 $x_i \neq y_j$ 결과에 따라 $\text{len}[i-1][j]$, $\text{len}[i][j-1]$, $\text{len}[i-1][j-1]$ 이 필요하다. 따라서, i , j 가 점점 커지는 방향으로 이차원 리스트를 채우면 된다

iv. 예: $X = ABCBDAB$, $Y = BDCABA$

		<i>j</i>	0	1	2	3	4	5	6
		<i>i</i>	B	D	C	A	B	A	
<i>i</i>	<i>j</i>	0	0	0	0	0	0	0	0
		1	A	0					
2	B	0							
3	C	0							
4	B	0							
5	D	0							
6	A	0							
7	B	0							

v. Pseudo code

def $\text{LCS}(X, Y)$:

```

n, m = |X|, |Y|
# prepare 2d list len and initialization
for j in range(0, m+1): len[0][j] = 0
for i in range(0, n+1): len[i][0] = 0
# filling DP table len
for i in range(1, n+1):
    for j in range(1, m+1):
        if X[i] == Y[j]: # 마지막 두 글자가 같으면
            len[i][j] = len[i-1][j-1] + 1
        else: # 마지막 글자가 다르면
            len[i][j] = max(len[i-1][j], len[i][j-1])
return len[n][m]

```

- vi. 위 테이블을 채우면 $\text{len}[7][6] = \text{LCS}$ 의 길이인 4가 저장되어야 한다
- vii. 위의 len 테이블을 채우는 데 걸리는 시간은?
 - 1. 테이블의 엔트리 갯수 \times 하나의 엔트리를 계산하는 데 걸리는 시간
 $= O(n^2) \times O(1) = O(n^2)$
- viii. 위의 len 테이블은 LCS 길이만을 저장한다. LCS 자체를 알고 싶다면, 추가적으로 무엇을 더 해야 할까?
 - 1. 마지막 문자가 같은 경우에는 LCS에 포함하기 때문에, len 을 모두 채운 후, $\text{len}[n][m]$ 부터 출발해서 LCS의 끝 문자부터 재구성하는 방법을 생각한다
 - 2. $X[n] == Y[m]$ 이었다면, 이 문자가 LCS의 가장 마지막 문자로 선택되었다는 의미이므로 LCS 마지막 문자를 바로 알 수 있다. LCS의 나머지 길이는 $\text{len}[n-1][m-1]$ 에 있으므로 $X[n-1] == Y[m-1]$ 인지 아닌지를 검사하는 과정을 반복하면 된다
 - 3. $X[n] != Y[m]$ 이라면, $\text{len}[n][m] = \max(\text{len}[n-1][m] + 1, \text{len}[n][m-1])$ 로 계산되므로 두 값 중에서 최대값을 주는 칸으로 이동해 반복하면 된다. 즉, 최대 값이 $\text{len}[n-1][m]$ 이라면 $X[n-1] == Y[m]$ 인지 아닌지 검사하면 된다
 - 4. 이 검사를 0번 행 또는 0번 열에 도착할 때까지 반복하면 된다
 - 5. 따라서 LCS는 len 테이블이 채워지면 $O(n)$ 시간에 재구성 가능하다

10. 예7: 편집거리(Edit Distance) (inspired by LCS problem)

- a. 문자열 X 의 어떤 문자를 지우거나(delete) 새로운 문자를 삽입해서(insert) 다른 문자열 Y 를 만들 수 있다
- b. 예를 들어, $X = "HUMAN"$ 에서 $Y = "CHIMPANZEE"$ 를 만들기 위해서,
 - i. HUMAN → HMAN → CHMAN → CHIMAN → CHIMPAN → CHIMPANZ → CHIMPANZE → CHIMPANZEE
- c. 편집거리 문제는 최소 횟수의 삽입 또는 삭제 연산으로 X 를 Y 로 변경할 수 있는지를 계산하는 문제이다
 - i. 위의 예에서는 1번의 삭제와 6번의 삽입으로 총 7번의 연산이 최소이다
- d. [?] 최소 횟수는 $\text{LCS}(X, Y)$ 와 매우 깊은 관련이 있다 (어떻게?)
 - i. X 와 Y 에 공통으로 들어 있는 문자는 그대로 두는 것이 연산을 최소화할 수 있다는 사실로부터 $\text{LCS}(X, Y)$ 에 해당하는 문자만 남기고 X 에는 있지만 Y 에 없는 문자는 삭제하고, X 는 없지만 Y 에 있는 문자는 삽입하는 연산을 하면 된다
 - ii. 위의 예에서는 $\text{LCS}(\text{HUMAN}, \text{CHIMPANZEE}) = \text{HMAN}$ 이므로 이 네 문자만 남기고, HUMAN의 U는 X에만 있는 문자이므로 삭제하고, C I P Z E는 Y에만 있는 문자이므로 맞는 위치에 삽입을 하면 된다. 즉, 1번의 삭제와 6번의 삽입 연산이 필요하다
- e. 만약, 삽입(insert), 삭제(delete)와 함께 교체(substitute) 연산도 할 수 있다고 하자. 연산의 종류가 하나 더 추가되었다. 그리고 각 연산마다 비용이 주어진다면 어떻게 DP로 풀 수 있을까?

- i. $w_{ins}(z)$ = 문자 z 를 X 에 삽입할 때의 비용
 - ii. $w_{del}(z)$ = 문자 z 를 X 에서 삭제할 때의 비용
 - iii. $w_{sub}(z, z')$ = X 에서 문자 z 를 z' 으로 교체할 때의 비용
 - iv. 이 세 비용은 응용에 따라 다르게 주어지고, 교체 비용이 다른 두 비용보다 큰 게 자연스럽겠죠?
- f. $D[i][j]$ 를 X_i 를 Y_j 로 변환하는 데 필요한 최소 비용으로 정의한다. 여기서 X_i 는 X 의 첫 i 개의 문자로 구성된 prefix 문자열, Y_j 는 Y 의 첫 j 개의 문자로 구성된 prefix 문자열이다
- i. 초기 조건은 무엇일까?
 - ii. $D[i][0] = X_i$ 를 Y_j ($= \emptyset$)로 변환하는 것이므로 X_i 의 각 문자를 삭제하는 비용의 합
 - iii. $D[0][j] =$ 빈 문자열 X_i 에서 삽입을 해 Y_j 로 변환해야 하므로 삽입 비용의 합
 - iv. 이제 $D[i, j]$ 에 대한 일반 점화식을 LCS와 유사하게 만들어보자!
- $D[i][j] =$ 두 문자열의 마지막 문자 x_i 와 y_j 의 관계를 고려
- $$= (X_{i-1} \rightarrow Y_j) \text{ 경우}, (X_i \rightarrow Y_{j-1}) \text{ 경우}, (X_{i-1} \rightarrow Y_{j-1}) \text{ 경우 중 적은 비용}$$
- $$= x_i \text{의 삭제 비용}, y_j \text{의 삽입 비용}, x_i \text{를 } y_j \text{로 교체하는 비용을 각각 고려}$$
- $$= \min(D[i-1][j] + w_{del}(x_i), D[i][j-1] + w_{ins}(y_j), D[i-1][j-1] + w_{sub}(x_i, y_j))$$
- g. 이 점화식을 이용해 D 를 채우는데 필요한 시간은? _____

11. 예8: 괄호 만들기

- a. 설명 동영상 2편 (\rightarrow)
 - b. 행렬 곱셈을 할 때 어떤 순서로 하느냐에 따라 기본 연산의 횟수가 크게 달라진다
 - i. 예를 들어, $a \times b$ 행렬과 $b \times c$ 행렬의 곱셈을 위해 필요한 기본 연산(두 수의 덧셈과 곱셈 횟수)은 $O(abc)$ 정도이다.
 - ii. 만약, 2×3 행렬 A, 3×2 행렬 B, 2×5 행렬 C가 있다면, 곱 ABC를 실제 시행하는 순서는 $((AB)C)$ 와 $(A(BC))$ 두 가지가 존재한다. 두 가지 모두 결과는 같다
 - iii. $((AB)C)$ 경우: AB를 위해 $2 \times 3 \times 2 = 12$ 정도의 비용이 들고 결과 행렬이 2×2 이므로 C와 곱셈을 위해 $2 \times 2 \times 5 = 20$ 정도의 비용이 된다. 따라서 총 32 정도의 비용이 필요하다
 - iv. $(A(BC))$ 경우: BC를 먼저 하기 때문에 $3 \times 2 \times 5 = 30$ 비용이 들고, 결과 행렬이 3×5 이므로 A와 곱셈을 하면 $2 \times 3 \times 5 = 30$ 의 비용이 필요해, 총 60 정도의 비용이 필요하다
 - v. 두 경우에 대해 필요한 비용이 거의 두 배 정도 차이가 난다. 결국, 어떤 곱셈 순서를 따라 행렬 곱을 해야지 총 비용을 최소로 할 수 있는지를 아는 게 핵심이다!
- 
- 

c. 행렬 $M[1], M[2], \dots, M[n]$ 이 주어진다. $M[k] = p_k \times p_{k+1}$ 행렬이라고 가정하자

i. $n = 3$ 인 경우

1. $(M[1]M[2])M[3]$ vs. $M[1](M[2]M[3])$ 두 개의 괄호치기 존재

ii. $n = 4$ 인 경우

1. $((ab)c)d, (a(bc))d, (ab)(cd), a((bc)d), a(b(cd))$ 로 5가지

2. 이건 세 쌍의 왼쪽, 오른쪽 괄호 쌍을 배열하는 것과 같다:

$((())), ()(), ()(), ()(), ()()$

iii. 일반적인 n 인 경우에는 $(n-1)$ 개의 괄호 쌍을 배열하는 경우의 수와 같다

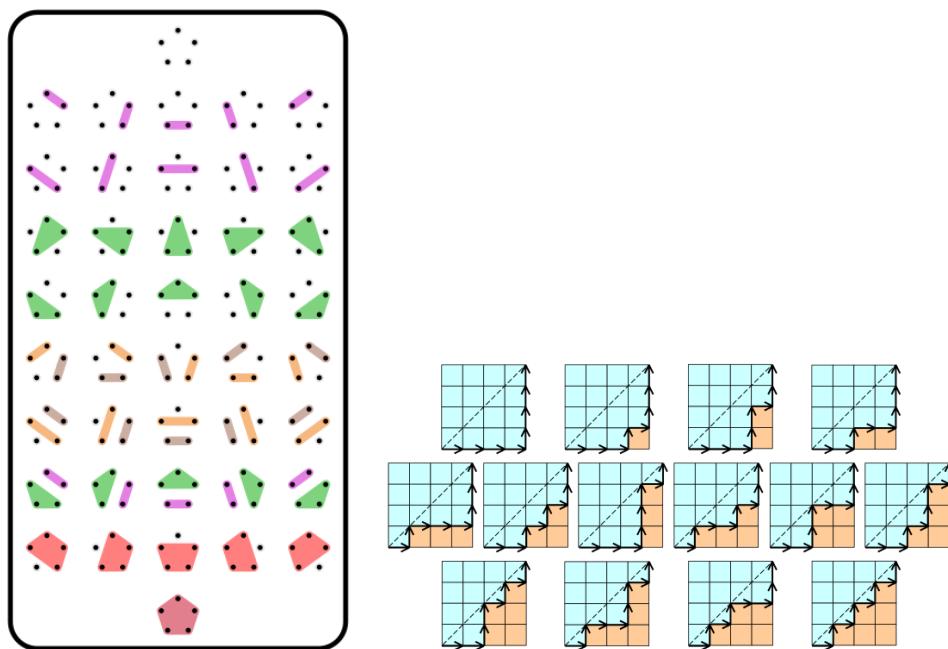
1. 이미지 출처: https://en.wikipedia.org/wiki/Catalan_number

2. 이 경의 수를 Catalan 수라 부른다. n 번째 Catalan 수 C_n 은 거의 4^n 으로
지수식이 되어 매우 큰 수이다

$$C_n \sim \frac{4^n}{n^{3/2} \sqrt{\pi}},$$

Catalan 수열을 보면 아래와 같다:

1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, ...



iv. 가장 쉬운 방법은 C_{n-1} 개의 괄호 치기 경우 각각에 대해 비용(기본 연산의 횟수)을 계산하여 가장작은 비용을 갖는 괄호 치기를 선택하는 것이다

1. 이 방법은 C_{n-1} 개 경우 \times 각 경우의 비용 계산 시간 = $O(nC_{n-1})$

2. 그런데 $C_{n-1} = O(4^n)$ 이 너무 커, 수행 시간이 의미가 없다

v. 훨씬 빠른 다항시간 DP 알고리즘을 설계해보자

d. DP 알고리즘: 4 단계 분석

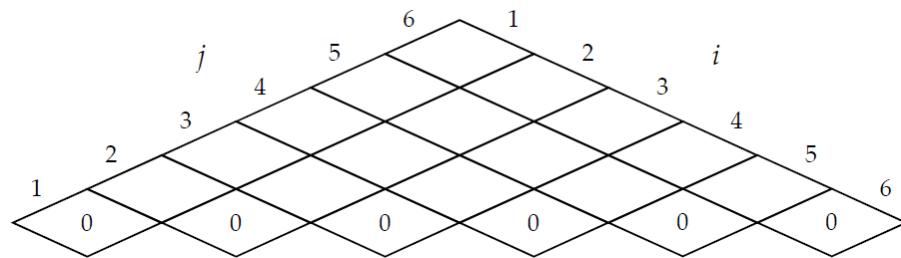
- i. **부문제:** 모든 $i \leq j$ 에 대해, $M[i] \dots M[j]$ 에 대한 괄호치기의 최소비용을 구하는 문제
 1. 이 최소 비용은 DP 테이블 $DP[i][j]$ 에 저장된다고 하자
 2. 그러면 우리가 최종적으로 원하는 건 $DP[1][n]$ 의 값이 된다
- ii. **관계식:**
 1. $(M[i] \dots M[k]) (M[k+1] \dots M[j]) \leftarrow$ 이 곱셈이 최소 비용 곱셈 순서 중에서 가장 마지막 곱셈이라고 가정해보자
 2. 여기서 인덱스 k 가 무엇인지 모른다. 이 k 값을 알아내는 게 목표이다
 - a. k 값은 $i, i+1, \dots, j-1$ 중에서 어떤 값도 가능하기에, 이 값을 모두 가정해보고 비용을 계산해 본 후 최소 비용을 주는 k 값을 선택하면 된다
 3. 특정 k 값에 대한 비용은 세 가지이다
 - a. $M[i] \dots M[k]$ 의 최소 행렬 곱셈 비용 $\rightarrow DP[i][k]$
 - b. $M[k+1] \dots M[j]$ 의 최소 행렬 곱셈 비용 $\rightarrow DP[k+1][j]$
 - c. 각각 계산된 두 행렬 (첫 번째 행렬은 $p_i \times p_{k+1}$ 행렬이 되고 두 번째 행렬은 $p_{k+1} \times p_{j+1}$ 행렬이 됨)의 곱셈 비용 $\rightarrow p_i p_{k+1} p_{j+1}$
 - d. 따라서 비용은 세 비용의 합인 $DP[i][k] + DP[k+1][j] + p_i p_{k+1} p_{j+1}$ 이 된다
 4. 그러면 $DP[i][j]$ 는 $k = i, i+1, \dots, j-1$ 에서의 비용 중에 최소 값으로 정의된다. 즉,

$$DP[i][j] = \min_{k=i, \dots, j-1} (DP[i][k] + DP[k+1][j] + p_i p_{k+1} p_{j+1})$$

5. $DP[i][j]$ 계산 시간은 당연히 $O(n)$ 을 넘지 않는다. 즉, DP 테이블의 한 엔트리 계산 시간이 $O(n)$ 이라는 뜻이다

iii. 테이블 계산 순서:

1. $DP[i][j]$ 를 위해선 $i \leq k < j$ 의 모든 k 에 대해서, $DP[i][k]$, $DP[k+1][j]$ 필요하므로 먼저 계산이 되기만 하면 된다
2. $DP[1][1], DP[2][2], \dots, DP[n][n] = 0$ 임에 주의! ($DP[i][i]$ 는 행렬 M_i 를 곱셈하는 데 필요한 최소 비용이라는 뜻인데, 행렬이 하나 뿐이므로 곱셈이 정의되지 않아 비용은 0이 된다)
3. 예: (아래 그림에서 i, j 의 증가 순서를 다르게 표시했음에 유의)



$$M_1 = 2 \times 5, M_2 = 5 \times 3,$$

$$M_3 = 3 \times 5, M_4 = 5 \times 10,$$

$$M_5 = 10 \times 2, M_6 = 2 \times 4,$$

iv. Pseudo code: 스스로 해볼 것!

v. 수행시간 = 테이블 엔트리 수 \times 엔트리 계산 시간 = $O(n^2) \times O(n) = O(n^3)$

12. 예 9: 최장 증가 부수열 문제 (LIS: Longest Increasing Subsequence)

- a. 예를 들어, $A = [2, 8, 5, 10, 18, 13, 20, 4]$ 인 경우에 가장 긴 증가 부수열을 찾는 문제로 2, 5, 10, 18, 20와 8, 10, 13 모두 증가 부수열이지만, 앞의 부수열의 길이가 더 길며 모든 증가 부수열 중에서 제일 길다
- b. 예 5 최장 zig-zag 수열 찾기 문제에서는 증가-감소 또는 감소-증가가 반복되는 가장 긴 부수열을 찾는 문제와 유사하다. 여기서는 계속 증가하는 가장 긴 수열을 찾는 것으로 오히려 더 단순하다고 볼 수 있다
- c. 두 가지 DP 알고리즘으로 해결되는 매우 좋은 연습 문제이다
- d. DP 알고리즘1: $O(n^2)$ 시간 알고리즘
 - i. $LIS[i] = A[0] \dots A[i]$ 부수열에 대해, $A[i]$ 로 끝나는 증가 부수열 중에서 가장 긴 부문자열의 길이라고 정의해보자
 - ii. 정답 LIS는 $A[0], A[1], \dots, A[n-1]$ 중 하나의 값으로 반드시 끝나므로 LIS 길이는 $\max_i(LIS[i])$ 가 됨을 알 수 있다.
 - iii. $LIS[i]$ 를 분해해 보자. (**hint**: zig-zag 수열을 구하는 경우와 유사!)
 - 1. $A[j] < A[i]$ 인 모든 j 에 대해, $LIS[i] = \max_j(LIS[j] + 1)$
 - iv. Pseudo code:


```
LIS[0] = 1 # A[0] 자체가 길이 1인 LIS
for i in range(1, n):
    LIS[i] = 1 # A[i] 자체가 길이 1인 LIS
    for j in range(i-1, -1, -1):
        if A[j] < A[i]:
            LIS[i] = max(LIS[i], LIS[j] + 1)
return max(LIS)
```
 - v. 수행시간: 이중 반복문을 분석해보면, 쉽게 $O(n^2)$ 시간임을 알 수 있다
- e. DP 알고리즘2: $O(n \log n)$ 시간 알고리즘
 - i. zig-zag 수열을 계산하는 $O(n \log n)$ 시간 알고리즘과 본질적으로 같음!
 - ii. $M[k] =$ 길이가 k 인 증가 부수열 중에서 가장 작은 마지막 값의 인덱스
 1. $M[k]$ 의 k 는 증가 부수열의 길이이고, $M[k]$ 는 A 의 값이 아닌 부수열의 최소 마지막 값의 인덱스임에 유의!
 2. 길이가 k 의 부수열의 오른쪽에 A 의 값을 추가해 길이 $k+1$ 인 부수열을 만든다면, 길이 k 의 부수열의 마지막 값이 작을수록 추가할 수 있는 오른쪽 값의 범위가 넓기 때문에 이런 식으로 DP 테이블을 정의한 것이다

iii. 예: $A = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]$
 index 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

1. 길이 k 는 1 이상 n 이하의 범위를 갖는다
2. 예를 들어, 길이가 $k = 2$ 인 증가 부수열은 $[0, 8]$, $[0, 4]$, $[0, 2]$, $[8, 12]$, $[2, 10]$, ..., $[3, 11]$, $[3, 7]$, $[7, 15]$ 등 매우 많다. 이 중에서 가장 작은 값으로 끝나는 증가 부수열은 $[0, 1]$ 이므로 $A[8] = 1$ 의 인덱스 값 8이 $M[2] = 8$ 이다
3. 최장 증가 부수열은 $0, 2, 6, 9, 11, 15$, 길이 $L = 6$ 이다
4. 그러면 $M[1] = 0$, $M[2] = 8$, $M[3] = 12$, $M[4] = 14$, $M[5] = 13$, $M[6] = 15$ 가 된다

iv. 사실 1: $A[M[1]], A[M[2]], \dots, A[M[L]]$ 은 증가 수열이다
 1. 증명: 거의 당연하므로 직접 해보기 바람

v. 알고리즘 아이디어:

1. A 의 값을 왼쪽부터 차례로 보면서 M 의 값을 업데이트한다
 - a. $A[i]$ 를 보는 단계라면, $A[M[k]] < A[i] < A[M[k+1]]$ 을 만족하는 $A[M[k]]$ 를 찾는다. 그러면 $A[M[k]]$ 는 길이가 k 인 증가 부수열의 마지막 값이 되고, $A[i]$ 보다 작으면서 왼쪽에 있기에 $A[i]$ 를 포함하면 길이가 $k+1$ 인 증가 부수열이 된다
 - b. $M[k+1]$ 의 값을 i 로 수정하면 된다
2. 사실 1에 의해 $A[M[1]], A[M[2]], \dots, A[M[L]]$ 이 오름차순으로 정렬되어 있으므로, $A[M[k]] < A[i] < A[M[k+1]]$ 을 찾는 것은 $O(\log n)$ 시간의 이진탐색으로 가능하다

vi. Pseudo 코드 (Wikipedia 코드를 조금 수정)

```

L = 0          # 현재까지 찾은 best LIS 길이 (결국, 우리가 찾는 값)
              # 초기 값은 0이고 계속 큰 값으로 업데이트 됨!
M = [0, ..., 0]
for i = 1 to n-1:
  1. A[0], ..., A[i]에 대해, A[i]로 끝나는 증가 부수열 최장 길이 newL을
     찾는다! 최장 길이 newL을 [1..L]에 대한 이진탐색으로 찾는게 핵심
     low = 1, high = L
     while low <= high: # A[M[low-1]] < A[i] < A[M[low]]
       mid = (low+high)//2
       if A[M[mid]] < A[i]: # 길이가 mid보다 큼
         low = mid + 1
       else:
         high = mid - 1
  2. newL = low          # low는 찾고 있는 길이 = k+1
  3. M[newL] = i          # 길이 newL 수열이 A[k]로 끝남

```

```

4. parent[k] = M[newL-1]      # LIS를 재구성하기 위해
5. L = max(L, newL)          # update L
return L, parent

vii. A = [0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15]
idx  0  1  2  3  4  5  6  7  8  9 10  11 12  13 14 15

L과 M을 계산해보자!

```

viii. 수행시간: $A[k]$ 값 각각에 대해 $[1, L]$ 범위에 대해 이진탐색을 하므로 $O(n \log L)$ 이지만 $L \leq n$ 이므로 $O(n \log n)$ 이면 충분

f. [?] 아래 코드를 분석해보자

```

Q = []
Q.append(A[0])
for i in range(1, n):
    k = index of first value of Q that is ≤ than A[i]
    if k == None: # no such index k
        Q.append(A[i])
    elif A[k] > A[i]:
        A[k] = A[i]
    else: pass
print(len(Q))

```

i. 무엇을 계산하는 코드인가?

ii. 수행시간은 어느 정도면 충분한가?

g. [고급 - skip 가능] LIS와 유사하게 가장 긴 감소 수열 (LDS: Longest Decreasing Sequence)을 정의할 수 있다. n 개의 서로 다른 수의 수열이 입력되는 경우, 이 입력 수열에는 항상 길이가 $n^{\frac{1}{2}}$ 이상의 LIS 또는 LDS가 존재한다는 사실이 이미 증명되었다

i. 증명: [Erdős-Szekeres theorem](#) (pigeon-hole 원리를 이용해 증명)

13. [🎤 인터뷰 문제 - 2D: 조각 높이 쌓기] 직사각형의 모양의 나무 조각이 n 개 있다. i 번째 나무 조각은 폭 $w[i]$ 와 높이 $h[i]$ 의 크기이다. 이 조각들을 차곡 차곡 쌓고 싶은데, 어떤 조각이 다른 조각 위에 놓으려면 반드시 위 조각의 폭이 아래 조각의 폭보다 작아야 한다. 이 조건을 만족하면서 최대한 높이 쌓고 싶다. 즉, 쌓아 올린 전체 높이가 최대가 되도록 하고 싶다. 어떻게 해야 할까?

- a. [힌트] Sorting + LIS 스타일의 DP?
- b. 핵심은 폭이 큰 조각부터 작은 조각 순서로 선택을 해야 한다는 것이다. 우선 폭 값의 내림차순으로 정렬을 하고, 정렬된 순서에 따라 하나씩 보면서 높이의 합이 최대가 되도록 선택을 하면 된다. 정렬된 순서에 따라 가장 큰 조각부터 차례대로 고려하기 때문에 이렇게 선택된 조각은 모두 조건을 만족하게 된다
- c. i 번째 조각을 고려한다고 하자. 그러면 0번째 부터 i 번째 조각에 대해 선택하는데, 가장 마지막에 i 번째 조각을 선택한 경우에 가장 큰 높이 합을 고려하면 된다. 이 때의 높이를 $\max[i]$ 에 저장한다고 하자
- d. $\max[i]$ 에 대한 점화식을 세워보자. i 번째 조각이 마지막에 선택되는 것이므로, 그 전에 선택된 j 번째 조각은 $w[j] < w[i]$ 조건이 성립해야 하는데, 조각들이 폭의 내림차순으로 정렬되어 있기에 $j < i$ 인 j 번째 조각은 이 조건을 당연히 만족한다. 그러면 $j < i$ 인 모든 j 에 대해, j 번째 조각이 i 번째 조각 바로 직전에 선택된다고 하면, 높이는 $\max[j] + h[i]$ 가 된다. 그러면 $\max[j]+h[i]$ 가 가장 클 때의 합을 $\max[i]$ 에 저장하면 된다. $\max[i] = \max[i]$ 로 초기 값을 지정한 후에, $\max[i] = \max_{j < i} (\max[i], \max[j] + h[i])$ 점화식을 이용해 업데이트하면 된다
- e. 사실, 이 점화식은 LIS 문제의 $O(n^2)$ 시간 DP 알고리즘의 점화식과 동일하다!

14. [🎤 인터뷰 문제 - 3D: 상자 높이 쌓기] 3차원 상자가 있다. 상자의 크기는 (높이, 폭, 깊이)로 정의되며, 위에 놓이는 상자의 폭과 깊이가 아래에 있는 상자의 폭과 깊이보다 각각 작아야 한다. 단, 상자는 회전이 가능하고, 동일한 상자를 회전해서 원하는 만큼 쌓는데 사용해도 된다. 조건에 맞게 쌓아 올릴 때, 전체 높이가 최대가 되도록 하려면 어떻게 해야 할까?

- a. 먼저, 문제를 잘 읽어봐야 한다. 회전을 해서 놓을 수 있다는 점과 같은 상자 여러 개를 사용할 수 있다. 즉, $(\text{높이}, \text{폭}, \text{깊이}) = (1, 2, 3)$ 인 상자가 있다면, $(1, 2, 3)$ 인 상자와 이 상자를 회전한 $(2, 1, 3), (3, 1, 2)$ 등의 상자도 원한다면 쌓는 데 사용할 수도 있다는 의미이다. 사실 이 조건이 없다면 오히려 더 풀기 어려운 문제가 된다 ^^
- b. 바닥에 놓이는 면은 폭과 깊이 값을 따져야 한다. 항상 폭 $<$ 깊이인 상태에서 상자를 쌓도록 하면 알고리즘이 더 단순해진다. 따라서, $(1, 2, 3)$ 의 상자가 있다면, 회전한 상자들 중에서 고려할 것은 $(2, 1, 3)$ 과 $(3, 1, 2)$ 두 경우 뿐이다
- c. 그러면 입력으로 주어진 n 개의 상자와 이 상자의 회전한 상자 두 개씩을 추가해 총 $3n$ 개의 상자를 폭과 깊이에 대한 내림차순으로 정렬한 후, 2D 조각 쌓기 알고리즘을 그대로 적용하면 된다
- d. 따라서 $O(n^2)$ 시간 DP 알고리즘으로 해결 가능하다

7. Greedy algorithm (욕심쟁이 알고리즘)

1. Warming-up 문제 1: Give-me-the-change, man!

- a. 1원, 10원, 50원, 100원 동전이 있을 때, 최소 개수의 동전으로 373원을 거슬러 주고 싶다. 어떤 동전들을 몇 개씩 사용해야 하나?
- b. 어떤 방식으로 거슬러 줄 동전을 선택했나?
- 372원을 100원 동전으로 최대한 많이 거슬러 주는 게 최소 개수를 얻는데 유리할 것 같아, $100\text{원} \times 3\text{개} = 300\text{원}$ 을 먼저 거슬러 준다고 생각한다
 - 남은 73원에 대해, 거슬러 줄 수 있는 최대 금액의 동전으로 되도록 많은 액수를 거슬러 준다. 즉, $50\text{원} \times 1\text{개} = 50\text{원}$ 을 추가로 거슬러 준다
 - 남은 금액 22원에 대해선, $10\text{원} \times 2\text{개} + 1\text{원} \times 2\text{개}$ 를 거슬러 준다
 - 결국, 총 $3 + 1 + 2 + 2 = 8$ 개의 동전으로 거슬러 주는 게 최소이다
- c. 이 방법이 항상 최소 개수를 보장할까?
- 만약 100원을 3개 보다 적게 사용하면서 전체 개수를 더 줄일 수 있는 방법이 있을까?
 - 있다면, 100원 1개를 대신해 50원 2개, 50원 1개와 10원 5개, 10원 10개, 1원 100개 등 최소 2개 이상의 작은 동전으로 100원을 대신해야 한다. 이건 애초의 의도와 다르게 더 많은 개수의 동전을 사용하게 되는 **모순**이 발생한다
 - 따라서 이 경우에는 항상 최대 단위로 가장 많이 교환해주는 방법이 최소 개수의 동전 교환을 보장한다
- d. 만약, 1원, 5원, 7원이 동전 단위인 경우, 10원을 거슬러주고 싶다면?
- 앞에서 사용했던 방법을 그대로 사용해서 거슬러 준다면 몇 개의 동전이 사용할까?

 - 이 동전 개수보다 더 작게 거슬러 줄 수 있나? (YES!) 정답은 몇 개인가? **50원 2개**
 - 왜 이 경우는 다를까? 앞의 동전 단위와 비교해 다른 점이 무엇인지 먼저 살펴보자
 - DP 알고리즘으로 해결할 수 있을까?
 - $\text{DP}[x]$ 는 x 원을 거슬러 주는 데 필요한 **최소 동전 개수**로 정의하면, $\text{DP}[x]$ 의 점화식을 세울 수 있을까?
 - 동전 단위 1원, 5원, 7원을 각각 하나씩 사용하는 경우 중에서 최소 동전 개수를 $\text{DP}[x]$ 에 저장하면 된다. 따라서 아래와 같은 점화식이 성립한다
$$\text{DP}[x] = \min (\text{DP}[x-1] + 1, \text{DP}[x-5] + 1, \text{DP}[x-7] + 1)$$
 - DP 알고리즘의 수행시간을 계산해보면, $\text{DP}[x]$ 를 계산하기 위해, 동전 단위 n 개를 모두 고려해 최소 값을 구하는 방식이므로 $O(n)$ 시간이 필요하다. DP 테이블의



엔트리 개수가 $DP[1], \dots, DP[x]$ 이므로 $O(x)$ 이다. 총 시간은 $O(xn)$ 이 된다. x 는 입력 값의 개수가 아닌 값 자체이므로 매우 클 수 있다. 경우에 따라서는 n 값과 관계 없이 엄청나게 큰 값일 수도 있다는 뜻이다. 이렇게 값의 개수 (입력의 크기)가 아닌 값 자체가 수행시간의 항으로 표현되는 경우의 수행 시간을 **유사 다항 시간 (pseudo polynomial time)**이라 부른다. 유사 다항 시간이 아닌 다항 시간 알고리즘을 설계하는 것이 바람직하다

2. Warming-up 문제 2: Workaholic, man!

- a. 할 일이 5가지가 있고, 각 일에 대해선 필요한 시간이 있다. $A = [2, 1, 4, 6, 3]$
- b. 그런데, 나에게 주어진 시간 $T = 9$ 시간 뿐이다. T 시간 이내에 최대한 많은 일을 하고 싶다. 여러분은 어떤 일들을 먼저 해야지 T 시간 내에 가장 많은 할 수 있을지 알아내야 한다
- c. 알고리즘
 - i. 가장 많은 일을 하기 위해선, 시간이 적게 드는 일부부터 하는 게 직관적
 - ii. A 를 오름차순으로 정렬: $A = [1, 2, 3, 4, 6]$
 - iii. 차례로 시간 합이 T 를 넘지 않을 때까지: $A[0] + A[1] + A[2] = 6 < T = 9$
 - iv. 그러면 이 세 개의 일을 하는 게 가장 많은 일을 하는 것임
- d. 이 알고리즘은 항상 최대한 많은 일을 선택하는가? 증명할 수 있나?
 - i. 만약, 다른 알고리즘에 의해 더 많은 일을 선택할 수 있었다고 *반대로* 가정해보자
 - ii. 즉, 3개가 아닌 4개 이상의 일을 선택할 수 있는 방법이 있었다면, 위의 알고리즘도 사실은 같은 개수의 일을 선택할 수 있어야 한다는 결론에 도달할 수 있다. 이는 원래 알고리즘이 3개만 선택할 수 있었다는 사실에 모순이 되어 더 많은 일을 선택할 수 있는 방법이 애초에 없다는 것을 의미한다

3. Greedy (그리디, 욕심쟁이) 방법

- 문제의 목적(goal)은 특정 값이 최소가 되든지 최대가 되어야 하는 게 일반적이다
- 현재 상태에서 목적을 달성하기 위해 가장 유리한 선택을 한다 → 사용한 선택 기준을 greedy 기준이라 부른다
- 더 이상 선택할 수 없을 때까지 반복한다.

4. 예3: 강의 배정 문제 (Activity Selection Problem)

- 문제: 강의실이 하나 뿐인데, 이 강의실을 쓰고 싶은 강의는 많다.
강의가 서로 겹치지 않고 최대한 많은 강의를 배정하라

- 입력: n개의 강의가 주어진다. 강의 i번은 $S[i]$ 에 시작해서 $F[i]$ 에 끝난다
- 출력: 최대한 많은 수의 강의를 배정한다



- 위의 warmup 문제 2와 유사

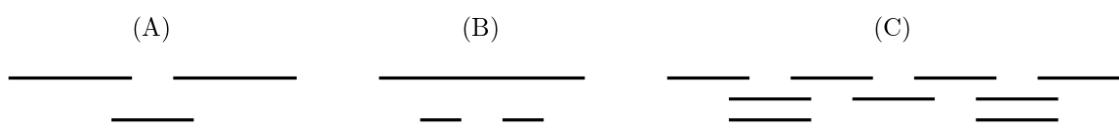
- 예: 11개 (0번 강의 - 10번 강의)의 시작 시간을 저장한 S , 끝나는 시간을 저장한 F

$$\begin{aligned} S &= [1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12] \\ F &= [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14] \end{aligned}$$

- 다음 greedy 기준을 생각해보자 (이 기준이 답을 주지 못하는 반례가 존재하는지 먼저 생각)

아래 그림 3개는 3개 기준에 대한 반례다. 어떤 기준이 어떤 반례를 갖는지 찾아보라

- [기준1] 수업시간이 가장 짧은 강의부터 선택한다 - 반례 (A)
- [기준2] 가장 적게 겹치는 강의부터 선택한다 - 반례 (C)
- [기준3] 가장 빨리 시작하는 강의부터 선택한다 - 반례 (B)
- [기준4] 가장 빨리 끝나는 강의부터 선택한다 - 반례 없음



- 기준 4에 따라 선택을 하는 greedy 알고리즘은 아래와 같다

```
Greedy-Iterative-Lecture-Selector(S, F):
1   Sort lectures by their finish times
2   L = [0]
3   k = 0
4   for i = 1 to n-1 :
5       if S[i] ≥ F[k]:
6           L = L.append(i)
7           k = i
8   return L
```

- 위의 예제에 대해선 선택한 강의 $L = [0, 3, 7, 10]$ 임

- g. 이 기준4의 그리디 알고리즘은 항상 최대 개수의 강의를 배정할까?
 (정확성 증명 → 오른쪽 동영상에서 설명)

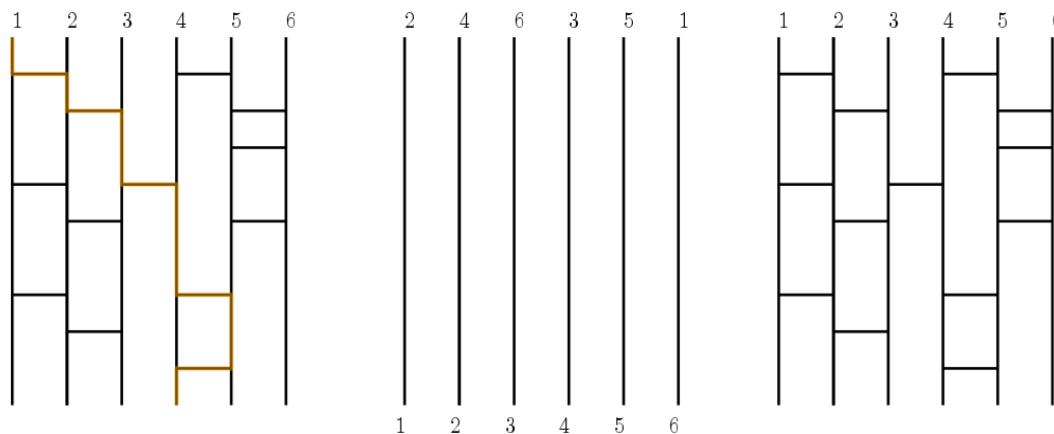


5. 예 4: 사다리타기

Wikipedia의 설명

Ghost Leg (Chinese: 畫鬼腳), known in Japan as **Amidakuji** (阿弥陀籤, "Amida lottery"), so named because the paper was folded into a fan shape resembling Amida's halo^[1]) or in Korea as **Sadaritagi** (사다리타기, literally "ladder climbing"), is a method of lottery designed to create random pairings between two sets of any number of things, as long as the number of elements in each set is the same.

- a. 사다리는 세로 기둥들과 가로막대들로 구성



- b. 질문1: 사다리의 가장 위에 1부터 n까지 주어지고 사다리의 가로 막대들이 입력으로 주어지면, 사다리의 가장 밑의 출력은 어떻게 계산할 수 있을까?

- i. 각 가로 막대는 (a, b, y) 세 자연수로 입력되는 데, 두 사다리 a, b 를 연결하는 막대이며 y -좌표는 y 라는 의미다. 두 사다리는 이웃해야 하므로 $a = b-1$ 을 만족한다
- ii. 간단한 방법은 가장 위쪽 가로막대부터 차례대로 고려하면서 가로막대를 통해 교환되는 두 수를 계속 업데이트하면 된다. $O(n)$ 시간에 가능하다

- c. 질문2: 입력으로 주어지는 수가 1부터 n까지의 임의의 순열(permutation)이고 출력은 왼쪽부터 오른쪽으로 오름차순으로 출력하려면, 가로막대를 어디에 몇 개를 그려야 할까?

- i. 핵심은 하나의 가로막대는 인접한 두 수의 위치를 바꾸는 것으로 버블 정렬의 교환(swap) 연산과 동일하다는 것이다.

- ii. 버블 정렬에서 교환 횟수만큼 가로막대를 그리면 된다. 가로막대는 교환 연산과 동일하고 교환은 인접한 두 수가 inversion일 때에만 발생한다. 두 수에 대해 왼쪽 수가 오른쪽 수보다 크다면 두 수는 하나의 inversion 쌍으로 정의된다. 인접한 두 수가 inversion인 경우에 가로막대를 통과하면 inversion이 하나 해소된다. 오름차순으로 정렬된 순열에서는 inversion이 존재하지 않는다. 따라서 모든 inversion이 해소되어야 한다. 가로막대는 인접한 두 수가 inversion일 때 해당 inversion만 해소하기에 inversion 개수가 정확히 하나 감소한다. 결국, 입력 순열의 inversion 개수만큼 가로막대가 필요하다
 - iii. 버블 정렬의 교환 연산이 가로막대와 동일한 효과를 보이기 때문에 입력 순열을 버블 정렬한다고 하고, 교환이 일어나는 두 수에 대응되는 인접한 두 세로막대 사이에 가로막대를 그리면 된다. 다음에 그리는 가로막대는 y-좌표를 증가시켜 그리는 식으로 하면 오름차순으로 정렬된 출력을 보장하는 사다리를 완성할 수 있다
 - iv. 버블 정렬이 $O(n^2)$ 시간이 가능하다. 따라서 사다리도 같은 시간에 완성할 수 있다. 그러나 inversion 개수는 합병 정렬(merge sort) 방식으로 $O(n \log n)$ 시간에 계산할 수 있다. 반면에 inversion 개수는 최대 $n(n-1)/2$ 개까지 가능하므로 가로막대 수도 최악의 경우에 그만큼 필요하다. 따라서 모든 가로막대의 위치를 출력해야 한다면 $O(n^2)$ 시간을 피할 수는 없다
- d. **질문3:** 사다리의 가장 위 순열이 1부터 n까지이고, 가로 막대들과 가장 아래 순열이 모두 입력으로 주어진다고 하자. 출력 순열을 그대로 유지하면 가로 막대 중에서 중복되는 막대를 최대한 많이 제거하고 싶다면 어떻게 해야 할까?
- i. ICPC 2019 Seoul Regional Contest의 "Ladder Game" 문제로 출제되었다 (오른쪽 QR 코드 링크에서 원본 문제 다운로드 가능)


 - ii. **질문1과 질문2의 답**으로부터 쉽게 해결 방법을 알 수 있다
 1. 사다리의 가장 아래에 주어지는 순열이 무엇인지 계산한다 (**질문1**)
 2. 사다리의 아래의 순열을 위로 올려 보내 위의 오름차순 순열이 된다고 거꾸로 생각한다. 즉, 아래에서 윗쪽 방향으로 정렬하는 사다리라고 생각한다
 3. 가장 아래의 가로막대부터 차례대로 통과하는데, 현재 가로막대가 inversion을 없애는 가로막대면 그대로 두고, inversion을 생성하는 가로막대는 없애는 식으로 판단하면 된다. 어차피 inversion의 개수만큼 가로막대가 있어야 하기에, inversion을 없애는 가로막대만 남기고 나머지는 (중복이므로) 모두 없애면 된다 (**질문2**)
 - iii. 결국, $O(n^2 + \text{가로막대수})$ 시간에 계산이 가능하다

6. 예 5: Huffman Code (허프만 코드) 계산 문제

- a. 키보드로 입력하는 모든 문자 키(key)에 대해선 고유의 정수번호가 할당되어 컴퓨터 내부에 저장되어 처리된다. 이 코드 체계를 ASCII (American Standard Code for Information Interchange)라 부른다.
 - i. 0부터 255 (= 2^8-1)까지 코드가 문자에 할당되어 있음
 - ii. 예를 들어, NULL = 0, 'A' = 65, 'a' = 97, '+' = 43
- b. ASCII는 문자 하나에 8 비트가 할당되는 코드 체계이므로, 만약 100개의 문자로 구성된 문서가 있다면 컴퓨터에서는 $100 \times 8 = 800$ bits의 메모리가 필요하다
 - i. 이를 고정길이코드(fixed-length code)라 한다.
- c. 문제는 문자들이 나타나는 빈도수(frequency)가 다르기 때문에 자주 사용되는 문자에게는 짧은 길이의 코드를, 자주 등장하지 않는 문자에게는 조금 더 긴 코드를 할당하는 게 전체 필요한 비트 수를 줄이는 경제적인 방법이다
 - i. 예를 들어, 영어 알파벳에서는 a, e, i, o, u와 같은 모음이 q, z, y 등과 같은 자음보다는 훨씬 더 자주 사용된다
 - ii. 이렇게 코드의 길이가 가변적이라면 이를 가변길이코드(variable-length code)라 한다.
- d. 예: 문자-빈도수-고정길이코드-가변길이코드

문자	빈도수	고정길이코드	가변길이코드
a	43	000	0
b	13	001	101
c	12	010	100
d	16	011	111
e	9	100	1101
f	7	101	1100

- i. 총 빈도수가 100이므로 고정길이 코드로 암호화 한 경우엔 $100 \times 3 = 300$ 비트가 필요하고, 가변길이 코드로 암호화 한 경우엔 $1 \times 43 + 3 \times (13+12+16) + 4 \times (9+7) = 230$ 비트가 필요하다
 - 1. 이 예에서는 가변길이코드가 고정길이코드보다 20% 정도 메모리를 더 적게 사용
- e. Prefix-free code:
 - i. 한 코드가 다른 코드의 prefix에 나타나지 않는 코드 체계를 prefix-free 코드라 부른다
 - ii. 예를 들어, 'a' = 01 'b' = 1 'c' = 011라고 하면, 011이란 이진코드가 주어지면 이 코드가 'ab'와 'c' 중 어느 것인지 결정할 수가 없게 된다
 - iii. 따라서, 모호성이 없는 코드 체계는 항상 prefix-free 코드여야만 한다
- f. 코드의 성능
 - i. 문자의 코드의 비용은 코드 길이 * 빈도수로 정의한다

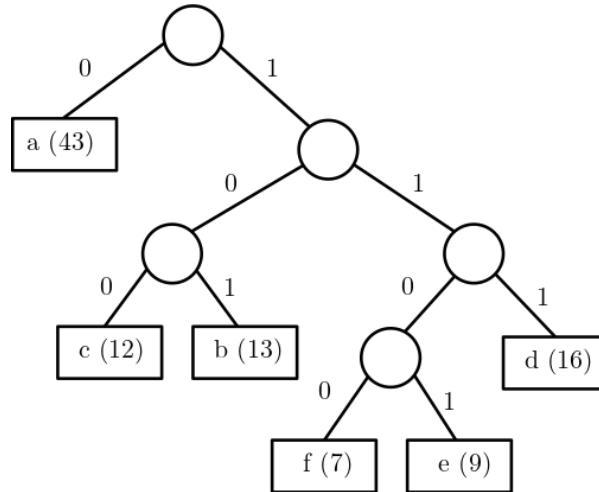
ii. 코드 전체 비용은 문자의 비용을 모두 더한 값으로 정의한다

g. **Huffman Coding Problem:**

i. 주어진 문자의 빈도수에 대해, 코드 비용이 최소가 되는 prefix-free 코드를 찾아라!

h. **Huffman Tree T**

i. 코드 할당을 표현한 이진 트리



i. 코드 비용 = $\text{cost}(T)$

i. $d_T(x)$: 문자 x의 T에서의 깊이(depth) = x의 코드 길이(비트수)

ii. $f(x)$: 문자 x의 빈도수

$$\text{cost}(T) = \sum_x f(x)d_T(x)$$

j. 결국 $\text{cost}(T)$ 가 최소인 Huffman Tree T를 계산하는 문제와 동일하다!

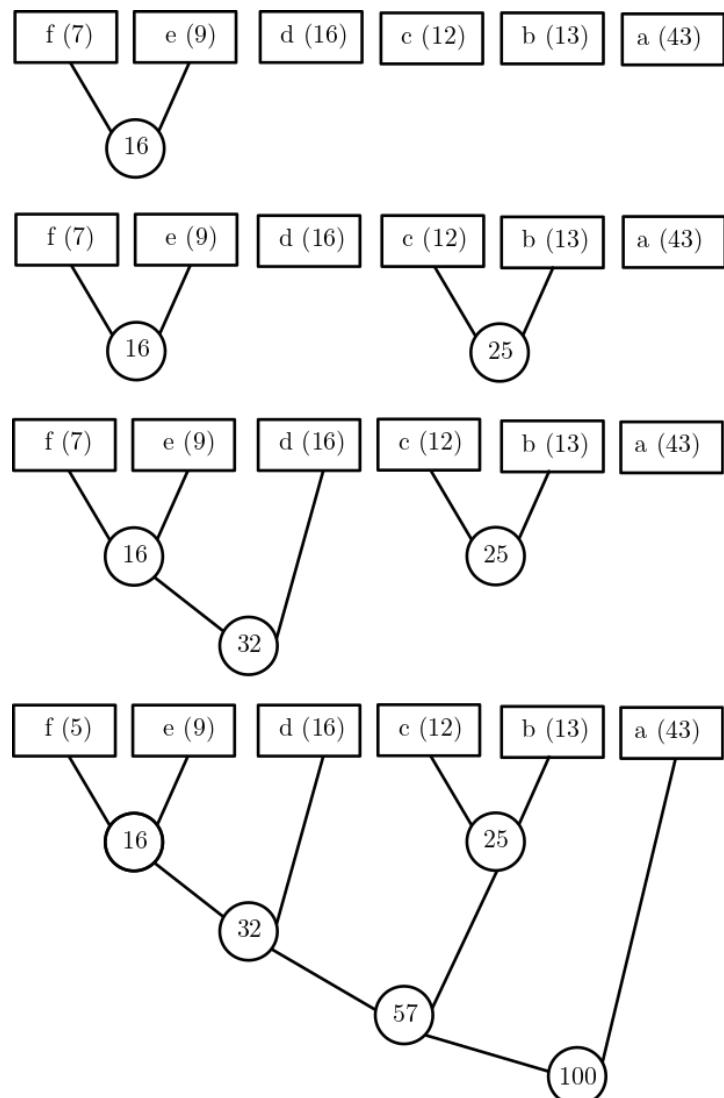
k. **Huffman Algorithm**

i. **IDEA:** 빈도수가 낮은 문자는 트리의 깊은 곳에, 높은 문자는 낮은 (루트에 가까운) 곳에 두는 그리디 기준을 적용하면 $f(x)d_T(x)$ 값을 줄일 수 있을 것 같다!



ii. **Greedy 기준:**

1. 빈도수가 낮은 문자를 먼저 짹을 지어 트리를 점진적으로 구성
2. 먼저 짹을 지어 구성하므로 빈도수가 낮은 문자가 루트 노드에서 멀어지게 되는 효과가 있다



I. 필요한 자료구조: heap

- greedy 선택을 위해, 매 라운드마다 가장 작은 두 개의 빈도수를 알아야 함 → `delete_min` 연산 두 번 호출하면 됨
- 매 라운드마다 (두 빈도수를 더한) 새로운 빈도수를 나머지 빈도수가 저장된 자료구조에 삽입해야 함 → `insert` 함수 호출하면 됨
- 이는 정확히 힙(heap) 자료구조이다 (min-heap 자료구조로 가장 작은 값이 루트노드에 저장)
 - `delete_min`과 `insert` 함수는 $O(\log n)$ 시간 필요

m. [Python] heapq 모듈에서 heap 관련 함수를 그대로 제공한다

- `import heapq`
- 힙 자체는 리스트를 사용한다: `h = []`
- 지원연산:
 - `heappush(h, key)`: 힙 `h`에 `key` 값을 삽입 (= `insert`와 동일)
 - `heappush(h, (key, value))`처럼 튜플 삽입 가능
 - `heappop(h)`: 최소값을 지우고 리턴 (`delete_min`의 역할)
 - `h[0]`: 힙의 최소값을 알고 싶다면

n. Pseudo code (by using heapq):

```

H = []
for x in range(n):
    heappush(H, (f[x], 'x')) # 튜플을 삽입
    # 튜플의 첫 번째 항은 f, 두 번째 항은 x

while |H| > 1:
    a = heappop(H) # 가장 작은 빈도수 item
    b = heappop(H) # 두 번째로 작은 빈도수 item
    heappush(H, (a.f + b.f, '(a.x b.x)'))

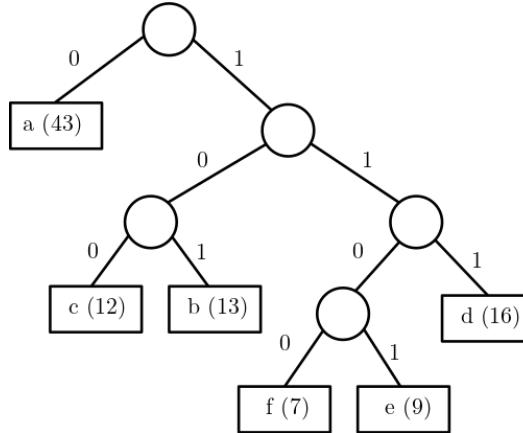
tree_string = heappop(H)[1]

```

o. 예를 들어, $f = [43, 13, 12, 16, 9, 7]$ 을 입력으로 준다면:

```
tree_string = (0 ((2 1) (3 (5 4))))
```

- i. $tree_string$ 은 트리를 어떻게 표현한 것인가? $(a, (b, c))$ 형태는 리프 노드 b와 c를 먼저 부모 노드로 연결하고, 리프 노드 a와 이 부모 노드를 새로운 부모 노드로 연결했다는 의미이다. 트리를 그려보면 다음과 같다



- ii. $tree_string$ 으로부터 각 문자에 할당된 bit 수는 어떻게 계산할 수 있을까?

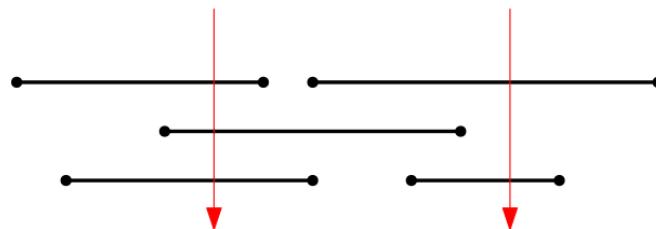
1. 문자의 비트 수는 해당 리프 노드의 깊이와 같다
2. 깊이는 트리 문자열에서 해당 입력 문자를 감싸고 있는 괄호 쌍의 개수와 같음을 쉽게 파악할 수 있다
3. 따라서 $tree_string$ 의 가장 왼쪽 문자부터 보면서 입력 문자 전에 나오는 왼쪽 괄호의 개수가 깊이가 된다
4. 오른쪽 괄호가 등장하면 그 전에 쌍이 되는 왼쪽 괄호가 존재하기에 깊이가 1 줄어들어야 한다
5. 이 과정은 문자열의 문자를 하나씩 보면서 왼쪽 괄호가 등장하면 깊이를 1 증가하고 오른쪽 괄호가 등장하면 깊이를 1 감소하면서 선형 스캔하면 $O(n)$ 시간에 충분히 수행된다

p. 코드 완성은 각자 마무리해보자!

- q. 이 greedy 알고리즘은 항상 올바른 Huffman code를 계산하는가?
(정확성 증명)
i. 영상으로 증명 과정을 설명합니다



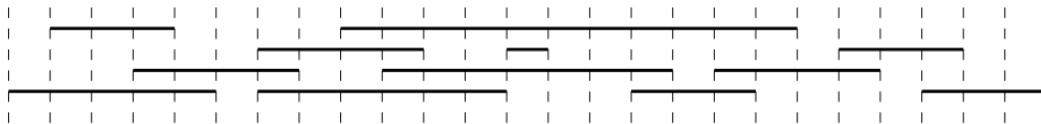
7. 예 6: 뭇 박기 (pinning)



- a. **문제 1:** n 개의 막대가 입력으로 주어질 때, 하나의 뭇으로 꽂을 수 있는 막대의 최대 개수는 어떻게 구할 수 있을까? (단, 막대의 끝 점을 뭇이 통과해도 그 막대를 꽂는다고 가정한다)
- i. 막대가 가장 많이 겹치는 구간을 구하는 것과 동일하다. 어떻게 그 구간을 계산할 수 있을까? 위의 예제에서는 첫 번째 빨간색 수직선 화살표에 뭇을 위치시키면 가장 많은 세 개의 막대를 통과할 수 있다
 - ii. 막대의 가장 왼쪽에 가상의 수직선이 있다고 가정하고, 이 수직선이 연속적으로 오른쪽으로 움직인다고 가정하자. 가장 첫 막대의 왼쪽 끝 점에 도달한 후에는 두 번째 막대의 왼쪽 끝 점에 도달할 때까지 수직선은 하나의 막대만 통과한다. 그 이후에는 두 개의 막대를 통과한다. 즉, 새로운 막대의 왼쪽 끝 점을 만날 때마다 수직선이 통과하는 막대의 수는 하나씩 늘어난다. 그러나 어떤 막대의 오른쪽 끝 점을 만나면 해당 막대는 더 이상 수직선과 만나지 않는다. 따라서 이 경우에는 수직선과 만나는 막대의 개수를 하나 줄여야 한다. 정리하면, 막대의 왼쪽과 오른쪽 끝 점을 모두 모아 오름차순으로 정리한 후에 정렬 순서에 따라 하나씩 보면서 왼쪽 끝 점이면 수직선이 통과하는 막대 개수를 1 증가시키고 오른쪽 끝 점이면 1 감소시키는 작업을 반복하면 된다
 - iii. 결국 한 번의 정렬과 끝 점을 정렬 순서대로 한 번씩 고려하는 선형 스캔이면 되기에 $O(n \log n)$ 시간이면 충분하다
- b. **문제 2:** n 개의 막대가 입력으로 주어질 때, n 개의 막대를 최소 개수의 뭇(pin)으로 꽂으려 한다. 모든 막대는 최소 하나 이상의 뭇이 박혀야 한다. 이 때, 최소 몇 개의 뭇이 필요한가?
(강의 배경 문제와 매우 비슷하다!)
- i. 위의 예제에서는 2개의 뭇으로 모두 꽂을 수 있다 (赍이 막대의 끝을 통과하더라도 인정하는 것으로 가정)
 - ii. **관찰:** 뭇을 막대의 오른쪽 끝에만 꽂으면서도 최소 개수의 뭇만 사용하는 해가 반드시 존재한다 (**왜?**)

- iii. **힌트**: 모든 막대는 최소한 하나 이상의 못에 의해 꽂혀야 한다 + 관찰 1

- iv. 아래 예제의 답은? 5개



- c. **문제 3**: n 개의 막대와 못의 개수 m 이 주어질 때, m 개의 못으로 꽂을 수 있는 최대 개수의 막대는 몇 개인가?

- 문제 2의 관찰이 여기서도 그대로 성립하는가? Yes
- 문제 2를 해결한 방식을 여기에도 적용할 수 있을까? 최대 개수의 막대를 보장하기 위해서는 중간의 막대 몇 개는 건너 뛸 수도 있기에 단순히 문제 1의 그리디 기준만으로 선택할 수 있는지는 분명하지 않다. 다른 방법을 먼저 생각해보자
- DP로 해결한다면, 해를 어떻게 작은 해의 식으로 표현할 수 있을까?
 - 막대를 먼저 오른쪽 끝 점의 오름차순으로 정렬 (왼쪽 → 오른쪽) 한다. 따라서 $O(n \log n)$ 시간이 필요하다. 못 역시 왼쪽부터 차례로 꽂아나간다고 가정하고, 오른쪽 끝 점 번호와 못 번호는 모두 1부터 시작한다고 가정
 - $P[k][j] =$ 오른쪽 끝 점 j 에 k 번째 못을 꽂았을 때, k 개의 못으로 꽂을 수 있는 막대의 최대 개수
 - 그러면 $P[k][j]$ 의 식을 마련해보자. $(k-1)$ 개의 못은 k 번째 못의 왼쪽에 존재한다. $(k-1)$ 번째 못 역시 어떤 막대의 오른쪽 끝 점에 놓여 있을 것이다. 이 끝 점을 i 라 하면, 당연히 $i < j$ 이다. k 번째 못만이 관통하는 막대는 무엇일까? 끝 점 i 와 j 사이에 왼쪽 끝 점이 있고 j 의 오른쪽에 오른쪽 끝 점이 있는 막대가 다른 못이 아닌 k 번째 못이 관통하는 막대이다. 이 개수를 $m(i, j)$ 라 하자. 그러면 아래 점화식이 성립한다
$$P[k][j] = \max_{i < j} (P[k-1][i] + m(i, j))$$
- $m(i, j)$ 는 $O(n)$ 시간에 쉽게 계산할 수 있다. 더 빨리 계산할 수 있는 방법은 없을까? [힌트: 구간 질의(query)가 주어지면 구간에 포함된 끝 점을 빠르게 답해주는 자료구조가 필요?]
- DP 테이블 P 의 초기화는? $P[1][j]$ 는 첫 번째 못을 j 번 끝 점에 놓을 때 관통하는 못의 개수가 된다. $j = 1, \dots, n$ 에 대해, $P[1][j]$ 같은 질문 1의 방법으로 $O(n)$ 시간에 계산할 수 있다
- 최종 답은 m 개의 못으로 관통할 수 있는 막대의 최대 개수이다. 따라서 $\max(P[m][1], \dots, P[m][n])$ 이 되는데, 사실 m 번째 못은 m 번 끝 점과 그 이후에 놓이는 것이 당연하므로 $\max(P[m][m], \dots, P[m][n])$ 도 맞다
- P 의 엔트리 수가 $O(mn)$ 개이고 한 엔트리를 채우는데 걸리는 시간이 t 라면, $O(mnt)$ 시간이 걸린다

iv. DP 이외의 다른 방법은? 다양한 그리디 기준을 적용하면 그리디 알고리즘도 가능?

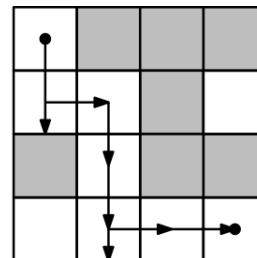
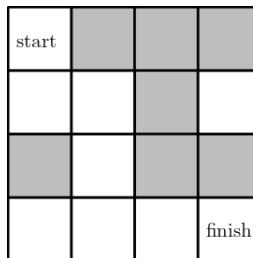
8. Backtracking algorithm



1. Warming-up 문제 1: 미로탈출

a. $n \times n$ 미로 M이 주어진다

- i. $M[i][j] = 0$ 이면 빈 칸이고, 1이면 장애물이라고 가정
- ii. 생쥐가 $M[0][0]$ 에서 출발하여, $M[n-1][n-1]$ 에 도착하면 탈출 성공!
- iii. 빈 칸만을 방문해야 하고, 오직 오른쪽에 이웃한 빈 칸 또는 아래쪽에 위치한 빈 칸으로 이동할 수 있다고 가정
- iv. 문제는 생쥐가 미로 지도를 갖고 있지 않기에 길을 미리 계산할 수 없다는 것이다. 성공적으로 탈출할 수 있는지 결정하는 문제!



b. 가장 직관적이고 간단한 방법은 출발 칸부터 차례대로 갈 수 있는 길을 모두 가보는 것!

c. 전략

현재 칸이 탈출 칸이라면: 성공!

아니라면:

1. 현재 칸이 장애물이라면, 그 전 칸으로 **후퇴 (backtrack)**!
2. 현재 칸이 빈 칸이고 처음 방문한 칸이라면:
 - a. 방문했다고 기록
 - b. 아래 칸으로 이동 (재귀적으로 반복)
 - c. 아래 칸으로 이동해서 목적지에 도달하지 못했다면:
 - i. 오른쪽 칸으로 이동 (재귀적으로 반복)
 - d. 오른쪽 칸으로 이동해서도 목적지에 도달하지 못했다면:
 - i. 실패 결과를 가지고 **backtrack**!

d. Python code: `find_way(0, 0)`을 호출하면 됨

```
def find_way(x, y):
    if x == n-1 and y == n-1: return True
    if M[x][y] is safe:
        try_down = find_way(x+1, y)      # 아래 칸으로 이동
        if try_down == True:              # 성공했다면:
            return True
        try_east = find_way(x, y+1)       # 오른쪽 칸으로 이동
        return try_east
```

```

    else:
        return False

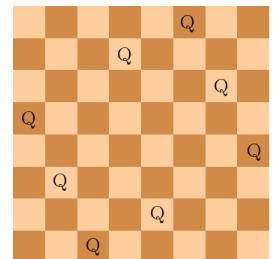
```

2. Warming-up 문제 2: N-queens

- a. $N \times N$ 체스 판에 N 개의 Queen을 서로 같은 행, 열, 대각선 위에 오지 않도록 배치하는 문제 (오른쪽 그림은 8-queens 예시)

- b. 관찰: 문제의 정의에 따라, 한 행과 열에 정확히 하나의 여왕을 배치해야 함

- c. 행 k 에 배치된 여왕의 열 번호를 x_k 라면, 결국 (x_1, x_2, \dots, x_N) 을 찾는 것과 동일



```

def NQueens( k ):
    # k 행에 놓을 x[k]를 결정하는 과정
    if k > N: return
    for c = 1, 2, ..., n:
        if k-th queen can place at (k, c): # bounding func B
            # if B(k, c) == True:
            x[k] = c
            NQueens( k+1 )

```

- d. $B(k, c)$: 여왕을 (k, c) 칸에 놓을 수 있으면 `True` 리턴, 아니면 `False` 리턴하는 학계 함수

- i. 1행부터 $(k-1)$ 행까지 놓인 여왕들과 충돌이 없어야 한다 (같은 열 또는 대각선에 놓인 여왕이 없어야 함!)
- ii. code는 각자 작성해보자!

- e. 해의 개수: $\text{pure} = \text{대칭이 아닌 해만 셀 경우}$

N	1	2	3	4	5	6	7	8	9	10		24
pure	1	0	0	1	2	1	6	12	46	92	...	28,439,272,956,934
all	1	0	0	2	10	4	40	92	352	724	...	227,514,171,973,736

3. Backtracking algorithm (general framework)

```

Backtrack( k ):    # recursive version
    if k > n:          # if solution is found!
        output solution
        return
    for each possible candidate value c for x[k]:
        if B(x[1], x[2], ..., x[k-1], c) == True:
            # (x[1], ..., x[k-1], c) is a part of solution
            x[k] = c
            Backtrack( k+1 )

```

4. 예제 1: Sudoku

- a. 유명한 퍼즐 문제로 9x9 판의 빈 칸을 규칙에 맞게 1부터 9사이의 수를 채우는 게임 (게임을 모르면 인터넷으로 검색해 볼 것!)

- b. 입력 예: 2차원 리스트로 표현

```
A = [[5, 3, 0, 0, 7, 0, 0, 0, 0],
      [6, 0, 0, 1, 9, 5, 0, 0, 0],
      [0, 9, 8, 0, 0, 0, 0, 6, 0],
      [8, 0, 0, 0, 6, 0, 0, 0, 3],
      [4, 0, 0, 8, 0, 3, 0, 0, 1],
      [7, 0, 0, 0, 0, 0, 0, 0, 6],
      [0, 6, 0, 0, 0, 0, 2, 8, 0],
      [0, 0, 0, 4, 1, 9, 0, 0, 5],
      [0, 0, 0, 0, 0, 0, 7, 0]]
```

- c. 리스트 A에 초기 9x9 판이 주어지면, 빈 칸을 찾아 1부터 9까지의 수를 차례대로 채워나간다. (물론, 그 수가 해당 빈 칸에 올 수 있는지 (valid한지의 여부)를 검사해서 올 수 없다면 다음 수를 채우는 식으로 진행한다)

```
def solve_sudoku(A):
    row, col = find_empty_cell(A) # 빈 칸이 없으면 None, None 리턴
    if row == None: # no empty cell means completed!
        return True

    for num in range(1, 10): # for each of possible nine candidates
        if is_safe(A, row, col, num): # if no conflict
            A[row][col] = num
            if solve_sudoku(A): # go on recursively
                return True      # "solution is found!", so return True

    A[row][col] = 0 # no safe number, so make it empty and backtrack
    return False
```

- i. **find_empty_cell**: A의 빈칸을 찾아 위치를 리턴하고, 빈 칸이 없다면 (즉, 퍼즐을 풀었다면) None, None을 리턴하는 함수

- ii. **is_safe**: A[row][col]에 값에 num이 valid한지 검사하는 함수

- d. 그러나 백트랙킹은 최악의 경우엔 모든 경우를 다 고려하는 것으로 매우 까다로운 입력에 대해선 상당히 오래 걸릴 수 있음!

e. 아래 코드 역시 백트래킹 코드인데, 올바르게 동작하는가?

```
def solve2(A):
    for row in range(len(A)):
        for col in range(len(A)):
            if A[row][col] == 0:
                for num in range(1, 10):
                    if is_safe(A, row, col, num):
                        A[row][col] = num
                        solve2(A)
                        A[row][col] = 0
    return # no possible value for A[row][col], so backtrack!

for i in range(len(A)):
    print(A[i])
print("")
input("want another solution if it exists? ")
```

i. 예: 아래 예는 백트랙킹으로 풀기엔 꽤 많은 시간을 요구함!

			3		8	5	
	1	2					
			5	7			
	4				1		
	9						
5					7	3	
		2	1				
			4			9	

(from [Wikipedia](#))

5. 예제 2: Subset Sum

- a. n개의 양의 정수 집합 A와 목표 값 S가 입력으로 주어질 때, n개의 수 중에서 몇 개를 뽑아 그 합이 S가 되는 경우가 존재하는지 결정하는 문제

- i. 다시 말하면, 합이 S가 되는 A의 부분집합이 존재하는지 검사하는 문제
- ii. $A = \{8, 6, 7, 5, 3, 10, 9\}$, $S = 15$ 이면 $\{8, 7\}$, $\{7, 5, 3\}$, $\{6, 9\}$, $\{5, 10\}$ 등 총 네 개의 부분집합의 합이 15가 되어 답은 YES!

- b. $A[i]$ 입장에서 생각해보자

- i. 부분집합에는 $A[i]$ 가 포함된 부분집합과 $A[i]$ 가 포함되지 않은 부분집합 두 가지 가능성뿐이다
- ii.

```
subsetSum(A, S) =
    subsetSum(A - A[i], S) or subsetSum(A - A[i], S - A[i])
```
- iii. Version 1

```
subsetSum1(A, S):
    if S == 0: return True
    elif S < 0 or A is empty: return False
    else:
        z = a value from A
        with = subsetSum1(A-{z}, S-z)      # T(|A|-1)
        wout = subsetSum1(A-{z}, S)        # T(|A|-1)
    return with or wout
```

- 수행시간: $T(n) = 2T(n-1) + \text{time_for_setminus}$

- iv. Version 2

```
subsetSum2(A, i, S):
    # {A[0],...,A[i]} has a subset whose sum is S?
    if S == 0: return True
    elif S < 0 or i == -1: return False
    else:
        with = subsetSum2(A, i-1, S-A[i])  # T(n-1)
        wout = subsetSum2(A, i-1, S)        # T(n-1)
    return with or wout
```

- 수행시간: $T(n) = 2T(n-1) + c$
- 실제 합이 S가 되는 부분집합을 모두 출력하기 위해서 어떤 코드를 추가해야 할까?



```

subsetSum2_output(A, i, S):
    if S == 0: return []
    if S < 0 or len(A) == 0:
        return None
    Y = subsetSum2_output(A, i-1, S)          # without
    if Y != None:
        return Y
    Y = subsetSum2_output(A, i-1, S-A[i])    # with
    if Y != None:
        return Y + [A[i]]
    return None

```

v. DP 알고리즘과 매우 유사하지 않나?

- `subsetSum(A, i-1, S)` or `subsetSum(A, i-1, S-A[i])`을 DP 식으로 변환해보자!
- $DP[i][S] = DP[i-1][S-A[i]] \text{ or } DP[i-1][S]$

- 수행시간은 DP 테이블의 엔트리 개수가 $O(ns)$ 개이고 상수시간에 엔트리 하나를 계산할 수 있기에 $O(ns)$ 이다
- 수행시간에 입력의 크기 n 뿐만 아니라 입력 S 의 값 자체가 포함되어 있다. 이렇게 입력으로 주어진 값 자체가 수행시간에 항으로 표시되는 수행시간을 **유사 다항 (Pseudo-Polynomial)** 시간이라고 한다. S 자체는 한 개의 값이지만, 값의 크기는 매우 클 수 있기 때문에 DP 알고리즘이 매우 느려질 수 있다. 가능하면 유사 다항 시간은 피하고 오로지 n 만 등장하는 다항 시간 알고리즘을 설계하는 게 바람직하다

vi. 합이 S 가 되는 모든 부분집합을 구하려면?

- 해를 나타내는 리스트 x 에서, $A[i]$ 가 해에 포함되면 $x[i] = 1$ 이고, 해에 포함되지 않는다면 $x[i] = 0$ 이 됨
 - $A=[8, 6, 7, 5, 3, 10, 9]$, $S=15$ 에서, $x=[1, 0, 1, 0, 0, 0, 0]$ 은 $A[0]=8$ 과 $A[2]=7$ 로 구성한 부분집합 $\{8, 7\}$ 이 해가 됨
 - 물론, $\{7, 5, 3\}$ 도 해가 되므로, $x=[1, 0, 0, 1, 1, 0, 0]$ 또한 해가 됨
 - x 의 모든 가능한 경우는 2^n 개이고, 이 중에서 답이 되는 4가지 경우가 존재한다

```

subsetSum_Backtrack1(k): # A, S, x는 모두 전역 변수라고 가정
    # current_sum = 현재까지 선택된 A 값의 합
    current_sum = sum(A[i]*x[i] for i in range(n))
    if k == |A|: # x[0]-x[n-1]를 모두 결정함
        if current_sum == S: # 해 하나가 완성
            print(x) # 해 출력!
    else:
        # x[k] = 1인 경우 - A[k]가 선택된 경우
        if current_sum + A[k] <= S:
            x[k] = 1
            subsetSum_Backtrack1(k+1)

        # x[k] = 0인 경우 - A[k]가 선택되지 않은 경우
        x[k] = 0
        subsetSum_Backtrack1(k+1)

```

- A가 오름차순으로 정렬되어 있다고 가정 (아니라면 정렬 후 호출)
- 정렬되어 있다면 위의 코드 중 어떤 부분을 어떻게 수정하면 더 빨라질까? (아래 코드 참조)
- A = [3, 5, 6, 7, 8, 9, 10], S = 15인 예를 가지고 아래 코드를 실행해보자

subsetSum_Backtrack2(k): # A, S, x는 모두 전역 변수라고 가정

```

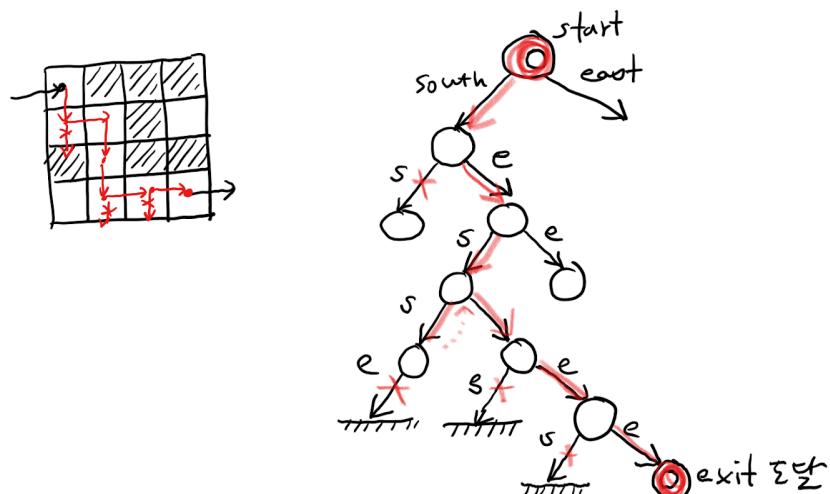
current_sum = sum(A[i]*x[i] for i in range(n))
if k == |A|: # x[0]-x[n-1]를 모두 결정되어 return
    return
else:
    if current_sum + A[k] <= S:
        # x[k] = 1인 경우 - A[k]가 선택된 경우
        x[k] = 1
        if current_sum + A[k] == S:
            # found a solution!
            print_subset()
    else:
        subsetSum_Backtrack2(k+1)

# x[k] = 0인 경우가 if 문 안으로 들어옴!
# 가능한 이유는? A가 오름차순으로 정렬되었기에
-----> x[k] = 0
-----> subsetSum_Backtrack2(k+1)

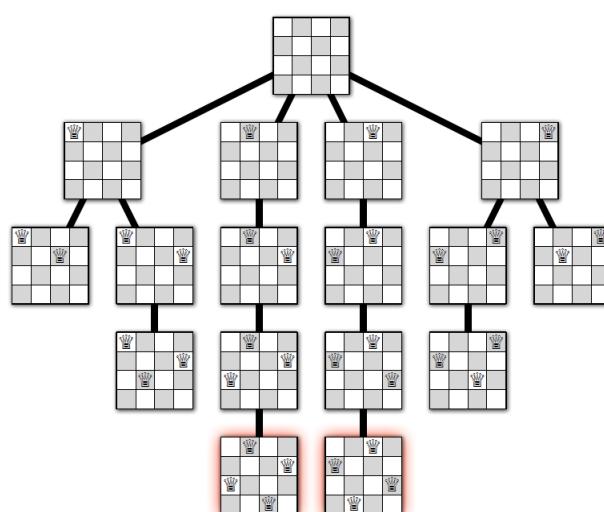
```

6. State Space Tree (상태공간트리)

- a. 상태공간트리는 초기 상태 (아무 것도 선택하지 않은 상태)를 루트 노드로 하고, 모든 가능한 경우를 리프 노드로 하는 이진 트리
- 노드에 연결된 에지는 $x[i]$ 값의 후보를 나타낸다. 해 $x[1], x[2], \dots, x[n]$ 를 n 개의 값을 $x[1]$ 부터 차례로 결정한다
 - 결국, 해에 도달하기 위해서는 해에 해당하는 리프 노드에 도달해야 한다. 만약 해가 없는 경우에는 최악의 경우에 모든 리프 노드 (가능한 모든 경우)를 방문해야 하기 때문에 가능한 모든 경우를 고려하는 단순 알고리즘과 본질적으로 같다
 - 경우에 따라서는 리프 노드까지 내려가지 않고도 해에 해당하는 노드를 확인할 수 있다 (예 3 - subset sum 문제 참고)
 - 상태 트리의 노드를 방문하는 순서는 DFS(Depth First Search, 깊이우선탐색)와 같다 (트리 또는 그래프에서의 DFS 설명 부분 참고)
 - 예 1: 미로탈출: 앞의 4×4 예제

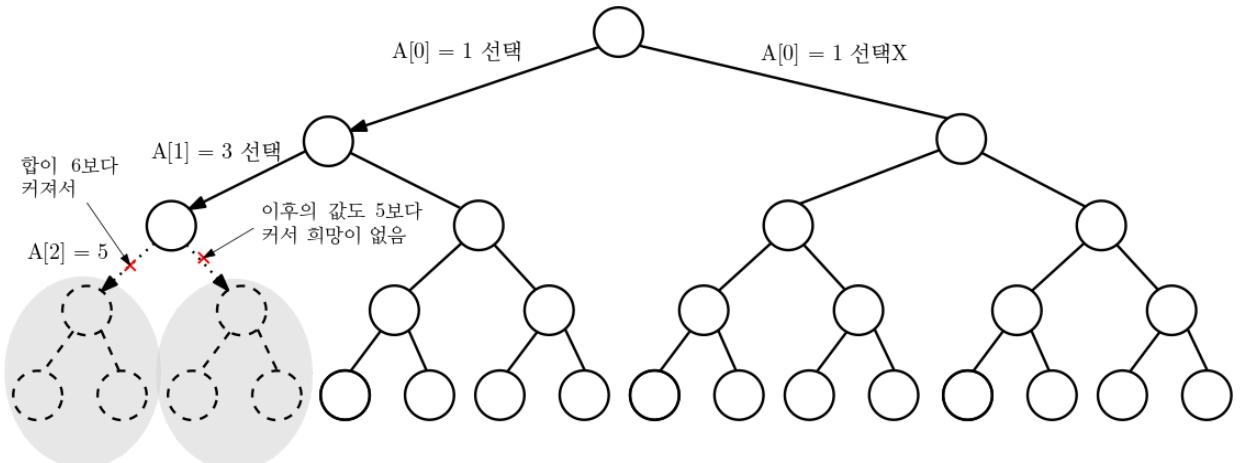


- v. 예 2: N-queens: $N = 4$

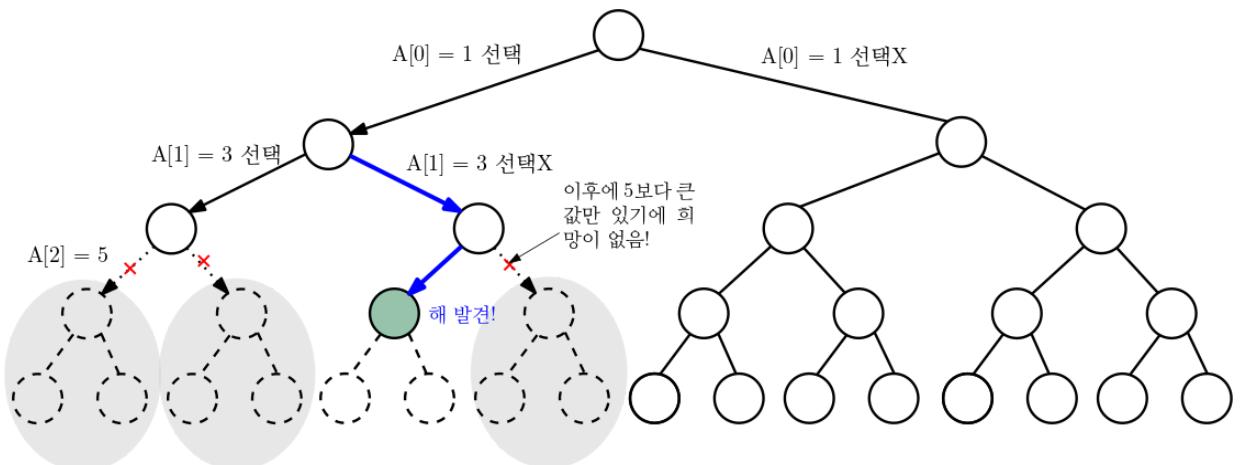


vi. 예 3: subsetSum: $A = [1, 3, 5, 6]$, $S = 6$

- subsetSum_Bactrack2 알고리즘의 결정을 상태 트리로 그려보자



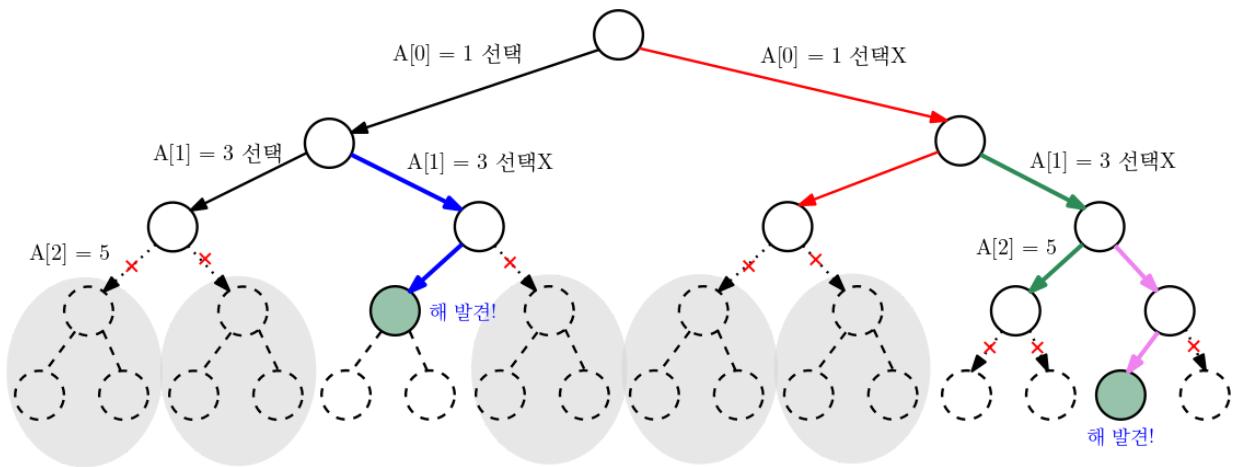
왼쪽에서부터 고려한다고 하면, $A[0] = 1$, $A[1] = 3$ 을 차례로 선택한다. 다음으로 $A[2] = 5$ 를 선택하는 경우에는 $1+3+5 > 6$ 이 되어 해당 에지를 따라 다음 노드로 진행할 필요가 없다. $A[2] = 5$ 를 선택하지 않더라도 이후에는 5보다 큰 값만 고려해야 하기 때문에 무조건 6보다 커져 내려갈 필요가 없다



다음으로 **파란색 화살표**를 따라가본다. $A[1] = 3$ 을 선택하지 않는 에지를 따라 내려가면 현재 1만 선택된 상태이고, 이 후에 $A[2] = 5$ 를 선택하는 에지를 따라 내려가면 합이 6이 되어 해에 해당하는 노드에 도착하게 된다. 사실, 해는 리프 노드에도 있지만, 리프까지 내려가지 않아도 지금처럼 중간에 해에 해당하는 노드를 쉽게 확인할 수 있다. 다른 해를 찾기 위해 탐색을 계속한다고 하면, $A[2] = 5$ 를 선택하지 않는 에지를 고려해야 한다. 이 에지는 따라갈 필요가 없다. 이후에 고려할 값들이 5보다 크고 따라서 6을 초과하기 때문에 이 에지 밑에는 해 노드가 존재하지 않음을 확신할 수 있기 때문이다.

이제는 루트 노드의 왼쪽 부트리는 전부 고려한 셈이 되었다. (이제 **빨간색 화살표**를 따라간다.) $A[0] = 1$ 을 선택하지 않는 루트 노드의 오른쪽 에지를 따라 가보자. 다시 $A[1] = 3$ 을 선택하는 에지를 따라간다. 현재까지 선택한 값이 3이므로 목표 값에 도달하지 않았기 때문에 무조건 따라가야 한다.

이후에 $A[2] = 5$ 를 선택하려고 했지만 $3+5 > 6$ 이 되어 따라갈 필요가 없고, $A[2] = 5$ 를 선택하지 않는 경우도 이후에는 5보다 큰 값만 고려해야 하기 때문에 따라갈 필요가 없게 된다



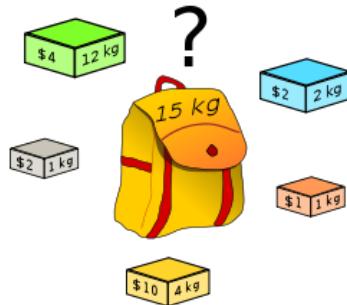
다음으로는 초록색 화살표와 보라색 화살표를 같은 방식으로 따라간다. 오직 6 값 하나만 선택하는 리프 노드에 도달하게 되는데, 이 노드가 두 번째 해가 된다

이 탐색에서 보듯이 상태트리의 모든 노드를 방문할 필요는 없다. A 의 값을 오름차순으로 정렬한 후 그 순서대로 고려하기 때문에 경우에 따라서는 어떤 노드 밑으로는 해가 존재하지 않음을 확신할 수 있고 그러면 방문하지 않아도 되어 탐색 시간을 절약할 수 있다. 백트래킹 알고리즘에서는 현재 방문중인 노드의 자손 중에 해가 있는지 없는지를 판별해 가지치기를 하는 게 중요하다. 가지치기를 많이 할 수록 탐색의 범위가 줄어들어 백트래킹 알고리즘의 수행시간이 짧아진다. 한마디로 백트래킹 알고리즘의 성능은 가지치기를 얼마나 효과적으로 할 수 있느냐에 달려있다

`subsetSum_Bactrack2` 알고리즘에서 가지치기의 기준이 되는 건, 고려 중인 에지를 따라 도달한 노드에서의 선택된 값의 합이 목표 값을 초과하는지의 여부이다. 초과하면 해 노드가 존재하지 않는다는 것이기 때문이다. 이 보다 더 효율적인 알고리즘을 위해서는 더 많은 가지치기가 가능한 기준을 마련하면 된다. 이러한 가지치기 기준을 **한계 함수(bounding function)**이라 부른다

7. [중요] 예제 3: Knapsack Problem

- a. 여러 물건의 무게와 가격이 주어지고, 배낭에 담을 수 있는 총 크기(용량 또는 무게) 제한 K 가 있다면, 배낭 용량 제한 K 를 넘지 않도록 물건을 골라 배낭에 담는 문제
- 목적: 선택한 물건의 가격(이익)의 합이 최대가 되도록 선택



- b. 여러 가지 버전의 문제 가능

- 0/1 knapsack 문제: 선택한 물건을 일부만 배낭에 넣을 수 없고 전체를 넣어야 함
- fractional knapsack 문제: 선택한 물건의 일부분만 잘라서 배낭에 넣을 수 있음

c. 동영상 강의 3편 ([완전정복 해보자!](#))



1/3



2/3



3/3

d. Fractional knapsack 문제

- $P = \text{가격} = [40, 30, 50, 10]$, $S = \text{크기} = [2, 5, 10, 5]$, $K = 16$
- 물건을 나눠 담을 수 있다는 조건이 중요!
 - 어떤 물건을 처음에 선택하는 게 좋을까? [선택기준은 [가성비](#)?]
- Greedy 알고리즘
 - 단위 크기당 가격이 가장 높은 물건을 선택해 최대한 많이 배낭에 넣는다
 - 예: 위의 예제에 적용해 보기
가성비가 각각 $20, 6, 5, 2$ 가 된다. 가장 높은 가성비를 갖는 물건부터 넣어보자. 처음 두 개를 모두 넣을 수 있다. 그러면 가격은 $40 + 30 = 70$ 이 되고 크기는 $2 + 5 = 7$ 이 된다. 세 번째 물건의 크기가 10이 되어 전체를 넣을 수 있다. 배낭의 남은 공간이 9이므로 9만큼만 잘라서 넣는다. 크기 대비 가격이 5이므로 크기가 9에 대해서는 가격이 45가 된다. 따라서 총 $70 + 45 = 115$ 가 된다
 - 이 선택 기준이 항상 답을 보장하는가? 그렇다. 일반적인 그리디 알고리즘의 정확성을 증명하는 방법 (귀류법)으로 어렵지 않게 증명 가능하다

e. 0/1 knapsack 문제

i. Greedy 알고리즘을 이 문제에 대해서도 사용할 수 있을까?

- 위의 예제에 적용해보자

- Greedy 답: 가성비 기준으로 차례로 넣으면, 첫 번째, 두 번째를 넣고, 세 번째는 크기는 10인데 남은 공간이 9라 넣지 못한다. 네 번째 물건은 통째로 넣을 수 있다. 따라서 $40 + 30 + 10 = 80$ 이 0/1 knapsack 문제의 답이다

- 정답: 실제 답은 첫 번째와 세 번째 물건을 선택하는 것이다. 크기의 합은 12이고 가격 합은 $40 + 50 = 90$ 이 된다

- 따라서 0/1 knapsack 문제를 가성비 그리디 기준으로 선택하는 그리디 방법은 올바른 답을 주지 못한다. (그런 입력 반례가 존재한다)

ii. F = fractional knapsack의 답, E = 0/1 knapsack의 답

- [질문] $F \geq E$? (맞다면 그 이유는?) 0/1 knapsack에서 선택한 물건을 가성비가 높은 순서대로 교체할 수 있음을 쉽게 보일 수 있다. 이렇게 교체된 물건은 fractional knapsack이 선택한 물건과 동일하다

iii. S 의 용량과 K 가 모두 양의 정수일 때: 동적계획법

- $DP[i][s]$ = 아이템 1부터 아이템 i 까지, 첫 i 개의 아이템 중에서 선택해 용량이 s 인 배낭에 넣을 때 선택된 아이템의 가격 합의 최대값
- 답이 아이템 i 를 선택할 수도 있고, 하지 않을 수도 있으므로, 두 가지 경우를 모두 해보고 가격 합이 최대인 것을 고른다
 - 아이템 i 를 선택한 경우: 남은 물건은 물건 1부터 ($i-1$)까지 ($i-1$)개에 대해, 용량이 $K - S[i]$ 인 배낭에 최대한 넣는 부분문제의 답을 이용 \Rightarrow (단, $K \geq S[i]$ 만족해야 함.)

$$DP[i-1][K-S[i]] + P[i]$$

- 아이템 i 를 선택하지 않은 경우: 경우: 남은 물건 (물건 1, ..., $i-1$) ($i-1$)개에 대해, 용량이 예전 그대로인 s 인 배낭에 최대한 넣는 부분문제의 답을 이용 $\Rightarrow DP[i-1][K]$

$$DP[i][K] = \max(DP[i-1][K-S[i]]+P[i], DP[i-1][K])$$

- 수행시간: $O(nK)$

- $O(nK)$ 에서 K 는 입력으로 주어지는 임의의 수이고, **입력의 크기가 아니다!** (입력의 크기 또는 입력의 길이는 n 임!) 입력의 크기가 아닌 항이 수행시간에 나타나기 때문에 유사 다항 시간(pseudo polynomial time)이 된다

iv. **P의 가격과 K가 모두 양의 정수일 때: 역시, 동적계획법 적용가능**

- $DP[i][p] =$ 아이템 1, ..., i 를 갖고 용량이 K 인 배낭을 채울 때 선택한 아이템의 가격 합이 p 이하인 경우의 **최소 크기 합**
- $DP[i][p] = \max(DP[i-1][p-P[i]] + S[i], DP[i-1][p])$

v. 일반적인 경우의 0/1 knapsack 문제: Backtracking 알고리즘

- 리스트 x 에 답을 저장: $x[i] =$ 아이템 i 가 선택되면 1, 아니면 0
- $x[i] = 1$ 또는 0이므로, 총 2^n 개의 경우 존재
- 모든 가능한 경우를 포함하고 있는 상태공간트리의 노드를 차례대로 방문하며 가장 좋은 선택을 계산하는 알고리즘
 - 여기서는 물건의 인덱스를 1부터 시작한다고 하자
 - $x[1] = 0, 1$ 인 경우 $\rightarrow x[2] = 0, 1$ 인 경우 $\rightarrow \dots \rightarrow x[n] = 0, 1$ 인 경우로 차례대로 결정하며 트리 탐색
- 몇 가지 표기 정의
 - $p(v) =$ 노드 v 에서 (루트부터) 선택된 물건의 가격의 합 (이익의 합)
 - $s(v) =$ 노드 v 에서 (루트부터) 선택된 물건의 크기의 합
 - $MP =$ 현재까지 가장 좋은 선택에 대한 가격의 합
 - $B(i+1)$
 - i. 물건 $i+1, \dots, n$ 에 대해, 배낭의 남은 크기를 넘기지 않으면서 추가로 선택할 수 있는 **최대 예상 이익**
 - ii. $B(i+1)$ 는 남은 물건에 대한 예상 이익의 최대 값이므로, 물건 $i+1, \dots, n$ 에 대한 fractional knapsack 알고리즘을 적용하여 얻은 가격의 합으로 정할 수 있음 (왜?)
- $x[1], \dots, x[i-1]$ 까지 결정한 상태에서 현재 노드 v 를 방문했다고 하자
 - 예를 들어, $B(i+1, size)$ 값이 20이라면, 남은 물건에 대해 취할 수 있는 예상 이익이 20을 넘을 수 없다는 의미다
 - 현재 $p(v)$ 는 현재까지 선택한 물건의 가격 합이다
 - 노드 v 의 두 자리를 차례로 따라 상태트리를 탐색해야 한다
 - $x[i] = 1$ 인 자리를 따라 간다면, 얻을 수 있는 총 이익은 $p(v) + P[i] + B(i+1)$ 를 넘을 수 없다. 만약, 이 값이 현재까지 얻은 가장 좋은 값인 MP 보다 크지 않다면, $x[i] = 1$ 인 자리를 따라 탐색할 이유가 전혀 없다 (탐색을 하더라도 현재 알고 있는 이익 합 MP 보다 더 큰 이익을 얻을 수 없기 때문에)
 - $x[i] = 0$ 인 자리를 따라 간다면, 얻을 수 있는 총 이익은 $p(v) + B(i+1)$ 가 된다. 마찬가지로 이 값이 MP 보다 크지 않다면 이 자리를 따라 탐색할 필요가 없다
 - 결국, 예상 이익이 MP 보다 클 때에만 탐색하면 된다

```

knapsack( i, size ): # x[i]=1/0 결정. 배낭의 남은 크기 = size

    if i > n or size <= 0: # 배낭에 들어 있는 (선택한) 물건을 출력
        print(x)
        return

    p(v) = sum( P[j] for j = 1..i-1 if x[j] == 1 )
    s(v) = sum( S[j] for j = 1..i-1 if x[j] == 1 )

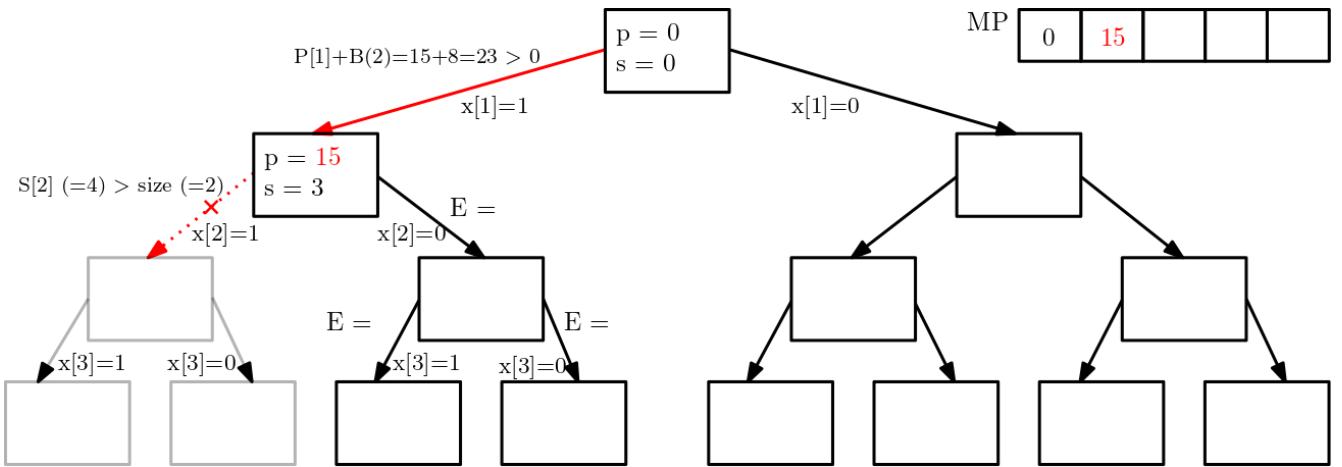
    # x[i] = 1을 따라가야하는지 결정
    if S[i] <= size: # 아이템 i가 크기 제한을 넘지 않아야
        B(i+1) = frac_knapsack(i+1, size-S[i])
        if p(v) + P[i] + B(i+1, size-s[i]) > MP:
            # Update MP
            if p(v) + P[i] > MP:
                MP = p(v) + P[i]
            x[i] = 1
            knapsack( i+1, size-S[i] )

    # x[i] = 0을 따라가야하는지 결정
    B(i+1) = frac_knapsack(i+1, size)
    if p(v) + B(i+1) > MP:
        x[i] = 0
        knapsack( i+1, size )

```

- vi. 다음 페이지에서 예제와 상태 공간 트리로 자세히 설명하고, 이해가 잘 되지 않는다면 동영상 시청을 추천한다

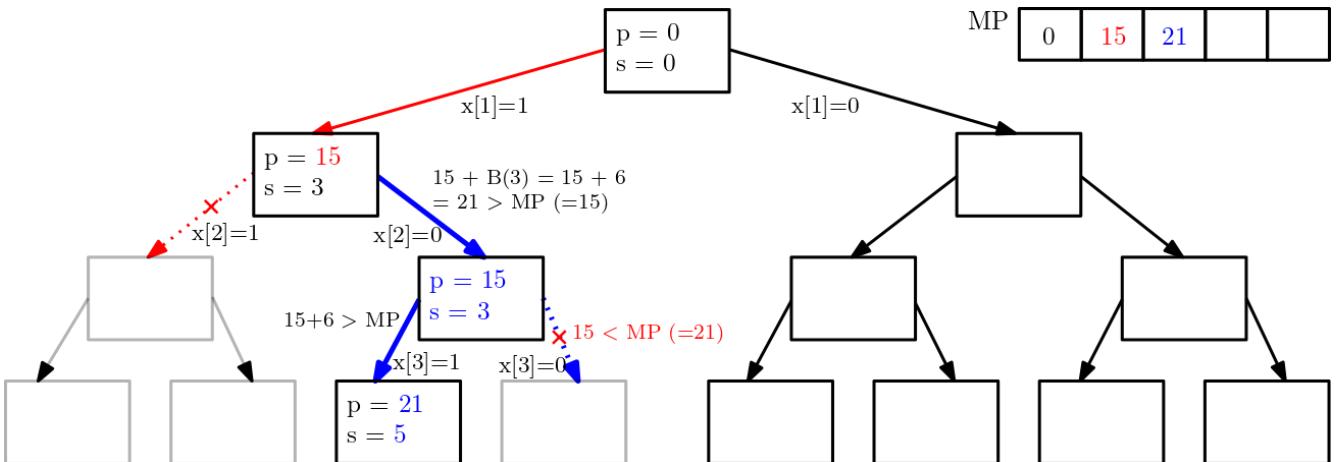
$$S = [3, 4, 2], P = [15, 16, 6], K = 5$$



아무것도 선택하지 않았으므로 $MP = 0$ 으로 초기화한다

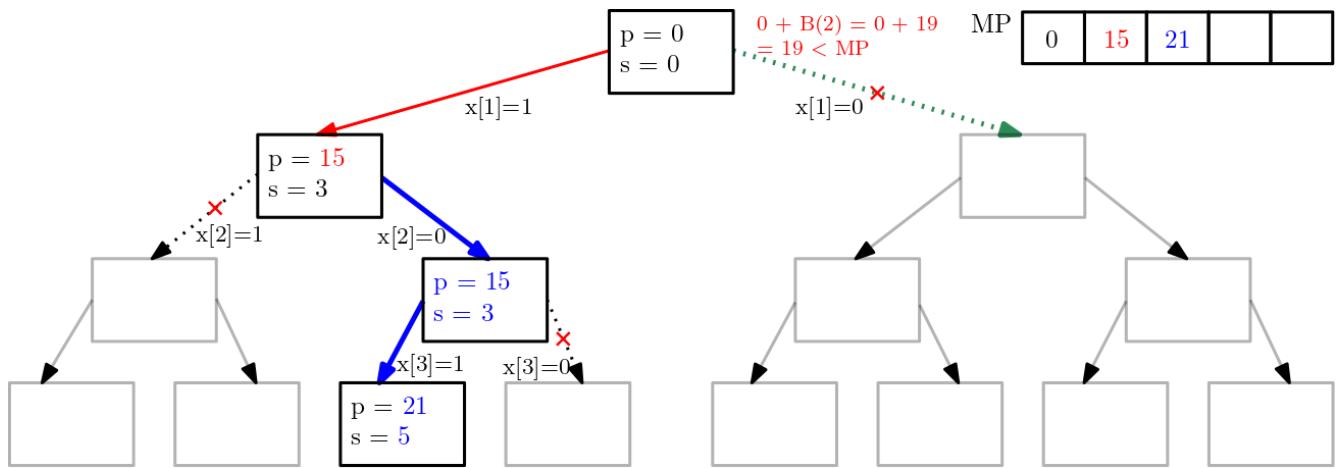
우선 첫 번째 물건을 선택하는 (**빨간색**) 에지를 고려한다. $S[1] = 3$ 이므로 배낭에 들어가므로 크기 조건은 만족한다. 남은 배낭 크기 2에 남은 물건 (두 번째와 세 번째)을 fractional knapsack으로 선택해 넣었을 때 얻을 수 있는 이익이 $B(2) = 8$ 이므로 $P[1] + B(2) = 15 + 8 = 23$ 이 에지를 따라 내려가 탐색을 통해 얻을 수 있는 최대 예상 이익이다. 그런데 현재까지 확보한 최대 이익은 $MP = 0$ 이므로 더 크다. 즉, 현재 최대 이익보다 더 큰 이익을 얻을 수 있을지도 모르므로 에지를 따라 내려가야 한다! 당연히 **MP = 15**로 업데이트되어야 한다

다음으로 두 번째 물건을 선택하려고 한다. 그런데 물건 크기는 4이고 남은 배낭 크기는 2이므로 크기 조건을 만족하지 않는다. 따라서 이 에지를 따라갈 필요가 전혀 없다. 그래서 아래의 세 노드는 탐색 범위에서 아예 제외된다



이제, 두 번째 물건을 선택하지 않는 경우를 고려한다. (파란색 화살표를 차례로 따라가보자) 그러면 남은 세 번째 물건을 fractional 조건으로 남은 배낭 크기 2에 넣는다면 $B(3) = 6$ 이 되어 현재 에지를 따라 내려가 얻을 수 있는 예상 이익은 최대 $15 + B(5) = 21$ 까지 될 수 있다. 이 예상 이익은 현재의 $MP = 15$ 보다 크기에 내려가봐야 한다. 다음으로, 세 번째 물건을 선택하는 에지를 따라 갈 수 있는지 보면, 세 번째 물건의 크기가 배낭의 남은 크기와 같기에 배낭에 넣을 수 있고 남은 물건이 더 이상 없기에 B 값은 계산할 필요가 없다.

따라서 $21 > MP = 15$ 이므로 내려가야 한다. 그리고 $MP = 21$ 로 업데이트 된다. 세 번째 물건을 선택하지 않은 경우는 최대 예상 이익이 MP 보다 작기에 내려갈 필요 없다



결국, 루트 노드의 왼쪽 부트리는 모두 고려되었고, 오른쪽 부트리를 탐색한다. 첫 번째 물건을 선택하지 않는 예지를 고려해야 하는데, 이를 위해서는 $B(2)$ 를 계산해야 한다. 남은 두 개의 물건을 남은 배낭 크기 5에 fractional하게 넣는 것이고 이 때 이익은 19가 된다. $19 < MP = 21$ 이므로 내려가봐야 현재 최대 이익보다 더 큰 이익을 얻을 수는 없음을 확신할 수 있다. 따라서 내려가지 않아도 된다. 최종적으로 상태 트리의 탐색이 완료되었다

이 예제의 경우에는 상태 트리에 총 15개의 노드가 있지만, 그 중에서 실제로 4개의 노드만 방문하고 나머지는 가지치기를 통해서 방문하지 않았다. 물론 입력에 따라 가지치기가 잘 되지 않아 거의 모든 노드를 방문해야 할 수도 있다

8. 예제 4: 최장 증가 부수열(문자열) 문제 (LIS: Longest Increasing Subsequence)

- a. 예를 들어, $A = [2, 8, 5, 10, 18, 13, 20, 4]$ 인 경우에 가장 긴 증가 부수열을 찾는 문제로 2, 5, 10, 18, 20와 8, 10, 13 모두 증가 부수열이지만, 앞의 부수열의 길이가 더 길며 모든 증가 부수열 중에서 제일 길다
- b. 이 문제는 DP 알고리즘을 이용하면,
 - i. $O(n^2)$ 시간에 해결할 수도 있고, 이진탐색을 함께 이용하면 $O(n \log n)$ 시간까지 줄일 수 있다
 - ii. DP 알고리즘은 본 교재의 동적계획법의 예 9에 자세히 설명되어 있다
- c. 비효율적이지만 backtracking으로도 해결할 수 있다
- d. Backtracking 알고리즘
 - i. 각 원소를 뽑아 부수열에 포함할지 안할지의 경우를 모두 고려하면, subset sum 문제처럼 총 2^n 가지의 경우가 존재
 - ii. 정답을 나타내는 리스트 x 를 정의: 위의 예에서는 $x = [1, 0, 1, 0, 1, 1, 0]$ 이 LIS 중 하나를 나타냄
 - iii. Backtracking 알고리즘의 틀을 그대로 따름:
 - lis = LIS의 길이를 저장하는 전역 변수
 - x = 해답을 나타내는 리스트 (모든 원소를 0으로 초기화)
 - $B(k)$: 한계 함수로 $A[k]$ 값을 선택해도 되는지 알려주는 함수 (True/False 리턴)
 - $A[k]$ 를 선택할 수 있으려면 현재까지 선택된 (왼쪽에 있는) 값들과 $A[k]$ 가 함께 오름차순이면 된다. 즉, $j = 0, \dots, k-1$ 에 대해, $x[j] = 1$ 인 $A[j]$ 가 오름차순이고 마지막 값보다 $A[k]$ 가 더 크면 $A[k]$ 를 선택할 수 있다

LIS(k):

```

if k >= n:
    return

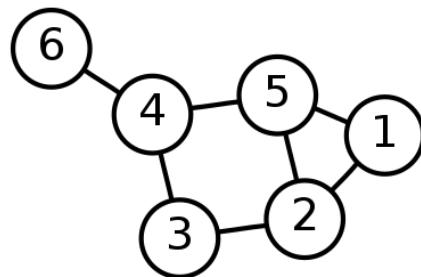
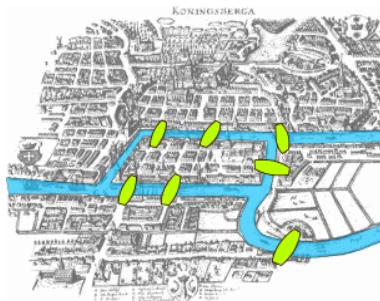
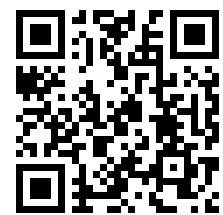
# x[k] = 1 → A[k]를 선택하는 경우
if B(k) == True: # B(k)는 어떤 조건을 검사해야 할까?
    x[k] = 1
    update lis
    LIS(k+1)
# x[k] = 0 → A[k]를 선택하지 않는 경우
x[k] = 0
LIS(k+1)

```

9. Graph algorithm

1. 두 노드 사이의 관계가 있는 경우 에지로 연결하여 표현하는 추상적이고 일반적인 자료구조

- a. 트리는 사이클이 없는 그래프이고, 연결리스트는 하나의 경로로 이루어진 트리이므로, 가장 단순한 형태의 그래프이다



2. 그래프 $G = (V, E)$

- a. V = 노드(node) 또는 정점(vertex) 집합을 의미. 그래프 이론 분야에서는 "정점" 선호
 b. E = 두 노드의 쌍으로 정의. 만약 $(u, v) \in E$ 라면 노드 u 와 v 가 서로 (방향이 없는) 에지로 연결되어 있다고 한다
 c. 위의 왼쪽 그림: (Wikipedia/Graph) Euler의 [Königsberg](#)의 다리로 유명

The paper written by [Leonhard Euler](#) on the [Seven Bridges of Königsberg](#) and published in 1736 is regarded as the first paper in the history of graph theory

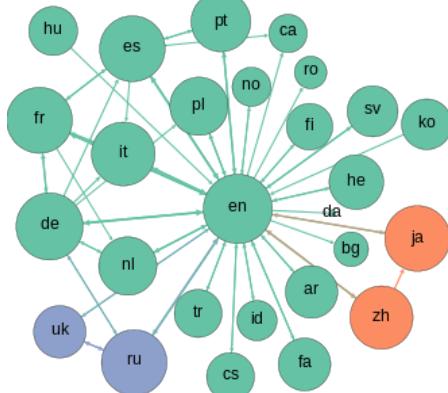
- d. 위의 오른쪽 그림: 지역 6곳을 노드로 정의하고 두 지역을 연결하는 다리가 존재하면 에지로 정의할 수 있다. 그러면 추상적인 "그래프"가 정의된다. 노드와 에지 집합은 아래와 같다
- $V = \{1, 2, 3, 4, 5, 6\}$ # 노드 번호를 0부터 시작해도 된다
 - $E = \{(1,2), (3,2), (1,5), (2,5), (4, 5), (6, 4), (3,4)\}$

3. 응용분야(applications)

- a. 주위의 거의 모든 관계를 그래프로 표현 가능하기에 응용 분야는 무궁무진하다
 b. Computer Science, Linguistics, Physics and Chemistry, Biology, Social Science



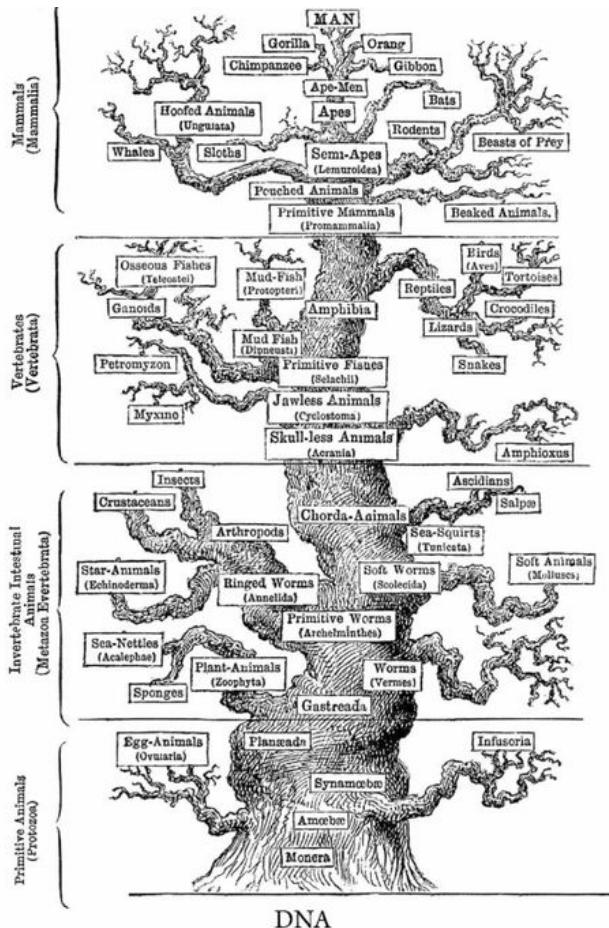
[US domestic-flights map from [Graphminator](#)]



[graph on web sites from Wikipedia]

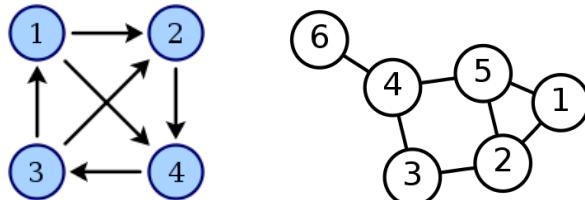


[subway map]

[Phylogenetic tree: [출처](#)]

4. 기본 용어 (basic graph terminology)

- a. 정점(vertex), 노드(node)
- b. 에지(edge), 링크(link)
 - i. 방향/무방향 에지 (directed/undirected edge)
- c. 무방향 그래프, 방향 그래프
 - i. 무방향 에지로 구성된 그래프와 방향 에지로 구성된 그래프



- d. 분지수(degree)
 - i. in-degree **in-deg(u)**: 노드 u로 들어오는 방향 에지의 개수
 - ii. out-degree **out-deg(u)**: 노드 u에서 나가는 방향 에지의 개수
 - iii. degree of vertex **deg(u)**: 노드 u에 연결된 무방향 에지의 개수
 - iv. degree of graph **deg(G)**: 무방향 그래프 G의 최대 노드 분지수
- e. 인접성(adjacency, incidence)
 - i. 에지 (u, v) 가 존재하면, u 와 v 는 서로 인접(adjacent)하다고 말함
 - ii. 에지 $e = (u, v)$ 에 대해, e 는 u 와 v 에 인접(incident)하다고 말함
- f. 경로(path)
 - i. 한 노드에서 다른 노드로 연결되는 선형 경로로 경로에 포함된 노드는 모두 달라야 한다 (방향 그래프에서의 경로는 한 방향 - 일방 통해 경로를 의미한다)
 - ii. 경로의 길이는 에지에 가중치 값이 없는 경우는 경로의 에지의 개수로 정의되고, 가중치 값이 있는 경우는 에지의 가중치 합으로 정의된다
- g. 에지/정점 가중치(edge/vertex weight)
 - i. 가중치는 주로 비용(cost)의 역할을 한다. 예를 들어, 그래프의 두 노드를 연결하는 경로의 길이는 경로에 포함된 가중치의 합으로 정의한다. 최단 경로는 경로 길이가 가장 짧은 경로로 정의된다
- h. 사이클(cycle)
 - i. 출발 노드와 도착 노드가 동일한 경로로 정의된다. 즉, 한 노드에서 출발해 해당 노드로 도착하는 원형 경로를 의미한다
- i. 트리(tree): 사이클이 없는 연결 그래프
 - i. 포리스트(forest): 하나 이상의 연결 트리의 집합
- j. 부그래프(subgraph)
 - i. 그래프의 정점 집합의 부분 집합과 그 부분 집합에 속하는 정점 사이에 정의된 에지 집합으로 정의되는 부분 그래프이다. 그래프가 트리인 경우에는 부트리 (subtree)라고 부른다
 - ii. 집합(set)과 부분집합(subset)의 관계와 유사하다

k. 연결성(connectedness)

- i. 두 노드가 연결.connected 되어 있다는 것은 두 노드를 연결하는 경로가 존재한다는 뜻이다
- ii. 그래프가 연결.connected 되어 있다는 것은 그래프의 모든 노드 쌍이 연결되어 있다는 뜻이다
- iii. 그래프의 연결 성분.connected component은 그래프의 연결된 부그래프를 의미한다. 그래프는 한 개 이상의 연결 성분으로 구성된다

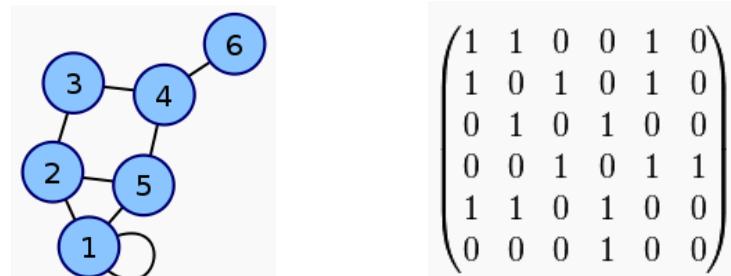
l. 신장 그래프(spanning subgraph)

- i. 부그래프 중에서 그래프의 모든 노드가 등장하는 부그래프를 의미한다

5. 그래프 표현 방법(graph representation)

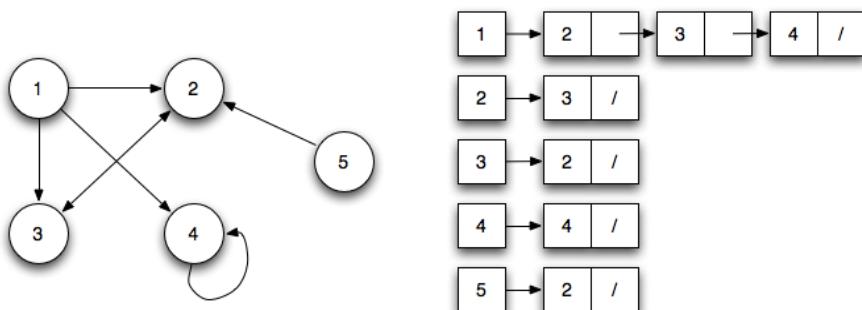
a. 인접행렬(adjacency matrix): 인접성을 행렬(2차원 배열/리스트)로 표현

b. 인접리스트(adjacency list): 정점에 인접한 에지만을 연결리스트로 표현



Python 이차원 리스트 표현:

```
G = [ [1,1,0,0,1,0], [1,0,1,0,1,0], [0,1,0,1,0,0],
      [0,0,1,0,1,1], [1,1,0,1,0,0], [0,0,0,1,0,0] ]
```



Python 리스트로 표현:

```
G = [ [], [2, 3, 4], [3], [2], [4], [2] ] # 정점 번호가 1번부터
```

c. 표현법에 따른 기본 연산 시간 비교 (표현법에 따른 장단점 파악 필요)

i. $n = |V|$ = 노드 개수, $m = |E|$ = 에지 개수

ii. Python 리스트에 저장한다고 가정

기본연산	인접행렬	인접리스트
(u, v)가 에지인가?	$G[u][v] == 1$ $O(1)$	$G[u].search(v) != None$ $O(n)$
u 의 인접한 모든 에지 (u, v)에 대해:	$G[u][v]$ for v in range(n) $O(n)$	for edge in $G[u]$ $O(\deg(u))/O(\text{out-deg}(u))$
새 에지 (u, v) 삽입	$G[u][v] \leftarrow 1$ $O(1)$	$G[u].pushFront(v)$ $O(1)$
에지 (u, v) 제거	$G[u][v] \leftarrow 0$ $O(1)$	$x \leftarrow G[u].search(v)$ $G[u].remove(x)$ $O(\deg(u))/O(\text{out-deg}(u))$
G 를 위한 메모리	$O(n^2)$	$O(m)$

d. 질문: 인접 리스트의 각 노드의 에지 리스트를 위해 연결 리스트 (또는 파이썬 리스트)가 아닌 해시 테이블을 사용하면 기본 연산 시간은 어떻게 될까?

i. 해시 테이블의 삽입, 삭제, 탐색은 충돌 확률이 테이블 슬롯 수에 반비례하고 빈 슬롯의 비율이 일정 수준으로 유지된다는 가정하에 평균 $O(1)$ 이다

6. Traversal: DFS, BFS



(1/2)



(2/2)

a. DFS(Depth First Search: 깊이 우선 방문)

- i. 현재 방문 노드에서 방문하지 않은 이웃 노드가 있다면 방문하는 방식

1. 재귀적인 방식

```
RecursiveDFS(v): # visiting v now!
    mark v as visited node
    for each edge (v, w):      # 에지 순서는 임의로~
        if w is unmarked:      # 인접한 노드 w가 미방문이면
            RecursiveDFS(w)
```

2. 비재귀적인 (반복) 방식

IterativeDFS(s):

```
stack.push(s)    # 나중에 방문할 노드를 스택에 대기시킴
while stack is not empty:
    v ← stack.pop()
    if v is unmarked:
        mark v as visited node
        for each edge (v, w):
            if w is unmarked:
                stack.push(w)
```

- ii. **수행시간:** 각 노드 v를 방문할 때, v에 인접한 미방문 노드 w가 스택에 push되고 적당한 때에 다시 pop된다. 즉, 에지 (v, w)에 대해 한 번의 push와 한 번의 pop이 이루어진다고 해석해도 된다. push, pop 스택 연산은 $O(1)$ 시간이 필요하므로 $O(m)$ 시간이면 충분하다. 추가로 노드 개수만큼의 시간 역시 필요하므로 정리하면 $O(n + m)$ 시간이 된다

- iii. [중요] DFS를 하면서 각 노드의 첫 방문 시간과 최종 방문 시간 기록하기 + 방문 순서 (부모 노드) 기록하기

1. 세 개의 리스트를 준비

- $\text{pre}[v]$ = 노드 v를 첫 방문한 시간을 저장
- $\text{post}[v]$ = 노드 v를 통해 방문할 이웃 노드가 없는 시간, 즉 v의 입장에서 DFS가 완료되는 시간을 저장
- $\text{parent}[v]$ = 노드 v를 방문하기 직전의 노드 (v의 입장에서 보면, 부모 노드)를 저장 (예: 노드 u를 거쳐 v를 방문했다면 $\text{parent}[v] = u$ 가 됨)

```

DFS(v):
    # (*)
    visited[v] = True    # mark v as a visited node
    pre[v] = curr_time    # record the first visit time
    curr_time += 1
    for each edge (v, w):
        if visited[w] == False:    # w is not visited yet
            parent[w] = v
            DFS(w) # Recursive version
        post[v] = curr_time # record the finish time
        curr_time += 1

DFSAll(G):
    for all nodes v:
        visited[v] = False    # initially not visited at all
        parent[v] = -1         # initially all parents are set as -1
        curr_time = 1           # time starts from 1
        count = 0                # 연결 성분 번호 나타내는 변수 (뒤에 설명)
        for all nodes v:
            if visited[v] == False:    # v is not marked → DFS at v
                count += 1    # 연결 성분 번호 증가
                DFS(v)

```

비재귀 코드: 아래 코드에서 pre와 post 값은 어떻게 지정해야 할까?

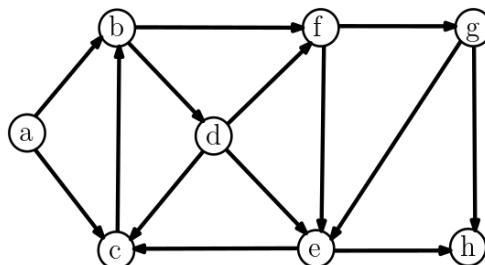
```

DFS(s):
    stack.push((None, s))    # tuple (parent, curr_node)
    while stack is not empty:
        p, v ← stack.pop()
        if visited[v] == False:
            visited[v] = True
            parent[v] = p
            for each edge (v, w):
                if visited[w] == False:
                    stack.push((v, w))

```

- iv. 예제: 노드 a부터 시작하고, 노드 번호가 작은 인접 노드부터 방문한다고 가정하자
 1. 아래 그래프 DFS 순서 (pre 값의 오름차순): a, b, d, c, e, h, f, g

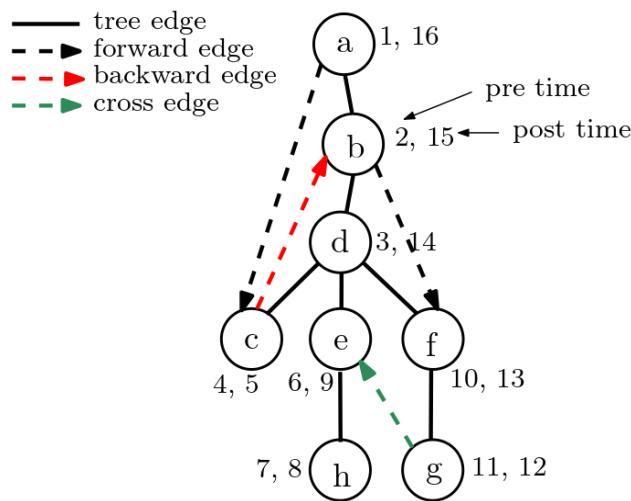
2. pre, post 시간은 다음 페이지 DFS 트리와 함께 표시되어 있음



v. DFS 트리

1. DFS 시작 노드가 루트 노드가 되며, 방문을 하면서 정의되는 부모 노드 - 자식 노드 관계에 의해 정의되는 트리가 DFS 트리이다
2. 신장 트리(spanning tree)이다. 즉, 그래프의 모든 노드가 연결된 트리이다.
3. DFS 트리의 나타난 에지 (u, v) 를 네 개의 타입으로 구별한다
 - **tree edge** - DFS 트리를 구성하는 에지
 - **forward edge** - 트리 에지가 아닌 조상 \rightarrow 자손으로 향하는 에지
 - **backward edge** - 트리 에지가 아닌 자손 \rightarrow 조상으로 향하는 에지
 - **cross edge** - DFS 트리, forward, backward도 아닌 그래프의 에지

위의 왼쪽 그래프에 대한 DFS 트리와 에지 종류는 아래 그림과 같다 (주의: 트리 에지는 모두 그렸지만 나머지 에지는 일부만 표시했음)



4. 무방향 그래프에서는 backward 에지와 forward 에지는 구별되지 않는다
5. $\text{pre}[u]$, $\text{pre}[v]$, $\text{post}[u]$, $\text{post}[v]$ 값의 대소 관계를 통해, 구별 가능하다. 어떻게 하면 될까?
 - tree edge: $\text{parent}[v] = u$
 - forward edge: $\text{pre}[u] < \text{pre}[v] < \text{post}[v] < \text{pre}[u]$
 - backward edge: $\text{pre}[v] < \text{pre}[u] < \text{post}[u] < \text{pre}[v]$
 - cross edge: $\text{pre}[v] < \text{post}[v] < \text{pre}[u] < \text{post}[u]$
6. 결국, pre 값과 post 값의 대소 관계를 통해 상수 시간에 forward, backward, cross 에지를 구분할 수 있다
7. 이런 구분이 어떻게 유용하게 쓰일 수 있을까?
 - 무방향/방향 그래프의 사이클(cycle)이 있는지 검사하고 싶을 때에는 backward 에지가 존재하지만 검사하면 된다. 왜? backward 에지는 자손 노드에서 조상 노드로 향하는 에지이다. 그런데 조상 노드에서 자손 노드까지는 트리 에지로 연결된 경로가 존재한다. 결국 조상 노드와 자손 노드를 연결하는 사이클이 존재한다는 의미이다

vi. DFS 응용

1. 연결 성분(connected component)을 구분해서 방문하기

- 연결된 부그래프를 의미하며 그래프는 하나 또는 그 이상의 연결 성분으로 구성된다
- 연결 성분을 구별해서 구하는 문제는 기본적인 그래프 문제로 연결 성분의 연결 정도에 따라 다르게 정의하기도 한다 (예: 2-connected component 등)
- DFS를 그대로 이용해서, 연결 성분의 번호를 count 전역 변수를 나타낸다. 각 노드가 속한 연결 성분의 번호를 기록하기 위해 comp라는 리스트를 준비한다
- 재귀 DFS 코드의 첫 줄에 `comp[v] = count` 기록하면 된다

2. 미로 탈출: 이 경우엔 동, 서, 남, 북으로 모두 이동 가능!

- DFS는 backtracking 방법과 동일하다!!
- 미로는 이차원 리스트 M에 저장되어 있음
- 입력 형식:
 - i. 1은 장애물, 0은 빈 칸 의미
 - ii. 바깥쪽 경계는 모두 1로 하여 외부로 나가지 못함
 - iii. 7 x 7 미로 예: (1, 1)이 입구, (5, 5)가 출구라는 의미

```

7
1 1 5 5
1111111
1000001
1111101
1000101
1011101
1000001
1111111

```

- 출력 형식:

- i. s = 입구 표시, f = 출구 표시, *는 탈출 경로 표시
- ```

1111111
1s****1
11111*1
10001*1
10111*1
10000e1
1111111

```

```

find_way_from_maze(r, c) # 현재 칸 (r, c) 방문 중

 visited[r][c] = True
 if (r, c) == exit:
 return True

 if 동쪽 이웃 칸이 빈 칸이고 미방문이라면:
 if find_way_from_maze(r, c+1):
 M[r][c+1] = '*'
 return True

 if 남쪽 이웃 칸이 빈 칸이고, 미방문이라면:
 ...
 if 서쪽 이웃 칸이 빈 칸이고, 미방문이라면:
 ...
 if 북쪽 이웃 칸이 빈 칸이고, 미방문이라면:
 ...
 return False

n = 미로 크기 (가장 자리 경계 제외)
입구 (sx, sy), 출구 (ex, ey) 입력, 미로 M 입력
visited = 초기화
find_way_from_maze(1, 1) # 호출

```

## vii. 사이클 찾기

### 1. 무방향 그래프에서는?

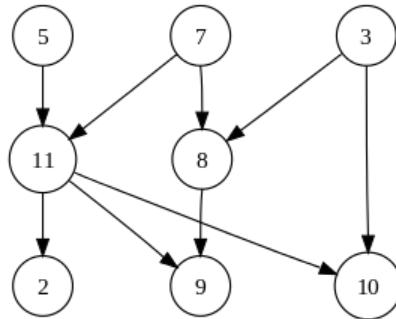
- backward 에지가 존재하는지 검사하면 된다
- **방법 1:** pre, post 시간을 구해서 backward 에지 존재하는지 검사
- **방법 2:** DFS에서  $v \rightarrow w$ 로 갈 때,  $\text{visited}[w] == \text{True}$ 라면  $w$ 를  $v$ 를 방문하기 전에 방문을 이미 한 후에  $v$ 에 도달했다는 의미다. 즉,  $w$ 에서  $v$ 까지의 경로가 존재한다는 것이고 다시  $v \rightarrow w$  에지가 있으므로 사이클이 존재함을 알 수 있다. (사실,  $v \rightarrow w$  에지는 backward 에지이다)

### 2. 방향 그래프에서는?

- 무방향 그래프에서의 알고리즘과 원칙적으로 같다
- 사이클이 없는 방향 그래프를 DAG(Directed Acyclic Graph)라 부르고 매우 많이 분야에서 등장한다
- DAG라면, outgoing 에지만 있는 노드와 incoming 에지만 있는 노드가 반드시 하나 이상 존재해야 한다. outgoing 에지만 있는 노드를 source 노드라 부르고, incoming 에지만 있는 노드를 sink 노드라 부른다

viii. 위상 정렬 (Topological Sort): DAG의 노드들을 순서에 따라 정렬해보자

1. DAG는 아래 그래프와 같이 사이클이 없는 방향 그래프이다. 이 DAG에는 세 개의 source 노드가 세 개의 sink 노드가 존재한다
2. 부품을 조립하는 조립 공정을 그래프로 표현하면 DAG가 된다. 소스 노드에서 다음 노드로 부품을 전달하고, 해당 노드에서는 incoming 에지를 통해 전달받은 부품을 조립해 outgoing 에지를 통해 다음 노드들로 전달한다. 이런 공정에서는 사이클이 존재할 수 없다. 존재한다면 공정이 사이클에서 끝나지 않고 영원히 반복될 것이기 때문이다
  - 아래 그래프에서 노드 11은 노드 5와 7에서 부품을 받아 조립해 결과물을 노드 2, 9, 10에 전달한다
  - 노드 5, 7, 3은 들어오는 에지가 없으므로 소스 노드이고, 2, 9, 10은 나가는 에지가 없으므로 싱크 노드이다



5, 3, 7, 8, 11, 2, 9, 10

3, 5, 7, 8, 11, 9, 10, 2

...

3. 조립 과정을 정렬을 해보자. 값의 대소 관계가 정의될 때 정렬이 정의된다. 정수와 실수 값은 정렬할 수 있는 건, 두 수의 대소가 정확히 정의되기 때문이다. 문자열도 사전순서에 따라 대소가 정의되므로 정렬이 가능하다. 조립 과정의 정렬 역시 두 노드의 대소가 정의되어야 한다. 노드  $v$ 에서 노드  $w$ 로 에지가 있다면  $v$ 의 부품이 먼저 완성되어야 그 부품을  $w$ 가 받아 조립을 시작할 수 있다. 따라서 순서적으로  $v$ 가  $w$ 가 앞선다. 즉  $v < w$ 의 대소 관계가 성립한다
4. 문제는 모든 노드 쌍 사이에 대소 관계가 정의되지 않는다는 것이다. 예를 들어, 소스 노드 사이에는 대소 관계가 없다. 마찬가지로 싱크 노드 사이에도 없다. 노드 5와 8 사이에도 대소 관계가 정의되지 않는다. 이렇게 모든 쌍에 대소 관계가 정의되지 않고 일부에 대해서만 정의되는 관계를 partial order라 부른다. DAG은 일종의 partial order를 나타낸다고 말할 수 있다
5. Partial order에서도 정렬을 할 수 있다. 그러나 정렬의 결과가 하나 이상일 수 있다. 위의 그래프의 정렬 순서는 아래와 같다

{5, 3, 7}, {8, 11}, {2, 9, 10}

6. {5, 3, 7}의 노드가 {8, 11}의 노드보다는 무조건 먼저 와야 하고, {8, 11}의 노드가 {2, 9, 10}의 노드보다 먼저 와야 한다. 그러나 {5, 3, 7}의 노드 사이에는 선후관계가 없기 때문에 어떻게 나열되도 상관없다. {8, 11}, {2, 9, 10}의 노드 사이에도 순서는 상관없다. 따라서 아래 정렬은 모두 올바른 순서이다

```
5, 3, 7, 8, 11, 2, 9, 10
3, 5, 7, 8, 11, 9, 10, 2
...
```

7. DAG에서의 정렬을 위상 정렬(topological sort, topological ordering)이라 부른다
8. **Algorithm 1:**

소스 노드들부터 차례대로 다음에 조립이 가능한 노드들을 찾아가는 방법: 가장 먼저 오는 노드는 in-deg가 0인 노드들이고 이 노드는 소스 노드들이다. 이 소스 노드들에서 나가는 에지(outgoing edge)를 지운 후, 노드들의 in-deg를 업데이트한다. 다시 in-deg가 0이 되는 노드들이 다음에 오면 된다. 이 방식으로 싱크 노드들이 남을 때까지 반복하면 된다

**TopologicalSort(G):**

```
S = []
for i in range(|V|):
 v ← any source node in G # how to find v?
 S.append(v)
 delete all outgoing edges from v
 # in-degrees of vertices are updated accordingly
return S
```

단점: "v의 outgoing 에지를 지운다"는 부분을 어떻게 구현해야 하나? in-deg를 일관성있게 update해야 하고, in-deg가 0이 되는 노드들의 리스트에 관리해야 한다

9. **Algorithm 2:**

**힌트:** DFS에서 가장 처음으로 DFS가 완료되는 노드는 post 값이 가장 작은 노드이다. 나가는 에지가 없기 때문이다. 따라서 이 노드는 무조건 sink 노드이다! 결국, post 값이 작아지는 순서 (내림차순)로 나열하는 것이 위상 정렬 순서이다? YES!

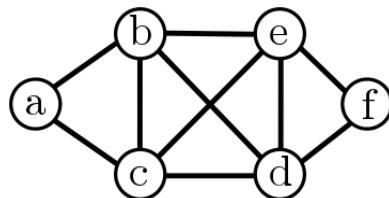
**TopologicalSort(G):**

1. add a new source node s with edge to all vertices
2. DFS(s) # post 값도 동시에 계산함
3. (s를 제외하고) post 값의 내림차순으로 노드 번호를 출력

10. **Algorithm 1 vs. Algorithm 2 (수행시간을 비교해보자)**

b. BFS(Breadth First Search: 너비 우선 방문)

- i. 현재 방문 노드에서 방문하지 않은 이웃 노드를 차례대로 방문하는 방식



- ii. 위의 예: 노드 a부터 출발한다면,  $a \rightarrow \{b, c\} \rightarrow \{e, d\} \rightarrow e$  순으로 출발 노드부터의 거리 순으로 노드들을 방문하게 된다. 즉, 출발 노드로부터 거리가 1인 (에지 하나로 갈 수 있는) 노드를 모두 방문하고, 거리가 2인 노드를 (거리가 1인 노드들의 이웃 노드들)을 모두 방문하는 식으로 진행한다
- iii. 큐(queue)  $Q = \{a\}$ 를 준비하여,  $v = Q.dequeue()$ 하여 노드  $v$ 를 방문하고,  $v$ 의 인접한 미방문 노드  $w$ 를 모두  $Q.enqueue(w)$ 하는 단계를 반복하면 된다
- iv. 리스트  $dist[v]$ 는 출발 노드로부터 거리 (출발 노드와  $v$ 를 잇는 경로의 에지 개수)가 저장된다

**BFS(G):**

```

visited = [False] * n # BFS 중간에 방문했는지를 기록
parent = [-1] * n # BFS 트리에서의 parent 기록
dist = [0] * n # source 노드로부터의 최단 거리 기록
for all source nodes s in G:
 Q.enqueue(s)
 visited[s] = True
 while Q is not empty:
 v = Q.dequeue()
 for each edge v → w:
 if not visited[w]:
 Q.enqueue(w)
 visited[w] = True
 parent[w] = v
 dist[w] = dist[v] + 1

```

- v. 수행시간: DFS와 유사하게 각 에지  $(v, w)$ 에 대해,  $w$ 가 한 번씩 enqueue, dequeue되기에  $O(n + m)$  시간이면 충분하다

## 7. 최단 경로 문제(Shortest Path Problem)

- a. 일상생활에서 자주 경험하는 문제로, 문제 자체의 역사도 깊고 응용 가치도 커 많은 연구가 이루어진 그래프 분야의 고전 문제이다
- b. 에지에 가중치(비용)가 주어진 방향 그래프를 대상으로 한다
  - i. 단, 가중치는 모두 양수라고 가정한다. 가중치가 음수인 경우는 실제로 발생하는 경우가 적고, 음의 가중치를 갖는 에지가 있다면 이를 허용하는 최단 경로 알고리즘을 설계해야 한다
  - ii. 에지를  $u \rightarrow v$ 라고 하면, 가중치는  $\text{weight}(u \rightarrow v)$  또는  $\text{weight}(u, v)$ 로 표기한다
- c. 출발 노드  $s$ 가 입력으로 지정되고, 출발 노드  $s$ 에서 다른 모든 노드까지의 최단 경로를 찾는 문제를 다룬다 (이 문제를 **Single Source Shortest Path Problem**이라 부른다)
  - i. 출발 노드와 도착 노드를 지정해 하나의 최단 경로를 찾지 않는 이유는 출발 노드에서 다른 모든 노드로의 최단 경로를 찾는 복잡도와 다르지 않기 때문이다
- d. 최단 경로의 기본 성질:
  - i.  $u \rightarrow v$ : 노드  $u$ 에서 노드  $v$ 로의 하나의 에지로 구성된 경로
  - ii.  $s \rightarrow v$ : 노드  $s$ 에서  $v$ 로의 경로를 표시하고, 중간에 여러 노드가 존재 가능
  - iii.  $s \rightarrow u \rightarrow v$  인 경우, 이 경로에서  $u$ 는  $v$ 의 predecessor 또는 parent이다
  - iv. [중요한 사실] 만약  $s \rightarrow u \rightarrow v$  가  $s$ 에서  $v$ 로의 최단 경로 중 하나라면,  $s \rightarrow u$  역시  $s$ 에서  $u$ 까지의 최단 경로 중 하나이다
    - 1.  $s$ 에서  $v$ 까지의 여러 경로 중에서  $u$ 를 통해  $v$ 에 도달하는 경로 중에 최단 경로가 있다는 의미이고,  $s \rightarrow u$ 를 먼저 계산하여 알고 있다면,  $s \rightarrow u \rightarrow v$ 는 쉽게 계산할 수 있다는 뜻이다
    - 2.  $\text{dist}[v] = "s$ 에서  $v$ 까지의 최단 경로의 길이"라고 정의하면,

```
dist[v] = min for_each u | u→v { dist[u] + weight(u→v) }
```

  - 3. 즉,  $v$ 로 들어오는 각 에지  $(u, v)$ 에 대해,  $\text{dist}[u] + \text{weight}(u \rightarrow v)$ 를 계산하고 그 중에서 최소 값이  $\text{dist}[v]$ 가 된다는 의미이다
  - 4. 어라? 이건 DP 식인데… 맞다!
  - 5. 그러면,  $\text{dist}[v]$  계산 전에  $\text{dist}[u]$ 가 먼저 계산되면 된다. 즉,  $v$ 의 predecessor  $u$ 에 대한  $\text{dist}[u]$ 를 먼저 계산하면 된다.
  - 6. 이런 논리를 계속 적용해나가면, 노드  $s$ 까지 거슬러 올라가게 된다
  - 7. 당연히  $\text{dist}[s] = 0$ 이고,  $s$ 가 아닌 노드  $v$ 에 대해선,  $\text{dist}[v] = \infty$ 로 초기 값을 갖는다.

8. s에서 나가는 에지 중에서, 최소 가중치를 갖는 에지가  $s \rightarrow v$ 라면,  $dist[v] = dist[s] + weight(s \rightarrow v)$ 이 되는데, 여기서  $dist[s] = 0$ 이므로  $dist[v] = weight(s \rightarrow v)$ 가 된다

- 왜 그럴까?  $s \rightarrow u \rightarrow v$ 로 가는 경로가 v의 최단경로가 될 수도 있지 않을까? 만약 이게 사실이라고 하자.  $weight(s \rightarrow v)$ 가 s에서 나가는 에지 중에서 최소 가중치를 갖는다고 정의되었기에  $weight(s \rightarrow u) \geq weight(s \rightarrow v)$ 이다. 따라서  $s \rightarrow u \rightarrow v$ 의 경로는  $s \rightarrow v$ 보다 무조건 더 길어야 한다 (증명 끝)
- 따라서,  $\min_{s \rightarrow v} weight(s \rightarrow v)$ 인 에지를 따라가면 s에서 v까지의 최단 거리가 된다

9.  $parent[v] = (s\text{에서 } v\text{까지의 최단 경로에서 } v\text{ 바로 직전 노드를 저장})$

- 최단 경로를 재구성하기 위해 parent 값 필요 (DFS, BFS에서의 parent 개념과 같다)

v. 위의 DP 식을 함수 **relax**란 이름의 함수로 정리하면:

```
def relax(u, v): # u를 통해 v로 오는 경로 길이를 dist[v]로 지정함
 dist[v] = dist[u] + weight(u → v)
 parent[v] = u
```

e. 다음을 실행하면 어떻게 될까?

```
for v in V: # 모든 노드 v에 대해, dist[v]를 매우 큰 값으로 초기화
 dist[v] = ∞

for i in range(1, n): # (n-1)번 반복
 for each edge u → v in G:
 if dist[v] > dist[u] + weight(u → v): # u를 통해 v로 오는 경로가 더 짧음
 relax(u, v) # dist[v]를 update
```

---

i. 이 코드가 s에서 다른 모든 노드로의 최단 경로의 길이를 올바르게 계산하나?

1. 우선, 모든 에지는 정확히  $(n-1)$ 번 if-relax 검사를 받는다. 이 반복을 round라고 하면, round 1, round 2, ..., round  $(n-1)$ 이 실행된다
2. round 1을 마치고 나면, 어떤 노드의 최단 경로가 계산된다. 어떤 노드인가?
  - 출발 노드 s에서 나가는 에지 중에 weight가 가장 작은 에지의 다른 노드를 u라고 하면,  $dist[u]$ 는  $weight(s \rightarrow u)$  값으로 relax되고, 이 값이 s에서 u까지의 최단 경로 길이이다
  - 이 노드 u는 출발 노드로부터의 최단 경로가 에지 하나로 구성된 경우이다. u 이외에도 에지 하나로 구성된 최단 경로는 모두 round 1에서 최단 경로 길이가  $dist[]$ 에 제대로 update되고, 이후의 round에서는 이 값이 변하지 않는다

3. 이 논리를 round  $k$ 로 확대 적용하면,  $(k-1)$ 개의 에지로 구성된 최단 경로의 길이는 round  $(k-1)$ 에서 제대로 update된다. round  $k$ 에서는 여기에 한 에지를 더 연결해,  $k$ 개의 에지로 구성된 최단 경로의 길이가 제대로 계산된다. 이유는 이 최단 경로가  $(k-1)$ 개의 에지와 마지막 한 개의 에지로 구성되는데, 이를  $s \rightarrow u \rightarrow v$ 라고 하면,  $s \rightarrow u$ 는  $(k-1)$ 개의 에지로 구성된  $u$ 까지의 최단 경로이고 round  $(k-1)$ 에서 제대로 계산된다. 따라서 round  $k$ 에서  $u \rightarrow v$ 에 해당하는 에지가 relax되면서  $s$ 에서  $u$ 까지의 최단 경로가 계산되는 것이다
4. 경로에 포함된 노드는 모두 달라야 하기에, 아무리 긴 최단 경로도 에지가  $n-1$ 개 보다 더 많이 포함할 수는 없다. 따라서  $(n-1)$ 번만 반복하면  $s$ 에서 다른 모든 노드로 가는 최단 경로 길이가 올바르게 계산된다

ii. 위의 알고리즘이 바로 유명한 [Bellman-Ford 알고리즘](#)이다.  
자세한 내용은 오른쪽 유튜브 설명을 참고하기 바란다



iii. 수행시간은 relax 함수가 상수 시간이므로 이중 반복문 회수에 비례하게 된다. 따라서  $O(nm)$ 이면 충분하다



f. 다음을 실행하면 어떤 걸 알 수 있을까?

```
for each edge $s \rightarrow u$:
 if $dist[s] > dist[s] + weight(s \rightarrow u)$:
 relax(s, u)
```

- i.  $s$ 에 인접한 노드  $u$ 에 대해,  $dist[u]$  값이 update된다. 그럼  $dist[u]$  값이 가장 작은  $u$ 는 최단 경로의 길이가 된다 (앞에서 이미 설명함)
- ii.  $u$ 에 인접한 노드들에 대해서 다시 relax를 반복하면,  $s \rightarrow u \rightarrow v$ 로 연결되는 최소 경로의 길이가  $dist[v]$ 에 저장된다
- iii. 이제는  $s$ 와  $u$ 에 인접한 에지들이 한 번 이상씩 relax가 된 상태다.  $s$ 와  $u$ 에 인접한 노드에 대해,  $dist[v]$ 가 가장 작은 값을 갖는 노드  $v$ 를 고려해보자.  $dist[v]$  값은  $s$ 에서  $v$ 로 오는 경로의 길이 중 현재까지 가장 작은 값이다. 그러면  $dist[v]$ 는  $s$ 에서  $v$ 까지의 최단 경로 길이임을 알 수 있다 (왜? 이보다 더 짧은 경로가 있었다면 그 전에 이미 고려되었어야 한다)
- iv. 결국, 현재의  $dist[u]$  값이 최소가 되는  $u$ 를 선택하면,  $dist[u]$ 는  $s$ 에서  $u$ 까지의 최단 경로의 길이가 되고 이 후에도 relax에 의해 변경되지 않는다.  $u$ 가 선택되면,  $u$ 의 인접한 노드  $v$ 에 대해 if-relax 문을 통해  $dist[v]$ 의 값을 (필요하면) update한다. 이 과정을 정리하면 다음 pseudo 코드와 같다

```

Dijkstra_SP_algorithm(G)
 s = 0 # 0 node is source node
 dist = [0, ∞, ..., ∞], parent = [None, ..., None]

 # 노드 v의 dist[v] 값을 key로 하는 힙 H
 # 여기서 힙은 당연히 min 힙이어야 한다!
 H ← all nodes v with key dist[v]

 while H is not empty:

 u = H.delete_min() # dist 값이 가장 작은 노드 선택
 # GREEDY choice!

 for each edge u → v:
 if dist[v] > dist[u] + weight(u → v):
 relax(u, v) # DP update!
 H.decrease_key(v, dist[v]) # decrease_key 연산

 # 주의점: 노드 v가 힙 H에 저장된 index를 알아야 한다! (왜?)
 # 즉, 노드 v는 자신의 key 값과 H에서의 index 정보 쌍을 알아야 한다
 # 그래서 적응형 힙(Adapted Heap)을 쓰면 편리 (힙 자료구조 편 참조)

 return dist, parent

```

g. 이 유명한 알고리즘을 **Dijkstra** 최단 경로 알고리즘이라 부른다

- i. Dijkstra는 **다익스트라**로 발음한다
- ii. Dijkstra 알고리즘 = Greedy 선택 + DP 전략을 혼용한 알고리즘!!

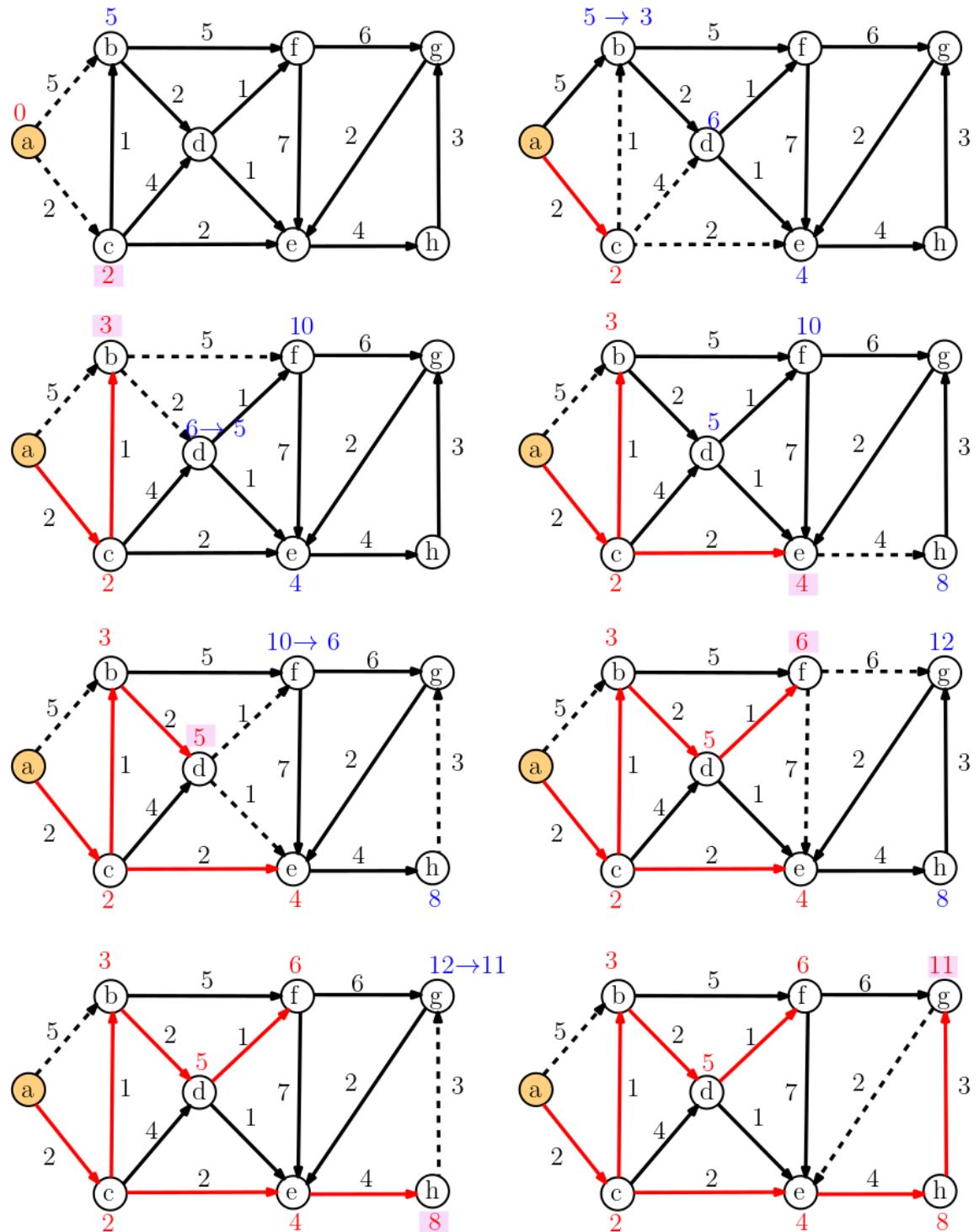


(알고리즘 설명)



(예제 설명)

아래 그래프에 Dijkstra 알고리즘을 적용해보자. 리스트 `dist`의 각 노드의 값은 출발 노드는 0, 다른 노드는 매우 큰 값(무한대)으로 초기화한다. 그림에서 노드 위에 표시된 값이 해당 노드의 `dist` 값이다. 무한대 값은 표시하지 않았다. 노드 a가 출발 노드이다. a의 인접한 노드들에 대해 `relax`를 하는데, 해당 에지를 점선으로 표시했다. 현재 `dist` 값 중에 제일 작은 값을 heap에서 꺼내 (`delete_min`) 해당 노드로의 에지를 최단 경로의 에지로 선택한다. 이 에지를 빨간색으로 표시했다. 이 과정을 모든 노드가 힙에서 `delete_min` 될 때까지 반복한다



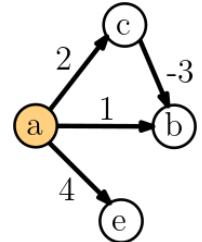
## h. 최단 경로 트리

- i. 최단 경로를 구성하는 빨간색 에지들을 모아보면 트리가 된다. 이 트리는 리스트 parent에 저장되어 있다. 이 트리를 최단 경로 트리라고 부른다
  - 1. 그런데 왜 트리 형태가 될까? 최단 경로의 부분 경로도 최단 경로라는 사실 때문이다
  - 2. 물론 최단 경로 트리가 유일한 것은 아니다. 어떤 노드에 도달하는 최단 경로가 하나 이상일 수도 있기에, 그 중 하나의 경로만 최단 경로 트리에 등장한다
- ii. 최단 경로 트리 (즉, parent 정보)만 있으면 출발 노드에서 도착 노드까지의 최단 경로를 쉽게 재구성할 수 있다
- i. 구현 이슈:
  - i. 개별 노드의 현재 dist 값을 힙에 넣어 관리해야 한다. 힙에는  $(\text{dist}[v], v)$ 의 정보가 저장된다. key 값은  $\text{dist}[v]$ 이고, 이 값이 노드 s에서 v까지의 거리임을 나타낸다. 이 힙에서는 두 가지 연산이 필수적인데,
    - 1. `delete_min` 연산: 최소 dist 값을 갖는 노드를 알아야 한다
    - 2. `decrease_key` 연산: 에지  $u \rightarrow v$ 에 대해 relax 연산이 수행되어  $\text{dist}[v]$  값이 감소하면 힙에서의 위치가 조정되어야 한다
  - ii. 힙마다 두 연산의 수행시간이 다를 수 있다
    - 1. binary heap: 자료구조 시간에 배우는 일반적인 배열에 저장되는 힙
      - o `insert`, `delete_min`, `decrease_key` 모두  $O(\log n)$  시간 필요
    - 2. Fibonacci heap: binary heap보다 복잡한 형태의 힙
      - o `insert`, `decrease_key`는  **$O(1)$**  시간 필요 ( $\leftarrow$  차이점!)
      - o `delete_min`은  $O(\log n)$  시간 필요
      - o wikipedia: [https://en.wikipedia.org/wiki/Fibonacci\\_heap](https://en.wikipedia.org/wiki/Fibonacci_heap)
  - iii. 수행시간: 사용하는 힙의 종류에 따라 다르다!
    - 1. 힙이  $\text{dist}$ 의 초기값을 가지고 구성된다 (`make_heap` 연산) -  $O(n \log n)$
    - 2. 노드는 한 번씩 힙에서 `delete_min`이 수행된다 -  $O(n \log n)$
    - 3. 에지마다 최대 한 번씩 `decrease_key`가 수행된다 -  $O(m \times d)$ 
      - o  $d$ 는 `decrease_key`의 수행시간
    - 4. 총 시간:  $O(n \log n + md)$ 
      - o binary heap의 경우:  $d = O(\log n) \rightarrow O((n+m)\log n)$
      - o Fibonacci heap의 경우:  $d = O(1) \rightarrow O(n \log n + m)$

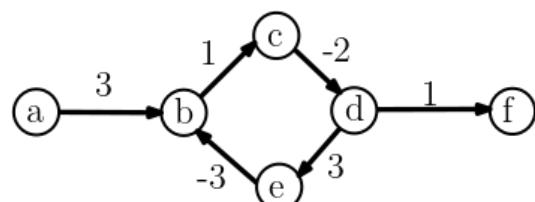
j. [생각해 볼 이슈] 가중치가 음수인 에지가 있다면?

- i. 두 가지 경우로 나눠 생각해 볼 수 있다. 음수 가중치를 갖는 에지가 존재하지만, 사이클의 가중치 합 (사이클의 에지의 가중치 합)이 음수가 되는 경우와 그렇지 않은 일반적인 경우

1. 가중치가 음수인 에지가 있지만 사이클의 가중치 합은 모두 음수가 아니라면, Bellman-Ford 알고리즘은 문제없이 동작한다. 그러나 Dijkstra 알고리즘은 올바르게 동작하지 않을 수도 있다. 어떤 그래프일 때 제대로 동작하지 않을까? 오른쪽 그래프를 보면, 출발 노드 a의 세 에지 중에서 최소 가중치 값을 갖는 에지 (a, b)가 선택되어  $\text{dist}[b] = 1$ 로 고정된다. 즉, b까지의 최단 경로 길이는 1로 결정된다. 그러나 c를 거쳐 b로 가는 경로의 길이가 -1이 되어 더 짧은데 발견하지 못한다. 그러나 Bellman-Ford 알고리즘은 제대로 동작한다. 왜 그럴까? 위의 그래프를 보면, round 1에서는  $\text{dist}[b] = 1$ ,  $\text{dist}[c] = 2$ ,  $\text{dist}[e] = 4$ 로 relax되고, round 2에서는  $\text{dist}[b] = \text{dist}[c] + -3 = -1$ ,  $\text{dist}[c] = 2$ ,  $\text{dist}[e] = 40$ 이고, round 3에서는 변화가 없다.  $(n-1)$ 번 모두 에지에 대해 relax 연산을 하기에 음의 사이클이 없는 한 제대로 계산하게 된다.



2. 사이클 가중치 합이 음수가 아래 그림과 같은 그래프처럼, 사이클 가중치 합이  $1+(-2)+3+(-3) = -1$ 이 되어 음수된다면, 사이클을 계속 반복할 수록 가중치 값이 감소하게 된다. 그러면 노드 a에서 노드 f로 가는 최단 경로는 정의할 수 없다



3. 이 상황에서는 최단 경로가 정의가 안된다는 걸 인식할 수 있어야 한다. Dijkstra 알고리즘은 음의 에지가 존재하는 경우에도 제대로 동작하지 않기 때문에, 이 상황을 제대로 인식하지 못한다. Bellman-Ford 알고리즘에서는 이 상황을 인식할 수 있을까?  $(n-1)$ 번의 round를 거친 이후에는  $\text{dist}$  값이 각 노드까지의 최단 경로 길이로 결정되어야 한다. 만약 한 번 더 반복을 해서,  $n$ 번째 round를 추가로 실행해서  $\text{dist}$  값 중 일부가 더 작은 값으로 업데이트 된다면 음의 사이클이 존재한다는 의미가 된다. 따라서 Bellman-Ford 알고리즘은 추가로 한 번 round를 수행해 음의 사이클의 존재여부를 검사할 수 있다

### 8. [🎤 인터뷰 문제] 최단경로문제에서 한 걸음 더

- a. Dijkstra 알고리즘은 출발 노드에서 다른 모든 노드까지의 최단경로를 모두 계산해주는 One-to-All 알고리즘이다. 만약, 각 노드에서 목적 노드까지의 최단경로를 알고 싶다면 어떻게 해야 할까? 즉, All-to-One 문제는 어떻게 풀어야 하나?
  - i. **무방향 그래프**인 경우: 무방향 에지이기 때문에, 노드 a에서 노드 b로의 최단 경로가 b에서 a로의 최단 경로다. 따라서 One-to-All 경로를 거꾸로 출력하면 All-to-One 경로가 된다
  - ii. **방향 그래프**인 경우: 에지에 방향이 있기 때문에, 에지의 방향을 반대로 바꾼 그래프를 새로 정의해서 새 그래프에서 One-to-All 알고리즘을 수행한다. 그 결과가 All-to-One 경로가 된다
- b. [난이도 높음] Second Best Shortest Path 문제: 두 노드 s와 t 사이의 최단 경로가 유일할 수도 있지만, 두 개 이상 존재할 수도 있다. 만약, 두 번째로 빠른 경로를 알고 싶다면 어떻게 계산해야 할까? 최단 경로가 2개 이상이면 두 번째로 빠른 경로 역시 최단 경로가 되고, 최단 경로가 하나 뿐이라면, 두 번째로 빠른 경로는 최단 경로보다 더 길다. 두 번째로 빠른 경로의 길이를 계산하는 알고리즘을 생각해보자.
  - i. 네비 길찾기에서 가장 빠른 경로와 두 번째로 빠른 경로를 동시에 보여주고 사용자에게 선택하도록 하는 응용에 사용 가능하다
  - ii. 최단 경로와 두 번째로 빠른 경로의 차이를 분석하는 게 우선 할 일!
  - iii. 두 번째로 빠른 경로는 최단 경로와 같은 에지를 따라 가지 않고 다른 에지 하나는 사용할 것이다. 예를 들어, s에서 최단 경로의 에지를 차례대로 따라 가다가 어떤 노드 u에서 다음 노드 v를 따라가지 않고 u에서 w로 다른 에지를 따라 가게 된다. w에선 목적 노드 t까지 w-t 최단 경로를 따라 가야 한다. (왜?) 여기서 u = s일 수도 있음에 유의하자.
  - iv. 즉, 두 번째로 빠른 경로는 s에서 u까지 최단경로는 그대로 따라가고, u에서 v가 아닌 u에서 w로 가는 에지를 선택하는 순간이 반드시 존재해야 한다. (아니라면 s-t 최단 경로를 그대로 따라가게 되므로)
  - v. ( $u, v$ ) 에지가 아닌 ( $u, w$ ) 에지를 선택했기 때문에, 이 에지를 통해 t로 가는 어떤 경로도 최단 경로의 길이와 같거나 더 커야 한다. 우리는 두 번째로 빠른 경로를 구하는 것이기에 w에서 t까지는 최단 경로로 가야 한다. 결국, 두 번째로 빠른 경로는

**(s에서 u까지의 최단 경로) + ( $u \rightarrow w$ ) + (w에서 t까지의 최단 경로)**

의 모양이 될 것이다. 결국, s에서 t까지의 최단 경로 상에 있는 모든 u에 대해, 최단 경로 상의 에지가 아닌 다른 에지를 이용해서 t에 도착할 수 있는 길을 조사해서 그 중 길이가 제일 짧은 경로가 두 번째로 빠른 경로가 된다.

Dijkstra 알고리즘을 이용해서 구하고 싶다. 구체적으로 어떻게 해야 할까?

All-to-One 최단 경로 방법이 힌트~

## 9. All-to-all shortest path (all-pairs shortest path) 알고리즘

- a. Dijkstra 알고리즘은 특정 노드에서 출발해서 다른 모든 노드로 가는 최단 경로를 구하는 one-to-all shortest path 알고리즘이다
- b. **방법 1:** All-to-all shortest path 알고리즘은 그래프의 모든 노드 쌍에 대해, 두 노드 사이의 최단 경로를 계산하는 알고리즘을 의미한다



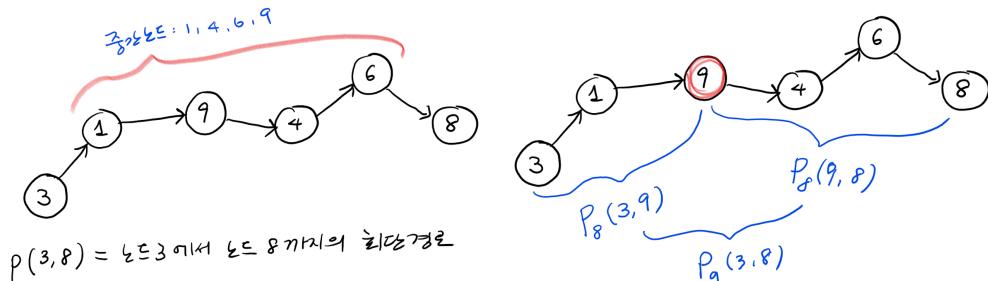
```
all_pairs_SP_simple(G):
 for each (source) node s in G:
 dist[s][] = SSSP(s) # SSSP = Single Source SP algorithm
```

수행시간:  $O(n \times \text{time of SSSP})$

- i. **경우 1:** 모든 에지의 가중치가 같은 경우
  - 1. 이 경우엔 BFS가 특정 소스 노드에서 다른 모든 노드로의 최단 경로를 찾아주므로 SSSP = BFS가 되면 됨
  - 2. 수행시간:  $O(n \times (n+m)) = O(n^2 + nm)$
- ii. **경우 2:** 모든 에지의 가중치가 양수인 경우
  - 1. Dijkstra 알고리즘을 SSSP 알고리즘으로~
  - 2. 수행시간: binary heap을 이용한 경우:  $O(n(n+m)\log n)$
- iii. **경우 3:** 에지의 가중치가 음수가 있는 경우 (단, 음수의 사이클이 없다고 가정)
  - 1. Bellman-Ford 알고리즘을 SSSP 알고리즘으로~
  - 2. 수행시간:  $O(n^2m)$

### c. 방법 2: Dynamic programming 방법

- i. 노드  $i$ 와 노드  $j$  사이의 최단 경로를  $p(i, j)$ 라 하자
  - 1.  $i$ 와  $j$ 는 1과  $n$  사이의 수로 노드 id를 의미
  - 2. 주의: 노드의 id가 0이 아닌 1부터 시작한다
- ii.  $p(i, j)$ 에 나타나는 중간 노드를 고려해보자. 중간 노드의 id는  $\{1, \dots, n\} - \{i, j\}$  중 하나다. 즉,  $i$ 와  $j$ 를 제외한 1부터  $n$ 까지의 어떤 id의 노드도 경로 중간에 등장 할 수 있다
- iii. 그림으로 설명하면 아래와 같다



- iv.  $p_k(i, j) =$  노드  $i$ 와  $j$  사이의 경로 중에서 중간 노드들이  $\{1, \dots, k\}$ 의 id를 갖는 노드들로 구성되는 최단 경로로 정의

1. 그러면, 우리가 원하는  $p(i, j)$ 는  $p_n(i, j)$ 가 됨을 확인할 수 있음!
2. 위의 그림은  $p_9(3, 8)$ 을 의미

v.  $p_k(i, j)$ 에 대한 DP 식을 만들어보자

1. 이 경로의 중간에 노드  $k$ 가 포함될 수도 안 될 수도 있다

- 포함되는 안되는 경우:  $p_k(i, j) = p_{k-1}(i, j)$  (**왜?**)
- 포함되는 경우:
  - i. 노드  $i$ 부터  $k$ 까지와  $k$ 부터  $j$ 까지로 두 개의 부경로(subpath)를 나눌 수 있음
  - ii.  $i$ 부터  $k$ 까지의 경로의 중간에는  $k$ 가 나타나지 않으므로  $p_{k-1}(i, k)$ 로 표기가 가능하고,  $k$ 부터  $j$ 까지는  $p_{k-1}(k, j)$ 로 표기가 가능하다
  - iii. 따라서  $p_k(i, j) = p_{k-1}(i, k) + p_{k-1}(k, j)$ 가 된다
  - iv. 결국, 이 두 가지 경우 중 짧은 길이의 경로가 된다
- $p_k(i, j) = p_{k-1}(i, k) + p_{k-1}(k, j)$ 는  $i \rightarrow k \rightarrow j$ 로 경로가 구성되는데,  $i \rightarrow k$ 까지의 경로에 있는 노드가  $k \rightarrow j$ 까지의 경로에 다시 나타날 수도 있지 않나? 이는 경로에 나타나는 모든 노드는 서로 달라야 한다는 정의에 어긋나는 것 아닌가?
  - i. 그런 경우가 발생 가능하지만 최종적인 최단 경로로 채택되지 않는다. 왜?

2.  $p_0(i, j)$ 는 무슨 의미인가?

- 중간에 어떤 노드도 오지 않는다는 뜻이므로  $(i, j)$  에지가 존재하면 그 에지 자체가  $p_k(i, j)$ 이다. 에지가 존재하지 않으면 직접 에지 하나만을 이용한 경로는 없다는 의미이다
- 따라서  $dist[i][j]$ 의 초기 값은  $(i, j)$  에지가 존재하면 에지의 가중치  $w(i, j)$ 로, 에지가 없다면 초기 값을 무한대 (매우 큰 값)으로 정하면 된다
- $dist[i][i]$ 인 경우는? 자기 자신으로의 경로는 당연히 존재하는 것이므로 가중치(비용)는 0으로 정하면 된다

3. 경로의 중간 노드가  $\{1, \dots, k\}$ 까지인 경우에  $i$ 부터  $j$ 까지의 최단경로의 길이를  $dist[i][j]$ 에 저장한다면, 현재 상태에서는

$dist[i][j] = \min( dist[i][j], dist[i][k] + dist[k][j] )$

```

1 dist = a list of minimum distances with ∞
2 pred = a list of predecessors with None
pred[i][j] = a predecessor of j of p(i,j)

2 for each edge (i,j)
3 dist[i][j] \leftarrow w(i,j)
 pred[i][j] \leftarrow i

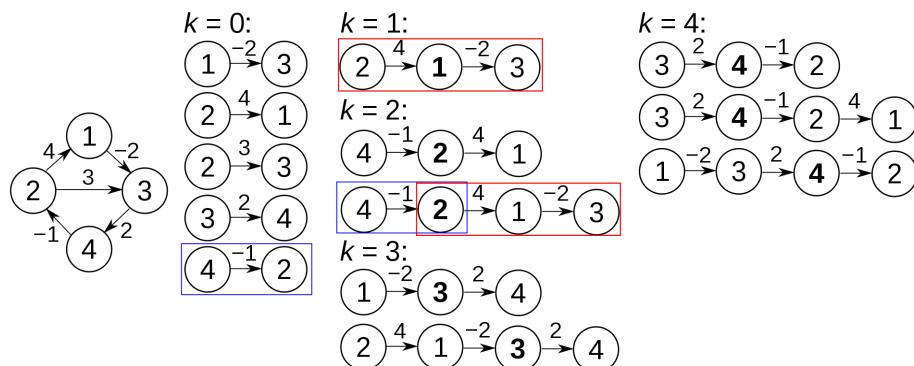
4 for each vertex i
5 dist[i][i] \leftarrow 0

6 for k from 1 to |V|
7 for i from 1 to |V|
8 for j from 1 to |V|
9 if dist[i][j] > dist[i][k] + dist[k][j]
10 dist[i][j] \leftarrow dist[i][k] + dist[k][j]
11 pred[i][j] \leftarrow pred[k][j] # why?

```

vi. 위 알고리즘을 [Floyd-Warshall](#) 알고리즘이라 부른다

1. Wikipedia 예제:



아래는 dist 리스트의 변화를 k 값에 따라 보여준다

|     |   | $j$      |          |          |          |
|-----|---|----------|----------|----------|----------|
|     |   | 1        | 2        | 3        | 4        |
| $i$ | 1 | 0        | $\infty$ | -2       | $\infty$ |
|     | 2 | 4        | 0        | 3        | $\infty$ |
| $i$ | 3 | $\infty$ | $\infty$ | 0        | 2        |
|     | 4 | $\infty$ | -1       | $\infty$ | 0        |

|     |   | $j$      |          |    |          |
|-----|---|----------|----------|----|----------|
|     |   | 1        | 2        | 3  | 4        |
| $i$ | 1 | 0        | $\infty$ | -2 | $\infty$ |
|     | 2 | 4        | 0        | 2  | $\infty$ |
| $i$ | 3 | $\infty$ | $\infty$ | 0  | 2        |
|     | 4 | $\infty$ | -1       | 1  | 0        |

|     |   | $j$ |    |    |   |
|-----|---|-----|----|----|---|
|     |   | 1   | 2  | 3  | 4 |
| $i$ | 1 | 0   | -1 | -2 | 0 |
|     | 2 | 4   | 0  | 2  | 4 |
| $i$ | 3 | 5   | 1  | 0  | 2 |
|     | 4 | 3   | -1 | 1  | 0 |

$k = 0$ : 에지  $(i, j)$ 에 대해,  $dist[i][j] = weight(i, j)$

$k = 1$ : 중간노드가 없거나 1번 노드를 갖을 수 있는 경우 중 최단 경로 길이

...

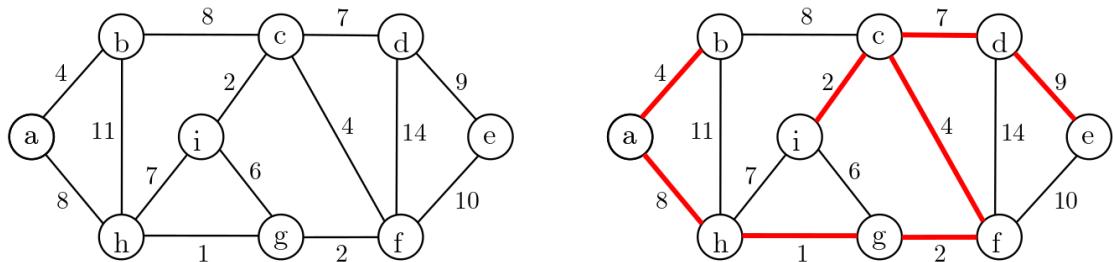
$k = 4$ : 중간노드가 없거나 1번- 4번까지의 노드가 올 수 있는 경우 중 최단 경로 길이

$k = 5$ : 중간노드가 없거나 모든 노드들이 올 수 있는 경우 중 최단 경로의 길이 (답)

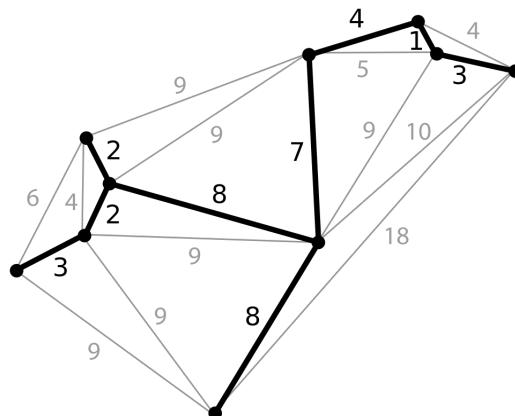
- vii. 이 Floyd-Warshall 알고리즘의 수행시간은? 삼중 **for** 루프안에  $O(1)$  시간 연산이 수행되므로  $O(n^3)$  시간이 필요하다
- viii. **Q1:** all-to-all path를 직접 출력하려고 한다면, 어떤 정보를 기록해 놓아야 할까?
  1. `pred[i][j]`에  $i$ 에서  $j$ 까지의 최단 경로에서  $j$ 의 바로 전 predecessor (parent)를 저장
  2.  $k$ 번째 루프에서 `pred[i][j]`를 update한다 ( $p(i, j)$ 가  $k$ 번 노드를 포함하지 않는 경우와 포함하는 경우로 나눠서 처리)
  3. 위의 코드를 참고할 것!
- ix. **Q2:** 길이가 음수인 사이클이 있는 경우엔 어떻게 찾아낼 수 있을까?

## 9. 최소 신장 트리 문제 (Minimum Spanning Tree Problem: MST)

- a. (무방향) 그래프의 에지 중  $n-1$ 개를 선택해 그래프의 모든 노드를 연결하는 트리를 정의할 수 있다면, 그 트리를 신장 트리(spanning tree)라 부른다
- 신장 트리는 여러 개 존재 가능 (예: 아래 그래프에서 신장트리를 2개만 찾아보자. 오른쪽 그래프의 빨간색으로 그려진 신장트리가 MST 중 하나이다)



- b. 신장트리의 가중치(weight)는 신장트리 에지의 가중치의 합으로 정의한다
- 위에서 찾은 신장트리의 가중치는 얼마인가?
- c. 최소신장트리는 신장트리 중에서 가중치가 가장 작은 신장트리로 정의한다
- 당연히, 최소신장트리도 유일하지 않을 수 있음!
  - 위의 그래프에서 최소신장트리 중 하나를 찾아보자. 최소신장트리의 가중치 합은 얼마인가?
  - Wikipedia example:



- d. 응용 분야 [from Wikipedia]
- 노드를 도시로, 에지를 두 도시를 연결하는 도로로 표현하고, 에지의 가중치는 해당 도로의 건설 비용인 경우: 재정 여건상 그래프처럼 모든 도로를 건설할 수 없고 일부만 건설해야 한다면, 모든 도시를 연결하고 전체 비용이 최소가 되는 것이 바람직함 → 이 조건을 만족하는 구조가 바로 최소신장트리!
  - Cluster analysis: clustering points in the plane
  - Constructing trees for broadcasting in computer networks.<sup>[24]</sup>
  - Image registration<sup>[25]</sup> and segmentation<sup>[26]</sup> - see minimum spanning tree-based segmentation.
  - Curvilinear feature extraction in computer vision.<sup>[27]</sup>
  - Handwriting recognition of mathematical expressions.<sup>[28]</sup>

### e. 중요 성질 0: 트리 성질

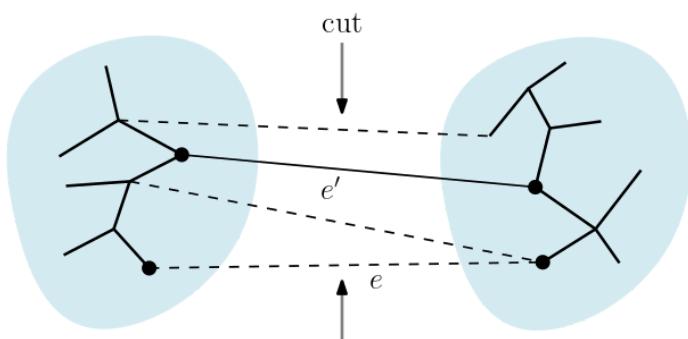
- i.  $n$ 개의 노드를 연결한 트리이므로 에지의 개수는  $n-1$  개이다
- ii. 트리이기 때문에 임의의 에지 하나를 제거하면 두 개의 부트리로 분할된다
- iii. 두 노드를 연결하는 에지가 없는 경우, 두 노드를 연결하는 에지를 트리에 추가하면 해당 에지를 포함하는 사이클이 생긴다

### f. 중요 성질 1: Cut property

- i. **cut** : 그래프의 노드 집합을 두 개로 분할할 때 사용되는 에지의 부분집합 (즉, cut에 포함된 에지를 지우면 그래프가 두 부분으로 분할된다)

그래프의 **cut** 중에서 어떤 에지  $e$ 의 가중치가 다른 모든 에지보다 작다면, 그 에지  $e$ 를 포함하는 MST가 반드시 존재한다

**증명:** 아래 그림에서 에지  $e$ 가 어떤 cut의 최소 가중치라고 하자. 그런데 이 에지  $e$ 를 포함하는 MST가 존재하지 않는다고 가정하자. (귀류법으로 증명하기 위해, 결론을 부정했다) 이 cut에는  $e$  이외의 다른 에지가 하나 이상 반드시 존재한다. cut이  $e$  하나만으로 구성되어 있다면 이 에지는 MST에 당연히 포함되어야 하기 때문이다. cut의 에지 중 하나는 반드시 MST에 포함되어야 하고, 그 에지를  $e'$ 이라 하자. 가정에 의해  $\text{weight}(e) \leq \text{weight}(e')$ 이다. 에지  $e$ 를 MST  $T$ 에 추가하면,  $e$ 와  $e'$ 를 포함하는 사이클이 정확히 하나 만들어진다. (성질 0의 3번째 항목 참고)  $e'$ 을 제거하면 다시 트리가 된다. 이 트리를  $T'$ 이라 하자.  $T'$ 의 다른 에지는 그대로이고  $e'$ 이  $e$ 로 대체되었기에  $\text{weight}(T') = \text{weight}(T) + \text{weight}(e) - \text{weight}(e') \leq \text{weight}(T)$ 가 성립한다. (이유는  $\text{weight}(e) \leq \text{weight}(e')$ 이기 때문에) 이 식은  $T'$ 의 가중치 합이 MST인  $T$ 의 가중치 합과 같거나 작다는 것이다. 이는  $T'$ 이 MST이거나 더 작은 가중치를 갖고 있다는 뜻이다.  $T'$ 은  $e$ 를 포함하고 있기 때문에, **e가 포함된 MST가 존재하지 않다는 가정**에 위배된다 (증명끝)

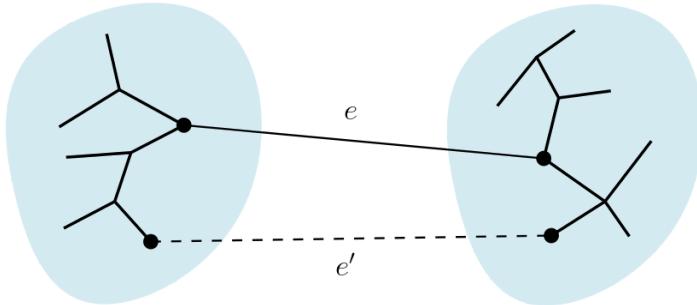


### g. 중요 성질 2: Cycle property

그래프의 사이클  $C$ 에서 어떤 에지  $e$ 의 가중치가  $C$ 의 다른 모든 에지보다 크다면 ( $e$ 가  $C$ 의 에지 중 최대 가중치라면) 에지  $e$ 는 MST에 절대 포함되지 않는다

**증명:** Cut property 증명처럼 에지  $e$ 를 포함하는 MST  $T$ 가 존재한다고 하자. 아래 그림 참조.  $T$ 에서  $e$ 를 제거하면 두 부분으로 분할된다. 그러면 이 두 부분의 노드를 연결하는 그래프의 에지들이 존재한다. 당연히 이 에지 집합은 cut 중의 하나이다. 그러면 이 cut에는 사이클  $C$ 의 다른 에지  $e'$ 이 반드시 존재해야 한다. (존재하지

않는다면, 사이클  $C$  자체가 존재하지 않게 된다)  $e$ 가  $C$ 의 최대 가중치 에지이므로  $\text{weight}(e') \leq \text{weight}(e)$ 가 성립한다.  $T$ 에서  $e$ 를 지우고 대신  $e'$ 을 삽입하면 새로운 신장 트리  $T'$ 을 정의할 수 있다. Cut property 증명처럼  $\text{weight}(T') \leq \text{weight}(T)$ 임을 보일 수 있다. 이는 가정에 위배된다. (증명 끝)



#### h. 대표적인 알고리즘

- i. **Prim 알고리즘**: Cut 성질을 이용한 그리디 알고리즘
- ii. **Kruskal 알고리즘**: Cycle 성질을 이용한 그리디 알고리즘

### i. Prim algorithm : Cut property를 이용한 Greedy 알고리즘

알고리즘 요약:

- i.  $F = \{v\}$  # 임의의 노드  $v$ 부터 시작
- ii.  $T = \{\}$
- iii.  $(F, V-F)$  분할하는 cut 에지 중에서 최소 가중치의 에지  
 $e = (v, w)$  선택
  1. 여기서  $v$ 는  $F$ 에 속하고,  $w$ 는  $V-F$ 에 속하는 노드임
- iv.  $T = T + e, F = F + w$  #  $T$ 에 에지  $e$ 를 추가하고,  $F$ 에 노드  $w$  추가
- v.  $|T| = n - 1$  이거나  $V-F = \{\}$  이면 return  $T$ , 아니면 단계 iii부터 다시 반복



- i. Correctness 증명: Cut-property에 의해 쉽게 증명됨

#### ii. Pseudo code

```

노드 v 는 집합 $V - F$ 에 있는 노드라고 가정
weight[v] = v 에서 F 의 노드에 연결된 에지 중에서 최소 weight
E[v] = 최소 weight를 갖는 에지의 끝 노드 번호
즉, $(v, E[v])$ 가 최소 weight 에지: v 는 $V-F$ 의 노드, $E[v]$ 는 F 의 노드

1. for each node v :
 weight[v] = $+\infty$
 E[v] = None
 F[v] = False # True인 노드만 집합 F 를 구성. 처음엔 모두 False

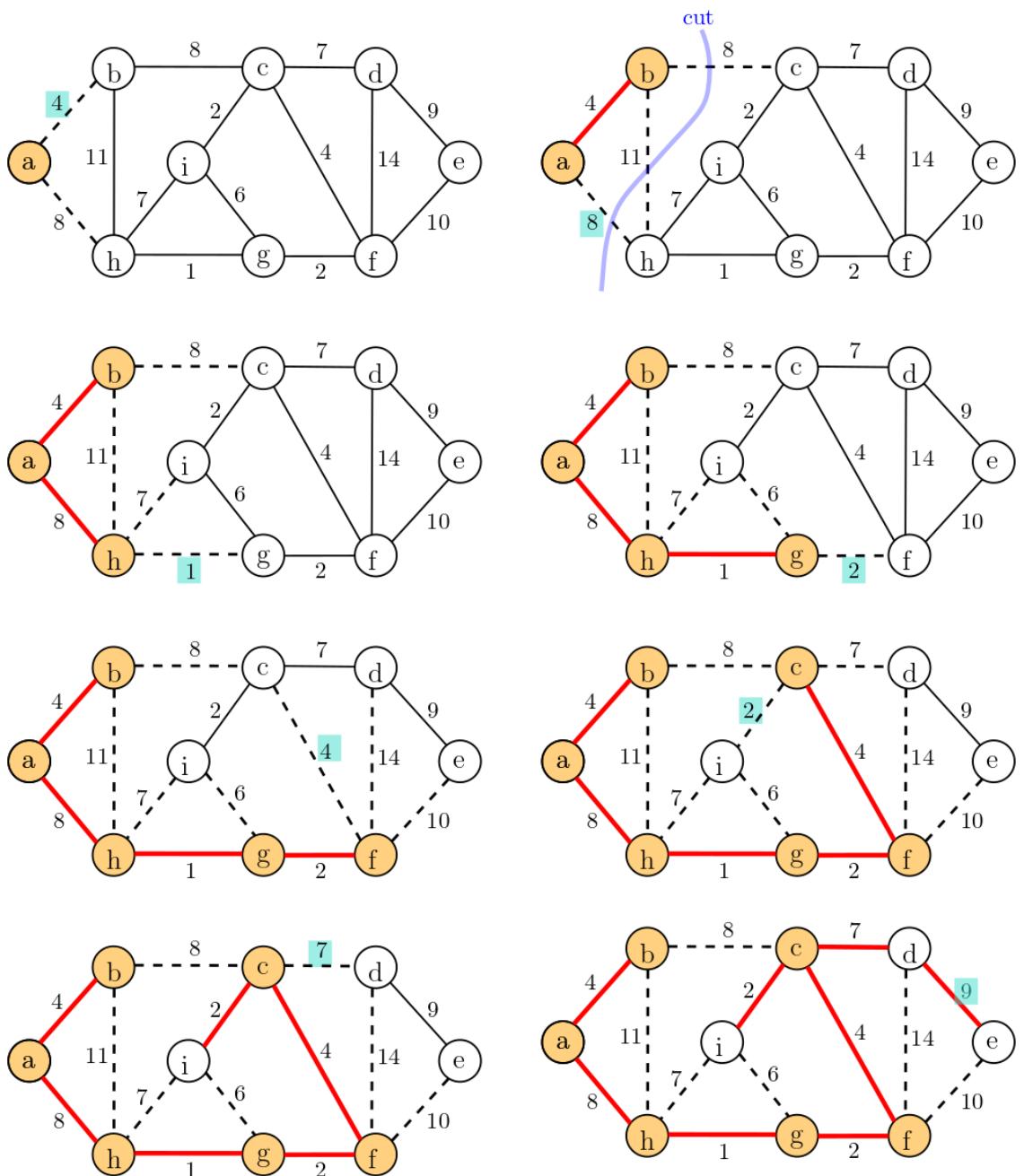
2. $T = []$ # MST를 구성하는 에지 집합. 처음엔 공집합 (빈 리스트)
3. $Q = \text{Heap}(V, \text{weight})$ # 각 노드 v 에 대해, key 값을 weight[v]로 하여 min 힙 구성

4. while Q is not empty:
 a. $v = Q.\text{delete_min}()$ # $(v, E[v])$ is the min edge
 b. $F[v] = \text{True}$
 c. if $E[v] \neq \text{None}$: # 첫 번째 루프 이외엔 항상 참이 된다! (왜?)
 T.append((E[v], v))
 d. for each edge (v, w) incident to v :
 if $F[w] == \text{False}$ and $\text{weight}(v, w) < \text{weight}[w]$:
 1. $\text{weight}[w] = \text{weight}(v, w)$ # decrease_key 발생!
 2. $Q.\text{decrease_key}(w, \text{weight}[w])$
 # w 의 key값을 $\text{weight}[w]$ 로 decrease_key
 3. $E[w] = v$

5. return T

```

## iii. 예제 그래프에서 단계별로 수행해보기



임의의 노드 (예제에서는 노드 a)를 선택한 후, 집합 F에 넣는다. F의 노드에 인접한 에지가 (a, b), (a, h) 두 개이고, 이 에지 중 가중치가 작은 에지가 (a, b)이다. 이 에지가 F = {a}와 V - F를 분할하는 cut 에지 중에 가중치가 제일 작은 에지가 된다. cut 성질에 의해, 이 에지를 포함하는 MST가 존재하기 때문에 선택할 수 있다. F = {a, b}가 된다. b에 인접한 에지도 다음 후보가 되며, a와 b에 인접한 에지 중에서 가중치가 제일 작은 에지를 선택한다. 그런 에지가 하나 이상일 수도 있다. 예제에서는 가중치 8을 갖는 에지가 두 개 존재한다. 어떤 에지를 선택해도 상관없다. 이 과정을 계속 반복하면 된다.

iv. 시간복잡도 :  $O(n\log n + m\log n)$  또는  $O(n\log n + m)$

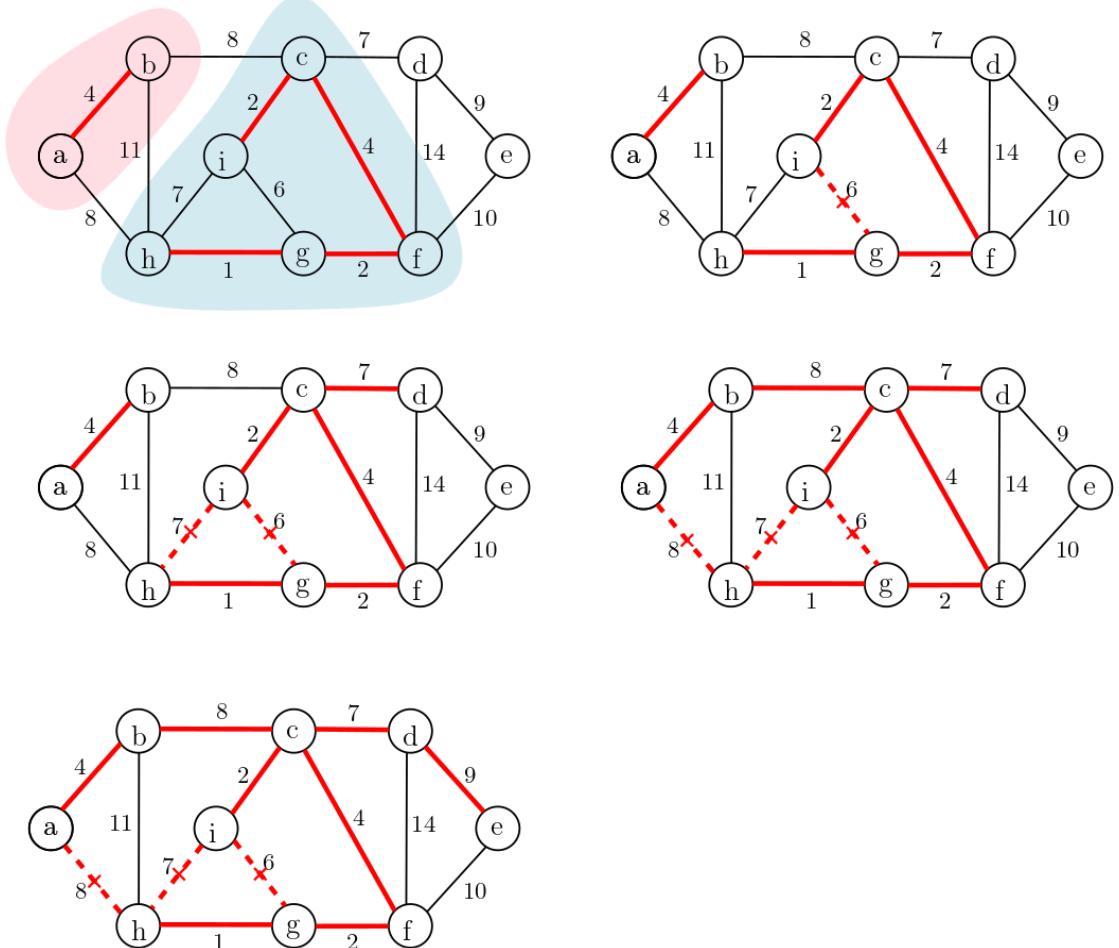
1. 단계 1의 수행시간:  **$O(n)$**
2. 단계 3의 수행시간:  **$O(n \log n)$**
3. 단계 4의 수행시간: Q를 어떤 큐(힙)를 사용하느냐에 따라 달라짐! (단계 5.d의 수행시간이 중요함!)
  - a. binary heap 사용:  **$O(m \log n)$**
  - b. Fibonacci heap 사용:  **$O(m)$**

#### j. Kruskal algorithm : Cycle-property를 이용한 Greedy 알고리즘

1.  $T = \{\}$
2.  $E$ 의 에지를 가중치의 오름차순으로 정렬
3.  $E$ 의 에지  $e$ 를 오름차순의 순서에 따라:
  - $e$ 를  $T$ 에 추가해서 사이클이 발생하면 무시하고, 아니면  $T$ 에 추가
  - **if**  $|T| == n - 1$ :  
**return**  $T$



- i. 알고리즘의 시작 단계에서 각 노드가 하나의 독립된 트리로 간주된다
- ii. 단계 3의 루프를 돌면서, 가중치의 오름차순으로 에지를 고려하는데, 에지의 양 끝 노드가 속한 두 트리가 합쳐지면서 더 큰 트리로 합병된다.
  1. 에지의 양 끝 노드가 동일한 트리에 속한다면, 당연히 그 에지가 포함된 사이클이 정의되어 해당 에지는 선택되지 않는다. (트리에 에지를 추가하면 무조건 하나의 사이클이 생김.)
- iii.  $T$ 는 이러한 작은 트리들의 집합 (forest)이다. 이 병합 과정을 모든 노드가 참여하는 하나의 트리 (즉, MST)가 만들어질 때까지 반복한다
- iv. 알고리즘에서 key 스텝은 사이클의 발생 여부를 판단하는 부분이다. 어떻게 효과적으로 알 수 있을까?
  1. 에지  $e = (u, v)$ 를 현재 고려한다고 하자. 그럼  $u, v$ 가 동일한 트리에 속해 있다면  $e$ 의 추가가 사이클을 생성한다. 만약  $u, v$ 가 서로 다른 두 트리에 각각 속해 있다면 사이클이 만들어지지 않고 두 트리를 하나의 더 큰 트리로 병합하게 된다.
  2. 그럼  $T = \{T_1, T_2, \dots, T_k\}$ 가 있을 때, 각  $T_i$ 에 속하는 노드를 union-find의 집합으로 관리해보자. 그러면 각 집합의 루트 노드에 저장된 값이 집합을 대표하기 때문에,  $u$ 와  $v$ 가 속한 집합의 루트 노드의 값을 비교하면 같은 트리에 속해 있는지 아닌지를 확인할 수 있다. 즉  $\text{find}(u)$ 와  $\text{find}(v)$ 를 비교하면 된다. 서로 다른 트리에 속해 있다면, 두 트리를 union 함수를 이용해 병합하면 된다
  3. 아래 그래프에 대해서 단계별로 적용해보자



에지의 가중치를 오름차순으로 정렬한 후, 가장 작은 가중치의 에지부터 고려하는데, 지금까지 선택한 에지들과 사이클을 만들지 않으면 cycle 성질에 의해 무조건 선택해도 된다. 왜? 두 번째 그림의 에지 (*i*, *g*)처럼 사이클이 만들어지면, 이 에지의 가중치가 사이클의 다른 에지의 가중치보다 (같거나 더) 크기 때문이다. (가중치의 오름차순으로 에지를 고려하고 있기에 당연히 가장 크다.) 문제는 현재 고려하는 에지와 이미 선택한 에지들 사이에 사이클이 만들어지는지 빠르게 검사하는 것이다. 이를 위해 딱 맞는 자료구조가 바로 앞에서 설명한 union-find 자료구조이다. 집합의 union 연산을 효율적으로 지원하는 자료구조인데, 어떤 원소가 속한 집합을 찾아주는 find 연산과 두 원소가 속한 두 집합을 하나의 집합으로 합하는 union 연산이 제공된다. 위의 (*i*, *g*) 에지의 경우에는 *i*와 *g*가 같은 집합 (트리)에 속해 있기 때문에 *i*와 *g*를 연결하면 당연히 트리 성질에 의해 사이클이 만들어진다. 만약 다른 집합에 속한다면 사이클이 만들어지지 않기에 해당 에지를 선택하면 된다.

union-find 자료구조는 두 가지 버전이 있다. 자세한 내용은 자료구조 책이나 wikipedia [https://en.wikipedia.org/wiki/Disjoint-set\\_data\\_structure](https://en.wikipedia.org/wiki/Disjoint-set_data_structure)를 참조바란다. 두 가지 버전에 따라 연산의 수행시간이 다르지만 두 연산의 수행시간이 **O(logn)** 으로 보장된다. (사실, 더 빠르게 구현할 수도 있다)

## v. Pseudo code [from Wikipedia]

```

T = [] # MST를 구성하는 에지의 집합. 빈 리스트로 초기화
for each v ∈ V: # 노드 하나로 구성된 집합 n개로 시작
 make_set(v)

Sort E in non-decreasing order of weights

for each e=(u,v) in E:
 if find(u) ≠ find(v): # 같은 트리의 노드라면
 # 즉, T+e가 사이클을 생성 안하면
 T.append(e)
 union(find(u), find(v)) # union-find

if |T| == n - 1:
 return T

```

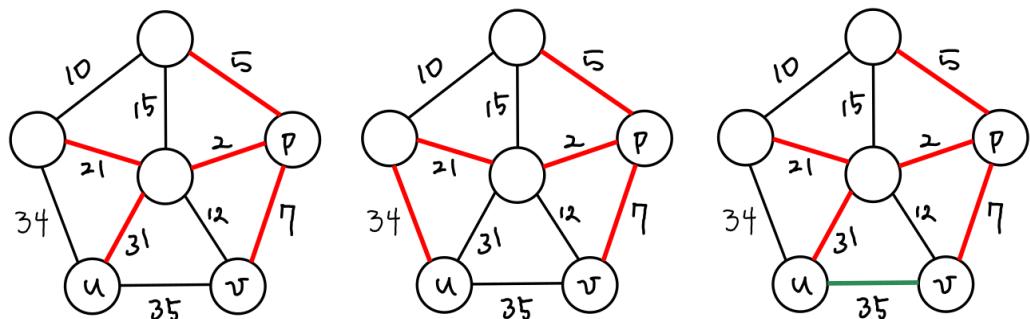
## vi. Correctness 증명: Cycle property에 의해 바로 증명

## vii. 시간복잡도:

1. 메인 **for**문은 에지 개수만큼 반복. 즉, 에지 하나당 2번의 **find** 연산과 1번의 **union** 연산을 수행함 → 전체적으로  $3m$  번의 **find**와 **union** 연산을 수행
2.  $m$  번의 **find**와 **union** 연산을 수행해야 하고, 각 **find**, **union** 연산은  $O(\log n)$  시간이 걸리기 때문에 (정렬 시간을 포함해서)  $O(m \log m)$  시간이면 충분

k. [💡 코딩 대회 문제 - 난이도 상] Second Best 신장트리 문제

- i. 두 노드 사이의 최단 경로를 제외하고 가장 짧은 길이의 (second best shortest) 경로를 찾는 문제와 비슷한 유형의 문제
- ii. 문제 정의: 그래프 G의 모든 신장트리 중에서 MST T를 제외하고 비용이 가장 작은 신장트리 S을 계산하는 문제
- iii. Second best 최단 경로 문제에서처럼 T와 S의 차이를 파악해야 한다
  1. S의 에지와 T의 에지가 모두 같으면 안된다 (당연!)
  2. 그러면 하나의 에지만 다른가? 아니면, 두 개 이상의 에지가 다른 경우도 있을까?
  3. 사실: S와 T의 에지는 정확히 하나의 에지만 다르다 (증명은 어렵지 않으니 각자 해보자)
    - a. 예제의 가장 왼쪽 빨간색 트리가 MST T이고, 가운데 빨간색 트리가 second best 트리 S이다 (31 값의 에지가 빠지고 34 값의 에지가 삽입된 모양)
    - b.  $\text{weight}(T) = 2+5+7+21+31 = 66$
    - c.  $\text{weight}(S) = 2+5+7+21+31 - \textcolor{red}{31} + \textcolor{green}{34} = 69$



- iv. 가장 먼저 생각할 수 있는 방법은 T의 특정 에지가 빠진다고 가정하고, 그 에지를 (가상으로) 제거한 상태에서 MST를 구해 보는 것이다. T의 모든 에지에 대해 이렇게 구해진 MST 중에서 비용이 제일 작은 트리가 당연히 S임
- v. 방법 1: 단순 알고리즘 1:  $O(nm\log n)$ 
  1. G의 MST T 계산 (Kruskal 알고리즘 이용)
  2. E의 에지는 가중치의 오름차순으로 이미 정렬되어 있다고 가정
  3. **for** each edge e of T:
    - a. 에지 집합 E - {e}를 가지고 MST T' 계산
    - b. S = T' 중에서 최소 비용을 갖는 신장트리 계속 update
  4. **return** S

- 1번과 3.a번은 Kruskal 알고리즘을 적용  $\rightarrow O(m\log n)$  시간

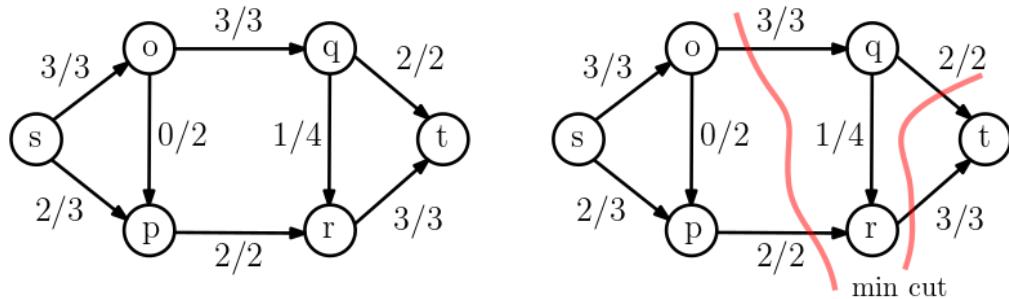
- 3번 루프는 총 n번 돌고, 매번 Kruskal 알고리즘 수행시간만큼 필요하므로  $O(nm\log n)$  시간이면 충분
- vi. 위의 방법 1은 T의 에지 e를 대체하는 에지 f를 찾는 것인데, 이를 거꾸로 해보자. 즉, T에 없는 각 에지 f가 참여하는 대신 T의 어떤 에지 e를 제거해야 하는지 따져보자. 방법 1을 조금 수정해 보면 다음과 같은 두 번째 알고리즘을 만들 수 있다
- vii. **방법 2: 단순 알고리즘 2:  $O(nm)$**
1. G의 MST T 계산 (MST 계산하면서, E의 에지는 가중치의 오름차순으로 이미 정렬되어 있다고 가정)
  2. **for** each edge f of E - T: # 트리에 없는 에지 f에 대해
    - a. T + {f}를 하면 사이클이 만들어짐, 그 사이클의 에지 중에서 에지 e를 선택해 제거
      - i. 어떤 에지를 선택해야 할까? 사이클에 있는 에지의 weight는 모두 f보다 작거나 같다 (왜? cycle property에 의해)
      - ii. 그럼 f의 weight에 가장 가까운 weight를 갖는 에지를 찾아 교체해야 한다 (그 에지는 바로 사이클에 T의 에지 중 weight가 제일 큰 에지가 된다) 이 에지가 e가 된다
    - b.  $S = T - \{e\} + \{f\}$  중에서  
제일 작은 비용을 갖는 신장트리를 계속 update
  3. **return S**
- 3번 루프는  $O(m)$ 번 반복되지만, 3.a 단계에서 고려하는 에지 e는 결국 T의 에지이므로  $O(n)$  시간이면 충분하다 →  $O(nm)$
  - 1번 단계의 시간이  $O(m\log n)$ 이므로 전체 수행시간 →  $O(nm)$
- viii. 더 빠른 방법은 없을까? 방법 2의 루프에서, E-T의 에지 f를 위해 제거할 T의 에지 e는  $T + \{f\}$ 의 사이클에서 weight가 가장 큰 에지임을 알았다
- ix. 예를 들어, 앞의 예제의 왼쪽 그림에 에지  $f = (u, v)$ 를 T에 삽입하면 오른쪽 그림처럼 (7-2-31-35 순서로) 사이클이 만들어진다. 사이클 에지의 weight는 f의 35보다 모두 크다. 그럼 이 세 에지 중 어떤 에지를 제거해야 할까? 31 에지를 제거해야 한다. 왜냐면 f의 35와의 차이가 제일 적은 31 에지를 제거해야 신장트리의 비용이 가장 적게 증가하기 때문이다. 결국 제거할 에지 e는 사이클에서 (f를 제외하고) weight가 가장 큰 에지여야 한다
- x. 사이클에서 최대 weight를 갖는 T의 에지는 방법 2에서처럼 사이클의 에지를 일일이 비교해서  $O(n)$  시간에 찾을 수 있다. 이 보다 더 빨리 찾아 보자
- xi. T의 아무 노드나 루트로 정하면 루트가 있는 트리가 된다. 위의 그림에서는 가장 위에 있는 노드를 루트로 정했다고 가정한다.  $f = (u, v)$ 라고 하면, T에서 u와 v의 LCA (Lowest Common Ancestor, 최근접 공통 조상) 노드를 p라고 하자. (그림 참조) 그러면 u에서 p까지의 경로에 있는 에지 중에서 최대 weight 값과 v에서 p까지의 경로의 에지 중에서 최대 weight 값 중 최대값이 우리가 찾는 에지 e의 값이 된다. 두

경로에 대한 최대 weight 값을 미리 계산해서 알고 있다면 두 값 중 최대 weight만 계산하면 되기 때문에 에지 e를 쉽게 찾을 수 있다

- xii. 이 계산은 LCA를 찾는 알고리즘을 그대로 따라하면 된다
  - 1.  $O(n \log n)$  시간의 전처리 과정을 거친 후, 두 노드 u와 v가 주어지면 LCA를  $O(\log n)$  시간에 찾을 수 있다 ([자료구조 교재 트리 편의 LCA 문제 설명 부분 참조](#))
- xiii. 위의 예제에서 35인 에지를 삽입하면 31 에지를 제거해야 한다.  $35 - 31 = 4$ 만큼 비용이 증가했다. 34인 에지를 삽입하면 31 에지가 제거되어 3만큼 비용이 증가했다. 12 에지는 7 에지가, 10 에지는 21 에지가, 15에지는 5 에지가 선택되어 제거된다. 이 중에서 비용이 가장 적게 증가한 경우는 34인 에지를 삽입하고 31 에지를 제거한 경우이다. (예제의 가운데 그림)
- xiv. **방법 3: 빠른 알고리즘:  $O(m \log n)$** 
  1. G의 MST T 계산 (MST 계산하면서, E의 에지는 가중치의 오름차순으로 이미 정렬되어 있다고 가정)
  2. LCA 알고리즘과 같은 방식으로 T를  $O(n \log n)$  시간에 전처리하여 다음 질의를  $O(\log n)$  시간에 처리하도록 함
    - a. 에지  $(u, v)$ 가 질의로 주어지면,  $\text{LCA}(u, v) = p$ 를 계산하고, u에서 p 경로에 있는 최대 weight 값과 v에서 p까지의 최대 weight 값을  $O(\log n)$  시간에 알려줌
  3. **for** each edge  $f = (u, v)$  of  $E - T$ : # 트리에 없는 에지 f에 대해
    - a.  $T + \{f\}$ 를 하면 사이클이 만들어짐, 그 사이클의 에지 중에서 f와 weight 차이가 가장 적은 에지 e (= 사이클에서 f를 제외하고 가장 큰 weight를 갖는 에지)를  $O(\log n)$  시간에 선택 (단계 2의 준비를 통해 가능)
    - b.  $S = T - \{e\} + \{f\}$  중에서 제일 작은 비용을 갖는 신장트리를 계속 update
  4. **return S**
    - 3번 루프는  $O(m)$ 번 반복되지만, 3.a 단계에서  $O(\log n)$  시간에 에지 e를 찾을 수 있기 때문에 결국  **$O(m \log n)$**  시간이면 충분

## 10. [고급] Network Flow 문제

- a. source 노드  $s$ 와 sink 노드  $t$ 가 하나씩 존재하는 **방향** 그래프  $G = (V, E)$ :
- $s$ 에서  $t$ 를 향하는 그래프의 에지를 통과하는 어떤 흐름(flow)을 가정
  - 에지  $(u, v)$ 의 용량(capacity)  $c(u, v)$  : 에지  $(u, v)$ 를 통해 흐를 수 있는 최대 용량
- b. 예: 에지의  $a/b$ 는 최대  $b$ 만큼 흐를 수 있고, 현재는  $a$ 만큼 흐르고 있음을 의미 (총 5만큼이  $s$ 에서 흘려 보내면 중간에 여러 노드를 거쳐  $t$ 에 5만큼 도착 가능함.)



- c. 플로우 (flow of  $G$ )  $f$  : 아래 두 조건을 만족하는  $G$ 의 흐름
- capacity constraint:** (용량 제한 조건)  
에지  $(u, v)$ 에 실제 흐르는 양  $f(u, v)$ 이라 하면, 항상  $c(u, v)$ 보다 크지 않아야 한다:  
 $f(u, v) \leq c(u, v)$
  - conservation of flows:** (용량 보존 법칙)  
 $s, t$ 를 제외한 어떠한 노드  $v$ 에 대해서,  $v$ 로 들어오는 flow 양의 합과  $v$ 에서 나가는 flow 양의 합은 항상 같아야 한다  
⇒ 노드  $v$ 의 모든 incoming, outgoing 에지의 flow의 합은 항상 같아야 한다
  - skew symmetry:**  $f(u, v) = -f(v, u)$   
⇒  $u$ 에서  $v$ 로  $f$ 만큼 flow가 있다면,  $v$ 에서  $u$ 로  $-f$ 만큼 flow가 있다고 해석 가능  
⇒  $v$ 에서  $u$ 로  $f$ 만큼 flow를 되돌려줄 수 있다는 의미로도 해석 가능

- d. 플로우 값 (flow value) =  $|f|$  = ( $s$ 에서 나가는 flow의 합)
- $s$ 에서 나간 flow는 용량 보존 법칙에 의해, 모두  $t$ 에 도달한다. 따라서 또는  $t$ 로 들어오는 flow 합과 같다

## e. Maximum flow 문제

- 입력 그래프  $G = (V, E)$ 에서 **플로우 값이 최대**가 되는 플로우  $f$ 를 찾아라! (위의 그래프 예에서는 최대 플로우 값이 5가 됨)
- f.  **$s-t$  컷** ( $s-t$  cut)  $C$ , 컷 집합 (s-t cut set)  $X_C$ , 컷 용량 (s-t cut capacity)  $c(s, T)$ :
- $s-t$  cut  $C = (S, T)$ 는  $V$ 를 두 부분 집합  $S$ 와  $T$ 로 나눈 분할 (단,  $S$ 에는  $s$ 가  $T$ 에는  $t$ 가 속함)
  - $s-t$  cut set  $X_C$ :  $S$ 에 있는 노드와  $T$ 에 있는 노드를 연결하는 에지의 집합

- iii. **s-t cut capacity  $c(S, T)$** :  $x_c$ 에 있는 에지의 용량 합
- iv. 예: 위의 그래프 예를 보고 세 가지 정의를 따져보자

#### g. Minimum cut 문제

- i. 입력 그래프  $G = (V, E)$ 에서 컷 용량(플로우 아님!)이 **최소인**  $s-t$  컷을 찾아라!
- ii. 위의 그래프 예에서  $S = \{s, o, p\}$ ,  $T = \{q, r, t\}$ 인 컷과  $S = \{s, o, p, q, r\}$ ,  $T = \{t\}$ 인 컷이 최소이며 이때 컷 용량은 5이다

#### h. Maximum Flow 문제 vs. Minimum Cut 문제

- i. 어떤 관계일까?
- ii. 직관적으로 특정  $s-t$  컷을 생각해보면,  $s$ 에서 출발한 플로우가 이 컷의 에지들을 통과해  $t$ 로 가는 것이므로 이 컷의 용량을 절대 초과할 수 없다. 결국, 최소 컷의 용량을 초과할 수는 없게 된다
- iii. 그럼 이 최소 컷의 용량이 최대 흘려 보낼 수 있는 플로우의 양인가? 그렇다!

#### i. [중요] Max-Flow Min-Cut Theorem

**Maximum Flow의 값 = Minimum Cut의 용량**

#### j. Maximum Flow Algorithms: 방향 그래프 $G = (V, E)$

- i. Ford-Fulkerson algorithm:  $O(|E| \max|f|)$  - 1956
    - 1. 최대 플로우 값  $f$ 가 수행시간이 포함되어 있음 → pseudo polynomial 수행 시간
  - ii. Edmonds-Karp algorithm:  $O(|V||E|^2)$  - 1972
  - iii. Dinic's algorithm:  $O(|V|^2|E|)$  - 1970
  - iv. Fastest algorithm:  $O(|V||E|)$  - 2013
- k. 가장 고전적이면서 대표적인 두 알고리즘 - Ford-Fulkerson 알고리즘과 Edmonds-Karp 알고리즘을 차례대로 설명한다

## I. Ford-Fulkerson algorithm vs. Edmonds-Karp algorithm

- i. **Idea:** flow가 없는 상태에서 s에서 t로의 flow가 흐를 수 있는 경로를 반복해서 찾아가며 점진적으로 flow를 증가시키는 **그리디(greedy)** 알고리즘
- ii. 잉여 그래프 (residual graph)  $G_f$ 가 중요 역할
  1.  $G_f$ 의 노드 집합은  $G$ 의 노드 집합과 동일
  2.  $G$ 의 에지  $(u, v)$ 를  $G_f$ 에서도 정의:
    - a.  $c(u, v) - f(u, v) > 0$ 인 경우에만  $c_f(u, v) = c(u, v) - f(u, v)$ 의 용량으로 해당 에지를  $G_f$ 에서 정의 (즉,  $(u, v)$ 로는  $c(u, v) - f(u, v)$ 만큼 더 흘려보낼 수 있는 여유가 있음을 의미 - residual flow가 존재함을 의미)
    - b. 반대방향 에지  $(v, u)$ 를  $G_f$ 에서 정의함 (유의!) 용량  $c_f(v, u) = f(u, v)$ 으로 정의 (즉, 노드  $v$ 의 입장에서는  $f(u, v)$ 을 노드  $u$ 로부터 받고 있다. 그런데 이 양이 나중에 조정되어 오히려 줄 수도 있다. 가장 많이 줄면  $-f(u, v)$ 만큼 줄 수 있다. 이런 변화를 반영하기 위해, 거꾸로  $v$ 에서  $u$ 로 가는 (backward) 에지를 정의하고  $f(u, v)$  만큼의 잉여 용량을 할당한다. 즉,  $v$ 가  $u$ 로부터 받은 양의 일부를 다시 돌려주는 경우를 고려한다는 의미이다. 최대 받은 만큼 돌려 줄 수 있으므로  $c_f(v, u) = f(u, v)$ 로 정의한다)
  3. 만약 잉여 그래프  $G_f$ 에서  $s$ 에서  $t$ 로 가는 경로가 존재한다는 의미는 그 경로의 에지 중 최소 잉여 값을 갖는 만큼 더 흐르게 할 수 있다는 의미이다. 이 경로를 **augmenting 경로**라 부른다
- iii. 이 두 알고리즘은 잉여 그래프에서 반복적으로 augmenting 경로를 찾아 플로우의 값을 점진적으로 늘려가는 방법이다
- iv. **Pseudo code** [from Wikipedia]
 

```

 $G_f \leftarrow \text{copy}(G)$
for all edges (u, v) in G :
 $f(u, v) = 0$ # 초기화

while there is an augmenting path P from s to t in G_f such that
 $c_f(u, v) > 0$ for all edges of P (*): # augmenting path

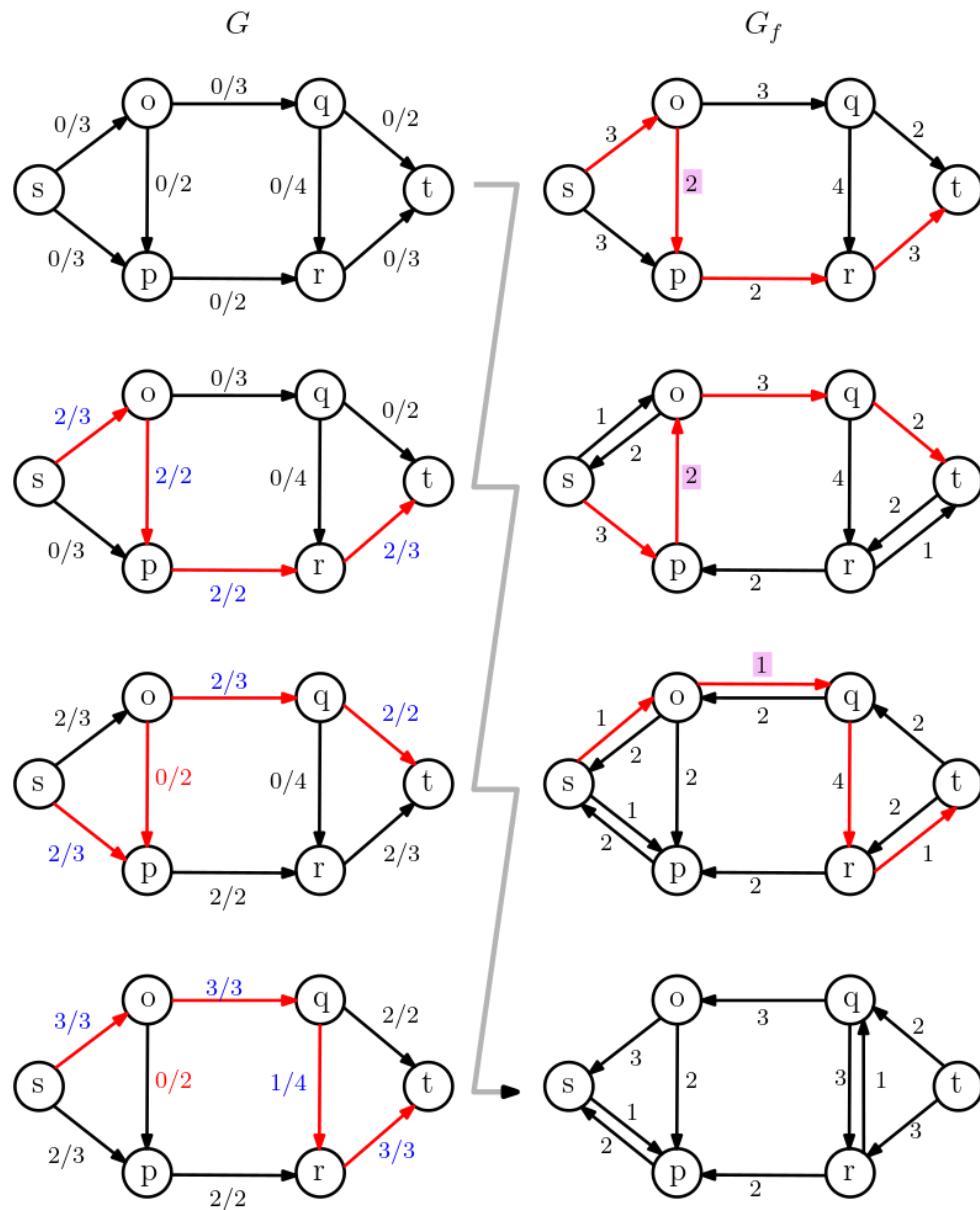
 find $c_f(P) = \min\{ c_f(u, v) \mid (u, v) \text{ in } P \}$
 # 최소 여유 용량 $c_f(P)$ 는 P 의 모든 에지에 그 용량만큼
 # flow 양을 늘릴 수 있다는 뜻

 # P 의 에지는 flow 증가, 반대 방향으로는 감소시킴

1. for each edge (u, v) in P : # P = augmenting path in G_f
 # add $c_f(P)$ to the flow of edges on P
 $f(u, v) = f(u, v) + c_f(P)$
```

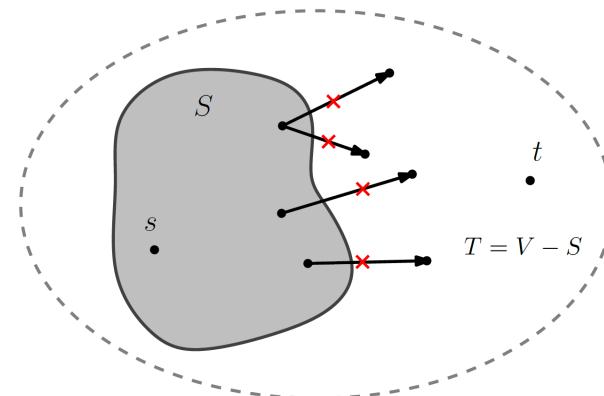
```
update G_f
G_f[u][v] = G_f[u][v] - c_f(P)
G_f[v][u] = G_f[v][u] + c_f(P)
```

- v. (\*)에서  $G_f$ 의 경로  $P$ 를 DFS로 찾으면 Ford-Fulkerson 알고리즘이라 하고, BFS로 찾으면 Edmonds-Karp 알고리즘이라 한다
- DFS와 BFS의 결정적인 차이는 BFS로 찾은 경로는  $s$ 에서  $t$ 로 가는 에지 수가 제일 작은 경로라는 것이다
- vi. (\*) while 반복문은  $G_f$ 의 경로  $P$ 가 존재하지 않을 때까지 반복된다. 즉, 잉여 그래프  $G_f$ 의 source 노드  $s$ 에서 sink 노드  $t$ 까지의 경로가 하나도 존재하지 않는다는 의미이다.
- vii. 앞 장에 제공된 그래프 예제를 가지고 Ford-Fulkerson 알고리즘을 돌려보자



**그림 해설:** 왼쪽이 원 그래프  $G$ , 오른쪽이 잉여 그래프  $G_f$ 이다.  $G_f$ 의 초기 형태는  $G$ 와 같고, 에지마다  $G$ 의 용량을 할당한다.  $G_f$ 에서  $s$ 에서  $t$ 까지의 경로를 하나 찾는다. 이 경로의 에지의 최소 용량만큼 흘려보낼 수 있다. 이 에지에 해당하는  $G$ 의 에지에 최소 용량만큼 흘려보낸다.  $s \rightarrow o \rightarrow p \rightarrow r \rightarrow t$ 로 2만큼 흘려보냈다. 그러면  $G_f$ 의 에지의 용량을 적절히 변경해야 한다.  $s \rightarrow o$ 는  $2/3$ 이므로 2만큼 흘려보냈으니  $G_f$ 에서  $s \rightarrow o$ 의 잉여 용량은  $3 - 2 = 1$ 로 감소한다. 2만큼  $o$ 로 흘러간 양이  $s$ 로 돌아오는 경우도 고려해야 한다. 즉, 흐른 양 2가 고정된 것이 아니라 늘어날 수도 줄어들 수도 있다. 줄어드는 현상을  $G_f$ 에 반영하기 위해  $o \rightarrow s$ 로 가는 반 대방향 에지를 만들어 최대 2만큼 돌려줄 수 있으므로 2의 잉여 용량을 부여한다. 다음으로,  $o \rightarrow p$ 는  $2/2$ 가 되어 용량만큼 흐르게 되니 잉여 그래프에서는  $o \rightarrow p$ 로의 에지는 사라지고, 반대 방향으로 용량 2만큼의 에지만 새로 추가된다. 이렇게 업데이트된  $G_f$ 에서 다시  $s$ 에서  $t$ 로의 경로를 찾는다. 이 경로는  $s \rightarrow p \rightarrow o \rightarrow q \rightarrow t$ 이다. 최소 잉여 용량이 2이기 때문에 해당 경로의 에지에 대응되는  $G$ 의 에지에 2만큼 흘려 보낸다. 주의할 것은  $p \rightarrow o$ 는 실제  $G$ 에 존재하지 않는 에지로  $o \rightarrow p$ 의 반대 방향 에지다. 따라서  $o \rightarrow p$ 로 흘려보냈던 양 중에 2만큼 줄여야 한다 (원래 이런 효과를 위해, 역방향 에지를  $G_f$ 에 삽입했던 것이므로…). 이런 방식으로  $s$ 에서  $t$ 까지 경로가  $G_f$ 에 더 이상 존재하지 않을 때까지 반복한다.

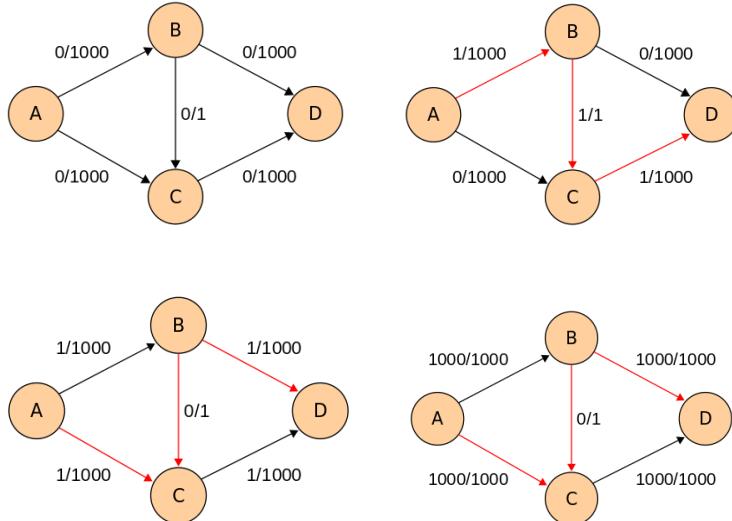
viii. 아래 그림처럼  $G_f$ 에서  $s$ 에서 도달 가능한 노드의 집합을  $S$ 라 하고  $T = V - S$ 로 정의한다.



1. 원 그래프  $G$ 에서  $S$ 의 노드  $u$ 에서  $T$ 의 노드  $v$ 로 향하는 에지의 집합은  $s-t$  컷으로 정의되며,  $c(u, v) = f(u, v)$ 가 성립한다. (아니라면  $G_f$ 에서  $c_f(u, v) > 0$ 이 되어  $v$ 가  $S$ 에 속해야하는 모순이 발생한다)
2. 원 그래프  $G$ 에서 이런 에지의 집합은  $s-t$  cut이고  $c(u, v) = f(u, v)$ 이므로  $c(S, T) = f(S, T)$ 이 된다. **f(S, T)**는 동시에 max flow이다. 다음의 여섯 가지 (부)등식을 따라가면  $f(S, T) = \text{max\_flow}(s, t)$ 임을 증명 가능하다
  - a.  $c(S, T) \geq \text{min\_cut}(s, t)$  (이유?)
  - b.  $f(S, T) \geq \text{min\_cut}(s, t)$  (이유:  $f(S, T) = c(S, T)$ )
  - c.  $\text{min\_cut}(s, t) = \text{max\_flow}(s, t)$
  - d.  $f(S, T) \geq \text{max\_flow}(s, t)$
  - e.  $f(\text{any cut}) \leq \text{max\_flow}(s, t) \Rightarrow f(S, T) \leq \text{max\_flow}(s, t)$
  - f.  $f(S, T) = \text{max\_flow}(s, t)$  (d와 e 두 식이 성립해야 하므로)

## ix. (\*) 부분의 경로 P를 DFS로 찾는 Ford-Fulkerson 알고리즘

1. 수행시간: **while** 루프를 x번 돋다고 하면,  $O(x * \text{DFS 시간})$ 이 된다. DFS는  
에지 개수에 비례하는 시간만큼 필요하므로  $O(|E|)$ 가 된다.
2. 그럼 **while** 루프를 최악의 경우엔 몇 번이나 돌 수 있을까?
3. 아래 그림과 같은 최악의 경우에는 max-flow 양만큼 즉,  $\max|f|$  번 돌 수도  
있다.
4. 따라서, Ford-Fulkerson 알고리즘의 수행시간은  $O(|E| \max|f|)$



1998번 반복 후

5. 에지의 용량이 무리수인 경우에는 알고리즘이 끝나지 않는 경우도 존재한다.  
[이미지 출처: [Non-terminating example](#) from Wikipedia] 그러나 용량이  
정수인 경우에는 (시간이 오래 걸려도) 최대 flow를 항상 찾는다

## x. (\*) 부분의 경로 P를 BFS로 찾는 Edmonds-Karp 알고리즘

1. BFS로 탐색하면, s에서 t까지의 (에지 개수가 가장 적은) 최단 경로를  
찾는다. 이 최단 경로를 P로 선택한다는 의미이다
2. 최단 경로를 선택하여 augmenting을 하면 Ford-Fulkerson 방법에서처럼  
 $\max|f|$  번 만큼 augmenting 하는 상황은 발생하나? 위의 그래프에  
적용해보자
3. 수행시간:  $O(|V||E|^2)$ 
  - a. BFS는 에지를 모두 탐색하는 것이므로  $O(|E|)$  시간
  - b. augmenting 경로 P를  $O(|V||E|)$  번 발견하여 augmenting하면 더  
이상 augmenting 경로가 존재하지 않음을 보일 수 있음:  $O(|V||E|)$   
시간
  - c. 결국,  $O(|V||E|^2)$  시간이면 충분

4. 증명: augmenting을 최대  $O(|V||E|)$  번만 반복하면 더 이상 augmenting 경로가 존재하지 않는다 (주장1 + 주장2)
5. 주장 1: 임여 그래프에서 augmenting을 반복할 수록, s에서 다른 모든 노드까지의 최단 거리는 절대 감소하지 않는다. (즉, 같거나 증가한다)
  - a. BFS 트리: 첫 레벨엔 source 노드 s만 포함하고, s에 인접한 노드들로 두 번째 레벨을 구성하고, 두 번째 레벨의 노드에 인접한 노드들로 세 번째 레벨을 구성하는 식으로 정의
  - b. 최단 경로는 레벨 별로 정확히 하나씩 선택한 노드들로 구성됨
  - c. augmenting 할 때마다 최소 하나 이상의 에지  $(u, v)$ 는 임여 용량을 모두 소진한다 (saturated 된다고 표현)  $\Leftarrow$  augmenting 단계 1, 2 참조
  - d.  $(u, v)$ 가 현재 augmenting에서 saturated 된다고 가정하면, 당연히  $u$ 와  $v$ 의 레벨은 인접해야 함.  $u$ 는 레벨  $i$ ,  $v$ 는 레벨  $i+1$ 이라고 가정
  - e. saturated 되면 이 에지는 임여 그래프에서는 **사라지고**, 역방향 에지인  $(v, u)$ 만 남게됨. 그리고 BFS 트리에 있는 **다른 에지는 변하지 않음**에 유의. 최단 경로가 에지  $(u, v)$ 를 통했다면 더 이상 그럴 수 없다. s에서  $v$ 로 간 후, 에지  $(v, u)$ 를 이용해서  $u$ 로 움직인 다음에 목표 노드로 이동할 수는 있다
  - f. 중요한 사실은  $(v, u)$ 는  $v$ 에서  $u$ 로 가는 에지이므로 레벨  $i+1$ 에서  $i$ 로 향하는 에지임!  $\Rightarrow$  경로에 포함되더라도 레벨  $i+1$ 의  $v$ 를 방문한 후, 레벨  $i$ 의  $u$ 로 거꾸로 간 후, 다시 레벨  $i+1$  이하의 노드를 방문하기 때문에 경로 길이가 오히려 더 늘어남  $\Rightarrow$  따라서 이 에지는 **다음 번 augmenting 경로의 길이를 늘리는 데 도움이 되어도 줄이는 데 전혀 도움이 되지 않는다는 사실!**
6. 주장 2: 임의의  $(u, v)$  에지는 최대  $|V|/2$ 번 (saturation을 위한) augmenting 가능하다
  - a.  $(u, v)$  에지가 임여 그래프에서 사라지지만 부활하여 augmenting 경로에 다시 포함될 수 있다. 부활하기 위해선 **반드시** 반대 방향 에지  $(v, u)$ 가 augmenting 경로로 적어도 한 번은 선택되어야 한다
  - b.  $(v, u)$ 가 augmenting 경로로 선택되기 위해선  $v$ 로 와서  $u$ 를 방문해야 하는데, 그러면  $v$ 의 레벨은 최소  $i+1$ 이므로  $u$ 의 레벨은 최소  $i+2$  이상의 레벨이 되어야 한다
  - c. 즉,  $u$ 의 이전 레벨  $i$ 보다 최소 2 증가한  $i+2$  레벨에 있는 셈이다. 이후에  $(u, v)$ 가 saturation 되기 위해 경로에 포함된다면  $v$ 의 레벨은 최소  $i+3$ 이 되어야 한다. 다시  $(v, u)$ 가 경로에 포함되기 위해  $u$ 의 레벨은 최소  $i+4$ 이다. 결국,  $u$ 의 레벨은  $i \rightarrow i+2 \rightarrow i+4 \rightarrow \dots$  처럼 2씩 증가하게 된다. 그런데 레벨의 최대값은  $V$  이하이다 (왜? 경로에 포함된 노드는 정의상 모두 달라야 하므로) 따라서  $(u, v)$ 의 에지가

선택-삭제-부활하는 augmenting 과정은 최대  $|V|/2$ 번을 넘지 않는다!

7. 모든 에지가 최대  $|V|/2$ 번 augmenting 가능하고, 에지는  $|E|$ 개 존재하므로 발생 가능한 augmenting은 최대  $|V||E|/2 = O(|V||E|)$ 번이다

xi. [Python Code] Edmond-Karp Algorithm:

1. 그래프  $G$ 는 2차원 리스트로  $G[u][v]$ 는 [**flow**, **capacity**]의 리스트를 저장한다고 가정 (즉, 인접행렬로 표현)
2.  $G_f$ 는  $G$ 의 residual 그래프 역시 인접행렬  $G_f[u][v]$ 로 표현. 에지  $(u, v)$ 가  $G$ 에 존재한다면,  $G_f[u][v] = G[u][v][1] = c(u, v)$ 로 초기화 되어 있다고 가정. 에지  $(u, v)$ 가 존재하지 않는다면 0으로 초기화
3. parent 리스트는 BFS 경로에 포함된 노드의 부모 노드 (이전 노드)를 저장하기 위해 사용
4. Queue 자료구조를 위해 **from queue import Queue** 필요
5.  $G$ ,  $n$ ,  $source$ ,  $sink$  모두 전역 변수

```
def find_path_BFS(source, sink):
 parent = [-1] * n
 parent[source] = -2
 q = Queue()
 q.enqueue((source, float("inf")))
 while q.empty() == False:
 x = q.dequeue()
 u, flow = x[0], x[1]
 for v in range(n):
 if $G_f[u][v] == 0$: # no edge (u, v) in G_f
 continue
 if parent[v] == -1 and $G_f[u][v] > 0$:
 parent[v] = u
 new_flow = min(flow, $G_f[u][v]$)
 if v == sink: # find an aug. path
 return new_flow, parent
 q.enqueue((v, new_flow))
 return 0, None
```

```

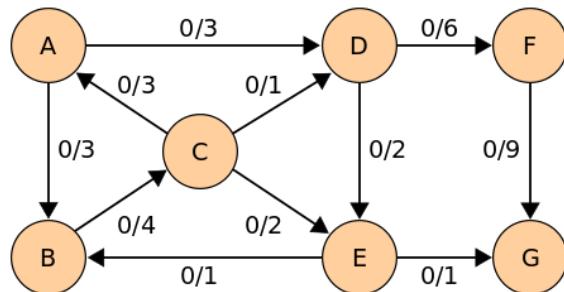
def max_flow(source, sink, Gf):
 flow = 0
 while True:
 new_flow, parent = find_path_BFS(source, sink)
 if new_flow == 0: # no more augmenting path
 return flow
 flow += new_flow
 curr = sink
 while curr != source:
 prev = parent[curr]

 # update Gf graph
 Gf[prev][curr] -= new_flow
 Gf[curr][prev] += new_flow

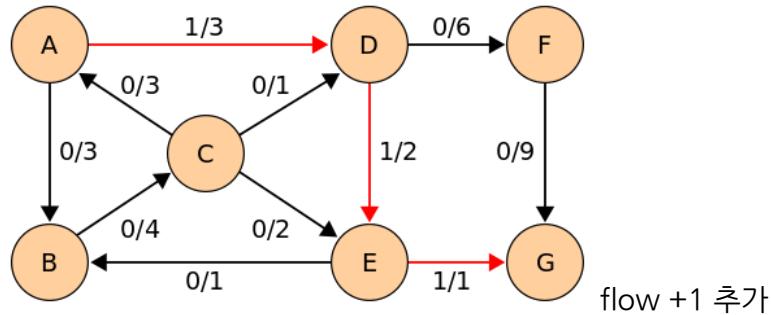
 # update flow of edges on path
 G[prev][curr][0] += new_flow
 curr = prev

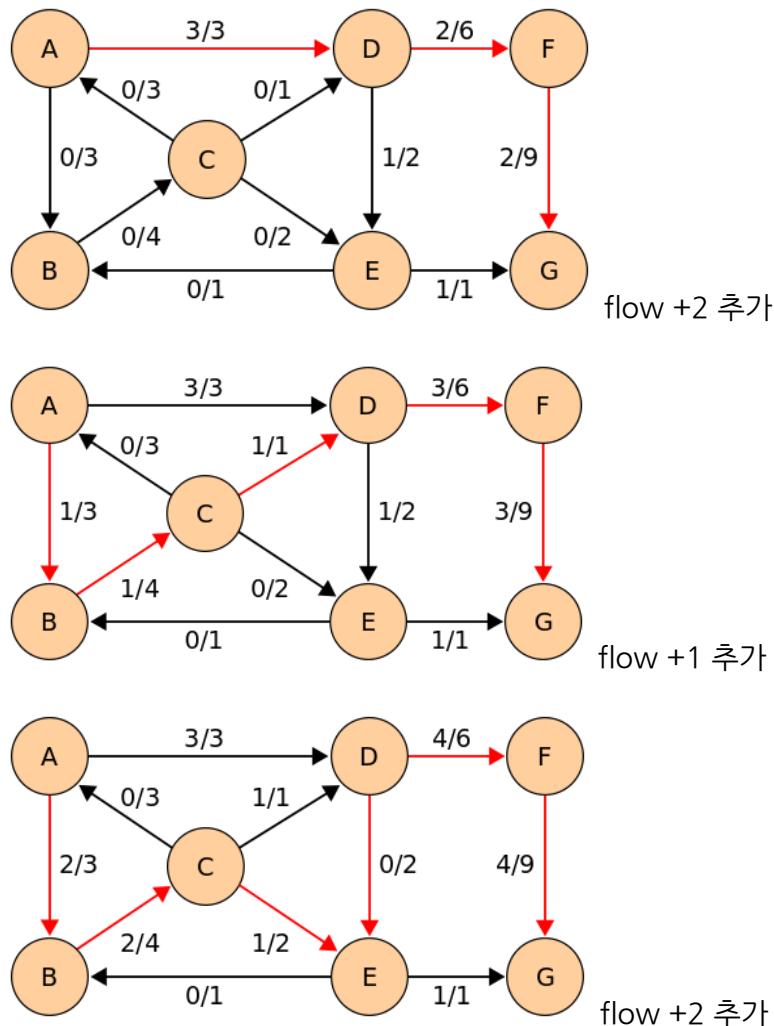
```

## xii. 예제 (wikipedia)



네 개의 augmenting path를 찾아 capacity와 flow를 update한다





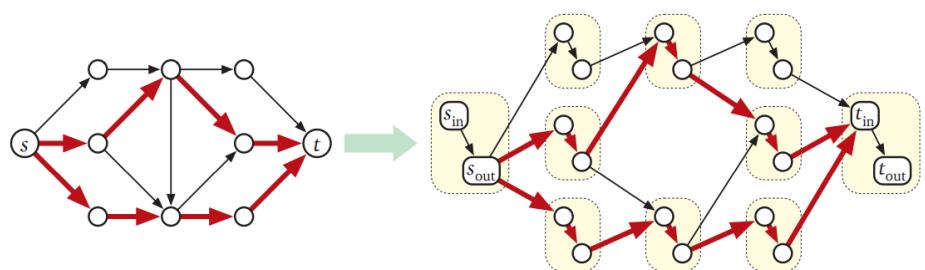
m. 현재 가장 빠른 maximum flow 계산 알고리즘

i. Orlin (2013):  $O(|V||E|)$  시간 알고리즘

1. 여러 기법과 자료구조를 사용하는 복잡한 알고리즘
2. 에지 수가  $O(|V|)$ 개라면  $O(|V|^2/\log |V|)$  시간에 수행 가능함

### n. 응용 문제 1: Edge-disjoint paths, Vertex-disjoint paths

- i. 여러  $s-t$  경로 중에서 경로의 에지가 중복이 되지 않는다면  $s-t$  경로들을 edge-disjoint라고 부른다. 노드가 중복이 되지 않는다면 vertex-disjoint라고 부른다.
- ii. 주어진 그래프에서 최대 개수의 edge-disjoint 경로와 vertex-disjoint 경로를 찾고 싶다면 어떻게 해야 할까?
  1. Disjoint path 문제는 응용성이 높은 문제이다. 두 노드를 연결하는 disjoint 경로가 많다는 건 경로 중 일부가 훼손되더라도 두 노드가 연결된다는 뜻이므로 연결도 (connectivity)가 높다는 의미이다. 연결도는 결국 네트워크의 신뢰성 (reliability)의 핵심이다
- iii. Edge-disjoint 경로
  1. 그래프의  $G$ 의 모든 에지에 capacity = 1로 정한 후, maximum flow를 구한다. 각 에지에는 flow가 0 아니면 1이 흐름으로 maximum flow에 의해 선택되는  $s-t$  경로에 포함된 에지들은 서로 edge-disjoint이다.
  2. Conservation rule에 따라 한 노드에 들어오는 flow의 합이 나가는 flow의 합과 같기에, 해당 노드를 지나는 경로의 개수가 flow의 합만큼 존재함을 알 수 있다. 따라서,  $f^*$ 가 maximum flow라면  $|f^*|$ 개의 edge disjoint  $s-t$  경로가 존재한다. 이 개수의 경로가 당연히 최대이다
- iv. Vertex-disjoint 경로
  1. maximum flow에 속한 경로가 서로 다른 노드만으로 구성되기 위해서는 한 노드를 지나는 경로는 최대 하나만 되도록 해야 한다
  2. 이 성질이 만족되도록 입력 그래프를 아래 그림과 같이 수정한다

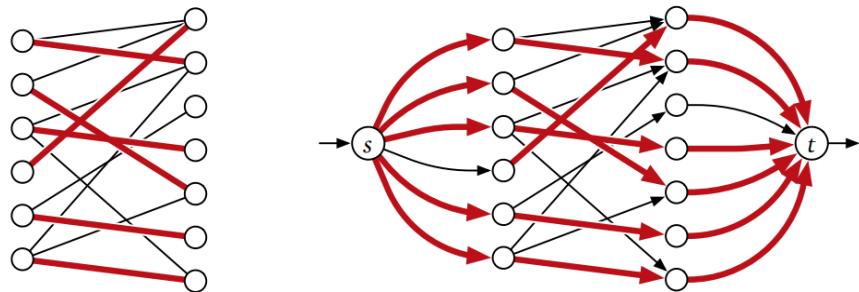


Jeff Erickson 알고리즘 강의 노트에서 인용

3. 입력 그래프  $G$ 의 각 노드  $v$ 를 두 개의 노드 ( $v_{in}$ 과  $v_{out}$ )로 교체해서 새로운 그래프  $G'$ 을 위 그림처럼 정의한다. 그리고 각 에지에 capacity = 1을 지정한다
4. 그러면  $G'$ 에서  $s_{out} - t_{in}$  maximum flow를 계산하면, 한 노드를 통과하는 flow는 ( $v_{in}$ ,  $v_{out}$ )의 capacity가 1이므로 오직 하나 뿐이다. 따라서 maximum flow의 값이 vertex-disjoint 경로의 최대 개수가 된다

### o. 응용 문제 2: Bipartite Matching

- i. 이분 그래프 (bipartite graph)  $G = (X, Y, E)$ :
  - 1. 노드 집합이  $X$ 와  $Y$ 로 구성되고 ( $V = X \cup Y$ ), 에지는  $X$ 의 노드와  $Y$ 의 노드 사이에서만 정의되는 그래프
  - 2. 즉,  $X$ 의 노드 사이에는 에지가 없고,  $Y$ 의 노드 사이에도 에지가 없는 그래프로 여러 응용 문제에서 자주 등장
  - 3. 예:  $X$ 는 사람(people) 집합,  $Y$ 는 음식(food) 집합인 경우, 사람이 좋아하는 음식 사이에 에지를 정의하면 이분 그래프가 정의됨
- ii. 매칭 (matching)
  - 1. 에지의 부분 집합으로 정의되며, 그래프의 각 노드는 최대 하나의 매칭에 포함된 에지에 인접해야 한다
  - 2. 즉, 한 노드에 인접한 두 개 이상의 에지가 매칭에 포함되면 안된다
  - 3. 예를 들어, 사람과 좋아하는 음식이 중복되지 않게 짹을 짓는 경우
- iii. 매칭과 flow의 관계
  - 1. 소스 노드  $s$ 와 싱크 노드  $t$ 를 하나씩 이분 그래프에 추가해보자
  - 2. 소스 노드  $s$ 에서 이진 그래프의  $X$ 의 모든 노드로의 에지를 추가하고,  $Y$ 의 모든 노드에서 싱크 노드  $t$ 로 향하는 에지를 추가한다
  - 3. 새로 정의한 그래프를 보면 아래와 같다



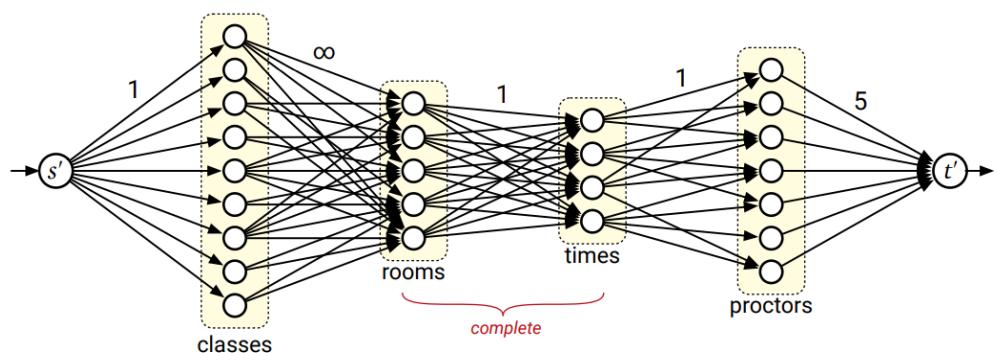
Jeff Erickson 알고리즘 강의 노트에서 인용

- 4. 모든 에지에 capacity 1을 할당한 후, maximum flow를 계산해보자
  - a.  $X$ 의 각 노드는  $s$ 로부터 최대 1의 flow를 받을 수 있기에 최대 1의 flow를 에지 하나를 통해  $Y$ 의 노드로 내 보낼 수 있다. 즉,  $X$ 의 노드에서 나가는 에지 중에 최대 하나의 에지에만 1의 flow가 흐를 수 있다
  - b.  $Y$ 의 노드에서  $t$ 에 연결된 에지를 통해, 최대 1의 flow만 보낼 수 있기에  $X$ 의 노드에서 들어오는 에지 중에서 (최대) 한 에지의 flow만 받을 수 있다
  - c. 결국, 한 노드에 인접한 두 에지에 flow가 동시에 흐르지 않기 때문에 maximum flow에 해당하는  $X$ 와  $Y$  사이의 에지 집합은 매칭이 된다
  - d. 역으로, 임의의 매칭이 있다면, 매칭에 속하는 각 에지에 1의 flow가 흐르는 flow를 항상 정의할 수 있다
  - e. flow의 값이 매칭의 에지 개수와 같기에, maximum flow = maximum matching이 된다
- 5. Ford-Fulkerson 알고리즘을 적용해보자

- a. residual 그래프에서 augmenting path를 찾아서 flow를 1씩 증가시키는 과정을 몇 번 반복하면 끝날까?
- [질문] augmenting path는 X와 Y의 노드를 번갈아 지나갈 수 있다. 이 경로의 의미를 생각해보자
  - 과정마다 매칭되는 에지의 개수가 하나씩 증가하기에 최대  $\min(|X|, |Y|)$ 번만 하면 더 이상 augmenting path가 존재하지 않게 된다
- b.  $\min(|X|, |Y|) = O(|V|)$ 이고 augmenting path는  $O(|E|)$  시간에 찾을 수 있으므로  $O(|V||E|)$  시간이면 충분하다

p. 응용 문제 3: Tuple Selection

- Bipartite matching 문제를 일반화해보자
- 예를 들어, 컴퓨터공학과의 기말고사를 위해, 수업별로 시험 장소와 시간대와 감독관을 배정하려고 한다
- $n$  개의 수업,  $r$ 개의 시험 장소 (room),  $t$ 개의 시간대,  $p$ 명의 감독관이 주어진다
  - 각 수업별로 수강생 수가 주어지고 시험 장소별로 좌석의 개수가 주어진다
  - 각 감독관은 감독이 가능한 시간대가 주어진다. 감독관은 최대  $k$ 개 이하의 시간대만 감독할 수 있다고 가정한다
- 하나의 배정은 (수업, 장소, 시간대, 감독관) 튜플을 결정하는 것이다
- 배정 원칙:**
  - 한 수업은 한 장소에만 배정 (한 수업의 시험을 두 장소에서 보지 않는다는 의미)
  - 한 장소는 하나의 시간대에만 배정 (여러 시간대에 나눠 시험을 보지 않는다는 의미)
  - 한 감독관은 최대  $k$ 개의 시간대에만 배정 (주 52시간 노동시간을 지키기 위해)



Jeff Erikson 알고리즘 강의노트에서 인용

- vi. 소스 노드  $s'$ , 수업, 장소, 시간대, 감독관, 싱크 노드  $t'$ 으로 연결된 방향 그래프를 위의 그림처럼 정의한다
1.  $s'$ 에서 수업들로 에지를 정의하고 capacity = 1로 지정 (하나의 flow는 하나의 수업에 대한 시험 배정을 의미)
  2. 수업에서 장소로의 에지는 수업의 수강생 수가 장소의 좌석 수보다 크지 않는 경우에 정의. 에지의 capacity = 1로 지정 (그림에서는 infinity 큰 값으로 지정되어 있지만, 어차피 수업으로 들어오는 flow는 0 또는 1이므로 하나의 에지에만 flow가 흐르게 되고, 하나의 수업당 하나의 장소만 배정되어야 하기에 1로 지정해도 된다)
  3. 한 장소에서 모든 시간대로 에지 정의한다. 한 장소는 수업 하나에 대해 정확히 하나의 시간대만 배정되어야 하므로 capacity = 1로 지정
  4. 각 시간대에 감독이 가능한 감독관들에게 에지 정의. 한 수업의 감독관은 한 명이므로 capacity = 1로 지정
  5. 한 명의 감독관은 최대  $k = 5$ 개의 시간대를 감독할 수 있다면, 감독관에서 싱크 노드  $t'$ 로 향하는 에지를 정의하고 capacity = 5로 지정. (최대 5개의 시간대로부터 flow를 받을 수 있기 때문에)
- vii. 위와 같은 그래프에서 Ford-Fulkerson 알고리즘을 수행해 maximum flow를 찾으면 된다. 이는 여러 개의  $(s' - t')$ -flow로 구성되는 데, 하나의 flow가 하나의 배정을 나타낸다

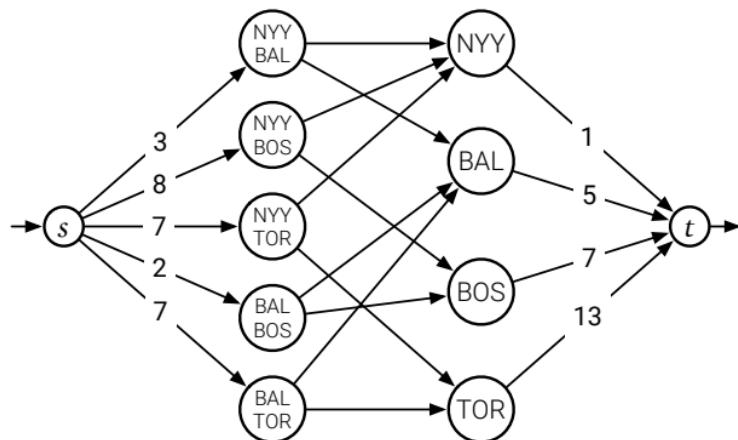
q. 응용 문제 4: Baseball elimination

- i. 현재 야구 성적과 남은 경기 일정을 토대로 특정 팀 (꼴지 팀)이 우승할 가능성이 있는지 결정하는 문제로 매우 유명한 문제다
- ii. 예를 들어, 아래 표는 1996년 8월 30일 MLB American League East의 현재 성적이다 (Jeff Erikson의 강의노트에서 인용)
  1. 현재의 승-패, 남은 게임과 앞으로 남은 같은 리그의 팀과의 경기 수이다
  2. Detroit Tigers가 꼴찌인데, 남은 27 게임을 모두 이긴다면 우승이 가능할까? 그러면 76승이 된다. New York Yankees가 남은 게임을 모두 진다면 가능하지 않나? 그렇지 않다. Yankees가 Boston과 8 게임에서 Boston이 모두 이기기 때문에 Boston은 최소 77승이 되어 Tigers는 우승할 수 없다

| Team              | Won-Lost | Left | NYY | BAL | BOS | TOR | DET |
|-------------------|----------|------|-----|-----|-----|-----|-----|
| New York Yankees  | 75-59    | 28   |     | 3   | 8   | 7   | 3   |
| Baltimore Orioles | 71-63    | 28   | 3   |     | 2   | 7   | 4   |
| Boston Red Sox    | 69-66    | 27   | 8   | 2   |     | 0   | 0   |
| Toronto Blue Jays | 63-72    | 27   | 7   | 7   | 0   |     | 0   |
| Detroit Tigers    | 49-86    | 27   | 3   | 4   | 0   | 0   |     |

3. 이런 분석 방법이 아닌 maximum flow 문제로도 변환해 알 수 있다

- iii.  $W[i]$  = 팀  $i$ 의 현재의 승수,  
 $G[i][j]$  = 두 팀  $i$ 와  $j$  사이에 남은 경기 수  
 $R[i]$  = 팀  $i$ 의 남은 경기 수
- iv.  $W[n]$ 은 꼴찌 팀의 승수이고 남은 게임을 모두 이긴다고 하면 최종 승수는  $W[n]+R[n]$ 이 된다. 팀  $n$ 이 우승하기 위해선, 다른 모든 팀의 최종 승수는  $W[n]+R[n]$ 보다 많으면 안된다. 즉, 팀  $i$ 의 남은 게임 중에서 최대  $W[n]+R[n]-W[i]$ 을 이기면 팀  $i$ 의 최종 승수가  $W[n]+R[n]$ 을 넘지 않게 된다. 그러면 팀  $n$ 이 우승 가능하다
- v. 아래와 같은 그래프를 정의해보자 (Jeff Erickson 강의 노트에서 인용)



- 1. 꼴찌인 타이거스 팀을 제외하고 나머지 4개 팀만으로 그래프를 구성한다
- 2. 소스 노드  $s$ 에서 각 경기를 나타내는 노드로 에지를 그리고 에지의 capacity를 해당 경기의 남은 시합 수로 정한다. 즉,  $s$ 에서 팀  $i$ 와  $j$  사이의 경기 ( $i, j$ )로 향하는 에지의 capacity는  $G[i][j]$ 로 정한다
- 3. 각 경기에 참여하는 두 팀으로 에지를 각각 연결한다. 이 에지에는 capacity를 제한하지 않는다. 예를 들어 (NYY, BAL) 사이에는 3 게임이 남았고, NYY나 BAL로 승리 횟수에 따라 flow가 흐르게 될 것이므로 굳이 용량을 제한할 필요는 없다
- 4. 각 팀  $i$ 를 나타내는 노드에서 싱크 노드  $t$ 로 에지를 연결하고,  $W[n]+R[n]-W[i]$  만큼의 용량을 배정한다
- vi. 위 그래프에서 flow가 어떻게 흘러야 꼴찌팀이 우승할 수 있는지 따져보자

- 1. 우선, 꼴찌팀을 제외한 네 팀은 남은 게임을 모두 치루어야 한다. 즉,  $s$ 에서 나가는 에지의 용량만큼 꽉 채워 flow가 흘러야 한다는 뜻이다 ( $s$ 에서 나가는 에지가 모두 **saturated**되어야 한다는 의미와 같다)
- 2. 이 flow가 중간의 두 노드 층을 통과해  $t$ 에 도달할 수도 있고 도달하지 못할 수도 있다. 도달한다는 의미는 각 팀  $i$ 에서  $t$ 로  $W[n]+R[n]-W[i]$  이내의 flow만 흐른다는 것이고, 결국 어떤 팀도 꼴찌 팀보다 더 좋은 성적을 얻을 수 없다는 뜻이다

3. 결론은  $s$ 에서  $t$ 로 가는 flow가 존재하고  $s$ 에서 나가는 모든 에지가 saturated된다면 꼴찌팀은 우승할 수 있는 가능성이 존재한다는 뜻이고, 그렇지 않다면 우승할 수 없다는 뜻이다
- vii. 위 그래프에서는  $s$ 에서 나가는 에지의 용량의 합이 27이고  $t$ 로 들어오는 에지의 용량의 합이 26이기에  $s$ 에서 나가는 27의 flow를  $t$ 가 모두 받을 수 없다. 따라서 타이거스 팀은 우승 가능성이 없다

# 10. String searching 알고리즘

## 1. 긴 문자열에서 특정 문자열(패턴)을 찾는 문제에 대한 알고리즘

- $S = \text{"Some books are to be tasted, others to be swallowed, and some few to be chewed and digested."}$
- $T = \text{"to"}$
- 문자열  $S$ 에서 패턴  $T$ 가 존재하는가? (존재 여부)
- 문자열  $S$ 에서 패턴  $T$ 가 존재한다면 어디에 총 몇 번 등장하는가? (등장 위치)

## 2. 왜 중요한 문제인가?

- 대규모 문서에서 특정 패턴을 찾는 일은 매우 다양한 분야에서 수행되는 매우 기본적이고 중요한 작업
- 예1: 문서에서 특정 패턴 찾기 [[wikipedia](#)]

### String (computer science)

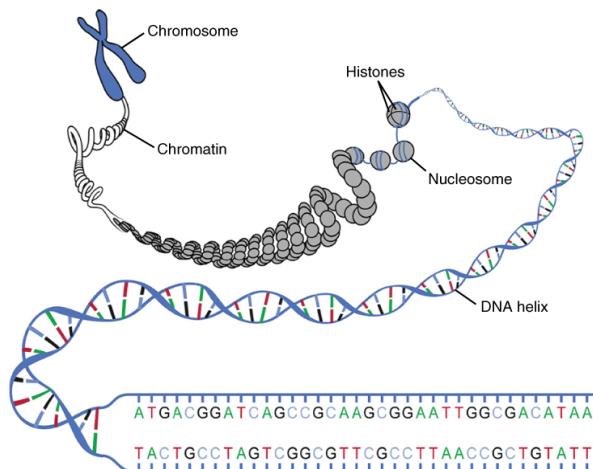
From Wikipedia, the free encyclopedia

This article is about the data type. For other uses, see [String \(disambiguation\)](#).

This article needs additional citations for verification. Please help improve this article by a citations to reliable sources. Unsourced material may be challenged and removed.  
 Find sources: "String" computer science – news · newspapers · books · scholar · JSTOR (March 2015) ([Learn when to remove this template message](#))

In computer programming, a **string** is traditionally a sequence of characters, either as a **literal constant** or as some kind of variable. The latter may allow its elements to be mutated and the length changed, or it may be fixed (after creation). A **string** is generally considered as a **data type** and is often implemented as an **array data structure** of **bytes** (or **words**) that stores a sequence of elements, typically characters, using some **character encoding**. **String** may also denote more general **arrays** or other sequence (or **list**) data types and structures.

- 예2: bioinformatics (생물정보학): DNA 구조는 A, G, C, T 네 가지 심볼의 문자열임!



[출처: wikipedia]

3. 현재 알려진 대표적인 유명 알고리즘 [Wikipedia]

a.  $|S| = n$ ,  $|T| = m$ 일 때, 탐색 알고리즘의 수행시간은  $n$ ,  $m$ 의 함수로 표현됨

| 알고리즘                                                                | 전처리 시간          | 패턴 발견 시간                                         | 메모리         |
|---------------------------------------------------------------------|-----------------|--------------------------------------------------|-------------|
| Naïve string-search algorithm                                       | none            | $\Theta(nm)$                                     | none        |
| Rabin-Karp algorithm                                                | $\Theta(m)$     | average $\Theta(n + m)$ , worst $\Theta((n-m)m)$ | $O(1)$      |
| Knuth-Morris-Pratt algorithm                                        | $\Theta(m)$     | $\Theta(n)$                                      | $\Theta(m)$ |
| Boyer-Moore string-search algorithm                                 | $\Theta(m + k)$ | best $\Omega(n/m)$ , worst $O(mn)$               | $\Theta(k)$ |
| Bitap algorithm<br><i>(shift-or, shift-and, Baeza-Yates-Gonnet)</i> | $\Theta(m + k)$ | $O(mn)$                                          |             |
| Two-way string-matching algorithm[2][3]                             | $\Theta(m)$     | $O(n+m)$                                         | $O(1)$      |
| BNDM (Backward Non-Deterministic DAWG Matching)[4][5]               | $O(m)$          | $O(n)$                                           |             |
| BOM (Backward Oracle Matching)[6]                                   | $O(m)$          | $O(mn)$                                          |             |

4. 단순 알고리즘 (naive algorithm): 단순하게 왼쪽에서 오른쪽으로 한 글자씩 이동하며 비교

a. 예:

```
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
T: ABCDABD
j: 0123456
```

```
I = []
for i in range(n-m+1): # why n-m+1 ?
 found = True
 for j in range(m):
 if S[i+j] != T[j]:
 found = False
 break
 if found == True:
 I.append(i+j)
return I
```

b. 수행시간:  $O(nm)$

5. KMP (Knuth-Morris-Pratt) 알고리즘

a. 예: [Wikipedia]

```
111111111222
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE #S[3] != T[3] (i = 3, j = 3)
T: ABCDABD
j: 0123456
두 문자열의 문자를 차례대로 매치해 나간다. D에서 매치가 처음으로 안됨
T를 S[1]부터 다시 비교하는 naive 알고리즘을 따르면 S[1] = 'B' == T[0] = 'A' 비교
그러나 이미 mismatch임을 알고 있다. 다음 'A'가 등장하는 곳부터 비교하면 된다
S의 "BC"에 T의 첫 문자 'A'가 없으므로 다음 match는 S[3] == T[0]?
```

```
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE # i = 3, j = 0, S[3]부터 비교 시작
T: ABCDABD # S[3] != T[0], i = 3, j = 0
j: 0123456
비교하자마자 mismatch이므로 다음 문자와 다시 비교
111111111222
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE # i = 4, j = 0, S[4]부터 비교 시작
T: ABCDABD # S[10] != T[6]
j: 0123456
S[8:10] = "AB" = T[0:2]이므로 다음엔 S[8]부터 시작함 (이전에는 "AB"가 없기에)
i = 8, j = 2로 setting하면, S[i+j-2] = S[8]부터 비교를 시작하면 됨
```

```
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
T: ABCDABD
j: 0123456
가장 첫 경우와 유사함
```

```
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
T: ABCDABD
j: 0123456
두 번째 경우와 유사함
```

```
i: 01234567890123456789012
S: ABC ABCDAB ABCDABC DABDE
T: ABCDABD
j: 0123456
ABCDAB 와 ABCDAB의 AB가 일치하므로, 그 위치로 점프해서 다시 비교시작
```

```
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
T: ABCDABD
j: 0123456 # 마침내!!! 물론 패턴이 없을수도 있음!
```

### 다음 세 가지 관찰이 중요하다

- b. **관찰1:** 인덱스 i는 계속 증가한다. (1 증가하거나 1보다 더 크게 증가하거나)
- c. **관찰2:** 1보다 크게 증가한다면 얼마나 증가해야 하나?

```
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
T: ABCDABD
j: 0123456
```

이 경우엔 ABCDAB까지는 일치한다. 즉, while 루프에서 i=4, j=6이 되어, S[4:4+6] == T[:6]를 의미한다. 다음 비교는 아래와 같다

```
i: 01234567890123456789012
S: ABC ABCDAB ABCDABCDABDE
T: ABCDABD
```

S[5:4+6]의 접미사(suffix)의 AB와 T[:6]의 접두사(prefix) AB가 서로 일치한다. 그러면 다음 번 match에서는 S[8:10] = "AB"의 시작 문자인 S[8]='A'에 T[0]를 맞춘 후 match를 진행하면 된다. 이 위치는 S[i+j-len("AB")] = S[4+6-2] = S[8]이 된다. 이 길이

`len("AB")`를  $S[i:j]$ 와  $T[j]$ 에 대한 다음 번 문자의 시작 위치 계산에 사용되는 값  $B[j]$ 를 정의한다

$S[i:i+j] == T[:j]$ 가 일치하는 경우,  $S[i:i+j]$ 의 suffix가  $T[:j]$ 의 prefix가 되는 길이를  $B[j]$ 로 정의하는데,  $S[i:i+j] == T[:j]$ 이므로  $S[i:i+j]$ 가 바로  $T[:j]$ 이다. 따라서, **Lps[j] = T[:j]의 suffix가 T[:j]의 prefix가 되는 가장 긴 길이로 정의**하면 된다. 단, 전체가 suffix이면서 prefix가 되는 경우는 제외한다. 이런 경우를 proper suffix 또는 proper prefix라 부른다. (**Lps**는 Longest proper Prefix which is Suffix의 약자이다. ^^) 예를 들어, AAA에 대해서는 Lps의 길이가 3이 아닌 2가 된다

- d. 관찰 3: 관찰 2의 예에서  $S[8]$ 에서 match를 시작하지만,  $S[8:10] = "AB" = T[0:2]$ 로 같다는 사실을 알기 때문에 이 부분을 비교할 필요는 없다! 즉,  $S[i:j]$ 와  $T[Lps[j]:j]$ 부터 비교를 시작하면 된다
- e. T 문자열만을 이용해 배열 Lps 값을 미리 계산할 수 있다는 사실이 중요하다!
- f. Lps 계산은 어떻게?

- i.  $Lps[j] = T[:j]$ 의 suffix가 되는  $T[:j]$ 의 proper prefix의 가장 긴 길이

```
예1: T: ABCDABD
 Lps: 0000120
예2: T: AABAAABB
 Lps: 01012230
예3: T: ABCDEF
 Lps: 000000
예4: T: AAAAAA
 Lps: 01234
예5: T: ABACABABC
 Lps: 001012320
```

- ii. 예를 통해 알고리즘을 설계해보자. Two Pointer 기법을 이용한다. 즉, T의 인덱스를 가리키는 두 개의 top, bot 포인터를 지정하고 세 가지 규칙(Rule1, 2, 3)에 따라 두 포인터를 이동하면서 Lps 값을 계산하는 방법이다. (이 설명과 코드는 [towardsdatascience.com](http://towardsdatascience.com/max-lefarov)의 Max Lefarov의 글을 참고했다.)

```
idx 012345678
top |
T: ABACABABC
bot | # Rule1: T[top]==T[bot]이 될 때까지 bot을 오른쪽 이동
Lps:000000000 # Lps는 0으로 초기화
```

```
idx 012345678
top |
T: ABACABABC
bot |
Lps:001 # 처음으로 같은 문자이므로 B값이 1 증가
```

처음으로 같은 문자를 찾았으니, 다음부터는 두 포인터를 한 칸씩 동시에 오른쪽으로 움직이며 서로 다른 문자가 나올 때까지 이동한다 (**Rule2**)

```
idx 012345678
top |
T: ABACBABCABC # Rule2: T[top] != T[bot]일 때까지 동시 이동 및 B값 증가
bot |
Lps:0010
```

Rule2를 적용한 후에는  $T[\text{top}] \neq T[\text{bot}]$ 인 상태이다. 이 경우에는 top을 왼쪽으로 이동해서  $T[\text{bot}]$ 와 일치할 수 있는 문자가 있는지 점검해야 한다. 아래 경우에는 왼쪽 인덱스가 0이므로 한 칸만 이동하면 된다. 이 규칙이 Rule3이다. Rule3는 이보다 조금 더 복잡한데 이후에 더 자세히 설명한다

```
idx 012345678
top | # 왼쪽으로 이동하면 바로 인덱스 0이 된다
T: ABACCABCABC # A != C이므로 다시 Rule1 적용
bot |
Lps:0010
```

#### Rule1 적용

```
idx 012345678
top |
T: ABACAABCABC # Rule1 적용: T[top] == T[bot]일 때까지 bot 이동
bot |
Lps:00101
```

#### Rule2 적용

```
idx 012345678
top |
T: ABACBABCABC # Rule2 적용: T[top] != T[bot]일 때까지 top, bot 이동
bot |
Lps:00101230 # T[top] == T[bot]이면 당연히 Lps 값 1씩 증가시킴
```

[중요]  $T[\text{top}] = C$ 이고,  $T[\text{bot}] = B$ 로 서로 다르다. 따라서 Lps 값은 초기 값 0 그대로다. 그 직전까지는 ABA가 일치했다. 이제는 Rule3을 적용해 top 포인터를 왼쪽으로 이동해야 한다. 정확히 어디로 이동하는 게 좋을까?

빨간색 **ABA**와 보라색 **ABA**는 일치한다. 보라색 suffix가 T의 prefix에 일치하는 문자열이 있다면, 그 다음 문자와 현재의 bot이 가리키는  $T[\text{bot}] = B$ 가 일치하는지 검사하면 불필요한 비교를 줄일 수 있다.

```

 top bot
 | |
T: ABAC...ABAB...
Lps:0010...123?
|
new top

```

보라색 A가 T의 prefix A와 일치한다. 그런데 prefix A의 다음 문자가 B이고 이는 현재  $T[\text{bot}] = \text{B}$ 와 일치한다. 이는 현재의 suffix AB가 prefix AB와 일치한다는 뜻이다. 그런데 빨간색 prefix ABA는 보라색 suffix ABA와 동일하기 때문에 A에 대한 Lps 값은  $\text{Lps}[A]$ 에 이미 저장되어 있고 이 값은  $\text{Lps}[\text{top}-1]$ 이 된다. 결국, top 포인터는  $\text{Lps}[\text{top}-1](=1)$ 로 이동(roll-back, 후진)하는 데, 이 roll-back 과정을 그 곳의 문자  $T[\text{Lps}[\text{top}-1]] = T[1] = \text{B}$ 와  $T[\text{bot}] = \text{B}$  문자가 같거나 인덱스 0에 도달할 때까지 반복하면 된다. (현재 예에서는 같다) 그 다음은 Rule2가 적용 가능한 상황 (두 문자가 같은 경우에 도달)이거나 Rule1(인덱스 0에 도달)이 적용 가능한 상황이된다

### Rule3 적용

```

idx 012345678
top | # top = Lps[top-1] 이동!
T: ABACABABC # T[top] (=B) == T[bot] (=B)이므로 stop → Rule2
bot |
Lps:00101232 # Lps[bot] ← top + 1

```

### Rule2 적용

```

idx 012345678
top | # top, bot 동시에 이동
T: ABACABABC # T[top] (=A) != T[bot] (=C)이므로 Rule3 적용
bot |
Lps:001012320

```

Rule3을 적용하면  $\text{top} = \text{Lps}[\text{top}-1] = 0$ 으로 이동되어 다음에는 Rule1을 적용해야 한다. bot을 증가시켜  $T[\text{top}] = T[0]$ 과 같은 문자가 나타날 때까지 bot을 증가시키면 되지만, 이미 bot이 마지막 인덱스이므로 끝내면 된다

```

def compute_Lps(T):
 Lps = [0] * len(T)
 top = 0 # top은 Rule3에 의해 roll-back 가능
 for bot in range(1, len(T)): # bot은 무조건 1씩 증가

 # Rule3: top 포인터를 Lps[top-1]로 roll-back 함.
 while top > 0 and T[top] != T[bot]:
 top = Lps[top - 1]

 # Rule2: T[top] == T[bot]이면 둘 다 1씩 증가
 # top은 직접 1 증가하고, bot은 for 루프에서 1 증가

```

```

if T[top] == T[bot]:
 top += 1
 Lps[bot] = top
 # Rule1은 for 루프 자체에서 수행중이므로 따로 처리 없음
return Lps

```

g. Lps 계산 알고리즘의 수행시간을 분석해보자

- i. bot은 for 루프에서 1씩 증가하여  $|T|$ 만큼 이동한다  $\rightarrow O(|T|) = O(m)$
- ii. top은 전진과 후진을 반복할 수 있다. 후진(roll-back)은 Rule3를 적용하는 경우에만 발생하는 데, Rule3를 적용하기 위해서는 먼저 Rule2를 적용해 전진을 해야 한다. Rule2에서 전진한다는 건 T의 prefix T[top]와 suffix 문자 T[bot]가 계속 일치한다는 것이고 현재 두 문자가 일치하지 않으면 Rule3의 적용을 받아 roll-back하게 된다. 아래 예에서 보면, top = 40이므로 top = Lps[top-1] = 30이 된다. T[3] = A != T[bot] = F이므로 다시 top = Lps[top-1] = 20이고 역시 A != F가 된다. 다시 top = Lps[top-1] = 10이고 역시 A != F, 다시 top = Lps[top-1] = Lps[0] = 0이 되어 while 루프를 빠져나오게 된다. 이처럼 계속 T[bot]와 일치하지 않아 while 루프를 반복하는 게 최악의 경우이다. 반복하는 횟수는 정확히 Rule2를 적용해 prefix와 suffix가 일치하는 문자열 길이 ( $|AAAA| = 4$ )이다. Rule3를 적용한 이후에는 다시 Rule2 또는 Rule1을 적용받게 되는데, 이 경우에는 top은 전진 (Rule2)하거나 움직이지 않고 (Rule1) 그대로 있게 된다. 결국, Rule3에서의 후진은 바로 직전 Rule2에서 bot이 전진한 만큼만 가능하고, bot은 계속 전진만하기 때문에 후진으로 이동한 거리를 다 모아도 bot이 전진한 거리를 넘지 않는다. bot은  $|T|$ 만큼 움직이므로 top의 후진 거리는  $|T|$ 를 넘지 않게 된다. top의 전진거리는 당연히  $|T|$ 를 넘지 않는다. 따라서 top의 전진과 후진 거리는 총  $2|T|$ 를 넘지 않는다

|                       |     |
|-----------------------|-----|
| top                   | bot |
|                       |     |
| T:    AAAAE...AAAF... |     |
| Lps: 01230...1234     |     |

- iii. 결론적으로 top과 bot은 for - while 이중 루프에서 총  $3|T|$  번 이동 (연산)하게 되어 compute\_Lps 함수의 수행시간은  $O(|T|) = O(m)$ 이 된다

h. 이제, compute\_Lps를 통해 계산된 Lps를 이용해 매칭 알고리즘을 설계해보자

- i. S의 인덱스  $i = 0$ , T의 인덱스  $j = 0$ 으로 초기화한 후, 앞에서처럼 세 가지 규칙에 따라 진행한다
- ii. Rule1: 우선  $j = 0$ 으로 지정한 후,  $T[j]$ 와 일치하는 S의 첫 문자를  $i$ 를 증가시키면서 찾는다. 이후에는 Rule2를 적용한다
- iii. Rule2: Rule1을 통해 일치한 첫 문자 쌍을 찾았다면,  $i$ 와  $j$ 를 동시에 증가시키면서 연속적으로 같은 문자가 매칭되는지 검사한다.  $S[i] \neq T[j]$ 인 경우에는 Rule3를 적용한다

## Rule1 적용

i: 01**2**3456789012  
 S: CB**A**BABACABADA  
 T: **A**BACABABC  
 j: 012345678  
 Lps: 001012320

## Rule2 적용

i: 01234**5**6789012  
 S: CB**A****B**BACABADA  
 T: **A****B**CABABC  
 j: 012**3**45678  
 Lps: 00**1**012320

- iv. Rule3: 위의 그림 참조.  $S[5] \neq T[3]$ 이므로, i는 그대로 유지하고,  $j = Lps[j-1]$ 로 후진한 후, Rule2 또는 Rule1을 다시 적용한다. 예에서는  $S[i] = T[j] = B$ 로 같기 때문에 Rule2를 이용해 계속 매칭해나간다

## Rule3 적용

i: 01234**5**6789012  
 S: CBAB**A**BACABADA  
 T: **A**BACABABC  
 j: 012345678  
 Lps: 001012320

## Rule2 적용

111  
 i: 01234567890**1**2  
 S: CBAB**A**BACABA**D**A  
 T: **A****B**CABABC  
 j: 0123456**7**8  
 Lps: 001012**3**20

## Rule3 적용

111  
 i: 01234567890**1**2  
 S: CBABABAC**A****B**ADA  
 T: **A****B****A**CABABC  
 j: 012**3**45678  
 Lps: 00**1**012320

## 다시 Rule3 적용

i: 01234567890**1**2  
 S: CBABABACAB**A**DA  
 T: **A****B****A**CABABC  
 j: 012345678  
 Lps: 00**1**012320

다시 Rule3 적용

|      |                        |
|------|------------------------|
| i:   | 01234567890 <b>1</b> 2 |
| S:   | CBABABACABADA          |
| T:   | <b>A</b> BACABABC      |
| j:   | <b>0</b> 12345678      |
| Lps: | 001012320              |

D != A가 다르고 j = 0에 도달했으므로 Rule1을 적용

|      |                       |
|------|-----------------------|
| i:   | 012345678901 <b>2</b> |
| S:   | CBABABACABADA         |
| T:   | <b>A</b> BACABABC     |
| j:   | <b>0</b> 12345678     |
| Lps: | 001012320             |

i가 마지막 인덱스에 도착했으므로 검사 완료 (원하는 매칭이 없음)

v. 이 과정을 코드로 옮기면 아래와 같다

```
def kmp(S, T):
 match_indices = []
 Lps = compute_lps(T)

 j = 0
 for i, ch in enumerate(S): # Rule1: implicitly applied in for loop

 # Rule3: roll-back j by using Lps
 while j > 0 and T[j] != ch:
 j = Lps[j - 1]

 # Rule2: keep matching; increment i and j
 if T[j] == ch:
 if j == len(T) - 1: # a match is found!
 match_indices.append(i - j) # record the match
 j = Lps[j] # keep matching
 else:
 # move j forward. i is incremented in for loop
 j += 1

 return match_indices
```

vi. 수행시간:

- 이 알고리즘도 Two Pointer 방식이고 수행시간 분석도 compute\_Lps와 본질적으로 같다. **O(n+m)** 시간이면 충분하다

## 6. Rabin-Karp 알고리즘

- a. hash 함수를 이용해 탐색을 하는 방법
  - i. hash 함수 및 hash 테이블의 자세한 내용은 자료구조 교재를 참고할 것!

```
def RabinKarp(S, T):
 1 I = []
 2 ht = hash(T) # 패턴의 해시값
 3 for i in range(n-m+1):
 4 hs = hash(S[i:i+m]) # S의 길이가 m인 substring의 해시 값 계산
 5 if hs == ht: # 두 해시 값이 우선 같아야 함!
 6 if S[i:i+m] == T: # 해시 값이 같다고 문자열이 같은 건 아님!
 7 I.append(i)
 8 return I
```

- b. 위 코드의 수행시간을 따져보자

- i. line 2: hash(T)를 계산하는 시간은 일반적으로  $O(m)$ , 오직 한 번만 계산
- ii. line 5: 두 해시 (실수) 값을 비교하는 건  $O(1)$ 이므로 전체적으로  $O(n-m)$
- iii. line 6: 두 문자열 비교는  $O(m)$  시간 필요. 그러나  $hs == ht$  인 경우에만 실행되므로 해시 값이 같은 횟수가  $k$ 라면  $O(mk)$  시간이 필요함
  - $k$  = 충돌(collision) 횟수 [hash 자료구조 참조]이므로 충돌을 줄여야 함
- iv. line 4:  $hash(S[i:i+m])$ 은 총  $O(n-m)$ 번 계산되므로  $O((n-m)m)$  시간 필요. 이 시간은  $O(nm)$ 으로 naive 알고리즘과 같다.
  - $hs$  값은 매 루프마다 갱신된다. 즉, 현재  $hs$ 는 전 루프의 해시 값을 가지고 있다. 이전 해시 값을 이용해 현재의 해시 값을  $O(1)$  시간에 만들어 낼 수 있으면, 총 시간이  $O(m+n-m) = O(n)$ 이 된다. 이런 식의 hash 함수를 rolling hash 함수라 부름

- c. Rolling hash 함수 예

base = 256, p = 101 (prime), S = "abrcbaabr" ( $|T|=3$ )  
 ASCII code: a = 97, b = 98, r = 114, c = 99

$$\begin{aligned} \text{hash("abr")} &= ('a' * 256^2 + 'b' * 256^1 + 'r' * 256^0) \% 101 \\ &= (97 * 256^2 + 98 * 256^1 + 114 * 256^0) \% 101 = 4 \end{aligned}$$

$$\text{hash("brc")} = ((\text{hash("abr")} - 'a' * 256^2) * 256 + 'c') \% 101$$

- i.  $\text{hash("abr")} \rightarrow$  상수시간  $\rightarrow \text{hash("brc")}$  계산 가능 (자세한 방법은 생략)
- ii.  $m$ 보다 충분히 큰 랜덤 소수  $p$ 를 정하면, 충돌하는 문자열의 평균 개수 =  $k = O(1)$ 로 작게 유지할 수 있다!
  - 최악의 경우:  $S = aa\dots a = a^n$ ,  $T = aa\dots a = a^m$
- iii. 결국, line 2:  $O(m)$ , line 5:  $O(n - m)$ , line 6: 평균  $O(m)$ , line 4:  $O(n)$   
 총 평균 시간 =  $O(n + m)$

[주의] 해시 함수의 성능이 평균적으로  $O(n + m)$ 을 보장한다는 의미이고, 최악의 경우에는  $k = O(n - m)$ 이 될 수 있어  $O(nm)$ 까지 커질 수 있다

- iv. 만약, 찾아야 할 패턴이 여러 개라면?

```
def RabinKarp(S, T): # T=list of pattern strings, |T|=k
 1 I = []
 2 hash_list = []
 3 for i in range(len(T)): # O(km)
 4 hash_list.append(hash(T[i]))
 5 for i in range(n-m+1): # (n-m) loops
 6 hs = hash(S[i:i+m])
 7 if hs in hash_list and S[i:i+m] in T: # O(k),O(1)?
 8 I.append(i)
 9 return I
```

- line 7에서 hs **in** hash\_list 과 S[i:i+m] **in** T를 빠르게 -  $O(1)$  시간으로 구현하는 방법은? [hint: hash table로 hash\_list와 T를 관리?]
- $O(1)$  시간에 line 7이 수행되면 for 루프는  $O(n-m)$  시간이 필요하고, line 3의 for 루프는  $O(km)$ 이므로 전체 시간은  $O(n+mk)$ 가 되어 하나의 패턴을 찾는  $O(n+m)$  시간 알고리즘을 k번 수행하는데 필요한  $O(nk+mk)$ 보다 적게 걸린다

## 11: 기하 알고리즘 (Geometric Algorithm)

1. 점, 선, 면, 삼각형, 원, 다각형 등의 2차원 또는 3차원 이상의 기하 객체(geometric object)와 관련된 문제를 해결하는 알고리즘을 기하 알고리즘이라 부른다

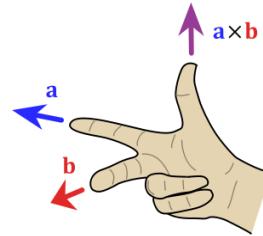
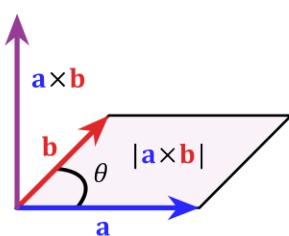
### 2. 삼각형 (triangle): 면적과 회전 (orientation)

a. 두 변을 나타내는 벡터가  $\mathbf{a}$ ,  $\mathbf{b}$ 이고 두 변 사이의 각이  $\theta$ 라면, 두 벡터가 이루는 평행사변형의 면적은 어떻게 되는가?

$$\text{Area} = |\mathbf{a}| |\mathbf{b}| |\sin \theta|$$

b. 두 벡터의 외적(cross product)은 다음과 같다. 이제 평행사변형의 (부호가 있는) 면적을 외적으로 표현해보자

- i. 이미지 출처: [https://en.wikipedia.org/wiki/Cross\\_product](https://en.wikipedia.org/wiki/Cross_product)
- ii. 오른손 법칙으로 외적 벡터 표현



c. 외적의 기본 성질

- i.  $\mathbf{a} \times \mathbf{b} = -\mathbf{b} \times \mathbf{a}$  (anticommutative)
- ii.  $\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = \mathbf{a} \times \mathbf{b} + \mathbf{a} \times \mathbf{c}$  (벡터 +에 대해 배분법칙 성립)

d. 2차원에 세 점  $\mathbf{p} = (p_x, p_y)$ ,  $\mathbf{q} = (q_x, q_y)$ ,  $\mathbf{r} = (r_x, r_y)$ 이 이루는 삼각형의 면적을 계산해보자. 외적은 3차원에서 정의되므로 각 점의 z-좌표를 0으로 정한다. 두 벡터  $\mathbf{a}$ ,  $\mathbf{b}$ 를 정의해보자

- i.  $\mathbf{a} = \text{벡터 } \mathbf{p} \rightarrow \mathbf{q} = \mathbf{q} - \mathbf{p} = (q_x - p_x, q_y - p_y, 0)$
- ii.  $\mathbf{b} = \text{벡터 } \mathbf{p} \rightarrow \mathbf{r} = (r_x - p_x, r_y - p_y, 0)$

e.  $\mathbf{i} = (1, 0, 0)$ ,  $\mathbf{j} = (0, 1, 0)$ ,  $\mathbf{k} = (0, 0, 1)$  단위벡터라면,

$$\begin{aligned} \mathbf{a} \times \mathbf{b} &= (a_x \mathbf{i} + a_y \mathbf{j} + a_z \mathbf{k}) \times (b_x \mathbf{i} + b_y \mathbf{j} + b_z \mathbf{k}) \rightarrow \text{외적 배분법칙 적용} \\ &= (a_y b_z - a_z b_y) \mathbf{i} + (a_z b_x - a_x b_z) \mathbf{j} + (a_x b_y - a_y b_x) \mathbf{k} \rightarrow a_z = b_z = 0 \text{이므로} \\ &= (a_x b_y - a_y b_x) \mathbf{k} = (0, 0, (q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)) \end{aligned}$$

f. [중요] 결국, 세 점  $\mathbf{p}$ ,  $\mathbf{q}$ ,  $\mathbf{r}$ 이 이루는 삼각형의 (부호가 있는) 면적

$$\begin{aligned} \text{Area}(\mathbf{p}, \mathbf{q}, \mathbf{r}) &= |(0, 0, (q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x))| \\ &= |(q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)| \end{aligned}$$

- g. 그럼  $(q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)$ 의 부호는 어떤 의미일까?
- $+$ :  $p$ 에서  $q$ 로 향하는 방향의 왼쪽에  $r$ 이 존재한다는 의미
  - $-$ :  $p$ 에서  $q$ 로 향하는 방향의 오른쪽에  $r$ 이 존재한다는 의미
- h. [중요] 결국, 부호가  $+$ 이면  $p$ 에서  $q$ 로 향하는 방향을 기준으로  $r$ 은 left turn을 의미하고-이면 right turn을 의미한다

```
def turn(p, q, r): # p, q, r 모두 2차원 좌표를 나타내는 tuple 형식이라 가정
 signed_area =
 (q[0]-p[0])(r[1]-p[1])-(q[1]-p[1])(r[0]-p[0])
 if signed_area == 0: return 0 # 세 점이 일직선 위에
 elif signed_area > 0: return 1 # left turn
 else return 2 # right turn
```

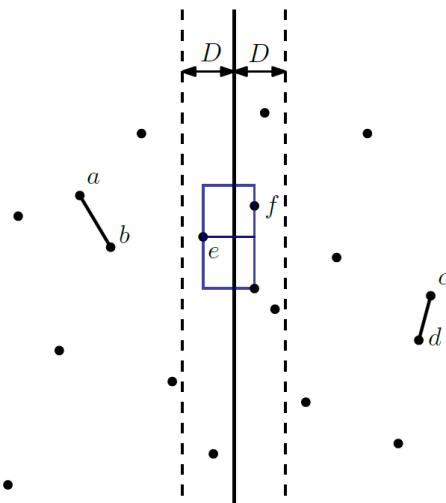
i. 응용: 두 선분의 교차 여부 및 교차점 계산하기

- 두 선분  $ab$ 와  $cd$ 가 만나기 위해선 어떤 조건이 만족되어야 하나?
  - 끝 점  $c$ 와  $d$ 는 선분  $ab$ 에 대해 서로 반대편 (하나는 left turn, 다른 하나는 right turn)에 위치해야 한다 (같은 편에 있다면 절대 만나지 않음!)  $\Rightarrow$  turn 함수를 호출하여 쉽게 검사 가능!
  - 서로 반대편에 있더라도 만나지 않을 수 있다. (어떤 상황?) 만나기 위해선 선분  $cd$ 에 대해서 끝 점  $a$ 와  $b$  역시 서로 반대편에 있는지 다시 검사하면 된다
- 교차 한다면, 실제 교차점은 두 선분의 식으로부터 계산할 수 있다

### 3. 가장 가까운 두 점 찾기 (Closest Point Pair Problem)

- 2차원 평면에  $n$ 개의 점이 주어지면, 가장 가까운 두 점 (최근접쌍)을 계산하는 기하 문제로 대표적인 분할정복 알고리즘
- 단순 알고리즘:  $O(n^2)$  시간
  - 모든 쌍에 대해, 거리를 계산하여 그 중 최소 거리가 되는 쌍을 출력!
- 분할정복 알고리즘
  - $x$ -좌표를 기준으로 오름차순으로 정렬한 후, 왼쪽  $n/2$ 개 점과 오른쪽  $n/2$ 개 점으로 분할한 후, 각 그룹에서 최근접 쌍을 재귀적으로 찾는 분할정복 알고리즘
  - 왼쪽 그룹의 최근접쌍을  $(a,b)$ 라 하고, 오른쪽 그룹의 최근접쌍을  $(c,d)$ 라 하자.  $d(a,b)$ 와  $d(c,d)$ 는 두 점 사이의 거리이다. 전체 점 집합에 대한 최근접쌍은  $(a,b)$ ,  $(c,d)$ , 또는 왼쪽과 오른쪽 그룹에 속한 두 점에 대한 최근접쌍, 이 세 쌍 중 하나이다
  - $D = \min(d(a,b), d(c,d))$ 라 하고, 세 번째 후보 쌍을  $(e,f)$ 라 하자. 어떻게  $(e,f)$ 를 계산할 수 있을까?

- (e, f)이 최근접쌍이 되기 위해선  $d(e, f) < D$ 이어야 한다. 그러므로 D 보다 작은 거리에 있는 왼쪽 그룹의 점 e와 오른쪽 그룹의 점 f를 찾으면 된다. (그런 쌍이 없을 수도 있음에 유의)



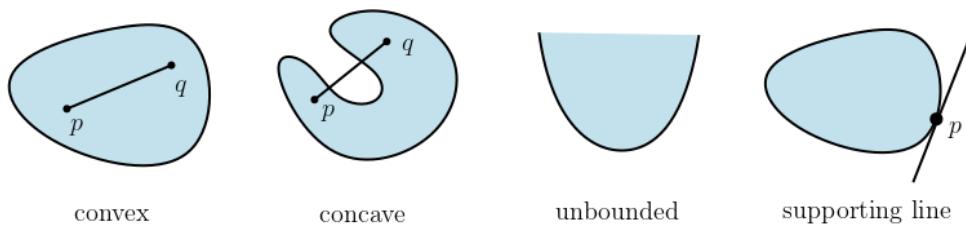
- 위의 그림에서, 가운데 분할 선의 좌-우로 D만큼 떨어진 두 수직선 사이의 띠 (strip) 영역안에 e와 f가 존재해야 한다. 이 띠 외부의 점은 절대 e와 f가 될 수 없다 (왜?)
- 이 영역 중에서 왼쪽 띠 영역에 있는 점이 e의 후보가 된다. f는 오른쪽 띠 영역의 점인데, 당연히 e에서 D 이내 거리에 있는 점이어야만 한다
- 그런 f가 존재한다면, 위 그림처럼 e의 위, 아래로 한변의 길이가 D인 정사각형 영역 안에 있어야 한다. (왜?)
- 결국, 점 e의 입장에선 이 두 정사각형 영역 내의 점들만 보면 된다
- 이 두 정사각형 영역에 상당히 많은 점들이 조밀하게 존재할 수 있을까? 이 점들의 최대 몇 개까지 가능할까? 5개
- 왼쪽 띠 영역의 각 점 e를 y-좌표 값이 작은 것부터 차례로 보면서 e의 y-좌표 값의 위와 아래로 D이내의 (오른쪽 띠 영역의 점) f가 존재하는지 검사하고, 있다면 e와 f의 거리를 계산해서 최소 거리를 기록한다
- 양쪽 띠 영역의 점들이 각각 y-좌표 값을 기준으로 정렬되어 있다면 정렬 순서를 따라가면서  $O(n)$  시간에 이 작업을 수행할 수 있다.
- 이 정렬은 merge 정렬을 이용하면 된다. 즉, 최근접쌍을 위해 재귀적으로 분할하는 단계에 merge 정렬 단계를 추가하면 된다. 결국, 양쪽 띠의 점들은 y-좌표 값에 따라 정렬이 된 상태가 되고 다음 재귀 단계를 위해, 양쪽 띠의 점들을  $O(n)$  시간에 "merge"하면 된다

#### iv. 수행시간

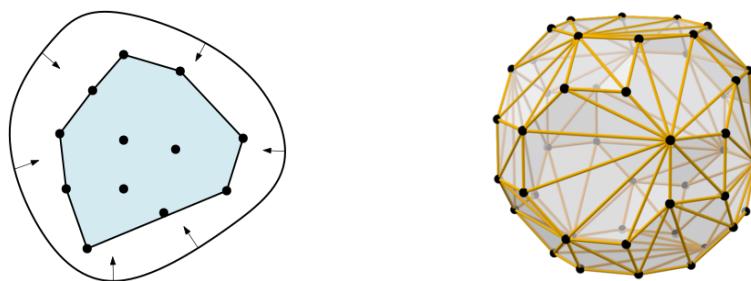
- $T(n) = 2T(n/2) + (\text{e와 f를 찾는 시간} + \text{양쪽 띠 영역의 점들 y-좌표값으로 정렬시간}) = 2T(n/2) + cn = O(n\log n)$

#### 4. 볼록 헬 (Convex Hull)

- a. 일부 내용은 Wikipedia와 메릴랜드 대학의 David Mount 교수의 [강의노트](#)의 내용을 기초로 작성했으며, 이미지 일부도 강의노트에서 가져옴 (<https://www.cs.umd.edu/class/fall2016/cmsc754/Lects/cmsc754-fall16-lects.pdf>)
- b. 볼록 집합 (convex set) : 집합의 임의의 두 점을 연결한 선분 역시 집합에 속하는 경우
- c. 오목 집합 (concave set) : 볼록하지 않은 집합
- d. 접선과 접점



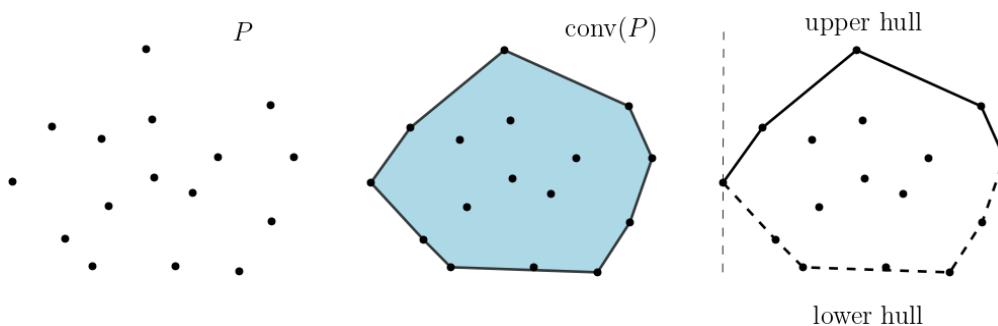
- e. 점 집합  $P$ 의 볼록 헬은  $P$ 를 포함하는 **가장 작은 볼록 집합**으로 정의 (예: 2D, 3D)



**왼쪽 그림:** 점을 평면에 박힌 못이라 가정하고, 모든 점을 포함하는 바깥쪽에 고무줄을 펴친 후 놓으면 안으로 수축하면서 그림처럼 바깥쪽 못에 걸려 더 이상 움직이지 않는 상태가 된다. 이때의 고무줄(파란색)은 최소 크기의 볼록 집합 (여기선 볼록 다각형)이 되고, 이를 점들에 대한 **볼록 헬**이라 부른다

**오른쪽 그림:** 3차원에 주어진 점 집합에 대한 3차원 볼록 헬 (3차원의 볼록 다면체로 정의됨)

- f. 점 집합  $P$ , 볼록 헬  $\text{conv}(P)$ , 위쪽 헬 (upper hull), 아래쪽 헬 (lower hull)



g. **볼록 헐 알고리즘:** 다양한 알고리즘 존재

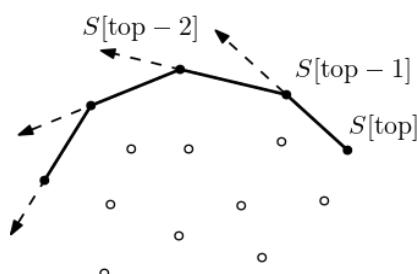
- i. 이차원에서는  $\text{conv}(P)$ 는 볼록 다각형이므로  $\text{conv}(P)$ 의 에지를 알아내면 된다
- ii.  $\text{conv}(P)$ 의 upper hull을 계산할 수 있으면 대칭적인 방법으로 lower hull을 계산할 수 있다. 두 hull을 단순 연결하면  $\text{conv}(P)$ 를 얻을 수 있다
- iii. Graham scan 알고리즘과 분할정복 알고리즘 두 가지를 각각 설명한다

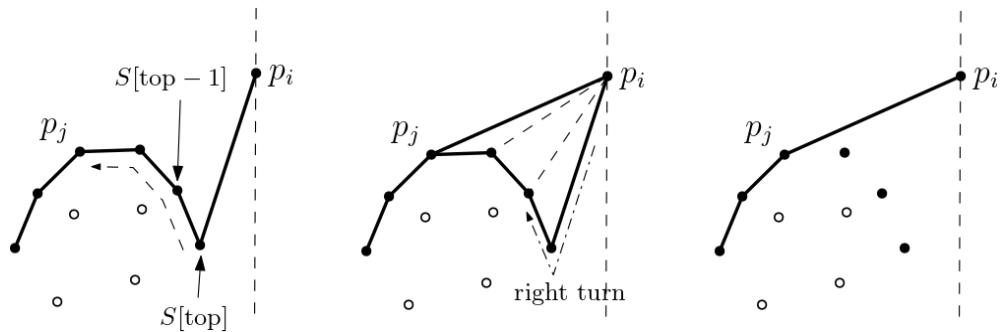
h. **단순 알고리즘:**

- i.  $\text{conv}(P)$ 의 에지는 두 점을 지나는 선분이다.
- ii. 두 점을 지나는 선분 (또는 직선)은 모두 몇 가지인가?
- iii. 이 선분 중에서  $\text{conv}(P)$ 의 에지가 될 수 있는 선분은?
- iv.  $\text{conv}(P)$ 의 에지를 지나는 직선을 생각해보자. 다른  $n-2$ 개의 점은 모두 이 직선의 한쪽 편에 존재한다! 결국,  $n-2$ 개의 점 모두가 한쪽 편에 존재하는지 검사하면 된다  
→ turn 함수를 호출!
- v. **수행시간:**
  1. 선분의 총 개수  $\times$  검사 시간 =  $O(n^2) \times O(n) = O(n^3)$

i. **Graham scan 알고리즘:**

- i. 점을 먼저  $x$ -좌표 값의 오름차순 순서로 정렬한다:  $P = p_0, p_1, \dots, p_n$
- ii.  $p_0$ 는  $\text{conv}(P)$ 의 가장 왼쪽 점으로 등장한다 (왜?)
- iii.  $P_i = \{p_0, p_1, \dots, p_i\}$ 라 하면,  $\text{conv}(P) = \text{conv}(P_n)$ 이 된다
- iv.  $P_1 = \{p_0, p_1\}$ 이라면,  $\text{conv}(P_1) = p_0p_1$ 이 되고  $p_2$ 부터 차례로 하나씩 이미 만들어 놓은 볼록 헐에 추가하여 최종적으로  $\text{conv}(P_n)$ 을 얻는 점진적인 알고리즘 (incremental algorithm)이다
- v. 핵심은  $\text{conv}(P_{i-1}) + p_i \rightarrow \text{conv}(P_i)$ 을 빠르게 계산하는 것이다
  1. 여기서  $\text{conv}(P_i)$ 는  $P_i$ 에 대한 upper hull을 의미한다 (lower hull은 같은 과정을 한 번 더 거치면 된다)
  2. upper hull에 속하는 점들은 스택  $S$ 에 차례대로 저장한다
  3. 이제  $i$ 번째 점인  $p_i$ 를 보면서 이 점이 upper hull에 속할지 아닐지를 판별해, 속한다면  $S$ 에 push하고 아니면 무시한다





- vi. 위의 그림의 가장 왼쪽처럼  $p_i$ 가 upper hull의 마지막 에지의 right turn이라면,  $p_i$ 는 upper hull에 포함되어어야 한다. 대신 기존의 upper hull에 포함된 몇 개의 점이 제외될 수 있다. upper hull의 점들은 가장 최근 점이 스택 S의 top에 위치하므로 하나씩 보면서 제외되어야하는지 점검하면 된다
- vii.  $p_i$ 에서 점  $S[\text{top}]$  방향에 대해,  $S[\text{top}-1]$ 이 오른편에 있다면, 점  $S[\text{top}]$ 은  $p_i$ 에 의해 upper hull에서 제외되어야 한다. S에서 pop한 후, 같은 검사를 계속하면 된다. 위의 가운데, 오른쪽 그림을 참고하자

viii. **Pseudo code:**

```

sort P # P = {p0, p1, ..., pn} - x-좌표값의 오름차순으로
S = {} # S = a stack storing upper hull points from left to right
S.push(p0)
S.push(p1)
for i in range(2, n):
 while |S| >= 2 and
 if turn(pi, S[top], S[top-1]) == right_turn:
 S.pop()
 S.push(pi)

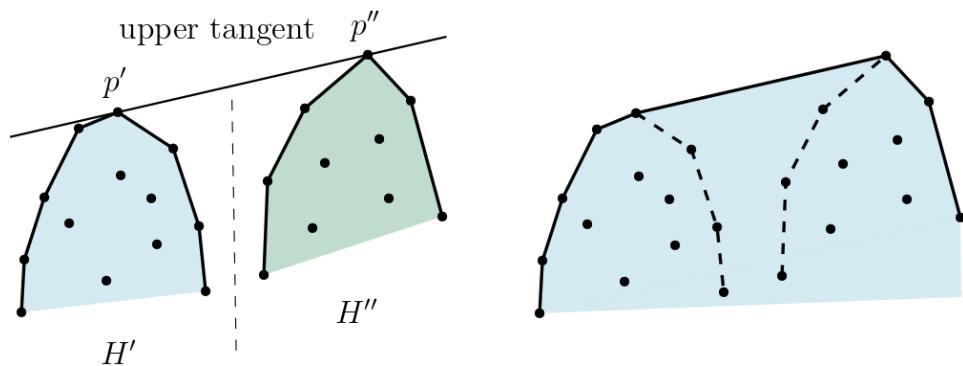
```

ix. **수행시간:**

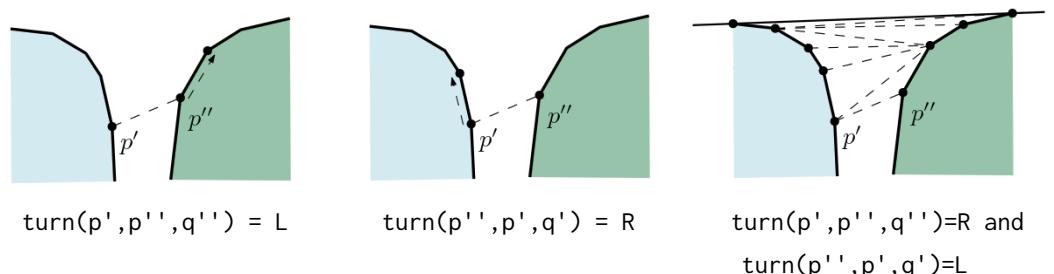
1. x-좌표 값의 오름차순으로 정렬:  $O(n \log n)$
2. 모든 점은 정확히 한번씩 push:  $O(n)$
3. upper hull에 속하지 않는 점들만 정확히 한번씩 pop 된 후 다시 push 되지 않음:  $\leq n-2 = O(n)$
4. 총 수행시간 =  $O(n \log n) \Rightarrow$  최적! (증명은 13.4절에 나와있다)

### j. Divide and conquer algorithm:

- i. x-좌표 값을 기준으로  $n/2$ 개의 왼쪽 점들과  $n/2$ 개의 오른쪽 점들로 분할한 후, 재귀적으로 upper hull을 계산한 후에 두 upper hull  $H'$ 과  $H''$ 을 merge해 하나의 upper hull  $H$ 을 구성하는 전형적인 분할정복 알고리즘이다



- ii.  $H'$ 과  $H''$ 을 merge하기 위해선 두 upper hull의 위쪽에서 support하는 직선을 찾아야 한다. 이 직선을 upper tangent line이라 부른다. 이 직선을 어떻게 찾을 것인가?
1. 두 upper hull을 위쪽에서 동시에 접하는 직선은 두 hull의 점들이 모두 직선의 아래에 위치해야 한다. 또한 이 직선은  $H'$ 의 한 점과  $H''$ 의 다른 한 점을 반드시 통과한다
  2.  $H'$ 의 오른쪽 끝 점  $p'$ 과  $H''$ 의 왼쪽 끝 점  $p''$ 부터 시작해서 이 두 점을 지나는 직선이 접선인지 검사하고 접선이 아니라면  $p'$  또는  $p''$ 를 다음 (왼쪽 또는 오른쪽) 점으로 이동하고 다시 두 점을 지나는 직선이 접선인지 검사하는 방식을 반복한다
  3. 쉽게 설명하면, 아래 그림처럼, 사다리를 오르듯이 지그재그 방식으로 점을 이동하는 식이다. 왼쪽 그림에서는  $q''$ 은  $p''$ 의  $H''$ 에서의 바로 오른쪽 점이다.  $\text{turn}(p', p'', q'')$ 이 왼쪽 턴이라면 직선  $p'p''$ 의 윗쪽에  $q''$ 이 존재한다는 의미이고 그렇다면 이 직선은 접선이 아니라는 의미다. 그리고  $p''$ 은 더 이상 고려할 필요없고 대신  $q''$ 을  $p''$ 으로 하여 고려하면 된다. 만약, 가운데 그림처럼  $\text{turn}(p'', p', q')$ 이 오른쪽 턴이라면 유사한 이유로  $p'$ 은 접선과 관계가 없으며  $q'$ 을  $p'$ 으로 해서 다시 고려하면 된다. 결국, 이 같은 방식으로 사다리 올라가듯 지그재그로  $p'$ 과  $p''$ 을 옮기면서 접선을 찾을 수 있다



### iii. 수행시간:

1.  $T(n) = 2T(n/2) + \text{merge time}$
2. merge time = upper tangent를 계산하는 시간 = 사다리를 오르는 과정에 걸리는 시간 = 사다리를 오르다 다시 후퇴해서 내려오지 않기 때문에 두 upper hull에 포함된 점의 개수에 비례하는 시간이면 충분 =  $O(n)$
- 3.  $T(n) = 2T(n/2) + cn = O(n\log n)$**

### iv. Python에서의 볼록 헐 계산

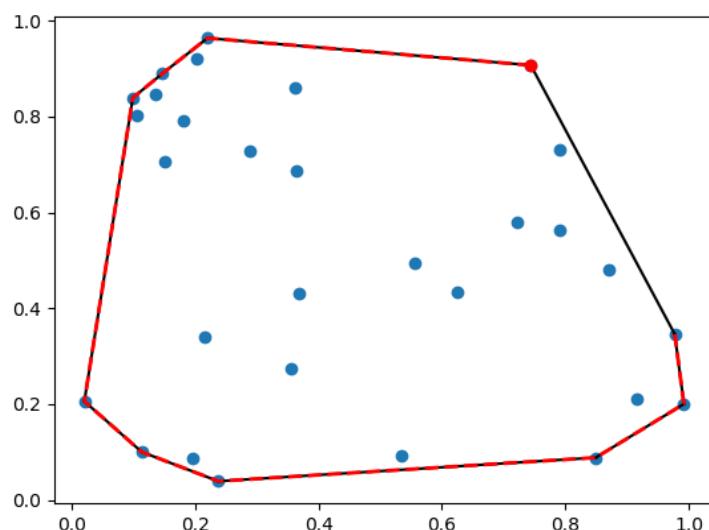
1. 파이썬에서 볼록 헐을 계산할 수 있는 모듈을 제공하는데, Qhull 볼록 헐 알고리즘을 사용한다 (<http://www.qhull.org/>)

```
from scipy.spatial import ConvexHull, convex_hull_plot_2d
import numpy as np
import matplotlib.pyplot as plt

points = np.random.rand(30, 2) # 30 random points in 2-D
hull = ConvexHull(points)

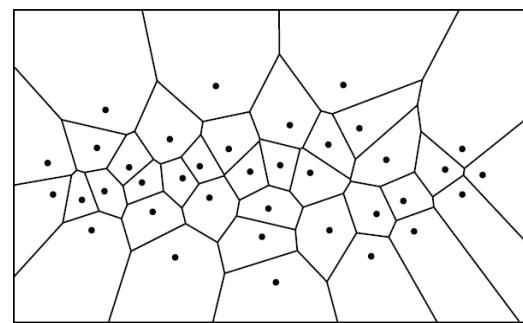
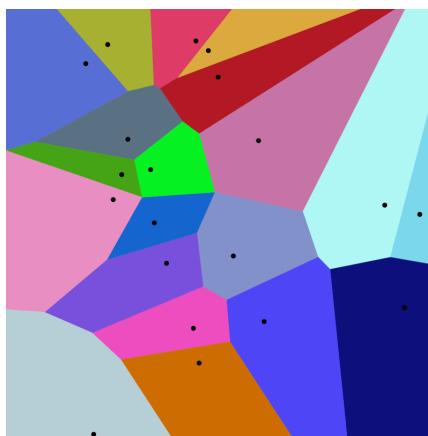
plt.plot(points[:, 0], points[:, 1], 'o')
for simplex in hull.simplices:
 plt.plot(points[simplex, 0], points[simplex, 1], 'k-')

plt.plot(points[hull.vertices, 0], points[hull.vertices, 1], 'r--', lw=2)
plt.plot(points[hull.vertices[0], 0], points[hull.vertices[0], 1], 'ro')
plt.show()
```

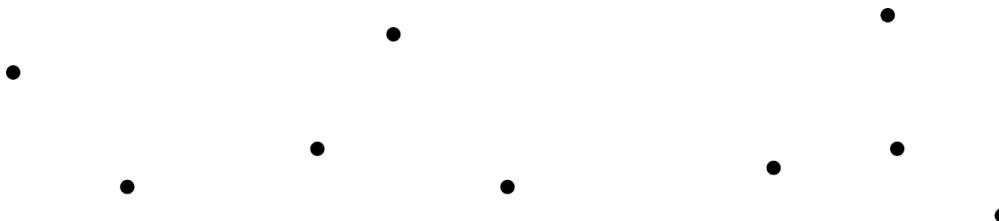


## 5. Voronoi diagram (보로노이 다이어그램)

- a. 도시에 소방서가 여러 개 있다고 하자. 어떤 집에서 화재가 발생하면 이 집 가장 가까운 소방서에서 출동해야 한다. 아래 그림의 점들이 소방서라고 하면, 그 점을 포함하는 다각형 영역이 그 소방서가 다른 소방서보다 더 가까운 점들의 영역이다
- b. 예를 들어, 가운데 연두색 영역에 있는 집은 해당 영역의 소방서가 다른 소방서보다 가깝기 때문에 그 소방서에서 출동해야 한다



- c. 소방서가 두 군데 뿐이라면, 각 소방서가 출동할 영역은 어떻게 결정되는가? 또는 세 곳, 네 곳이라면? (아래 그림은 세 가지 경우 - 2점, 3점, 4점)



- d. n개의 사이트(소방서, 경찰서, 병원 등)가 입력으로 주어지고, 각 사이트까지의 거리가 다른 사이트까지의 거리보다 작은 점들의 영역을 보로노이(Voronoi) 영역이라 정의한다. 이

영역들은 이차원 평면을 분할 하는 데, 이 분할을 특별히 **Voronoi diagram / partition / decomposition**이라 부른다

- i. Georgy Fedosievych Voronoy의 이름에서 유래
- ii. Voronoi diagram은 일종의 그래프의 형태이고, 정점, 에지, 면으로 구성되어 있다. 면(face)은 보로노이 영역이고, 에지는 보로노이 에지, 정점은 보로노이 정점이라 부른다. (질문: 보로노이 에지에 있는 점에 가장 가까운 사이트는? 보로노이 정점에 있는 점에 가장 가까운 사이트는?)
- iii. 매우 다양한 응용분야 - [[Wikipedia](#) 참조]

e. 여러 조건을 고려 가능:

- i. 두 점 사이의 거리 기준  $L_p$  ( $p = 1, 2, \infty$ )
  - 1.  $L_p = \sqrt[p]{\sum_{i=1}^n (x_i - y_i)^p}$
  - 2.  $L_2$ 가 실생활에서 사용하는 Euclidean 거리
  - 3.  $L_1$ 은 Manhattan 거리
- ii. 사이트 모양: 점, 선분, 원, 볼록 다각형 등
- iii. High-order Voronoi diagram
  - 1. First-order Voronoi diagram = 일반 보로노이 다이어그램
  - 2. Second-order Voronoi diagram = 가장 가까운 점과 두 번째로 가까운 점에 대한 보로노이 영역으로 공간을 나누는 다이어그램 (예: 점이 a, b, c에 대해서는 (a, b)가 가장 가까운 두 점이 되는 보로노이 영역, (b, c), (a, c)가 각각 가장 가까운 두 점이 되는 보로노이 영역으로 나누어 만들어지는 다이어그램)
  - 3. k-order Voronoi diagram = 가장 가까운 k개의 점에 대해 보로노이 영역으로 공간을 나누는 다이어그램
- iv. High dimensional Voronoi diagram: 2차원, 3차원, ..., d차원으로 확장

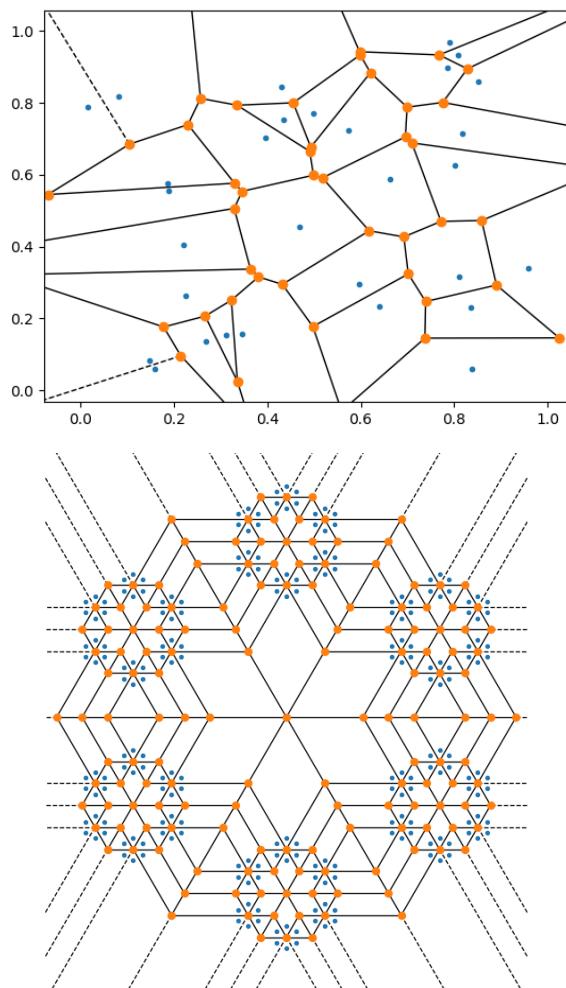
f. 대표적인 알고리즘:

- i. **1차원**: x-축 위에 점들이 주어진다고 가정하는 경우
  - 1. 쉽다! 여러분이 생각하는 알고리즘의 수행 시간은?
- ii. **2차원**: 알고리즘 자체가 꽤 복잡하고 정확히 구현하기가 쉽지 않음
  - 1. **Simple, slow algorithm**: 어떤 사이트 p의 보로노이 영역  $Vor(p)$ 을 반복해서 구하면 된다. 어떻게? (위의 점 4개인 경우 가운데 점을 예를 들어 설명)  $\Rightarrow O(n^2 \log n)$  시간 필요
  - 2. **Fortune's algorithm**: youtube 설명 동영상  
[\[https://www.youtube.com/watch?v=k2P9yWSMaXE\]](https://www.youtube.com/watch?v=k2P9yWSMaXE)
    - o  $O(n \log n)$  시간 알고리즘
    - o Javascript (직접 클릭해서 점 추가 가능)  
[\[http://www.raymondhill.net/voronoi/rhill-voronoi.html\]](http://www.raymondhill.net/voronoi/rhill-voronoi.html)
  - 3. **Qhull algorithm**: 알고리즘이 간단하고 구현 용이 (Python scipy.spatial 모듈에서 사용하는 알고리즘)

### iii. Python module: `scipy.spatial` 패키지의 Voronoi 모듈

```
from scipy.spatial import Voronoi, voronoi_plot_2d
import numpy as np
import matplotlib.pyplot as plt

points = np.random.rand(30, 2) # 30 points in 2d
vor = Voronoi(points)
fig = voronoi_plot_2d(vor)
plt.show()
```



(이미지: <https://docs.scipy.org/doc/scipy/reference/tutorial/spatial.html>)

## 12. FFT를 이용한 다항식 곱셈 알고리즘

### 1. FFT: Fast Fourier Transformation

2. 다항식:  $p(x) = a_0 + a_1x^1 + a_2x^2 + \dots + a_nx^n$ 
  - a.  $a_i$ : 계수, coefficient,  $a^n \neq 0$ 이라고 가정
  - b. 최고차 항이 n차이므로 n차 다항식
3. 두 다항식  $p(x)*q(x)$ 를 계산하려면 단순하게 전개하면 된다. 각 계수가 두 리스트 P, Q에 저장되어 있다면 아래와 같이  $O(n^2)$  시간에 가능

```
R = [0]*(2n+1)
for i in range(n+1):
 for j in range(n+1):
 R[i+j] += P[i]*Q[j]
```

#### a. 이보다 더 빠르게 할 수 없을까?

### 4. 다항식 표현법 세 가지

#### a. 계수 지정: 계수 $(n+1)$ 개를 정하면 됨: $p(x) = a_0 + a_1x^1 + a_2x^2 + \dots + a_nx^n$

예: 계수를 나타내는 리스트가  $A = [1, 3, -6]$ 이라면  $p(x) = 1+3x-6x^2$

#### b. 값(sample) 지정: n차 다항식에는 서로 다른 $(n+1)$ 개의 값이 주어지면 다항식을 유일하게 결정할 수 있기에, 좌표 값 $(n+1)$ 개를 정하면 됨: $(x_0, y_0), (x_1, y_1), \dots, (x_n, y_n)$

예:  $(1, -2), (0, 1), (-1, -8)$ 의 값을 주면  $p(1) = -2, p(0) = 1, p(-1) = -8$ 이 되어 3개의 선형 방정식이 나오고, 세 개의 계수  $a_0=1, a_1=3, a_2=-6$ 을 결정할 수 있음

#### c. 근(root)과 스케일 값 지정: n개의 근 $r_1, r_2, \dots, r_n$ (실근, 허근, 중근 모두 포함)과 1개의 스케일 값 s가 주어지면 계수가 결정됨: $p(x) = s(x-r_1)(x-r_2)\dots(x-r_n)$

예: 근이 2, 6이고  $s = 3$ 이면,  $p(x) = 3(x-2)(x-6) = 7x^2 - 24x + 36$

### 5. 다항식 중요 연산 세 가지: 함수 값 계산, 두 함수 덧셈, 두 함수 곱셈

#### a. 함수 값 계산: $p(x)$ 계산

i.  $x$  값이 입력으로 주어지면 함수 값  $p(x)$ 를 계산해보자

ii. 계수 지정:  $p(x) = a_0 + x(a_1 + x(a_2 + \dots + x(a_{n-1} + x a_n) \dots))$ 으로 표현할 수 있기에  $O(n)$ 번의 덧셈과 곱셈으로 계산이 가능하다. 따라서  $O(n)$  시간 (**Horner's Rule**)

iii. 값 지정:  $(n+1)$ 개의 값을 대입하면  $(n+1)$ 개의 선형식이 나오고, 가우스 소거법으로  $O(n^2)$  시간에 연립방정식을 풀어 계수 지정 표현법으로  $p(x)$ 를 표현할 수 있음. 이 식에  $x$  값을 대입해  $O(n)$  시간에 계산 가능. 따라서  $O(n^2)$  시간 필요

iv. 근 지정:  $p(x) = s(x-r_1)(x-r_2)\dots(x-r_n)$ 으로 표현되므로  $O(n)$  시간에 계산

b. 덧셈:  $p(x) + q(x)$

- i. 덧셈한 결과 다항식도 해당 표현법으로 나타내야 함에 유의
- ii. 계수 지정: 두 다항식의 차수가 같은 항의 계수끼리 더하면 되므로  $O(n)$  시간
- iii. 값 지정: 각 다항식에 대한 값이  $(n+1)$ 개씩 주어지므로 각  $x_i$  값이 같다면  $(n+1)$ 개의  $(x_i, p(x_i)+q(x_i))$  값을 계산할 수 있음:  $O(n)$  시간
- iv. 근 지정:  $p(x)+q(x)$ 의 근  $n$ 개를 알아야 한다. 그러나 5차 이상의 다항식의 경우엔 근을 정확히 구할 수 없기 때문에 효율적인 방법이 존재하지 않음!

c. 곱셈:  $p(x) * q(x)$

- i. 계수 지정: 두 다항식의 곱셈은 배분법칙에 따라 전개하면 되므로  $O(n^2)$
- ii. 값 지정: 다항식을 곱하면 결과 다항식의 차수는  $2n$ 이 되므로  $2n$ 개의 값을 마련해야 됨. 이 조건을 만족하려면, 두 다항식 모두  $2n$ 개의 값을 가지고 시작한다고 가정한다. 그러면  $O(n)$  시간에 가능
- iii. 근 지정: 곱은 덧셈과 달리 두 다항식을 단순히 곱하면 되므로  $O(n)$  시간

d. 정리하면 이 표와 같다

| 표현법   | 함수 값 계산  | 덧셈         | 곱셈       |
|-------|----------|------------|----------|
| 계수 지정 | $O(n)$   | $O(n)$     | $O(n^2)$ |
| 값 지정  | $O(n^2)$ | $O(n)$     | $O(n)$   |
| 근 지정  | $O(n)$   | impossible | $O(n)$   |

6.  $O(n^2)$  시간 걸리는 부분을 더 빨리 할 수 없을까?

a. 계수 지정 표현법  $\Rightarrow$  값 지정 표현법:  $O(S)$  시간 알고리즘이 존재하고, 값 지정 표현법  $\Rightarrow$  계수 지정 표현법:  $O(T)$  시간 알고리즘이 존재한다면:

- i. 함수 값 계산을 (1) 계수 지정  $\Rightarrow$  값 지정으로  $O(S)$  시간에 변환한 후, (2) 곱셈 계산을  $O(n)$  시간에 하고, (3) 다시 계수 지정 표현법으로  $O(T)$  시간에 변환함: 이 방법은 계수지정의 곱셈을  $O(S+T+n)$  시간에 할 수 있음을 의미함
- ii. 마찬가지로, 값 지정  $\Rightarrow$  계수 지정으로  $O(T)$  시간에 변환하고 계수 지정에서의 함수 값 계산을  $O(n)$  시간에 한 후, 계수 지정  $\Rightarrow$  값 지정으로  $O(S)$  시간에 변환하면, 값 지정 표현법에서의 함수 값 계산이  $O(S+T+n)$  시간에 가능함

b. 계수 지정 표현법  $\Rightarrow$  값 지정 표현법: Discrete Fourier Transform (DFT)에 의해  $O(n \log n)$  시간에 가능

c. 값 지정 표현법  $\Rightarrow$  계수 지정 표현법: Inverse DFT  $O(n \log n)$  시간에 가능

d. 결국,  $S = T = O(n \log n)$ 이 되어,  $O(n^2) \Rightarrow O(n \log n)$ 으로 개선 가능함!

## 7. 다시 정리 (근 지정 표현법은 더 이상 고려하지 않음)

| 표현법   | 함수 값 계산       | 덧셈     | 곱셈            |
|-------|---------------|--------|---------------|
| 계수 지정 | $O(n)$        | $O(n)$ | $O(n \log n)$ |
| 값 지정  | $O(n \log n)$ | $O(n)$ | $O(n)$        |

## 8. 계수 지정 표현법 $\Rightarrow$ 값 지정 표현법: Discrete Fourier Transform (DFT)

- a. **입력:** 리스트  $P$ 에  $a_0 + a_1x^1 + \dots + a_{n-1}x^{n-1}$ 의 계수가 저장되어 있다고 가정. 설명의 편의상, 최고차 항을  $n-1$ 로 지정 (그러면 입력의 크기가  $n$ 이 되어 일관성 있는 설명 가능)
- b. **출력:**  $(x_0, y_0), (x_1, y_1), \dots, (x_{n-1}, y_{n-1})$  (단,  $x_i$ 는 모두 달라야 함!)
- c. **방법 1:**  $x_i$ 를 임의로 정한 후,  $p(x_i)$ 를 계산하여  $\{(x_i, p(x_i))\}$  출력
  - i.  $p(x)$  계산은 위의 표에 따르면  $O(n)$ 에 할 수 있기 때문에  $n$ 개의  $x_i$ 에 대해선  $O(n^2)$  시간이 필요하다. 목표는  $O(n^2)$  시간보다 빨라야 하므로 이 방법은 의미가 없다
- d.  $p(x)$ 를 아래와 같이 두 개의 더 작은 다항식으로 분할해보자!

$$\begin{aligned}
 p(x) &= a_0 + a_1x^1 + a_2x^2 + a_3x^3 + a_4x^4 + \dots + a_{n-1}x^{n-1} \\
 &= a_0 + a_2(x^2)^1 + a_4(x^2)^2 + \dots + x^1(a_1 + a_3(x^2)^1 + a_5(x^2)^2 + \dots) \\
 &= p_{\text{even}}(x^2) + x \cdot p_{\text{odd}}(x^2)
 \end{aligned}$$

- e. **방법 2:** 분할정복방법으로 계산해보자.
  - i. 모든  $X = \{x_0, x_1, \dots, x_{n-1}\}$ 에 대해  $p_{\text{even}}(x^2)$ 와  $p_{\text{odd}}(x^2)$ 를 재귀적으로 계산한 후 더한다
  - ii.  $p_{\text{even}}()$  입장에서 입력은 무엇인가?  $x^2$ 이어야 하므로  $X^2 = \{x_0^2, x_1^2, \dots, x_{n-1}^2\}$  이 입력이 된다. 그러면 입력이  $n/2$ 로 줄지 않고, 경우에 따라선  $n$ 개 값이 모두 입력으로 필요하게 된다! (기존의 전형적인 분할정복법에서는  $n/2$  크기의 문제 두 개로 분할되었음을 기억하자)
  - iii. 이 방법의 점화식은 무엇인가?
    - $T(n, |X|) = 2T(n/2, |X|) + c(n+|X|)$ 
      - 일반적인 분할정복법에서는  $n = |X|$ 가 성립된다
      - $T(n) = 2T(n/2) + cn$ 이 되어  $O(n \log n)$ 이 가능
      - 그러나 지금은  $n$ 과  $|X|$ 는 다름! VERY STRANGE!
    - 결과적으로 이 점화식의 (최악의 경우의) 전개 결과는?  
[힌트: 점화식 트리를 그려 볼 것!]
  - iv. 예를 들어,  $n=4$ ,  $X=\{1, -2, 3, 2\}$ 이라고 하자. 이 경우에는  $X^2 = \{1, 4, 9\}$ 가 되어  $|X|$ 와 비교하면 1개의 적다. 재귀 호출을 할 때마다 입력이  $1/c$  씩 줄어들지 않으면 원하는 시간  $O(n \log_c n)$ 을 얻을 수 없게 된다!

f. 그러면 어떻게 해야 할까?  $X$ 의 값 중에서 두 값의 제곱 값이 같다면,  $X^2$ 에서는 두 수가 하나로 줄어든다. 이러한 성질을 최대한 만족하도록  $X$  값을 정할 수 있지 않을까?

- i. 예를 들어,  $X = \{1, -1\} \rightarrow X^2 = \{1\}$
- ii.  $X = \{1, -1, i, -i\} \rightarrow X^2 = \{1, -1\}$  (여기서  $i$ 는 허수임)
- iii. 이렇게 제곱을 할 때마다 값이 반씩 증복되어 결국 값의 개수가 반씩 줄어드는 집합을 **collapsing** 집합이라 한다

g.  $X$ 를 collapsing 집합으로 구성할 수 있다면,

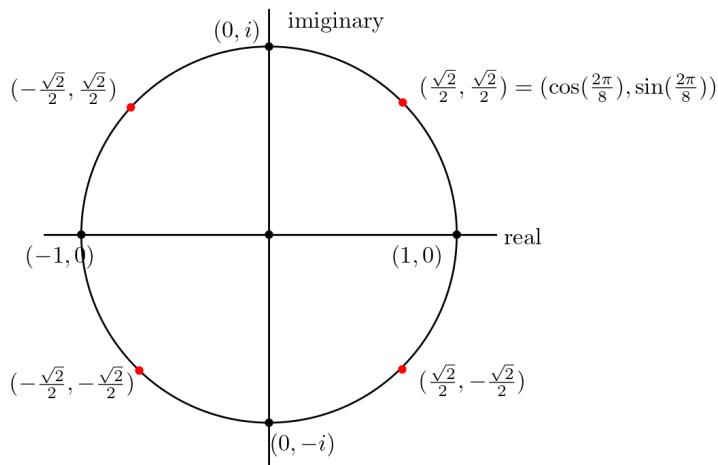
- i.  $T(n) = 2T(n/2) + cn$  이 성립하고,  $T(n) = O(n \log n)$ 이 된다
- ii. 그러면 쉽게 collapsing 집합을 만들 수 있는 방법이 있을까?

h. 단순하게  $n = 2^k$ 이라고 가정하자. ( $2^k$  아니라면 계수가 0인 항을 더 만들어  $2^k$ 이 되도록 채우면 (padding 하면) 된다)

- i.  $w^2 = 1$ 의 식의 해는?  $\{1, -1\}$
- ii.  $w^4 = 1$ 의 해는?  $\{1, -1, i, -i\}$
- iii.  $w^8 = 1$ 의 해는?  $\{1, -1, i, -i, a+ai, a-ai, -a+ai, -a-ai\}$ ,  $a = \sqrt{2}/2$
- iv.  $w^n = 1$ 의 해는? 이해 집합은 collapsing 집합인가?
  - $w^8 = 1$ 의 해를 제곱하면  $w^4 = 1$ 의 해와 같음을 알 수 있다. 즉, 8개의 해 중에서 4개의 해가 기존의 해와 같아져 4개로 개수가 줄게 된다
  - $w^4 = 1$ 의 해를 제곱하면 역시  $w^2 = 1$ 의 해와 같아져 수가 반으로 줄게 된다

i.  $w^n = 1$ 의 해를 **COMPLEX NTH ROOTS OF UNITY**라 부른다. 다음처럼  $k$ 번째 루트  $w_n^k$ 가 정의된다

- i. 허수 2차원 평면에 단위 원을 그려서 원 위의 점  $n$ 개를 등 간격으로 선택해보자. 원 위의 점을 극좌표(polar coordinate)로 표현하면  $(\cos \theta, \sin \theta)$  이 되고, 벡터로 표현하면  $\cos \theta + i \sin \theta$ 가 된다



- ii.  $\cos \theta + i \sin \theta = e^{i\theta}$ 이 성립됨이 증명되어 있다 [**Euler's formula**]
  - $e^{i\pi} = \cos \pi + i \sin \pi = -1 \rightarrow$  세 가지 심볼  $e, i, \pi$ 이 모두 들어간 세상에서 가장 유명한 식

- iii. 단위 원에서 n개의 점을 등간격으로 뽑는다는 것은  $2\pi/n$  각도 간격으로 뽑는다는 의미이다.  $2\pi = \tau$ 라고 표기하면,  $k=0, 1, \dots, n-1$ 에 대해, k번째 선택 점은  $e^{i\tau k/n}$ 이 된다는 뜻이다

- iv.  $(\cos \theta + i \sin \theta)^k = e^{i\theta k} = \cos k\theta + i \sin k\theta$ 이기 때문에 다음 식 성립

$$e^{i\tau k/n} = \left( \cos\left(\frac{\tau}{n}\right) + i \sin\left(\frac{\tau}{n}\right) \right)^k = \cos\left(\frac{\tau k}{n}\right) + i \sin\left(\frac{\tau k}{n}\right)$$

- 두 번째 항과 세 번째 항이 같은 이유는 De Moivre's formula 때문
- $k = 2$ 인 경우, 즉 제곱을 하면, 각이 2배가 되어 적용됨에 유의

- v.  $k$ 번째 해  $\omega_n^k = e^{i\tau k/n}$ 으로 정한다

- $(\omega_n^k)^n = e^{i\tau k} = \cos(k\tau) + i \sin(k\tau) = 1$ 이 되어 우리가 원하는 해 중의 하나임을 확인 할 수 있다

- vi. 성질 1:  $\omega_n^0 = 1, \omega_n^{n/2} = -1, \omega_n^{k+n/2} = -\omega_n^k$

- vii. 자세한 내용은 [https://en.wikipedia.org/wiki/Root\\_of\\_unity](https://en.wikipedia.org/wiki/Root_of_unity) 참조

- viii. n 개의 해  $\omega_n^0, \omega_n^1, \dots, \omega_n^{n-1}$ 를 x 값으로 정해보자

- 이 값들을 제곱하면, 그 중 반은 다른 반과 일치하게 된다
- 그래서 제곱을 하면 계속 반씩 줄게 되어 collapsing 집합이 된다. 단,  $n = 2^k$ 의 형태여야 정확히 반씩 줄어든다

- j. 결국,  $w^n = 1$ 의 해를 n개의 x 값으로 정하면 된다!

## k. Pseudo Code

- i. **입력:** 계수가 리스트  $P[0..n-1]$ 에 저장되어 주어진다 ( $n = 2^k$  형태)
- ii. **출력:** unity root  $x$ 에 대한  $p(x)$  값이 리스트  $Y[0..n-1]$  저장되어 리턴

```

from math import *
def DFT(P):
 n = len(P) # n is assumed to be 2k
 if n == 1: return P

 even = P[::2] # 짝수 계수 모으기
 odd = P[1::2] # 홀수 계수 모으기

 L = DFT(even)
 R = DFT(odd)

 # wn, w, Y[i] 모두 복소수로 정의되어야 함
 # python에서의 복소수는 complex 클래스 사용
 # 예: v = complex(2, 3) → v = 2 + 3i 의미
 wn = complex(cos(2*pi/n), sin(2*pi/n))

 w = complex(1) # w = 1
 Y = [complex(0)] * n # Y = [0, ..., 0]
 for j in range(n//2): # w0부터 wn//2까지만 고려해도 됨
 Y[j] = L[j] + w*R[j]
 Y[j+n//2] = L[j] - w*R[j] # 왜 -w인가? (성질1)
 w = w * wn
 return Y # return Y values only

```

- I. **결론:**  $(n-1)$ 차 다항식의 계수가 주어지면 이 다항식을 만족하는  $n$ 개의 샘플 값을  $O(n \log n)$  시간에 계산할 수 있다.

## 9. 값 지정 표현법 $\Rightarrow$ 계수 지정 표현법: Inverse DFT

a. (n-1)차 다항식을 행렬식으로 표현하면 아래와 같다

- i. 벡터  $a$ 는 계수를 모아 놓은 벡터이고,  $(x_i, y_i)$ 는 n개의 샘플 값임
- ii. 아래처럼  $Va = y$  행렬식으로 표현됨. 행렬  $V$ 를 Vandermonde matrix라 부름

$$\begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \vdots & x_{n-1}^{n-1} \end{bmatrix} \begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

- iii. 계수 벡터  $a$ 를 알면 행렬 곱셈을 수행해  $O(n^2)$  시간에  $x_i$ 에 대한  $y_i$  값을 계산할 수 있다 (당연함!)
- iv. 만약  $(x_i, y_i)$ 는 알고  $a$ 를 모른다면? 아래와 같이 역행렬을 구해  $a$ 를 계산할 수 있다

$$\begin{bmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & x_0 & x_0^2 & \cdots & x_0^{n-1} \\ 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n-1} & x_{n-1}^2 & \vdots & x_{n-1}^{n-1} \end{bmatrix}^{-1} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix}$$

- v. 그런데 일반적인 역행렬 계산법인 가우스 소거법은  $O(n^3)$  시간이 필요해 너무 느리다. 우리는  $x_i$  값들을  $w^n = 1$ 의 해로 지정했고, 이 경우엔 역행렬  $V^{-1}$ 을 더욱 쉽게 계산되지 않을까 하는 것이 유일한 희망이다

$$V = \begin{bmatrix} 1 & \omega_0 & \omega_0^2 & \cdots & \omega_0^{n-1} \\ 1 & \omega_1 & \omega_1^2 & \cdots & \omega_1^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \omega_{n-1} & \omega_{n-1}^2 & \vdots & \omega_{n-1}^{n-1} \end{bmatrix}$$

- vi.  $V^{-1} = \bar{V}/n$  임이 성립!

- $\bar{V}$  컬레 복소수 (complex conjugate)로 구성된 행렬을 의미
  - $a + bi$ 의 컬레 복소수는  $a - bi$
- 증명은 생략. unity의 근의 성질을 이용하면 어렵지 않음
- 각 원소를 컬레 복소수로 바꾼 후,  $n$ 으로 나누면 됨

- vii. 결국  $V^{-1}$ 에 벡터  $y$ 를 곱하면 된다. 즉,  $\vec{a} = V^{-1}y \rightarrow n\vec{a} = \bar{V}y$

- $\Rightarrow$  DFT에서  $V$ 에 벡터  $a$ 를 곱하는 것과 같은 본질적으로 같은 문제이므로, 앞에서 설명한 (계수  $\rightarrow$  값)으로 변경하는  $O(n \log n)$  시간 알고리즘을 그대로 적용한다!

- viii. Pseudo code

- 입력: unity root에 대한 n개의 y 값 리스트  $Y[0..n-1]$  (단,  $n = 2^k$ )

- 출력: 계수 벡터  $\mathbf{na}$ 를 저장한 리스트  $P[0..n-1]$  (**최종 값을 위해선  $n$ 으로 나눠줘야 한다**)

```

def Inverse-DFT(Y):
 n = len(Y) # 가정: n = 2k
 if n == 1: return Y

 even = Y[::2] # 짝수 계수 모으기
 odd = Y[1::2] # 홀수 계수 모으기

 L = Inverse-DFT(even)
 R = Inverse-DFT(odd)

 # wn, w, P 모두 complex 수로 초기화 필요
 wn = cos(2*pi/n) - i*sin(2*pi/n) # 켤레 복소수!
 w = 1
 P = [0]*n
 for j in range(n//2):
 P[j] = (L[j] + w*R[j])
 P[j+n//2] = (L[j] - w*R[j])
 w = w * wn
 return P

```

- ix. 결론:  $n$ 개의 샘플 값으로부터  $(n-1)$ 차 다항식의 계수를  $O(n \log n)$  시간에 계산할 수 있다

10. [응용1: 다항식 곱셈 - easy] 계수로 표현된 n차 다항식  $p(x)$ 와 m차 다항식  $q(x)$ 의 곱셈하기

- a. 곱셈한 결과도 계수로 표현되어야 함에 유의!
- b. 계수로 표현된 다항식의 곱셈은  $O(n^2)$  시간이면 충분
- c. 더 빨리 할 수 없을까? [hint: 두 다항식이 샘플 값으로 표현된다면  $O(n)$  시간에 가능!]
  - i.  $[O(n+m)]$  우선  $p$ 와  $q$ 의 차수를  $n+m$ 으로 늘림 ( $p$ 의 경우엔  $n+1$ 차부터  $n+m$ 차까지의 계수를 0으로 하면 되고,  $q$ 의 경우엔  $m+1$ 차부터  $n+m$ 차까지의 계수를 0으로 하면 됨) 결국 두 다항식 모두  $n+m$  차 다항식이 됨!
  - ii.  $[O((n+m)\log(n+m))]$   $y(p) = \text{DFT}(p)$ ,  $y(q) = \text{DFT}(q)$ 를 하여 다항식을 샘플 값으로 표현
  - iii.  $[O(n+m)]$   $y(p)$ 와  $y(q)$ 의 대응되는  $y$  값을 곱함. 단, 이 곱한 값을 계수로 표현해야 함:  $Y = y(p)*y(q)$
  - iv.  $[O((n+m)\log(n+m))]$  곱을 다시 계수로 표현해야 하므로  $\text{Inverse-DFT}(Y)$  수행
  - v. 모든 단계의 수행시간을 합하면  $O((n+m)\log(n+m))$  시간이면 충분!

## 11. [응용2: X\*Y - easy] 매우 큰 두 정수 X와 Y를 곱셈하기

- a. n-자리의 두 큰 정수를 곱할 때 일반적인 방법은  $O(n^2)$  번의 기본 덧셈과 곱셈을 수행
- b. 분할정복 파트에서 설명한 Karatsuba 알고리즘은  $O(n^{\log 3}) = O(n^{1.58})$  번으로 줄어든다. Karatsuba 방법도 더 빠르게 할 수 없을까? FFT를 이용하면  $O(n \log n)$ 까지 줄일 수 있다
- c. 그렇다면  $X = x_{n-1} \dots x_1 x_0$ ,  $Y = y_{n-1} \dots y_1 y_0$ 이라고 하자. 이 두 정수가 각각 두 리스트  $X = [x_0, x_1, \dots, x_{n-1}]$ ,  $Y = [y_0, y_1, \dots, y_{n-1}]$ 에 저장한다. 예를 들어,  $X = 3208$ ,  $Y = 2731$ 이라면,  $X = [8, 0, 2, 3]$ ,  $Y = [1, 3, 7, 2]$ 가 된다
- d. 우선  $X*Y$ 를 하면 자리수가 2배까지 늘어날 수 있으므로 두 리스트에 n자리의 0으로 채운 빈칸을 더 추가해 길이가 2n인 리스트를 마련한다
- e. 그러면  $X$ ,  $Y$ 의 값을 갖는 다항식  $P$ 와  $Q$ 를 정의하고, FFT로  $P*Q$ 를  $O(n \log n)$  시간에 계산한다. 그 결과를 리스트  $Z$ 에 저장한다
  - i. 예를 들어,  $283*407 = [3, 8, 2]*[7, 0, 4] = (3+8x+2x^2) * (7+4x^2) = 21+56x+26x^2+32x^3+8x^4 = [21, 56, 26, 28, 8]$ 이 된다. 그런데 리스트의 각 원소는 한 자리 수만 저장되어야 하므로, 낮은 자리부터 1의 자리만 남기고 10의 자리 값을 그 다음 자리로 carry 형태로 넘겨 덧셈 연산을 수행해 최종 결과를 얻어야 한다
  - ii.  $Z[0] = 21$ 에서 1만 남고 십의 자리의 2는  $Z[1]$ 의 값에 carry로 더해진다. 따라서  $Z[1] + 2 = 56 + 2 = 58$ 이 되어, 8이 10의 자리 값이 되고 5는 carry가 되어 다음 자리  $Z[2]$ 에 더해진다. 이 과정을 마지막 자리 값까지 반복하면 된다
  - iii. 이 과정은  $O(n)$  시간이면 충분하고, 최종적으로  $[1, 8, 1, 5, 1, 1] = 115181$ 을 얻는다
- f. 결론은  $O(n \log n)$  시간에 n 자리수 정수의 곱셈을 얻을 수 있다

12. [응용3:  $X+Y$  - medium]  $n$ 개의 정수 집합  $X$ 와  $Y$ 에 대해,  $X+Y = \{i+j \mid i \text{ in } X, j \text{ in } Y\}$  계산하기

- 단순한 방법으로는  $X$ 와  $Y$ 의 모든 쌍을 고려하는 것  $\rightarrow O(n^2)$ 이 되어 너무 느리다
- 두 다항식을 곱하면, 계수  $a_i$ 와  $a_j$ 는 서로 곱해지지만  $x^i$ 와  $x^j$ 의 차수는 서로 더해진다. 즉,  $x^{i+j}$ 가 된다
- 만약, 다항식  $p$ 의 각 항을  $X$ 의 값  $i$ 에 대해,  $x^i$  항으로 정의하고  $p$ 는 그런 항의 합으로 정의한다. 같은 방법으로 다항식  $q$ 를  $Y$ 의 값  $j$ 에 대한 항  $x^j$ 의 합으로 표현하면,  $X+Y$ 는 두 다항식의 곱  $p*q$ 를 계산하면 된다
- 이 계산은 응용1에 의해  $O((|X|+|Y|)\log(|X|+|Y|))$  시간이면 충분
  - 여기서  $|X|$ 는  $X$ 의 가장 큰 정수 값을 의미한다 (사실 이 값이 매우 크다면 매우 느릴 수도 있음에 유의)
- 그러면  $p*q$ 의 항 중에서 차수를 모든 모은 집합이 답  $X+Y$ 임을 알 수 있다

13. [응용4: 3SUM - medium]  $n$ 개의 서로다른 정수 리스트  $X$ 가 주어지면,  $a + b + c = 0$ 되는 세 수 찾기

- 이 문제를 3SUM 문제라 부른다 (<https://en.wikipedia.org/wiki/3SUM>)
- $O(n^2)$  시간에 가능할까?
  - 먼저  $a + b = 0$ 이 되는 쌍을 찾아보자. 최선의 방법은?
  - 위의 방법을 확장해  $a + b + c = 0$ 되는 세 수를 찾아보자. 수행시간은?
    - 정렬 알고리즘의 응용 문제로 소개한  $a + b = k$ 가 되는  $(a, b)$  쌍을 모두 찾는 문제와 매우 유사!
  - 특별한 자료구조를 사용한 방법을 생각해보자. [hint: 해시 테이블!] 수행시간은?
  - $O(n^2)$  시간보다 더 빨리 할 수 있을까? 오랫동안 더 빨리 하긴 매우 어려울 것으로 믿어 왔다. 그런데 2014년에 Allan Grønlund과 Seth Pettie에 의해  $O(n^2 / (\log n / \log \log n)^{2/3})$  시간에 수행되는 알고리즘이 제시되어,  $O(n^2)$  시간보다 약간 더 빠르게 해결할 수 있음이 증명되었다
- [고급] 만약  $X$ 의 수가 모두  $[-M, M]$  범위의 수라고 한다면  $O(M \log M)$  시간에 가능할까?
  - Hint: 응용3 문제에 적용했던 방법을 이용해 보자

14. [응용 5: 문자열 매칭 - hard] 문자열  $S$ 에서 특정 패턴 문자열  $T$ 가 등장하는 지의 여부 또는 등장 위치를 찾는 문자열 매칭 문제에 응용:  $O(n \log m)$  시간,  $n=|S|$ ,  $m=|T|$

a. 두 다항식  $P(x)$ 와  $Q(x)$ 의 곱셈은 아래와 같고, FFT에 의하면  $O(n \log n)$  시간에 계산 가능

$$P(x) = p_0 + p_1x^1 + \cdots + p_{n-1}x^{n-1}$$

$$Q(x) = q_0 + q_1x^1 + \cdots + q_{m-1}x^{m-1}$$

$$P(x)Q(x) = \sum_{i=0}^{n-1} \left( \sum_{j=0}^i p_j q_{i-j} \right) x^i$$

b. 매칭 문제를 FFT로 어떻게 변환할 수 있을까가 핵심

i.  $S = s_0s_1\dots s_{n-1}$ ,  $T = t_0t_1\dots t_{m-1}$

ii.  $T$ 가  $S$ 의  $i$ 번째 인덱스부터 매칭이 된다는 의미는  $t_0 = s_i$ ,  $t_1 = s_{i+1}$ , ...,  $t_{m-1} = s_{i+m-1}$ 이라는 뜻이다

iii. 만약, 문자를 서로 다른 정수로 변환해서 생각한다면 (예:  $a \rightarrow 1$ ,  $b \rightarrow 2 \dots$ )  $t_0 - s_i = 0$ ,  $t_1 - s_{i+1} = 0$ , ...,  $t_{m-1} - s_{i+m-1} = 0$ 이 된다

iv. 그러면 모든  $i = 0, \dots, n-1$ 에 대해,

$(t_0 - s_i) + (t_1 - s_{i+1}) + \cdots + (t_{m-1} - s_{i+m-1}) = 0$ 이 되는 인덱스  $i$ 가 존재한다면 패턴  $T$ 가  $S$ 의 인덱스  $i$ 부터 매칭이 된다는 뜻이다

v. 이 식을 다시 정리하면,

$$\sum_{j=0}^{m-1} (t_j - s_{i+j}) = 0$$

이 되는  $i$ 를 찾으면 된다

vi.  $t_j$ 와  $s_{i+j}$ 가 서로 곱해진 형식이 나와야 FFT 알고리즘을 이용할 수 있다. 이를 위해, 위의 합에서 각 항을 제곱해보자. 그래도 합은 0이 되는 인덱스를 찾으면 그 인덱스에서 매칭이 된다는 의미는 변하지 않는다

$$\sum_{j=0}^{m-1} (t_j - s_{i+j})^2 = 0$$

vii. 이 합을 전개하면,

$$\sum_{j=0}^{m-1} (t_j - s_{i+j})^2 = \sum_j t_j^2 + \sum_j s_{i+j}^2 - 2 \sum_j t_j s_{i+j} = 0$$

- 첫 합은 모든  $t_j$ 의 제곱의 합이므로  $O(m)$  시간에 미리 계산해 놓으면 상수 값처럼 사용하면 된다
- 두 번째 합은  $s_i^2 + s_{i+1}^2 + \cdots + s_{i+m-1}^2$ 로  $s_i$ 를 제곱해서  $i$  번째부터  $m$ 개의 값을 더한 값이다. 모든  $i$ 에 대해 이 합을 알아야 하는데, 그건  $O(n)$  시간에 미리 계산할 수 있다. (어떻게? 힌트: prefix sum)

- 세 번째 합은 어디서 많이 본 형태 아닌가?  $s_i$ 와  $t_j$ 를 계수로 하는 다항식을 곱한 후의 계수와 유사하다. FFT를 적용해서 구할 수 있을지 살펴보자

- viii.  $\sum_{j=0}^{m-1} t_j s_{i+j}$  항을 모든  $i$ 에 대해 계산해야 한다. 문제는 일반적인 두 다항식  $P(x)$ 와  $Q(x)$ 의 곱을 하면 계수는 다음과 같은 것이다

$$P(x)Q(x) = \sum_{i=0}^{n-1} \left( \sum_{j=0}^i p_j q_{i-j} \right) x^i$$

- 즉  $p_j q_{i-j}$ 의 곱으로 표현된다. 우리가 필요한 것은  $t_j s_{i+j}$ 로  $i-j$ 가 아닌  $i+j$  인덱스 형태이다
- ix. 다음과 같이  $S$ 를 위한 다항식  $S(x)$ 와  $T$ 를 위한 다항식  $T(x)$ 를 정의하자
- $$S(x) = s_0 + s_1 x^1 + s_2 x^2 + \cdots + s_{n-1} x^{n-1}$$
- $$T(x) = t_0 x^n + t_1 x^{n-1} + \cdots + t_{m-1} x^{n-m+1} + 0x^{n-m} + \cdots + 0x^0$$
- $T(x)$ 의 계수  $t_i$ 가  $x^i$ 가 아닌  $x^{n-i}$ 와 짹이 됨에 유의! (계수가 역순으로 짹을 맺음)
  - $x^{n-m}$ 부터  $x^0$ 까지의 계수는 실제하지 않으므로 모두 0으로 지정

- x.  $S(x)T(x)$ 에서  $x^{n+i}$ 의 계수는 무엇인가? 먼저  $i = 0$ 일 때  $x^n$ 의 계수부터 살펴보자.  $(t_0 s_0 + t_1 s_1 + \cdots + t_n s_n) x^n$  이 된다. 사실  $T(x)$ 의  $t_m$ 부터  $t_n$ 까지의 계수는 0으로 정의를 했기 때문에,  $(t_0 s_0 + t_1 s_1 + \cdots + t_{m-1} s_{m-1}) x^n$  으로 표현해도 된다. 만약, 이 계수가 0이라면 패턴이  $s_0$ 부터 매칭이 가능함을 의미한다

- xi.  $x^{n+i}$ 의 계수는  $(t_0 s_i + t_1 s_{i+1} + t_2 s_{i+2} + \cdots + t_n s_{i+n}) x^{n+i}$  가 되고,  $t_m$ 부터  $t_n$ 까지의 계수는 0으로 정의되기 때문에 계수 부분은 다음과 같이 다시 표기 가능하다

$$\sum_{j=0}^{m-1} t_j s_{i+j}$$

- xii. 결국, 이 계수가 우리가 필요한 값과 동일한 것임을 확인할 수 있다. 이로써 세 번째 항은 FFT로  $O(n \log n)$  시간에 계산이 가능하여 전체적인 패턴 매칭 시간은  $O(n \log n)$ 에 가능하다

- xiii. [?] 약간의 변형을 하면 매칭 시간을  $O(n \log m)$  시간으로 줄일 수 있다. 힌트:  $S$ 를  $2m$ 개씩 분할을 하고, 분할된 부문자열과  $T$ 를 패턴 매칭하는 방법을 생각해보자. 이 방법이 왜 올바른 답을 출력하는지, 수행 시간은  $O(n \log m)$ 으로 줄어드는지 확인해보자

## 13. 계산 복잡도(computational complexity) P vs. NP

### 1. 계산 모델 (Model of Computation)

- a. 계산 모델 (또는 가상 컴퓨터 모델)은 알고리즘의 성능(performance)인 수행 시간을 측정하기 위한 기본 모델로 필수적이다
- b. 현대 컴퓨터 구조는 Turing machine에 기초한 von Neumann 구조를 따른다
- c. 현재 가장 많이 사용하는 현실적인 계산 모델은 (real) **RAM**(Random Access Machine) 모델이다
- d. (real) RAM 모델은 CPU + memory + register + primitive operation으로 정의
  - i. 연산(operation, command)을 수행하는 CPU
  - ii. 임의의 크기의 실수 (즉, 정확한 실수 값을) 저장할 수 있는 무한한 워드(word)로 구성된 메모리
  - iii. CPU의 계산에 활용되는 충분한 개수의 레지스터(register) - 일종의 메모리
  - iv. 단위 시간에 수행할 수 있는 기본연산(primitive operation)의 집합
    - A = B (대입 또는 복사 연산: B의 값을 A 레지스터에 복사 또는 B 레지스터의 값을 A 메모리에 복사)
    - 산술연산: +, -, \*, / (나머지 % 연산이나 버림, 올림 연산 등은 원칙적으로 허용 안되나, 본 강의에서는 포함한다)
    - 비교연산: >, >=, <, <=, ==, !=
    - 논리연산: AND, OR, NOT
    - 비트연산: bit-AND, bit-OR, bit-NOT,bit-XOR, <<, >>
      - a. 비트 연산도 교재에서는 기본연산에 포함한다
- e. 요약하면, 이 교재에서 사용하는 Real RAM 모델은 (1) 정확한 실수를 저장할 수 있는 무한 크기의 메모리가 제공되고 (2) 메모리와 레지스터 사이의 값을 상수시간에 복사할 수 있고 (3) 사칙, 비교, 논리, 비트 연산을 상수 시간에 할 수 있는 모델이다
- f. 기본 연산만을 단위 시간에 처리 가능한 무한 메모리를 갖춘 (이상적이지만 단순한) 컴퓨터에서 코딩을 한다고 가정하면 큰 무리가 없다

### 2. 상한(upper bound)과 하한(lower bound)

- a. 어떤 문제를 관심있는 계산 모델에서 T 시간에 동작하는 알고리즘을 설계했다면, 그 문제의 상한은  $O(T)$ 라고 말한다 (즉, 모든 입력에 대해, 특정 시간 이내에 문제가 해결된다면 그 시간은 해당 문제의 상한 - upper bound - 이 된다)
  - i. 예를 들어, n개의 값을 정렬하는 문제에 대해, selection sort 알고리즘은  $O(n^2)$ 에 동작한다. 따라서 정렬 문제의 상한은  $O(n^2)$ 이다
  - ii. 그런데 merge sort 알고리즘은 이보다 빠른  $O(n \log n)$ 에 수행되므로 상한이  $O(n^2)$ 에서  $O(n \log n)$ 으로 감소한다. 즉, 모든 입력에 대해  $O(n \log n)$ 에 해결할 수 있음을 의미한다
- b. 반면에 하한은 관심있는 계산 모델에서 문제를 해결할 때, 최소 T 시간 이상이 반드시 필요한 입력이 존재한다면, 해당 문제의 하한은  $\Omega(T)$ 라고 한다

- i. 예를 들어,  $n$ 개의 값을 정렬하는 문제에서 두 값을 비교하며 정렬하는 경우엔  $cn\log n$  번의 비교를 반드시 할 수 밖에 없는 입력이 존재함이 이미 증명되어 있다
  - ii. 비교 횟수가 정렬 알고리즘의 수행시간이라 볼 수 있으므로, 비교를 통해 정렬하는 경우엔 이 세상 어떤 알고리즘도  $cn\log n$ 번 보다 적은 비교만으로 모든 입력을 정렬할 수는 없다는 뜻이다. 따라서 정렬 문제의 하한은  $\Omega(T)$ 이다
- c. 어떤 문제의 **상한과 하한이 일치**한다면, 꼭 필요한 시간 (하한)에 비례하는 시간으로 해결 (상한)했으므로 해당 문제는 완전히 해결되었다고 말한다
- i. 예를 들어, 정렬 문제에서는 merge sort와 heap sort 등의 알고리즘  $O(n\log n)$  시간에 항상 정렬하므로 하한의 수행시간과 일치한다. 결국 일반적인 값들을 정렬하는 문제는 완전히 해결되었다고 말해도 된다
  - ii. 문제의 상한과 하한이 모두  $T$ 로 일치하는 경우에는 해당 문제의 복잡도는  $\Theta(T)$ 라고 한다
- d. 예 1:  $n \times n$  행렬 A와 B를 곱셈하는 문제
- i. **상한:** 3중 for 루프를 이용하면 고등학교에서 배운 행렬 곱셈을 할 수 있으므로  $O(n^3)$  시간이면 가능하다. 이 알고리즘으로 상한은  $O(n^3)$ 이라 말할 수 있다. 이 후에 Strassen에 의해 제안된 분할정복 알고리즘은  $O(n^{2.807})$  시간에 수행되어 상한을 조금 더 줄였다. 현재까지 알려진 가장 빠른 알고리즘의 수행시간은  $O(n^{2.373})$ 이므로 문제의 현재 상한은  $O(n^{2.373})$ 이다
  - ii. **하한:** A와 B의  $n^2$ 개의 값을 모두 최소한 한 번씩 계산에 반영해야 하므로  $\Omega(n^2)$ 의 시간이 필요하다. 이 하한은 특별한 증명이 필요없는 기본적인 하한 (trivial lower bound)로 큰 가치는 없다. 그런데 이 당연한 하한 값보다 더 큰 하한은 아직 증명되지 않았다
  - iii. 결국, 하한  $\Omega(n^2)$ 과 상한  $O(n^{2.373})$ 의 차이(gap)가 존재한다. 아직 문제가 완전히 해결되지 않았다는 뜻이다. 완전한 해결을 위해선 두 가지 방법이 있다: (1) 더 빠른 수행시간의 알고리즘 설계해 상한 값을 줄이든지 (2) 더 큰 하한 값을 증명하여 하한 값을 키우든지 해야 한다
  - iv. 그러나 다행인 건 2차원 행렬 곱셈 문제의 상한과 하한이 일치하지 않더라도 차이는 그렇게 크진 않다는 것이다.
- e. 예 2: 그래프에서 해밀턴 경로 (Hamiltonian Path) 찾기
- i. 무방향 그래프의 경로 중에서 그래프의 노드가 정확히 한번씩 등장하는 경로를 해밀تون 경로라 부른다. 모든 노드를 방문하는 경로를 의미한다
  - ii. 그래프  $G = (V, E)$ 가 주어지면, 해밀턴 경로가 있는지 없는지를 판별하는 문제를 해결해보자
  - iii. **하한:** 그래프의 에지를 최소한 한 번 이상 고려할 수 밖에 없으므로 ( $|E|$ )  $\Omega(|E|) = \Omega(n^2)$ 이 당연한 하한이고, 이 보다 더 큰 하한은 알려져 있지 않다
  - iv. **상한:**
    - **알고리즘 1:** 그래프에 존재할 수 있는 서로 다른 경로가 최대 몇 개나 될까? 에지 개수가 가장 많은 무방향 그래프는 임의의 두 노드 쌍에 대해 에지가

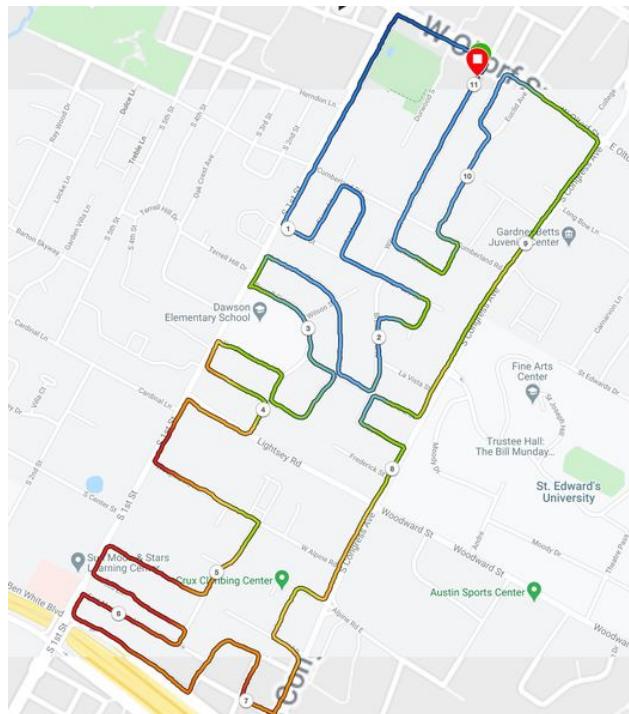
존재하는 경우다. 이 그래프를 완전 그래프 (complete graph)라 하고 에지는 수는  $n(n-1)/2$ 개가 된다. 이 완전 그래프에는 당연히 해밀턴 경로가 존재하고 쉽게 찾을 수 있다. 이유는  $n$ 개의 노드의 임의의 순열 (permutation)의 순서를 따르는 해밀턴 경로가 항상 존재하기 때문이다. 결국  $n!$ 개의 해밀تون 경로가 존재할 수 있다. 이 의미는 입력으로 주어진 그래프에 대해,  $n!$ 개의 경로에 대해, 해당 경로가 실제로 존재하는지 일일이 점검하면 된다는 것이다. 그래서  $O(n!)$  시간이면 충분하다.

- 알고리즘 2: DP 방법으로 조금 더 빨리 찾을 수 있다. 그래프 노드의 부분 집합  $S$ 와  $S$ 의 한 노드  $u$ 에 대해,  $S$ 의 모든 노드를 지나면서 노드  $u$ 에서 끝나는 경로의 존재하면 이차원 배열  $P[S][u] = \text{True}$ , 존재하지 않으면  $\text{False}$ 라고 하자. 그러면  $S = V$ 에 대해,  $P[V][u] = \text{True}$ 인 노드  $u$ 가 존재하면 해밀턴 경로가 존재하는 것이고,  $\text{True}$ 가 되는 노드가 존재하지 않는다면 해밀턴 경로가 존재하지 않는 것이다. 이차원 배열의 크기는  $V$ 의 모든 부분 집합의 개수  $\times$  노드 수가 되어  $2^n \times n!$ 이 된다.  $P[S][u]$ 에 대한 DP 점화식을 만들어보자 [?].  $P[S][u]$ 의 값은  $O(n)$  시간이면 충분히 계산할 수 있다. 따라서  $O(n^2 2^n)$ 이면 충분하다. 여전히  $2^n$  항이 들어있기 때문에 지수시간 알고리즘이다. 현재  $n$ 에 관한 다행시간 알고리즘은 알려져 있지 않다
- f. 이 문제의 경우엔 하한은 다행 시간이고 상한은 지수 시간이 되어 하한과 상한의 차이가 매우 크다. 이를 줄일 수 있는 지의 여부는 계산 복잡도 분야에서의 중요한 **화두**이다
- g. [?] 그래프에서 두 노드 사이의 최단 경로 문제 (shortest path problem)는 Dijkstra 알고리즘에 의해 다행 시간에 계산할 수 있으므로 상한은  $O(n^3)$ 을 넘지 않는다. 그래프의 에지는 모두 고려해야 하므로 하한은  $\Omega(n^2)$ 이다. 최적은 아니지만 차이는 작다. 그런데 두 노드 사이의 가장 긴 (최장) 경로를 계산하는 문제는 어떨까? 최단 경로처럼 다행 시간에 계산할 수 있을까?
- i. 에지의 가중치가 모두 동일하다고 하자. 그러면 최장 경로는 경로의 에지 수가 최대인 경로이다. 모든 노드를 포함하는 해밀턴 경로가 그래프에 존재한다면 그 경로가 (당연히) 최장 경로가 된다. 결국, 그래프에 해밀턴 경로가 존재하는지의 여부를 묻는 해밀턴 경로 문제를 풀어야 한다. 최단 경로 문제와는 다르게 최장 경로 문제는 상당히 어려운 문제가 된다
  - ii. 특정 조건을 최대화 또는 최소화하는 문제를 **최적화 문제** (optimization problem)라 부른다. 최장 경로 문제는 경로 중에서 길이가 최대인 경로로 찾는 최적화 문제다
- h. 예 3: 해밀턴 사이클 (Hamiltonian Cycle) 문제
- i. 해밀턴 사이클은 그래프의 모든 노드를 지나는 사이클로 정의된다
  - ii. 해밀턴 사이클 문제는 주어진 그래프에 해밀턴 사이클이 존재하는지를 결정하는 것이다. 해밀턴 경로 문제를 해결하는 알고리즘을 약간만 변경하면 같은 방법으로 결정할 수 있다
  - iii. 결론적으로 해밀턴 경로 문제와 해밀턴 사이클 문제의 복잡도는 같다

- 이를 증명하기 위해선, 해밀턴 경로 문제를 해밀턴 사이클 문제로 변경할 수 있어야 하고, 반대 방향으로도 변경할 수 있어야 한다 [?] 어떻게 하면 될까?

i. 예 4: 오일러 사이클 (Eulerian Circuit) 문제

- i. 그래프에서 오일러 사이클 문제는 "한붓그리기" 문제로 알려져 있다. 그래프의 노드는 여러 번 지날 수 있지만 그래프의 에지는 정확히 한 번씩만 지나야 한다 (그래서 Cycle이 아니라 Circuit이라고 했다)



(René van Oostrum's running circuit, under permission)

- ii. [?] 오일러 사이클이 존재하는 필요 충분 조건은 뭘까? (많이 알려진 조건임!)
  - 에지 하나를 지나 노드에 도착하면 다시 다른 에지를 이용해 나가야 하므로 모든 노드의 분지수 (degree)가 짹수이어야 한다! (분지수가 0인 노드가 있어도 된다) 따라서 이 조건은 꼭 필요한 조건이어야 한다 (필요조건)
  - 모든 노드의 분지수가 짹수이면 항상 오일러 사이클이 존재하는가? 즉, 이 조건이 오일러 사이클이 존재하는 충분한 조건인가? 이 사실만 증명하면 필요 충분 조건을 보인 것이다
  - 충분 조건을 보이기 위해선, 모든 노드의 분지수가 짹수인 입력 그래프에 대해서, 항상 오일러 사이클을 계산하는 알고리즘을 설계하기만 하면 된다. 이 알고리즘은 어떻게? (어려운 듯 아닌 듯 한데…)
  - 임의의 노드부터 시작해 에지의 연결 리스트를 점진적으로 만든다. 노드에 인접한 에지 중에서 아직 방문하지 않은 에지를 하나 선택해 연결 리스트에 삽입한다. 에지에 연결된 다른 노드에서 다시 방문하지 않은 에지를 선택해 연결 리스트에 삽입한다. 이런 선택은 노드의 분지수가 항상 짹수이르모 이런

선택은 항상 가능하다. 이 방식으로 모든 에지가 선택될 때까지 반복하면 된다

- iii. 필요 충분 조건은 모든 노드에서 인접한 에지 개수를 검사하면 되기 때문에,  $O(|E|)$  =  $O(n^2)$  시간이면 충분하다. 따라서 다행 시간에 오일러 사이클 문제가 풀린다
  - iv. **놀랍게도**, 모든 노드를 한 번씩 지나는 사이클의 존재 여부를 아는 것은 매우 어렵지만 모든 에지를 한 번씩 지나는 사이클의 존재 여부를 아는 것은 매우 쉽다!
    - 어떤 차이때문에 이런 현상이 발생하는걸까?
  - v. [?] 입력으로 주어진 그래프가 오일러 경로가 존재하는 경우, 실제 오일러 경로 하나를 계산하고 싶다면 어떻게 해야 할까?
- j. 앞의 몇 가지 예제에서 살펴 본 것처럼 문제에 따라 하한과 상한의 차이가 매우 큰 경우가 있다. 이는 본질적으로 해결하기 어려운 문제들이 존재할지도 모른다는 (어찌보면 당연한) 의심을 갖게 한다. 이런 의심이 사실이라면 문제의 어려움의 정도를 기준으로 여러 그룹으로 묶어서 분류할 수 있지 않을까? 그러면 어려운 문제를 빠르게 해결할 수 있다는 헛된 희망을 가지고 풀려는 헛수고를 줄일 수 있지 않을까?

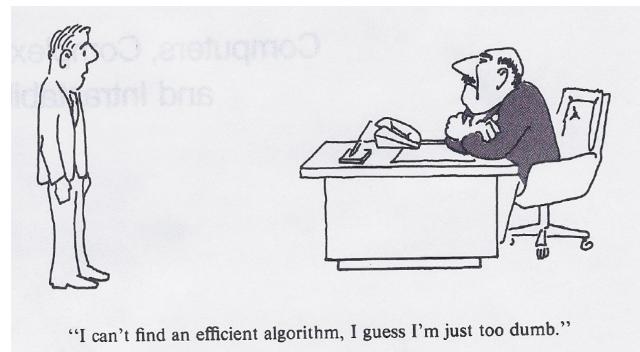
### 3. P vs. NP

#### a. 몇 가지 가정

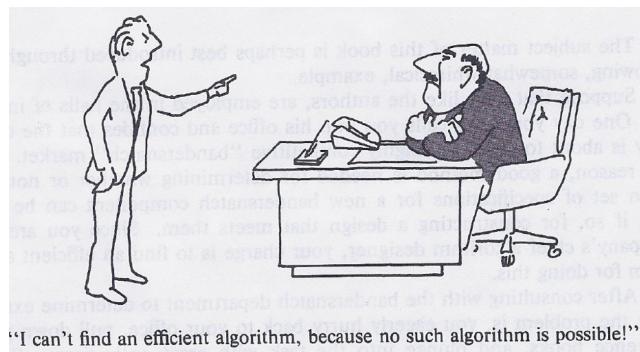
- i. 입력의 크기는  $n$ 으로 표기
- ii. 다향 시간이라는 것은 수행 시간이  $n$ 에 관한 다향식으로 표현되는 것을 말한다.  
교재에 등장하는 거의 모든 알고리즘은 다향 시간 알고리즘이다
  - 예:  $O(n^3)$ ,  $O(n^{4.5})$ ,  $O(n^2 \log n)$
- iii. 결정 문제 (decision problem)의 복잡도를 따진다
  - 예를 들어, 원래 해밀턴 경로 문제는 해밀턴 경로가 존재한다면 그 경로를 실제로 출력해야 하고, 그런 경로가 없다면 없다고 출력하는 것이다. 이 문제의 **결정 문제 버전**은 입력 그래프에 해밀턴 경로가 존재하면 **True** (Yes), 아니면 **False** (No)를 출력하는 것이다
  - 최장 경로 문제처럼 최적화 문제의 경우에는 그래프와 양의 정수  $k$ 가 주어지고, 그래프에 길이가  $k$  이상인 경로가 존재하는지 여부를 결정하는 형식이 된다
- iv. 결정 문제는 원래 문제보다 간단하지만, 복잡도는 본질적으로 차이가 없다. 그래서 계산 복잡도를 연구하는 분야에서는 결정 문제의 복잡도만으로 원래 (최적화) 문제의 복잡도를 증명하는 것이 일반적이다

#### b. 어려운 문제가 정말 어렵다는 것을 어떻게 증명할 수 있을까?

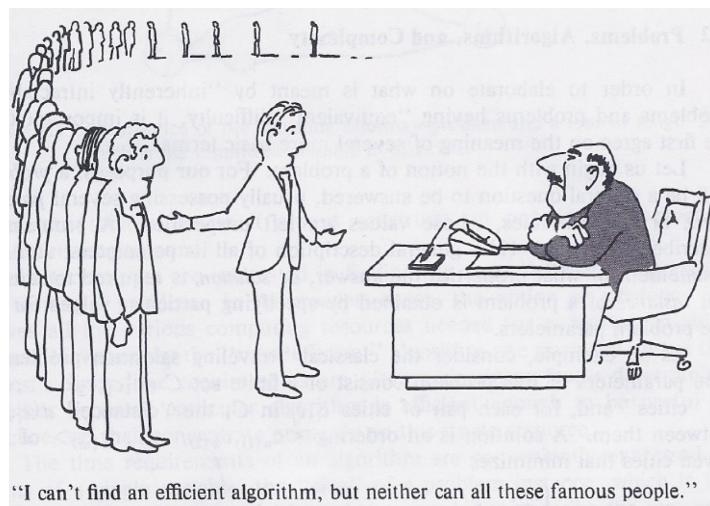
- i. 복잡도에 관한 가장 훌륭한 소개서라고 평가받는 M. Gary and D. S. Johnson, "Computers and Intractability" (1979) 책에 등장하는 카툰이 핵심을 말해준다



솔직하지만 무능함만 드러내는 꼴



효율적인 알고리즘이 없다는 증명을 할 수 없다는 게 약점



효율적인 알고리즘이 없다는 사실을 증명하기 어려운 상황에서  
가장 설득력이 있는 논리

- ii. 문제의 어려운 정도를 상대적으로 비교한다: 문제 A를 끈 사람이 아무도 없을 정도로 악명이 높다. 따라서 모두가 A는 매우 어려운 문제라고 인정한다. 그런데 내 문제 B도 A 만큼 어렵다는 걸 논리적으로 보일 수 있다. 그럼 B도 매우 어렵다고 당당히 말할 수 있다!

### c. NP 문제

- i. **Nondeterministic Polynomial**의 약자 (**Not 'Not Polynomial'**)
- ii. 대충(!) 말하면, 결정 문제의 답을 추측해서 그 추측(guess)이 정답(Yes)인지 다행 시간에 검사할 수 있는 문제들을 NP 문제라고 정의한다
- iii. 답을 추측하는 부분은 말 그대로 oracle(신탁)이 다행 시간에 알려준다고 가정하기 때문에, 결국 다행 시간의 **정답 체크 알고리즘**이 존재하는 문제를 NP 문제라고 정의할 수 있다. 따라서 이러한 문제의 범위는 매우 넓다
  - 신탁이 guess에 도움을 주기에 "nondeterministic"이란 단어가 사용되었음 → 비현실적인 모델로 복잡도 분석만을 위한 장치
  - 정확히 표현하면 답을 **nondeterministic polynomial time**에 **guess**하고, 그 답이 Yes인지 **deterministic polynomial time**에 **verify**하는 문제들의 클래스를 NP라 정의한다! (guess는 oracle이 verify는 내가 한다!)

NP = { decision problems with polynomial size certificate and polynomial time verifier for Yes input }
- iv. 예를 들어, 해밀턴 경로 문제는 NP이다. 출발노드에서 갈 수 있는 다음 노드는 최대 **n-1**개다. 이 중에서 어떤 노드로 가야 해밀턴 경로를 구성할 수 있는지를 정확히 신탁이 (상수시간에) 알려준다고 하자. 이 과정으로 계속 **n**번만 반복하면 (신탁의 도움을 받아서) 다행 시간에 guess 할 수 있다. 이 guess의 답이 Yes인지를 다행

시간에 판단해야 한다. 어떤 경로가 해밀턴 경로인지는 그 경로에 등장하는 노드가 빠짐 없이 한 번씩 등장하는지만 검사하면 되기 때문에  $O(n \log n)$  시간이면 충분히 확인 (verify) 가능하다

- v. 최장 경로 문제 역시 NP이다. 최장 경로 문제는 길이가  $k$  이상인 경로가 존재하는지를 결정하는 것이다. 주어진 경로의 길이가  $k$  이상인지는 경로의 에지의 가중치를 모두 더해  $k$  이상인지 비교하면 되기에  $O(n)$  시간에 쉽게 확인할 수 있다
- vi. [주의] 교재에서 현재까지 다른 모든 문제는 정답 여부를 다항시간에 검사할 수 있기 때문에 모두 NP 클래스에 속하는 문제이다

d. P 문제

- i. 정답을 다항 시간에 출력가능한 문제들의 클래스
  - 교재의 문제 중에선 백 트래킹 문제 몇 가지를 제외하고는 모두 P 클래스에 속함
- ii. 정답을 다항 시간에 출력할 수 있기에, 주어진 출력 값이 정답인지 아닌지도 당연히 다항 시간에 알 수 있다. 따라서,

$$P \subseteq NP$$

- iii. [ ? ? ? ] NP에는 속하지만 P에는 속하지 않는 문제가 존재하는가?

$$NP - P \neq \emptyset \text{ 또는 } P \not\subseteq NP$$

- 100만 달러 상금이 걸린 매우 유명한 질문이다. 질문의 답이 참이면,  $P \neq NP$ 이고 거짓이면  $P = NP$ 가 된다
- $P = NP$ 라면, 정답인지 검사하는 것과 정답을 실제 계산하는 것은 본질적으로 같다는 의미이다. 그러나 많은 연구자들은  $P \neq NP$ 라고 믿고 있다 (아직 증명되지 않았으니 도전해서 100만 달러의 주인공이 되길...)

#### 4. 문제 변환 (Reduction)

- a. 문제 A와 B가 있을 때, A 문제의 입력을 B 문제의 입력으로 변경할 수 있고 B 문제를 풀어 그 정답을 다시 A 문제의 정답으로 만들 수 있다면, B 문제의 알고리즘으로 A 문제를 해결할 수 있다. 이 과정을 변환 (reduction)이라 하고,  $A \rightarrow B$ 로 표기한다. 변환을 통해 B의 알고리즘으로 A를 해결할 수 있게 된다
  - i. A 문제의 입력을 B 문제의 입력으로 변경한다 ( $T_{\text{input}}$  시간 필요)
  - ii. B 문제의 알고리즘으로 B 문제의 정답을 계산한다 ( $T_B$  시간 필요)
  - iii. B 문제의 정답을 A 문제의 정답으로 바꾼다 ( $T_{\text{output}}$  시간 필요)
- b. 문제 A의 복잡도  $T_A$ 의 상한은  $T_A \leq T_{\text{input}} + T_B + T_{\text{output}}$ 이 된다. 이는 문제 B의 복잡도  $T_B$ 가 작다면 문제 A의 복잡도  $T_A$ 도 작아진다는 뜻이다
- c. 반대로 문제 B의 복잡도  $T_B$ 의 하한은  $T_B \geq T_A - T_{\text{input}} - T_{\text{output}}$ 이 된다
- d. 예1: 유일성 문제 → 최근접쌍 문제

A = **Uniqueness 문제**: n 개의 수가 모든 다른지 판별

B = **Closest pair 문제**: n개의 수 중 가장 가까운 두 수 계산

- i. A의 입력을 그대로 B의 입력으로 사용  $T_{\text{input}} =$  단순 복사 =  $O(n)$
- ii. B의 알고리즘은 n개의 값을 오름차순으로 정렬한 후, 가장 작은 값부터 보면서 인접한 두 수의 값 차이의 최소 값을 유지하는 식으로 진행한다. 따라서  $T_B = O(n \log n)$
- iii. B의 정답이 어떤 두 수 a, b일 때, A의 정답은  $|a-b|=0$ 이면 Yes, 아니면 No로 출력하면 된다. 따라서  $T_{\text{output}} = O(1)$
- iv. 결국 유일성 문제는  $O(n) + O(n \log n) + O(1) = O(n \log n)$ 에 풀린다

- e. 예2: 편집 거리 문제 → LCS 문제

A = **Edit distance 문제**: 문자열 X를 삽입, 삭제 연산의 최소 횟수를 적용해 Y 만들기

B = **Longest Common Subsequence 문제** : X, Y의 최장공통부수열 계산

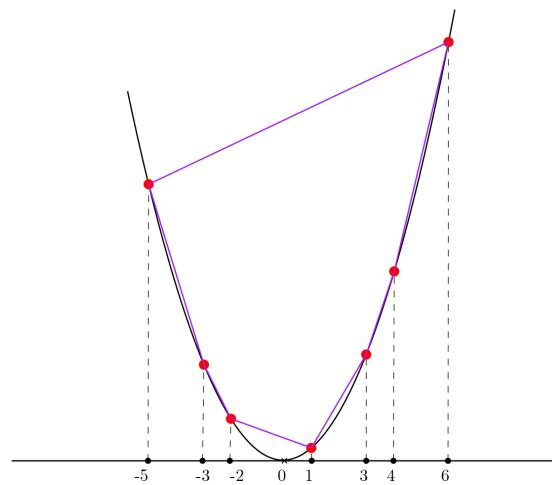
- i. 편집거리문제의 두 입력 문자열 X, Y를 그대로 LCS 문제의 입력으로
- ii. LCS(X, Y)를 계산한다. 최장공통부문자열의 길이가 k라고 하자
- iii. X에서 최장공통부문자열에 속하지 않는 문자들은 삭제하고  $(|X|-k)$ 번 삭제 연산 수행), X에는 없지만 Y에는 있는 문자들은 삽입한다 ( $(|Y|-k)$ 번 삽입 연산 수행)

- f. 예3: 정렬 문제 → 볼록 헐 문제 (유명한 변환 예)

A = **Sorting 문제** : n개의 수를 오름차순으로 배열

B = **Convex Hull 문제** : 2차원의 n개의 점의 볼록 헐을 계산

- i. n 개의 수  $x_1, x_2, \dots, x_n$ 을 각각  $(x_1, x_1^2), (x_2, x_2^2), \dots, (x_n, x_n^2)$ 으로 이차원 평면의 점으로 변환
- ii. 이 점들에 대한 볼록 헐 C를 계산한다
- iii. C에 포함된 점 중에서 x-좌표가 가장 작은 점부터 차례대로 C의 포함된 점의 x좌표를 출력하면, 출력되는 값이 정렬 순서와 일치한다 ( ? 왜 그럴까? 그림을 그려보면 이유를 쉽게 알 수 있다)

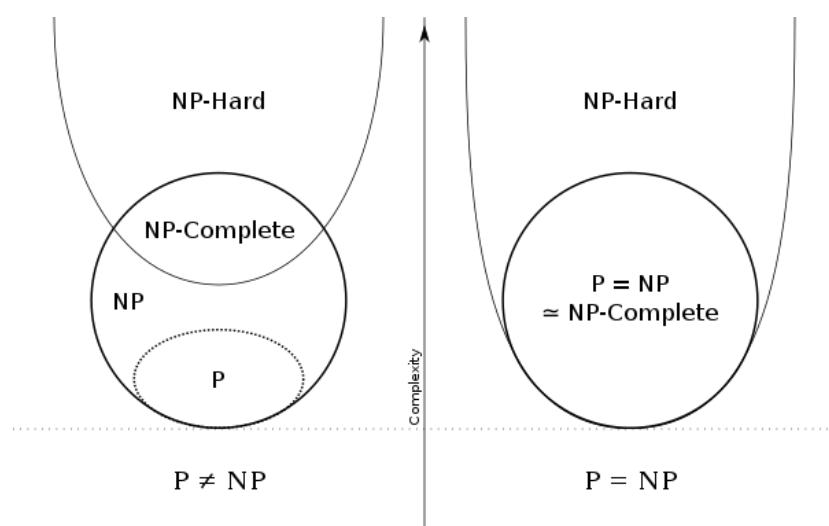


[ $n$ 개의 수 = [1, -2, 4, 6, -5, 3, -3, 4]라고 하면,  
각 수를 포물선 위의 빨간 점으로 변환]

- g. 두 수를 비교하는 연산을 통한 정렬 문제는 하한이  $\Omega(n \log n)$ 임이 알려져 있다. [정렬 문제 편  
참조] 정렬 문제를 볼록 헐 문제로 변환할 수 있기 때문에 볼록 헐 문제의 하한 역시 정렬  
문제의 하한인  $\Omega(n \log n)$ 과 동일함을 알 수 있다

## 5. NP-hard 문제와 NP-complete 문제

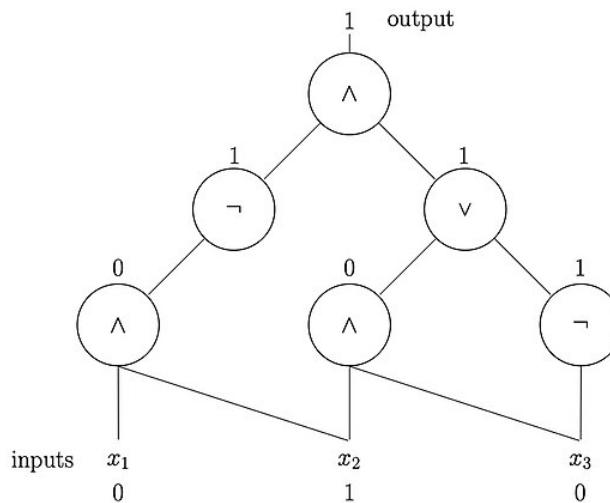
- a. **모든** NP 문제가 어떤 특정 문제 X로 다행 시간에 변환 (reduction)되면 X를 **NP-hard** 문제라고 한다. (여기서 핵심 단어는 "모든") NP 문제 모두가 X로 변환되었기 때문에 X가 효율적을 해결되면 변환의 정의에 의해 모든 NP 문제가 효율적으로 해결된다는 의미이다. 즉, X가 NP 문제들보다 어려운 문제라는 뜻이다 (X가 NP에 속해야 한다는 조건이 없음에 유의!)
- b. NP-hard 문제 X가 NP에 속하는 문제라면 X를 특별히 **NP-complete**이라 부른다. NP-complete 문제에 대한 다행 시간 알고리즘이 있다면, NP의 모든 문제가 다행 시간에 풀리게 되어  $P = NP$ 가 자동적으로 증명된다. 엄밀하진 않지만, "NP-complete 문제는 NP 문제 중에서 가장 어려운 문제들을 모아 높은 집합"이라고 말할 수 있다. 그러나 아직까지 NP-complete 문제에 대한 다행 시간 알고리즘이 알려져 있지 않고, 대부분의 학자들은 다행 시간 알고리즘이 존재하지 않을 것으로 믿고 있다. (존재하지 않을 것으로 추측하는 것이지 증명된 것은 아니다)
- c. 이 관계를 그림으로 표현하면 아래와 같다 [출처: wikipedia]



- d. 그럼 가장 첫 NP-complete 문제는 어떤 문제인가? 이 물음에 답하기 위해서는 모든 (무수히 많은) NP 문제가 첫 NP-complete 문제로 "변환"된다는 것을 "증명"해야 한다. 어떻게 이게 가능할까?

### i. Circuit Satisfiability 문제 (줄여서 **Circuit SAT** 문제라 부름)

- Boolean 함수: 두 bit 또는 한 bit를 입력으로 받아 1 또는 0을 출력하는 함수. 대표적으로 AND, OR, NOT 함수
- Gate: AND, OR, NOT bool 함수를 회로 게이트로 구현한 것
- Boolean Circuit: 입력으로 N개의 bit를 받아 AND, OR, NOT 게이트로 구성된 회로를 통해 M개의 bit를 출력하는 회로

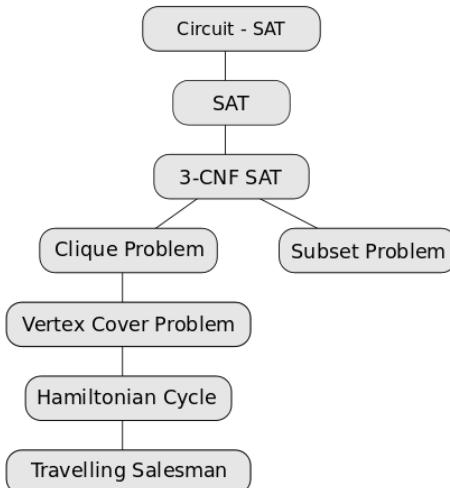


[그림 출처: wikipedia - circuit SAT 페이지]

- **Circuit SAT 문제:** 주어진 boolean circuit에 출력이 1이 되는 입력 bit 조합이 존재하는지 여부를 판별하는 결정 문제
  - a. 위의 그림에서는  $x_1 = 0$ ,  $x_2 = 1$ ,  $x_3 = 0$  으로 입력 비트를 정하면 출력이 1이 되므로 위의 회로에 대해선 circuit SAT 문제의 결과는 Yes이다
- **(1971) Cook-Levin Theorem:** Circuit SAT 문제는 NP-complete이다

**증명 IDEA:** 답인지 아닌지는 직접 각 게이트를 통과시켜  $O(n)$  시간에 검사할 수 있으므로 Circuit SAT 문제는 NP 문제이다. 따라서 모든 NP 문제가 Circuit SAT 문제로 "변환"됨을 증명하기만 하면 된다. 모든 NP 문제 - 결정 문제를 boolean circuit으로 설계할 수 있고, NP 문제의 입력을 encoding해서 circuit의 이진 비트로 변환할 수 있고, circuit의 출력이 1이면 NP 결정 문제의 답이 Yes, 0이면 No가 됨을 보이면 된다. 자세한 증명은 범위를 벗어나므로 생략한다

- 이제 첫 NP-complete 문제를 알게 되었다. 다른 NP-complete 문제는 어떻게 증명하면 될까?
  - NP-complete으로 이미 증명된 문제 A를 NP-complete 문제로 증명하고 싶은 문제 B로 "변환"한다. 즉,  $A \rightarrow B$
  - 문제 B는 NP 문제임을 증명한다
- 위의 단계 (i)는 (모든 NP 문제)  $\rightarrow A \rightarrow B$ 임을 의미하므로 (모든 NP 문제)  $\rightarrow B$ 가 된다는 의미이다 (변환은 정의에 의해 transitive하다. 즉,  $A \rightarrow B \rightarrow C$ 이면  $A \rightarrow C$ 가 성립한다)
- 이렇게 해서 지금까지 수 많은 문제가 NP-complete으로 증명되어 있음. 아래 그림은 대표적인 NP-complete 문제와 변환 관계를 나타낸다 (출처: wikipedia)



- i. 위 그림에서 보듯, 해밀턴 사이클 (또는 경로) 문제는 NP-complete이고 다행 시간에 풀기 매우 어려운 문제일 가능성이 매우 높은 문제이다
- ii. Subset Problem은 백 트래킹에서 살펴보았던 Subset Sum 문제이다. 이 문제 역시 3SAT 문제로부터 변환되기 때문에 NP-complete 문제이다
- h. NP-hard 문제는 어떻게 증명하나?
- i. 이미 NP-hard (NP-complete도 포함)로 알려진 문제를 증명하기 원하는 문제로 "변환"하기만 하면 된다. 해당 문제가 NP에 속하는지는 증명할 필요가 없는 점이 NP-complete 증명과의 차이점이다
- i. **NP-complete 문제 증명 예1** : Hamiltonian Cycle Problem (HCP) → Travelling Salesperson Problem (TSP)
- i. TSP: 에지에 음이 아닌 가중치 값을 갖는 그래프에서 사이클의 가중치의 합이  $k$  이하가 되는 해밀턴 사이클이 존재하는지 결정하라
  - ii. Idea: TSP도 해밀턴 사이클을 구하는 것이므로, TSP 문제에서 특정 가중치 이하의 해밀턴 사이클이 존재하기만 하면 HCP의 답이 Yes가 된다. 아니면 No이다. 결국  $k$  값을 정하기만 하면 된다
  - iii. [NP에 속하는가?] TSP는 NP에 속한다는 것을 쉽게 확인할 수 있다. 사이클이 해밀턴 사이클이면서 사이클의 가중치 합이  $k$  이하인지  $O(n)$  시간에 확인할 수 있기 때문이다
  - iv. [변환] HCP의 입력:  $G = (V, E) \rightarrow$  TSP의 입력:  $G' = (V', E'), k$ 
    - $V' = V$ 로 같은 노드 집합으로 정의
    - $E'$  = 모든 노드 쌍에 대한 에지를 정의하고, 그 에지가  $E$ 에 있으면 가중치 0을 없으면 가중치 1을 부여한다
    - 이렇게 모든 노드 쌍에 대한 에지가 존재하는 그래프를 완전 그래프 (complete graph)라 부른다
    - $k = 0$ 으로 한다

v.  $G'$ 과  $k = 0$ 에 대해 TSP 답을 구한다. 0 이하의 가중치를 갖는 해밀턴 사이클이 존재하면 Yes가, 아니면 No가 답이 된다

- $G'$ 이 완전그래프이기 때문에 해밀턴 사이클이 존재한다.
- 답이 Yes라는 의미는  $k = 0$  이하의 가중치를 갖는 해밀턴 사이클이 존재한다는 것이므로 사이클의 모든 에지가  $G$ 의 에지라는 것이다. 따라서  $G$ 의 해밀턴사이클이 존재하게 되어 HCP 결과로 Yes를 출력하면 된다. 사이클의 가중치가 0보다 크다는 뜻은 가중치가 1인 에지를 최소 하나 이상 사용했다는 뜻이므로  $G$ 에는 해밀턴 사이클이 존재하지 않는다고 판별할 수 있다 따라서 HCP의 결과로 No를 출력한다
- 결국, HCP의 답을 TCP의 답으로부터 구할 수 있고, 역으로 HCP의 답이 TCP의 답이 된다 ( $G$  has a Hamiltonian cycle if and only if  $G'$  has a cycle passing through all nodes once with at most length  $k$ .)

vi. 따라서 TSP 문제도 NP-complete이다

j. **NP-complete 문제 증명 예2:** Subset Sum → 0/1 Knapsack Problem

- i. 백 트래킹에서 다루었던 Subset Sum 문제는 3SAT NP-complete 문제로부터 변환이 가능해 NP-complete 문제이다
- ii. Subset Sum 문제를 0/1 Knapsack 문제로 변환해 0/1 Knapsack 문제 역시 NP-complete임을 증명해보자 (backtracking의 대표적인 문제로 이미 설명함)
- iii. **Subset Sum 문제:**  $n$ 개의 자연수 집합  $A = \{a_1, a_2, \dots, a_n\}$ 과 값  $s$ 가 주어진 경우, 합이  $s$ 가 되는  $A$ 의 부분집합이 존재하는지 여부를 결정하는 문제
- iv. **0/1 Knapsack 문제:**  $n$ 개의 아이템의 크기 (또는 무게)와 이익이 각각  $s_1, s_2, \dots, s_n, p_1, p_2, \dots, p_n$ 이고 배낭의 크기  $K$ 가 주어질 때, 배낭의 크기를 넘지 않도록 선택한 아이템의 이익의 합이 (역시 입력으로 주어진)  $P$  이상이 되는지 결정하는 문제
- v. **NP:** 0/1 Knapsack 문제는 NP에 속한다: 매우 당연 (직접 해보기를)
- vi. **변환:** 역시 매우 간단
  - 모든  $i$ 에 대해,  $s_i = a_i$ 로 지정,  $p_i = a_i$ 로 지정함
  - $K$ 와  $P$  모두  $s$ 로 지정함
  - 만약, 0/1 Knapsack의 답이 Yes라면, 총 크기는  $K (= s)$ 를 넘지 않고, 이익은  $P (= s)$  이상이 되도록 아이템을 선택할 수 있다는 뜻이다. 여기서,  $s_i = a_i$ 와,  $p_i = a_i$ 라는 사실과  $K = P = s$ 라는 사실이 중요하다. 이는 선택된 아이템의 크기의 합 (= 선택된  $a_i$ 의 합)  $\leq K = s$ 이면서 동시에 이익의 합 (= 선택된  $a_i$ 의 합)  $\geq P = s$ 가 된다. 이 두 부등식이 성립하려면 선택된  $a_i$ 의 합이 정확히  $s$ 가 되어야 함을 의미한다!
  - 만약, 0/1 Knapsack 답이 No라면, 크기 합이  $> K = s$ 이거나, 이익의 합이  $< P = s$ 인 경우이다. 이 경우는 당연히 선택된  $a_i$ 의 합이  $s$ 가 되지 않는 경우이다

## 6. NP-complete 문제 공략법

- a. 다행 시간 알고리즘이 존재하지 않을 가능성이 매우 높지만, 그렇다고 아무 노력도 하지 않고 포기할 수는…… 없다
- b. 욕심을 좀 줄이면 몇 가지 방법으로 공략이 가능하다
  - i. **입력/문제 조건 제한 방법**: 일반적인 입력이 아닌 제한된 입력에 대해 다행 시간 알고리즘이 존재할 수도 있다. 예를 들어, 해밀턴 사이클 문제에서는 충분히 많은 에지를 포함한 그래프가 입력인 경우에는 Yes일 가능성이 매우 높다. 또는 문제의 조건을 완화하는 방법도 있다. 예를 들어, fractional Knapsack 문제는 아이템을 나눌 수 있는 조건을 추가하여 다행 시간에 해결할 수 있다
  - ii. **근사 방법 (approximation)**: 정확한 해를 구하는 대신 해에 되도록 가까운 해 (근사해)를 계산하는 방법. 해에 얼마나 가까운지를 수치로 증명한다는 점이 까다로움
  - iii. **랜덤 방법 (randomization)**: 정확한 해를 구하는 알고리즘의 수행 시간은 최악의 입력을 고려해 분석하지만, 입력이 랜덤하게 주어지고 알고리즘에서의 선택도 랜덤하게 결정한다면 수행 시간을 평균적으로 분석할 수 있다. 크게 Monte Carlo 방법과 Las Vegas 방법으로 나눈다
  - iv. **휴리스틱 (heuristic)**: 정답을 얻을 수도 그렇지 않을 수도 있는 방법. 근사 방법과의 차이점은 정답과 얼마나 가까운지 수치로 증명되지 않았다는 것이다
  - v. 이 중에서 근사 방법과 랜덤 방법을 간단한 예를 들어 맛보기로 살펴보자

## 7. 랜덤 방법 (Randomization)

- a. quick select와 quick sort의 피봇을 랜덤하게 선택한다면, 나눠지는 두 부분의 크기가 평균적으로 전체의 반에 가까울 것이다. 이러한 방법이 randomness를 알고리즘에 이용하는 것이 랜덤 방법이다. 같은 입력을 여러 번 수행해도 피봇의 선택이 달라지기 때문에 수행시간이 달라짐에 유의하자
- b. 예: Finding maximum problem

```
def find_max(A, n):
 randomly shuffle A (*)
 current_max = A[0]
 for i in range(1, n):
 if current_max < A[i]:
 current_max = A[i] (**)
 return current_max
```

- i. (\*) 문에서 *A*의 원소를 임의로 섞는다
  - [?] 이 문장은 어떻게 구현할 수 있을까?
- ii. 질문: (\*\*) 문 (*current\_max* 업데이트 횟수)의 평균 횟수는 몇 번일까?

- 만약, A가 오름차순으로 정렬되었다면, 매번 업데이트 되고, 반대로 내림차순으로 정렬되었다면, 전혀 업데이트 되지 않는다. 모든 경우에 대한 업데이트의 평균 회수는 당연히 0과 n 사이의 값이 된다
  - `current_max < A[i]`가 참이 되어야 업데이트가 실행된다. 참이 되는 것은  $A[0], A[1], \dots, A[i]$ 까지 최대 값이  $A[i]$ 라는 의미이다. 그럼 그 확률은 얼마인가? [힌트: A의 값을 임의로 섞기 때문에  $A[0], A[1], \dots, A[i]$  중에서 어떤 원소가 최대 값이 될 확률은 모두 같을 수 밖에 없다]
- 
- 평균 횟수는 모든 i에 대해  $A[i]$ 가 최대 값이 될 확률을 모두 더하면 된다 (왜?)
  - 최종 평균 횟수는?

## 8. 근사 방법 (Approximation)

- 정답의 성능과 차이 또는 비가 어떤 입력에 대해서도 특정 값 이내로 제한됨을 수치적으로 증명해야 한다
- 예를 들어, TSP 문제의 경우는 가중치가 최소인 모든 노드를 지나는 해밀턴 사이클을 찾는 문제이다. 주어진 입력 그래프 G에 대한 정답 값이  $\text{opt}(G)$ 라고 하고 근사 알고리즘의 출력 값이  $\text{alg}(G)$ 라고 하자. 그러면 근사도 (approximate ratio) c는 아래와 같이 정의된다

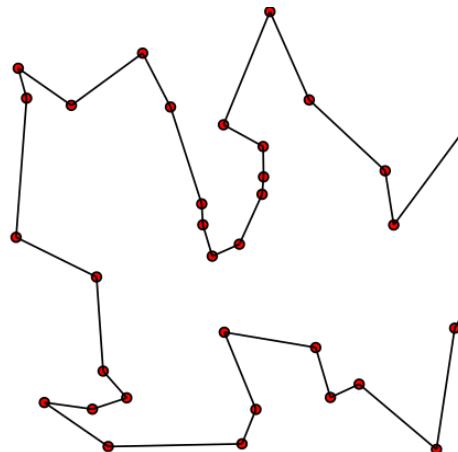
$$C = \max_{\forall \text{graph } G} \frac{\text{alg}(G)}{\text{opt}(G)}$$

- $\text{alg}(G)$ 가 정해라면  $c = 1$ 이 되기 때문에 일반적으로  $c > 1$ 이다
  - $c = 2$ 라는 것은 어떤 그래프에 대해서도 근사 알고리즘이 계산한 해밀턴 사이클의 가중치가 정답의 두 배보다 크지 않다는 의미이다. 즉, '성능'이 2배보다 나빠지지 않는다는 것이다
  - 당연히 c의 값이 1에 가까울수록 좋은 근사 알고리즘이다
- 최대화하는 문제인 경우에는 반대로 c의 값이 1 이하가 된다
  - 예 1: 0/1 Knapsack 문제에 대한 근사 해를 생각해보자
    - NP-complete 문제이므로 다행 시간 알고리즘이 현재 없다
    - 근사 방법 1: 이익이 높은 아이템부터 선택하는 그리디 방법
      - [?] 이 방법의 결과가 정답과 얼마나 차이가 날 수 있을까? (답: 매우 나쁠 수 있다. 얼마나 안 좋을까?)
    - 근사 방법 2: 크기당 이익이 높은 아이템부터 선택하는 그리디 방법
      - [?] 이 방법의 결과가 얼마나 안 좋을 수 있을까?
    - 근사 방법 3: 근사 방법 1과 2 중에서 더 좋은 결과를 취하는 방법

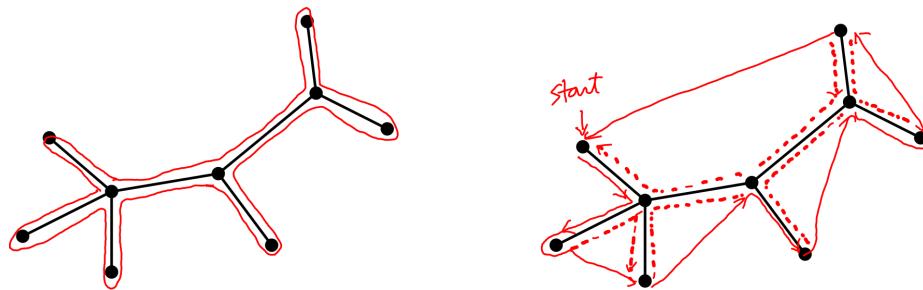
- 즉, 근사 방법 1의 결과 이익이 A, 근사 방법 2의 결과 이익이 B라고 하면,  $\max(A, B)$ 가 근사 방법 3의 결과 이익이 된다.
- 이 방법의 결과의 근사도는  $1/2$ 이다. 즉, 정답 이익의  $\frac{1}{2}$  이상은 항상 배낭에 넣을 수 있음을 뜻한다
- 정답의 이익은  $\text{opt}$ 라고 하면,  $\max(A, B)/\text{opt} \geq \frac{1}{2}$ 를 증명하고 싶다
- 주장:**  $A + B \geq \text{opt}$ 가 성립한다.  
[?] 증명 힌트 - B는 가성비 순서로 선택하는 데, 처음으로 배낭에 넣지 못하게 되는 아이템의 가치는 항상 A보다 작아야 한다 (왜?) 이 사실을 이용해 보자
- $A + B \geq \text{opt}$  이므로  $\max(A, B) \geq \text{opt}/2$  가 되어 증명 끝!

e. 예 2: Euclidean TSP 문제에 대한 근사 해를 생각해보자

- Euclidean TSP는 TSP 문제의 기하 버전이다. 2차원 평면에  $n$ 개의 도시(점, 노드)가 주어져 있고, 두 도시 사이의 거리는 Euclidean 거리 (좌표 값의 차의 제곱을 더 한 후 제곱근을 취한 값으로 실생활에서의 거리 기준)가 된다. 모든 도시를 정확히 한 번씩 방문한 후 시작 도시로 돌아오는 사이클 중에서 이동한 거리의 합이 제일 작은 사이클을 찾는 문제가 Euclidean TSP 문제이다 [아래 그림은 빨간색 입력 점들에 대한 Euclidean 사이클을 나타낸다. 출처: wikipedia]



- 이 문제 역시 그래프에서의 TSP 문제로 쉽게 변환할 수 있으므로 본질적으로 같은 문제이다. 따라서 NP-complete이다

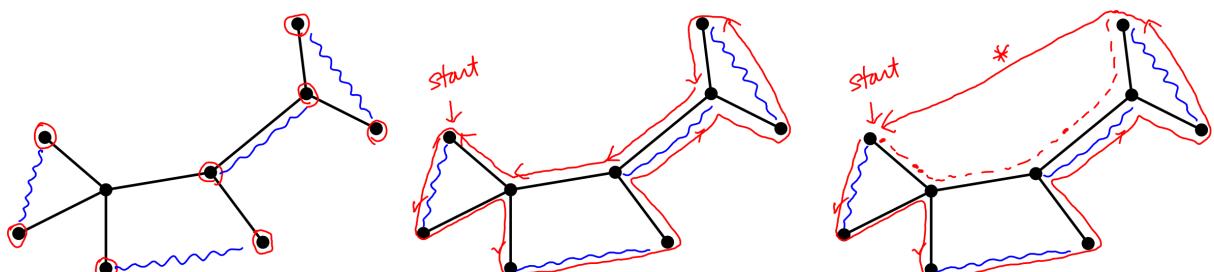


### iii. 근사 알고리즘 1: MST + edge doubling (근사도 2인 근사 알고리즘)

- MST는 n개의 점을 연결하는 최소 개수의 에지만을 갖는 그래프 (트리) 중에서 최소 거리의 그래프이다. MST의 각 에지를 위의 그림처럼 중복하면 일종의 Euler trail이 만들어진다
- $|MST|$ 를 MST의 에지의 거리 합이라 하고,  $|TSP|$ 는 정답 사이클의 거리 합이라 하자 (여기에서는  $opt = |TSP|$ )
- 주장 1:**  $|MST| \leq |TSP|$   
**증명:** [ ? ]
- MST의 에지를 중복해서 구성한 Euler trail을 각 노드가 한 번씩만 지나도록 수정해 보자: 임의의 리프 노드부터 출발해 DFS 방식으로 차례로 노드를 방문하면서 사이클을 구성한다. 이미 방문한 곳을 방문하면 그 다음에 처음으로 방문하는 노드로 직접 연결하는 식으로 구성한다
- 이 사이클은 에지끼리 교차할 수도 있지만 각 노드를 정확히 한번씩만 방문하게 된다. 이 사이클을 C라고 하자
- 주장 2:**  $|C| \leq 2|MST|$   
**증명:** [ ? ] (힌트: triangle inequality)
- 사실 1:**  $|C| \leq 2|TSP| \rightarrow C = \text{근사도가 } 2\text{인 TSP 사이클}$   
증명: 주장 1과 주장 2를 이용하면 자연스럽게 성립한다!

### iv. 근사 알고리즘 2: MST + matching (근사도 1.5인 근사 알고리즘)

- 근사 알고리즘의 1의 근사도는 2이다. 즉, TSP 사이클 길이의 2배를 넘지는 않는 근사 해를 찾을 수 있다는 뜻이다
- 이 근사도를 1.5로 낮춰보자
- MST의 노드의 분지수(degree)를 보면, 노드의 분지수가 홀수인 노드는 항상 짹수 개임을 알 수 있다 (증명은?)
- 분지수가 홀수인 노드를 모아 집합 O라 하자. O의 노드에 대한 **최소 거리 합의 완전 매칭** (perfect matching)을 구해보자
  - 매칭 (matching)이란 두 점씩 짹을 맺는 것을 말하고, 완전 매칭이란 모든 점들이 빠짐 없이 짹을 맺는 것을 의미한다
  - O의 노드가 짹수개이니 완전 매칭이 가능하다
  - 최소 거리 합의 완전 매칭이란 완전 매칭 중에서도 짹을 맺는 두 노드 사이의 거리의 합이 최소가 되는 것이다



*~~~ : perfect matching*

(그림: 왼쪽의 파란색 선은 매칭, 가운데 그림의 빨간색은 MST+M의 에지를 모두 지나는 Euler trail, 오른쪽 그림의 빨간색 선은 trail을 cycle로 변환한 것. \* 표시 에지가 이미 방문한 노드를 건너 뛰고 직접 연결한 에지임)

- d. 이러한 매칭은 다행 시간에 계산할 수 있다 [매칭 알고리즘을 인터넷에서 검색해서 조사해보기]
  - e. 여기서 구한 매칭을  $M$ 이라 하자
  - **주장 3:**  $|M| \leq |TSP|/2$
- 증명:** 짹수 분지수를 갖는 노드 집합을  $E$ 라 하면, 전체 노드 집합은  $E \cup O$ 가 된다. 그러면 triangle inequality에 의해,  $|TSP(O)| \leq |TSP(E \cup O)| = |TSP|$ 이 성립한다. (구체적으로?) 사이클  $TSP(O)$ 의 에지를 짹수번째 에지들과 홀수번째 에지들로 나눠 생각해보자. 이 두 그룹의 에지들은 홀수 분지수 노드에 대한 완전 매칭이다. 따라서 각 그룹은 최소 거리 매칭보다 거리 합이 작을 수는 없다.  $2|M| \leq |TSP(O)|$ 이 된다.  $2|M| \leq |TSP(O)| \leq |TSP|$ 가 되어  $|M| \leq |TSP|/2$ 이 성립한다 (증명 끝)
- MST + M 역시 그래프이다. 이 그래프에서 노드의 분지수는 모두 짹수가 된다. (당연하지만 왜?)
  - 노드 분지수가 모두 짹수이므로 MST와  $M$ 의 모든 에지를 정확히 한 번씩 지나는 Euler trail을 다행 시간에 만들 수 있다. 위 그림을 참조해보자
  - **주장 4:**  $|\text{Euler trail}| \leq 1.5|TSP|$   
**증명:**  $|MST| \leq |TSP|$ 이고  $|M| \leq 0.5|TSP|$ 를 이용하면,  $|\text{Euler trail}| = |MST| + |M| \leq |TSP| + 0.5|TSP| \leq 1.5|TSP|$ 가 된다
  - Euler trail은 한 노드를 두 번 이상 지날 수 있으므로, 앞에서 했던 방식으로 trail의 노드를 차례로 방문하면서 이미 방문했던 노드를 만나면 다음 번 처음으로 미방문 노드로 직접 연결한다. 이렇게 수정된 trail은 모든 노드를 한 번씩 방문하는 사이클이 된다. 이 사이클을  $C$ 라고 하자. 역시 triangle inequality에 의해,  $|C| \leq |\text{Euler trail}|$ 이 된다. 따라서  $|C| \leq |\text{Euler trail}| \leq 1.5|TSP|$ 가 되어, 근사도가 1.5인 사이클을 다행 시간에 얻을 수 있다
  - **사실 2:**  $|C| \leq 1.5|TSP| \rightarrow C = \text{근사도 } 1.5\text{인 사이클}$

- a. 이 근사 알고리즘은 1976년에 Chirstofides에 의해 제안
- v. 근사 알고리즘 1과 2를 통해 근사도를 2에서 1.5로 줄일 수 있었다. 이렇게 조금씩 줄이면 1에 매우 가깝게 줄일 수 있지 않을까? 그렇다! 가장 좋은 근사 알고리즘은 1에 원하는 만큼 가까운 근사도를 갖는 TSP 사이클을 찾을 수 있다 (단, 1에 가깝다는 거지 1이 아니므로 정답을 찾지는 못한다)
- Sanjeev Arora에 의해 1998년에 근사도가  $(1+\varepsilon)$ 인 근사 알고리즘이 제안되었다 (여기서  $\varepsilon$ 은 0과 1사이의 임의의 실수이다) 대신 근사 알고리즘의 수행시간이  $\varepsilon$ 에 지수적으로 증가한다

## 14. 알아두면 유용한 기법

### 1. 비트 연산 활용하기 (Bit Manipulation)

- a. 비트 연산자 & | ^ ~ << >> 는 단위 시간에 수행되는 기본 연산자라고 가정한다
  - i. 비트 연산자는 + - \* / 등의 산술 연산자보다 일반적으로 빠르다
- b. 주의할 점: 음수에 대해 >>, <<와 ~ 연산
  - i. 음수 >> k 를 하면 음수 값을  $2^k$ 로 정수 나눗셈을 한 것과 동일하고, 음수 << k 를 하면 역시 음수 값에  $2^k$ 를 곱한 것과 동일한 결과가 나옴!
    - 1. 음수의 MSB의 값 1은 계속 유지됨 → arithmetic shift 연산자
  - ii. ~ 연산을 음수에 적용하면 MSB의 값 1이 0으로 바뀌어 양수가 됨!
- c. 어떤 정수 값의 LSB는  $2^0$  비트로 가장 오른쪽 비트를 의미하며, MSB는 가장 왼쪽 비트를 나타낸다 (LSB의 bit\_index = 0이라고 가정)
- d. Bit mask: 특정 비트 값을 알아내거나 수정하는 작업

- i. get 함수: 특정 bit index의 값을 구하는 연산

```
def get_bit(value, bit_index): # value의 bit_index의 값
 return value & (1 << bit_index)
print(get_bit(0b101101, 3)) # 3번째 비트 값 $2^3 = 8$

def get_normalized_bit(value, bit_index): # 비트 자체 (0/1)
 return (value >> bit_index) & 1
print(get_normalized_bit(0b101101, 3)) # 3번째 비트는 1
```

- ii. set 함수: 특정 bit index의 값을 1로 set하거나 0으로 unset하는 함수

```
def set_bit(value, bit_index):
 return value | (1 << bit_index)
print(set_bit(0b101101, 4)) # 0b111101

def unset_bit(value, bit_index):
 return value & ~(1 << bit_index)

def toggle_bit(value, bit_index): # bit_index 값을 반전
 return value ^ (1 << bit_index)
```

- e. 예제 문제 1: n의 이진수 비트수를 세기

```
def bits_count(n): # assume n > 0
 cnt = 0
 while n > 0:
 cnt += 1
 n = n >> 1 # or n = n // 2
return cnt
```

- n의 비트를 오른쪽으로 한 비트씩 이동하면서 세는 방식. n이 양수라면 모든 비트가 이동하게 되면 0이 되기에 0이 되기 전까지 반복
- [?] 위의 함수의 n이 만약 음수라면, 제대로 동작하는가?

f. 예제 문제 2: n의 이진수의 1의 개수 세기 (n을 양수라고 가정)

i.  $O(\log n)$  시간 방법

```
def one_bits_count1(n):
 cnt = 0
 while n > 0:
 if n % 2 == 1:
 cnt += 1
 n = n >> 1
 return cnt
```

n의 각 비트를 보면서 1이면 cnt 값을 증가시키는 방식으로 비트수만큼의 시간이 필요 →  $O(\log n)$  시간

```
def one_bits_count2(n):
 cnt = 0
 while n > 0:
 cnt += n & 1 # if 문을 쓰지 않는 방법
 n = n >> 1
 return cnt
```

ii.  $O(\log k)$  시간 방법 ( $k = n$ 의 1의 개수)

```
def one_bits_count3(n):
 cnt = 0
 while n > 0:
 cnt += 1
 n = n & (n-1)
 return cnt
```

iii.  $n = n \& (n-1)$ 은 어떤 효과가 있나?

1. 예를 들어,  $n = 1001\ 1000$  인 경우,  $n-1$ 은  $1001\ 0111$ 이 된다. 즉, 가장 오른쪽 1과 1의 오른쪽에 있는 0 비트들은 ... $100\dots00$  형식이고 여기서 1을 빼면 ... $011\dots11$ 이 된다

$$\begin{array}{r} \dots 100\dots00 \\ & \& \dots 011\dots11 \\ \hline \dots 000\dots00 \end{array}$$

2.  $n \& (n-1)$ 을 하면 가장 오른쪽 1부터 오른쪽 끝까지는 모두 0이 되고 왼쪽 비트들은 변하지 않는다. 결국 가장 오른쪽 1만 0이 된다
3. 이 과정을  $n$ 이 0이 될 때까지 반복하면 충분하다. 따라서 루프는 1의 개수를  $k$ 개라고 하면  $O(\log k)$ 번 만큼 반복된다  $\rightarrow O(\log k)$  시간
- iv. [?] 두 이진수 A와 B가 주어진다. A의 비트를 적절히 변경해서 B와 일치하도록 하고 싶다. 몇 번의 비트 변경을 해야 하나? (힌트: XOR와 1의 개수 세기 이용)
- g. 예제 문제 3:  $n$ 의 가장 오른쪽 1의 비트 값 계산하거나 clear하기 ( $n$ 을 양수라고 가정)
- $n = 0101\ 1000$ 이라면, 가장 오른쪽 1의 bit index는 3이다. 비트의 자리 값은  $2^3 = 8$ 이다. 가장 오른쪽 1의 자리 값을 계산하는 문제이다
  - 가장 오른쪽 1의 자리 값을 추출하기 위한 트릭은?  $n \& -n$ 

$$\begin{array}{r} n = 0101\ 1000 \\ -n = 1010\ 1000 \quad (1\text{의 보수} + 1 = 1010\ 0111 + 1 = 2\text{의 보수}) \\ & \hline \\ & = 0000\ 1000 = 2^3 = 8 \end{array}$$
  - 수행시간:  $O(1)$
  - 응용: Bit Indexed Tree (또는 Fenwick 트리)에서 이용된다. Fenwick 트리는 자료구조 교재에서 설명되어 있다. Fenwick 트리에서는 특정 값을 여러 수의 합으로 표현해야 한다. 예를 들어,

$$0101\ 1000 = 0100\ 0000 + 0001\ 0000 + 0000\ 1000$$

으로 비트 1이 3번 등장하므로 3개의 수의 합으로 표현 가능하다. 이 세 이진수는 결국 가장 오른쪽 1의 비트 자리 값을 반복해서 계산하는 식으로 알 수 있다.

- h. 예제 문제 4:  $n$ 보다 같거나 큰 수 중에서 가장 작은  $2^k$  계산하기
- $n$ 을 이진수로 표현했을 때, 비트 수가  $x$ 개라면,
    - $n = 2^x$ 인 경우엔  $k = x$ 가 되지만
    - $n \neq 2^x$ 인 경우엔  $k = x+1$  된다
    - $n = 2^x$ 인 경우는 어떻게 판별할 수 있을까?
      - $n \& \sim(n \& (n-1))$
      - 예:  $n = 1000$ 이라면  $n \& (n-1)$ 은 가장 오른쪽 1을 clear하는 연산이므로  $n \& (n-1) = 0000$ 이 된다. 즉, 모두 비트가 0이 됨
      - 여기에  $\sim$ 연산을 하면 모두 1이 됨
      - MSB 비트만 추출해야 하기에,  $n=1000 \& 1111$ 을 해서  $2^x = 2^3$ 을 계산하면 됨
    - 그렇지 않은 경우엔  $n$ 의 비트수를 예제 1처럼 반복문을 이용해 세면 된다

```

def get_smallest_power_of_two(n):
 if n & (n&(n-1)) == 1: # n = 2k
 return n
 cnt = 0
 while n > 0:
 n = n >> 1
 cnt += 1
 return 1 << cnt

```

i. 예제 문제 5: 동전을 이용해 주사위 눈의 값을 랜덤하게 생성하기

- i. 앞-뒤가 나올 확률이 각각 1/2인 동전 하나를 이용해, 1부터 6까지의 수(주사위 눈)을 랜덤하게 생성하고 싶다. 즉, 1부터 6까지 나올 확률이 1/6이어야 한다는 의미이다 → **코딩 인터뷰 문제로 자주 언급**
- ii. 1부터 6까지 수 대신 0부터 5까지의 수로 대응해도 문제없다. 이 수들은 3 비트로 충분히 표현할 수 있다
- iii. LSB부터 시작해서 각 비트의 값 (0 또는 1)을 동전을 던져 결정한다. MSB를 던진 후의 최종 값이 [0,5] 범위면 리턴하고, 범위를 벗어나면 다시 처음부터 LSB부터 값을 정하는 과정을 반복한다

```

import random
def generate_random_dice():
 while True:
 result, i = 0, 0
 while (1 << i) < 8: # 3번만 반복 - 5 < 2^i < 8
 result = (result << 1) | random.randint(0,1)
 i += 1
 if result < 6:
 break
 return result + 1

```

- iv. 위 코드는 세 비트를 결정해서 그 값이 0부터 5까지의 값이 나오면 1을 더해 리턴한다. 나온 값이 6, 7이 나오면 바깥 while 루프를 계속 반복한다
- v. 수행시간은 루프의 반복횟수에 의해 결정된다. 반복횟수가 일정하게 정해지지 않고, 오직 기대값(expected number of loops)으로만 표현된다
  1. 6, 7이 나올 확률은  $2/8 = 1/4$ 임
  2. 첫 루프에서 결정못할 확률  $\frac{1}{4}$ , 두 번의 루프 모두에서 결정하지 못할 확률  $(\frac{1}{4})^2$ , … k번의 루프 모두에서 결정하지 못할 확률  $(\frac{1}{4})^k$ . 따라서,
$$\text{기대값} = 1(\frac{1}{4}) + 2(\frac{1}{4})^2 + \dots + k(\frac{1}{4})^k + \dots \text{ (멱급수!)}$$
  3. 위의 멱급수를 계산하면  $4/9$ 가 되어 1이 되지 않는다. 결국 평균적으로 한 번 정도만 루프를 수행하면 주사위 눈을 얻을 수 있다

- vi. 이 방법을 일반화해서 동전 하나를 여러 번 던져  $[a, b]$  구간의 랜덤 정수 값을 생성해보자
1.  $[a, b]$  범위 대신  $[0, b-a]$  범위에서 선택하는 것으로 바꿈
  2.  $b-a$ 보다 크면서 가장 작은  $2^t$ 의  $t$ 를 결정한다
  3.  $t$ 개의 비트의 값 0과 1을 LSB부터 차례대로 랜덤하게 (동전을 던져) 결정한다
  4. 결정한 값이  $[0, b-a]$ 에 포함되면 리턴하고 아니면 다시 반복한다
  5. 기대값 분석:
    - a. 성공할 확률은  $(b-a+1)/2^t$ 가 된다.  $b-a+1 \geq 2^{t-1}$ 이므로 성공 확률  $\geq 2^{t-1}/2^t = \frac{1}{2}$ . 반대로 실패할 확률은  $< \frac{1}{2}$ 이다
    - b. 루프를  $k = 1, 2, \dots$  번 반복하기에 기대값은
$$\text{기대값} = 1(\frac{1}{2}) + 2(\frac{1}{2})^2 + \dots + k(\frac{1}{2})^k + \dots$$
    - c. 이 값은 2를 넘지 않는다. 결국 평균 2번 이내로 루프가 반복한다
  6. 평균 2번 이내로 루프를 반복하고 루프 내에선  $t$ 개의 비트를 결정하기 때문에 총 시간은  $2t$ 를 넘지 않는다. 따라서 시간  $O(t) = O(\log(b-a))$

2. Coming soon in the next edition…

