알고리즘2 (2024-2)

1. 코딩 테스트 개요와 효율적인 알고리즘 구현

국립금오공과대학교 컴퓨터공학과

김 경 수

목차

- 코딩 테스트 개요
- 시간 복잡도 분석 리뷰
- 알고리즘의 효율적 구현
- 실전 문제 풀이

학습 목표

① 코딩 테스트의 정의와 특성을 파악하고 효과적인 코딩 테스트 대비를 위해 자료 구조와 알고리즘 지식이 필수적임을 이해할 수 있다.

② 코딩 테스트 문제의 출제 형태를 파악하고 이해할 수 있다.

③ 알고리즘의 시간 복잡도에 따른 계산량을 비교하고 이를 설명할 수 있다.

④ 알고리즘의 효율성을 빠르게 분석하는 방법을 이해하고 이를 실제 구현된 코드에 적용하여 효율성을 분석할 수 있다.

⑤ 간단한 코딩 테스트 문제 풀이를 통해 효율적인 알고리즘 적용의 필요성을 이해하고, 실제 구현 방법을 체득할 수 있다.

코딩 테스트 개요

코딩 테스트 개요

- 코딩 테스트는 코딩 능력을 평가하는 것이 아니라, **문제 분석 및 해결 능력**을 평가하는 것이 주 목표이다.
- 따라서, 주어진 문제를 정확하게 분석하는 과정이 제일 중요하다(※ 정보 올림피아드와 유사)
- 또한, 코딩 테스트는 명백히 "**시험(test)**"임을 인지한다.
  - ➢ 코딩 테스트 준비에서 벼락치기, 단기 완성 등은 통하지 않음
  - ➢ 코딩 테스트를 공부할 때는 실전처럼 제한 시간을 정해 놓고 문제를 푸는 연습을 충분히 수행해야 함
  - ➢ 이외로 문제를 분석하는 데 시간이 많이 소요되므로, 이를 감안하여 시간을 배분하는 연습이 필요함
- 내가 어떻게 문제를 풀었는지 다른 사람에게 말로 설명할 수 있다면, 해당 문제에 대해 비로소 정확하게 이해한 것이다. → 팀(team)을 구성하여 함께 대비하면 더욱 유리

문제를 효과적으로 분석하는 방법

① **문제를 쪼개서 분석할 것**
- ➢ 주어진 문제를 동작 단위로 쪼개서 분석하는 것이 유리함

② **제약 사항을 파악하고 테스트 케이스를 추가할 것**
- ➢ 문제에 주어지는 테스트 케이스는 발생 가능한 오류/예외를 가정하지 않음
- ➢ 따라서, 발생 가능한 예외나 오류를 가정한 나만의 테스트 케이스를 만들어서 활용하는 것도 중요함

③ **입력값을 분석할 것**
- ➢ 알고리즘의 시간 복잡도에 가장 큰 영향을 주는 요인 중 하나가 바로 입력의 크기
- ➢ 만약 입력의 크기가 100만이라면 $O(n^2)$ 알고리즘은 채택하기 어려움

④ **문제에 제시된 핵심 키워드를 파악할 것**
- ➢ 핵심 키워드에 따라 어떤 알고리즘 패러다임을 적용할 지 결정하는 데 용이함
- ➢ 핵심 키워드가 잘 판단되지 않는다면 아래 ⑤와 연계하여 문제를 그림으로 그려볼 것

⑤ **데이터의 흐름과 구성을 파악**
- ➢ 만약 데이터의 삽입과 삭제가 빈번하다면 힙(heap)의 사용을 적극 고려할 수 있음

핵심 키워드에 따른 알고리즘 선택 방법

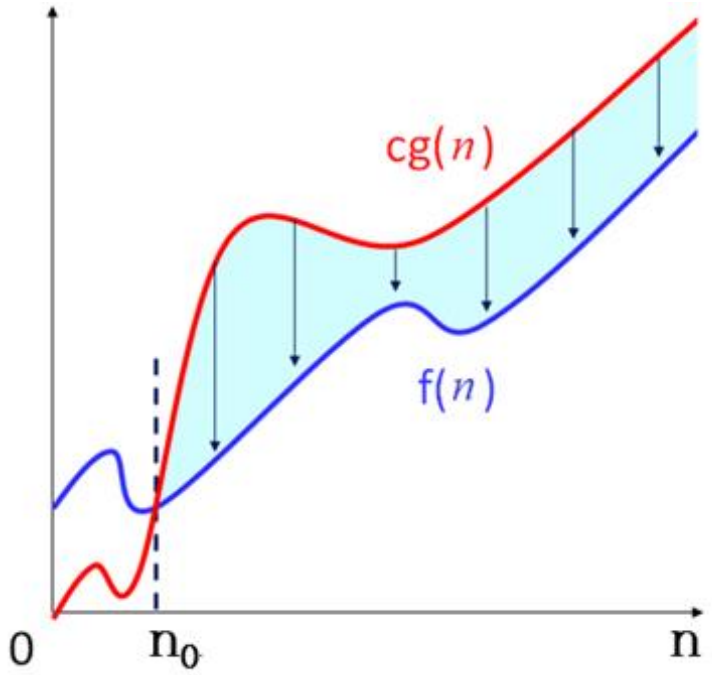
주요 알고리즘	문제에 제시된 키워드	상황
스택	쌍이 맞는지, 최근	• 무언가를 저장하고 반대로 처리할 때 • 데이터의 조합이 균형을 이뤄야 할 때 • 알고리즘이 재귀적 특성을 가질 때 • 최근 상태 추적
큐	순서대로, ~대로 동작하는 경우, 스케줄링, 최소 시간	• 특정 조건에 따라 시뮬레이션할 때 • 시작 지점부터 목표 지점까지 최단 거리
깊이 우선 탐색	모든 경로	• 메모리 사용량이 제한적일 때의 탐색 • 백트래킹 문제를 해결할 때
너비 우선 탐색	최적, 레벨 순회, 최소 단계, 네트워크 전파	• 시작 지점부터 최단 경로나 최소 횟수를 찾아야 할 때
백트래킹	조합, 순열, 부분 집합	• 조합 및 순열 문제 해결 시 • 특정 조건을 만족하는 부분 집합
최단 경로	최단 경로, 최소 시간, 최소 비용, 트래픽, 음의 순환, 단일 출발점 경로	• 다익스트라: 특정 정점 → 나머지 모든 정점 • 벨만-포드: 음의 순환 탐지, 음의 가중치를 가진 그래프에서 최단 경로 탐색 시 활용

시간 복잡도 분석 리뷰

시간 복잡도 리뷰

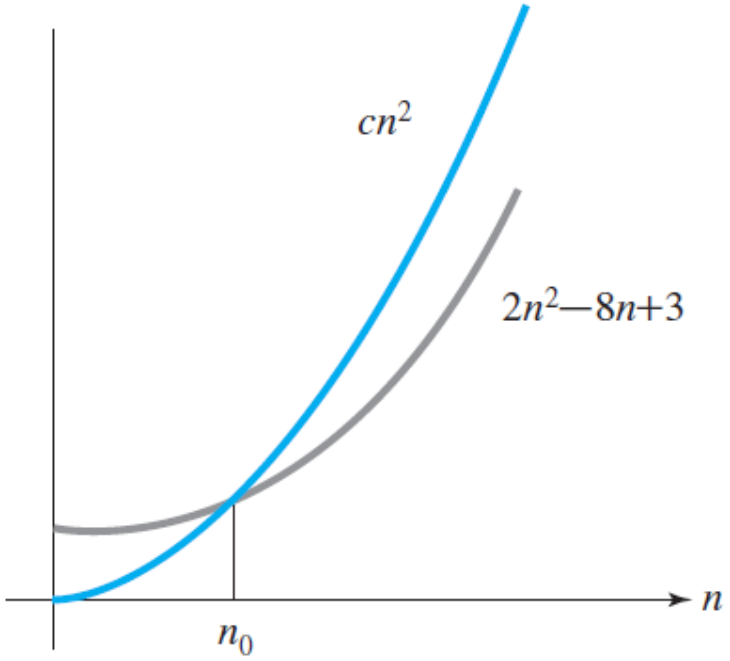
- 시간 복잡도는 **입력 크기에 대한 함수**로 표기된다.
  - 이 때, 시간 복잡도 함수는 여러 개의 항을 가지는 다항식으로 표현되며, 이를 입력의 크기에 대한 함수로 표현하기 위해 **점근적 표기 (Asymptotic Notation)**를 사용한다.

- **O-표기법 (Big-Oh notation)**
  - 모든 $n \geq n_0$에 대해서 $f(n) \leq cg(n)$이 성립하는 양의 상수 c와 n_0가 존재하면, $f(n) = O(g(n))$이다.
  - $f(n) = O(g(n))$이면, ***g(n)*은 *f(n)*의 상한(Upper Bound)**이다.



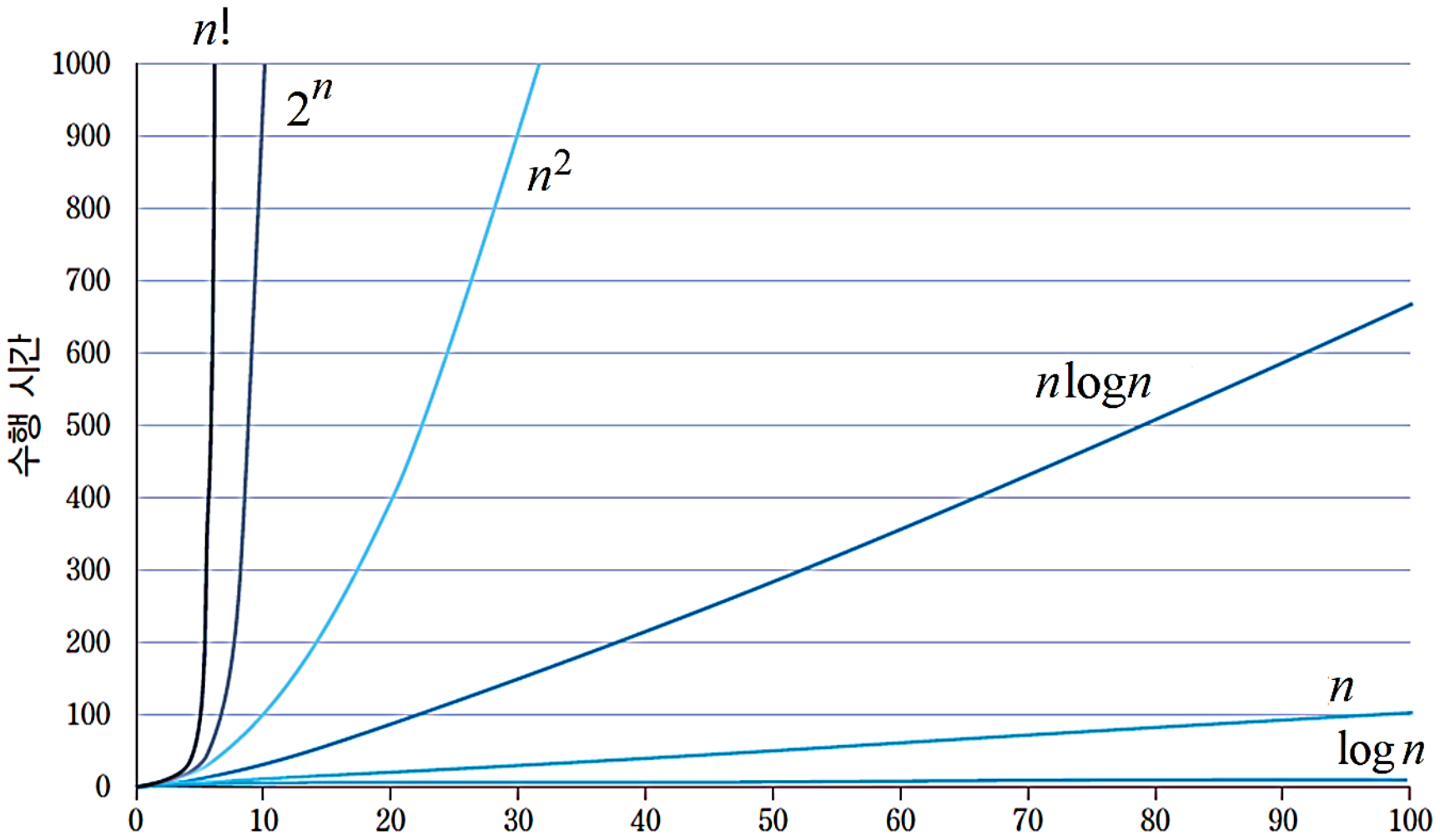
시간 복잡도 리뷰

- **예제. $f(n) = 2n^2 - 8n + 3$일 때 시간 복잡도**
  - $f(n)$의 O-표기는 **$O(n^2)$,** 단순화된 표현은 n^2
  - 즉, 단순화된 함수 n^2에 임의의 상수 c를 곱한 cn^2이 n이 증가함에 따라 $f(n)$의 상한이 된다. (※ 단, $c > 0$)



  - 예를 들어 c=5일 때, $f(n) = 2n^2 - 8n + 3$과 $5n^2$과의 교차점(n_0=1/3)이 생기는데, 이 교차점 이후 모든 n에 대해, 즉 n이 무한대로 증가할 때, $f(n) = 2n^2 - 8n + 3$은 $5n^2$보다 절대로 커질 수 없다.
  - 따라서 $O(n^2)$이 $2n^2 - 8n + 3$의 **점근적 상한**이 된다.

주요 시간 복잡도 비교



O(1) 상수 복잡도 (Constant complexity)

O(log*n*) 로그 복잡도 (Logarithmic complexity)

O(*n*) 선형 복잡도 (Linear complexity)

O(*n*log*n*) 로그 선형 복잡도 (Log-linear complexity)

O(n^2) 이차 복잡도 (Quadratic complexity)

O(n^3) 3차 복잡도 (Cubic complexity)

O(n^k) 다항 복잡도 (※ 단, k는 상수) (Polynomial complexity)

O(2^n) 지수 복잡도 (Exponential complexity)

시간 복잡도를 빠르게 확인하는 방법

- **가장 먼저 확인해야 할 사항**
  - ✓ 반복문과 재귀 호출의 사용 빈도 수
  - ✓ 입력의 크기에 따른 메인 메모리 사용의 허용 여부
  - ✓ 어떤 알고리즘 메커니즘을 적용하였는가?
  - ✓ 특히 "재귀식을 포함한 형태"로 구현되어 있고, 점화식을 도출할 수 있다면 마스터 정리(Master's Theorem)를 적용하면 보다 쉽게 정확한 시간 복잡도를 구할 수 있음.

시간 복잡도를 빠르게 확인하는 방법

- **적용된 알고리즘 메커니즘별로 집중 체크해야 할 부분**
  - ✓ 선형 구조 기반 ➔ 입력 리스트의 크기, 리스트의 탐색 횟수, 반복문 … 등
  - ✓ 분할 정복 메커니즘 기반 ➔ 문제 분할 및 재귀 호출 구조, 재귀 호출 횟수, 점화식 … 등
  - ✓ 동적 계획법 기반 ➔ 사용된 테이블의 크기, 반복문의 중첩, 점화식 … 등
  - ✓ 상태 공간 탐색 기반 ➔ 상태 공간 트리의 높이, 트리의 반복 탐색 횟수, 재귀 호출 횟수, 가지치기(pruning) 횟수 … 등

시간 복잡도를 빠르게 확인하는 방법

- **(예제 1)** 아래 코드의 시간 복잡도를 분석하시오.

```
// Input: List A[]; Value
// Output: Found value in the list A[]
  low = 0; high = n – 1;  n = length(A);
  while ( low <= high )
        mid = ( low + high ) / 2;
        if ( A[mid] > value ) high = mid – 1;
        else if ( A[mid] < value ) low = mid + 1;
        else return mid;
  return -1; /* Failure */
```

- 단일 반복문이되, 총 반복 횟수는 n회가 아님에 유의.
- 반복문이 1회 수행될 때 마다 탐색되는 리스트의 범위가 절반으로 감소하고 있음.
- 리스트의 길이가 1이하가 되면 종료됨(low <= high)
- 반복문 1회 수행 당 탐색되는 리스트의 길이를 나타내면, $n \rightarrow n/2 \rightarrow n/4 \rightarrow n/8 \rightarrow n/16 \rightarrow \dots \rightarrow n/2^n$ 임.
- 따라서, 총 시간 복잡도는 $\boldsymbol{O(\log n)}$.

시간 복잡도를 빠르게 확인하는 방법

- **(예제 2)** 아래 코드의 시간 복잡도를 분석하시오.

```
/* Input: 가중치 그래프 G=(V,E); 시작 정점 s */
/* Output: 정점 s에서 각 정점까지의 최단 거리 배열 dist[] */
1.  S = {s}; // 시작 정점 s로부터 최단 거리 계산이 수행된 정점의 집합
2.  for v ∈ V and v ≠ s do
3.      dist[v] = e(s,v);   /* e(s,v)는 정점 s와 v를 잇는 간선의 가중치 */
4.      Q.insert(v);   /* 정점 v를 우선순위 큐 Q에 삽입한다. */
5.  while Q is not empty do
6.      u = Q.delete();   /* Q의 맨 앞의 요소(정점)를 삭제 및 반환한다. */
7.      S = S ∪ {u};   /* 정점 u를 집합 S에 추가한다. */
8.      for v such that "e(u,v) ≠ ∞ and v ∉ S" do
9.          if dist[u] + e(u, v) < dist[v] then
                dist[v] = dist[u] + e(u, v);
10. return dist;
```

- 정점 n-1개를 우선순위 큐에 삽입함.
- 우선순위 큐에서 요소 하나를 삽입/삭제할 때 시간 복잡도는 $O(\log n)$

- 우선순위 큐가 완전히 빌 때까지 반복문이 수행됨.
- 즉, 반복문의 반복 횟수는 n-1회
- 한 번 반복 때마다 우선순위 큐에서 삭제 연산이 수행됨.
- 따라서, 총 시간 복잡도는 $O(n \log n)$

시간 복잡도를 빠르게 확인하는 방법

- **(예제 3)** 아래 코드의 시간 복잡도를 분석하시오.

```
// 입력: 배낭의 용량 C, n개의 물건, 각 물건의 무게 wi와 가치 vi
// 출력: 9번째 줄의 return 문 참고.
1. for i = 0 to n do K[i,0]=0;  // i는 물건의 종류
2. for w = 0 to C do K[0,w]=0; // w는 배낭의 (임시)용량
3. for i= 1 to n  do
4.    for w = 1 to C  do
5.       if( wi > w) then
6.          K[i, w] = K[i-1, w];
7.       else
8.          K[i, w] = max{K[i-1, w], K[i-1, w-wi]+vi};
9. return K[n, C];
```

- For문이 중첩되어 있음.
- n은 물건의 개수, C는 배낭의 허용 용량이므로, 이 두 값 사이에는 명확한 대소 관계가 성립하지 않음.
- 중간에 for문이 종료되는 가지치기와 같은 기능이 없음.
- 따라서, 시간 복잡도는 $\boldsymbol{O(nC)}$

시간 복잡도를 빠르게 확인하는 방법

- **(예제 4)** 아래 코드의 시간 복잡도를 분석하시오.

```
// Input: List A[]
// Output: List A[]
for  i = 1 to n – 1   do
    temp = A[i];
    j = i – 1;
    while   j >= 0 and A[j] > temp   do
        A[j+1] = A[j];
    A[j+1] = temp;
return A;
```

- 반복문이 중첩되어 있음.
- 외부 반복문은 n – 1번 수행됨.
- 내부 반복문의 경우 인덱스 j는 i보다 1 작은 값부터 시작하므로 외부 반복문 i에 따라 수행 횟수가 결정됨.
- 이때, 내부 반복문은 최대 j가 0이 될 때까지 수행됨.
- 따라서, 시간 복잡도는 $\boldsymbol{O(n^2)}$

알고리즘의 효율적 구현

효율적인 알고리즘 구현을 위한 확인사항

- 반복문의 중첩 구조 확인
- 복잡도가 높은 연산의 사용 여부 및 빈도 확인
  - (예) 지수 연산, 팩토리얼 연산 … 등
- 오버플로우(overflow) 및 언더플로우(underflow) 발생 가능성 확인
  - 특히, 처리되는 수의 크거나 작은 경우
- 재귀 함수(recursive function) 확인
  - 재귀 함수는 스택 오버플로우를 발생시킬 위험이 존재함.
  - 재귀 함수의 호출 구조를 파악하면 시간 복잡도를 보다 정확하게 계산할 수 있음.
- 사용된 라이브러리 함수의 시간 복잡도 확인
  - (예) 파이썬 리스트의 sort 메소드, C++ STL의 sort 함수

효율적인 알고리즘 구현 방법

✓ 반복문의 중첩 횟수는 최소화하고, 불필요한 반복은 피한다.
- 필요하다면, continue 및 break문을 적극적으로 사용할 것.

✓ 팩토리얼, 지수 연산은 최소화한다.

✓ 재귀 함수는 반복문으로 구현하기 쉬우면 가급적 반복문으로 구현한다.

✓ 시간 복잡도를 더욱 줄일 수 있는 방법을 연구한다.
- $O(n^k)$로 쉽게 아이디어가 떠오른다면? ➔
- $O(n^2)$으로 쉽게 아이디어가 떠오른다면? ➔
- $O(n)$으로 쉽게 아이디어가 떠오른다면? ➔
- 광범위한 상태 공간을 탐색해야 하는 상황이라면? ➔

✓ 아이디어가 잘 떠오르지 않는다면, 가장 간단한 brute-force로 구현했을 때, 어떤 부분을 어떻게 수정하면 비용을 줄일 수 있을지 고민해본다.

효율적인 알고리즘 구현 방법

✓ 언어에서 기본적으로 제공하는 다양한 요소 및 라이브러리를 적극 활용한다.

- 만약 라이브러리 사용이 불가하다면 문제에서 사전에 제한 조건으로 설명함.
- 언어에서 기본 제공하는 라이브러리는 효율성과 안정성 모두 우수함.
- 단, 기본 라이브러리를 제외한 외부 라이브러리(e.g., numpy)는 일반적으로 허용 안됨.
- 특히, **삼성 SW역량테스트 B형**, **현대 Softeer Level 4** 등 고급 레벨 유형에서는 **라이브러리 사용이 불가**하므로, 기본 자료구조 및 알고리즘을 스스로 구현할 수 있어야 함.
- 즉석에서 바로 구현할 수 있어야 하는 핵심 자료구조 및 알고리즘
  - **자료구조**: 연결리스트(C++), 스택, 큐, 원형 큐, 이진 트리, 그래프, 힙 … 등
  - **알고리즘**: 탐색, 정렬, 신장 트리, 최단 경로, 트리/그래프 탐색, 기본 DP 알고리즘 … 등

실전 문제 풀이

문제 1. 예산 지원이 가능한 부서의 수

제한시간: 15분

- S사에서는 각 부서에 필요한 물품을 지원해 주기 위해 부서별로 물품을 구매하는데 필요한 금액을 조사하였다. 그러나, 전체 예산이 정해져 있기 때문에 모든 부서의 물품을 구매해 줄 수는 없다. 그래서 최대한 많은 부서의 물품을 구매해 줄 수 있도록 하려고 한다.
- 물품을 구매해 줄 때는 각 부서가 신청한 금액 만큼을 모두 지원해 줘야 한다. 예를 들어 1,000원을 신청한 부서에는 정확히 1,000원을 지원해야 하며, 1,000원보다 적은 금액을 지원해 줄 수는 없다.
- 부서별로 신청한 금액이 들어있는 배열 d와 예산 budget이 매개변수로 주어질 때, 최대 몇 개의 부서에 물품을 지원할 수 있는지 return 하도록 maxBudgetDept() 함수를 완성하시오.
- 제한사항
  - ✓ d는 부서별로 신청한 금액이 들어있는 배열이며, 길이(전체 부서의 개수)는 1 이상 100 이하이다.
  - ✓ d의 각 원소는 부서별로 신청한 금액을 나타내며, 부서별 신청 금액은 1 이상 100,000 이하의 자연수이다.
  - ✓ budget은 예산을 나타내며, 1 이상 10,000,000 이하의 자연수이다.

문제 1. 예산 지원이 가능한 부서의 수

d	budget	result
[1,3,2,5,4]	9	3
[2,2,3,3]	10	4

- **입출력 예제 1 설명**
  - 각 부서에서 [1원, 3원, 2원, 5원, 4원]만큼의 금액을 신청한다.
  - 만약에, 1원, 2원, 4원을 신청한 부서의 물품을 구매해주면 예산 9원에서 7원이 소비되어 2원이 남는다.
  - 항상 신청한 금액만큼 지원해 줘야 하므로 남은 2원으로 나머지 부서를 지원할 수 없다.
  - 위 방법 외에 3개 부서를 지원해 줄 방법들은 다음과 같다.
    - 1원, 2원, 3원을 신청한 부서의 물품을 구매해주려면 6원이 필요함
    - 1원, 2원, 5원을 신청한 부서의 물품을 구매해주려면 8원이 필요함
    - 1원, 3원, 4원을 신청한 부서의 물품을 구매해주려면 8원이 필요함
    - 1원, 3원, 5원을 신청한 부서의 물품을 구매해주려면 9원이 필요함
  - 3개 부서보다 더 많은 부서의 물품을 구매할 수 없으므로 최대 3개 부서만 지원 가능하다.
- **입출력 예제 2 설명**
  - 모든 부서의 물품을 구매해주면 10원이 되므로, 최대 4개 부서의 물품을 구매해 줄 수 있다.

문제 1. 예산 지원이 가능한 부서의 수

```
def maxBudgetDept(d, budget):
    d.sort()
    count = 0

    for amount in d:
        if amount > budget:
            break
        budget -= amount
        count += 1

    return count if budget >=0 else count - 1
```

- 심화문제
  - ➢ 파이썬에서 제공하는 sort( ) 메소드를 사용하지 않고, 자신이 직접 구현하여 위 문제를 해결하시오.

d	budget	result
[1,3,2,5,4]	9	3
[2,2,3,3]	10	4

상기 예제 입출력 결과와 실제 구현 결과를 비교해 보자.

문제 2. 효율적인 조합 계산

제한시간: 20분

- 다음은 조합(combination)을 나타낸 공식이다.

$$\binom{n}{r} = \frac{n!}{(n-r)!\,r!}$$

- 두 변수 n과 r이 주어졌을 때, $\binom{n}{r}$ 을 계산하는 메소드 combination()을 구현하시오.
  - **예제 1:** 입력: combination(10, 5) → 출력: 252
  - **예제 2:** 입력: combination(20, 8) → 출력: 125970

문제 2. 효율적인 조합 계산

- **방법 1.** 반복문을 이용하여 구현하는 방법
  - 가장 간단하게 시도할 수 있는 방법
  - For문으로 팩토리얼 연산을 구현하여 $n!$과 $(n-r)!$ 및 $r!$을 계산한다.
  - 그렇지만, 이렇게 쉽게 해결할 수 있는 문제가 코딩 테스트에 출제될 가능성은 거의 없다. → 이렇게 구현할 때 문제점은 무엇일까?
    - n!은 매우 빠르게 증가하므로, n이 조금만 커져도 n!의 값은 기하급수적 그 이상으로 크기가 증가함
    - 따라서, 오버플로우가 발생할 위험이 매우 높음! (특히, C, C++)

문제 2. 효율적인 조합 계산

- **방법 2.** 점화식을 구한 후 동적 계획법으로 해결하는 방법
  - ➢ 아래 조합 공식을 점화식 형태로 기술하면 다음과 같다.

$$C(n,r) = \binom{n}{r} = \frac{n!}{(n-r)!\,r!} = \binom{n-1}{r-1} + \binom{n-1}{r}$$

※ 단, 0 < r <n
※ 만약 r=0 or r=n이면 C(n, r) = 1이다.

  - ➢ 위와 같이 점화식을 구한 후 우리가 해야 할 일은?
    - ★ **1부터 순서대로 대입해 가면서 규칙을 발견하는 것 ➔ 분할 정복법 OR 동적 계획법 확인**

      C(1,0) = C(1,1) = C(2,0) = C(2,2) = C(3,0) = C(3,3) = C(4,0) = C(4,4) = 1

      C(3,1) = 3, C(3,2) = 3, C(4,1) = 4, C(4,2) = 6, C(4,3) = 4, …
    - ✓ 즉, **이차원 배열을 이용한 동적 계획법**으로 문제를 해결할 수 있다.

자주 출제되는 문제들의 점화식들을 평소에 암기해 두면 실제 코딩 테스트에서 매우 유용하게 활용할 수 있다.

문제 2. 효율적인 조합 계산

- 실제 구현 후 테스트 케이스를 통해 정확하게 동작하는 지 확인한다.

```
def combination(n, r):
    B = [[0 for _ in range(r+1)] for _ in range(n+1)]

    for i in range(n+1):
        for j in range( min(i, r) + 1 ):
            if j == 0 or j == i:
                B[i][j] = 1
            else:
                B[i][j] = B[i-1][j-1] + B[i-1][j]

    return B[n][r]
```

Q & A