- 알고리즘 문제 해결 과정

직관과 비슷한 체계적인 접근.
수식화, 문제를 단순화, 그림으로 그려볼 수 있을까, 수식으로 표현할 수 있을까, 문제를 분해할 수 있을까, 뒤에서부터 생각해서 문제를 풀 수 잇을까, 순서를 강제할 수 있는지.

- 좋은 코드를 짜기 위한 원칙.

- 간결한 코드를 작성하기.
- 적극적으로 코드를 재사용하기
- 표준 라이브러리 공부하기
- 항상 같은 형태로 프로그램을 작성하기
- 일관적이고 명료한 명명법 사용하기
- 모든 자료를 정규화해서 저장하기
- 코드와 데이터를 분리하기

<인성면접>
자기소개
본인은 리더형인지 팔로워형인지. 리더십/팔로워십은 없는것인가.
지원한 분야가 본인하고 잘 안맞으면 어떻게 할 것인가
본인이 직무를 수행함에 있어서 어떤 장점이 있는지.
본인만의 창의적인 경험.
본인이 하고 싶은 업무가 있다면?
부당한 일을 마주한 경험이 있는지.
휴학기간에 무엇을 했는지.
프로젝트 이야기
묘비에 어떤 글귀가 적혀있었으면 좋겠는지
삼성전자에서 뭐하고 싶은지
가장 인상 깊게 들었던 수업은?
어떤 칭찬들었을때 가장 좋은지
여가시간에 뭐하는지
회사에서 중요하다고 생각하는 가치가 무엇인지
본인과 선배가 현장에서 일을 한다면 본인은 무엇을 할 것인지
자신의 단점 3 가지
대외활동이 많은데 학업과 어떻게 병행했는지
내가 남들보다 뛰어나다고 생각하는 점은?
개발할 때 가장 힘든 팀원은?
마지막으로 하고싶은 말

<직무면접> https://jcdgods.tistory.com/401

DB Transaction 문제 DB 무결성과 정합성 설명
2) DB 무결성의 4 가지 설명
3) DB 제약 조건의 종류와 무결성을 연관 설명
4) DB 정규화 단계 설명
5) 데이터 베이스 언어 (DDL, DML, DCL, TCL)에 대한 설명
6) 뷰 특징과 장단점 설명
7) 조인 (내부/외부 조인)에 대한 설명

8) JDBC 와 ODBC 의 차이점 설명
9) statement 와 preparestatement 의 차이점

- 웹

1) GET 과 POST 의 차이점
2) session 과 cookie 차이점과 사용 용도 설명
3) HTML 과 XML 의 차이점과 장단점 설명
4) MVC 모델이란, 프레임워크란?
5) Spring Framework 에 대한 설명
6) JSP 와 Servlet 의 차이점
7) JSP 와 ASP 그리고 PHP 의 차이점
8) AJAX 에 대한 설명

- JAVA

1) String 과 StringBuffer 의 차이점
2) Interface 와 Abstract 의 차이점
3) Static 의 의미와 선언하지 않은 것과의 차이점
4) Heap 과 Stack 에 대한 설명
5) Call by value 와 Call by reference 차이점
6) Java 와 Javascript 의 차이점

- 기타

1) 정렬 알고리즘중 가장 빠른 방식과 느린 방식을 설명와 이유
2) 탐욕 알고리즘이란?
3) 객체란?
4) 이진 탐색 방식을 시퀀스 서치와 비교하여 이점을 설명

- 자료구조

1) Tree 란?
2) Binary Tree 란?
3) Binary Tree 에서 Non-leaf 와 Leaf 노드 개수의 관계식은?
4) Heap Tree 란?
5) Heap Tree 의 데이터 삭제, 추가 과정을 설명하시오
6) Tree 에서 pre, in ,post order 를 설명하시오
7) B tree 란?
8) B Tree 의 데이터 삭제 추가 과정
9) Array List 와 Linked List 의 차이점은?
10) Graph 란?

- 알고리즘

1) Big O 란?

컴퓨터 알고리즘이 실제 동작함에 있어서 시간과 같은 절대적 수치가 아닌, 입력값에 비례하여 해당 알고리즘의 동작 시간을 나타내는 점근적 접근법에서 알고리즘의 최악의 시간 복잡도를 나타내는 표기법
2) Divide and conquer 를 사용한 알고리즘 예를 들어보라.

merge sort, quick sort
3) Divide and conquer 와 Dynamic Programming 의 차이점 및 유사점은?

두 가지 패러다임 모두 하나의 문제를 소문제로 나누어 처리하는 것을 기반으로 하고 있음. 다만 D&C 는 이미 계산한 소문제의 결과를 다시 확인하지 않는 경우에 활용할 수 있음. 반면 DP 는 소문제에 대한 결과를 계속 확인함을 효율적으로 처리하기 위하여 이를 저장하여 이용
4) Backtracking 과 Branch and Bound 의 차이점은?

- Backtracking : DFS, 유망한 노드인지 루트를 이용하여 검사(Pruning)하고 그런 경우에만 하위 자식으로 내려가면서 검사하는 방법
  - Branch and Bound : BFS, 가망이 있는 branch 와 그렇지 않은 것을 구분하여 경우의 수를 줄임
  = 둘 모두 경우의 수를 줄이기 위한 방법
  = 다만 두개의 주요 차이점은 백트래킹의 경우 constraint satisfaction 문제에서 가능한 조합의 후보군을 만들어 가는 방법이고, Branch and bound 는 최적화 문제에 사용된다.
5) Merge/Quick/Heap sort 의 best, average, worst 케이스의 시간 복잡도는?

- 알고리즘 최선 / 평균 / 최악
  = Quick : O(nlogn) / O(nlogn) / O(n^2)
  = Merge : O(nlogn) / O(nlogn) / O(nlogn)
  = Heap : O(n) / O(nlogn) / O(nlogn)
6) Merge sort 와 Quick sort 의 차이점과 방식은? - 둘의 가장 큰 차이점은 O(n)이 필요한 동작을 Divide 되기 전에 하느냐 Combine 할때 하느냐 차이
7) Quick sort 와 Bubble sort 의 차이점은? = Quick Sort 는 Divide and Conquer
  = Bubble Sort 는 Brute Force 정도..?
8) Floyd 알고리즘이란?

그래프 자료구조에서 하나의 노드에서 서로 다른 노드까지의 최단 거리를 구하는 알고리즘
= i 노드에서 j 노드로 이동할 때 k 노드를 거쳐서 가는것이 빠른가 그렇지 않은 가로 명제를 세워 for(k, i, j) [i, j] = max([i,j], [i,k]+[k,j])로 알 수 있음.
9) Dijkstra 알고리즘이란?

- 다익스트라 알고리즘은 하나의 시작점에서 출발하여 다른 노드까지의 최단 거리를 구하는 알고리즘
10) Floyd 와 Dijkstra 알고리즘의 차이점과 정의는?

두 알고리즘의 주요한 차이점은 모든 노드에 대한 최단 거리를 구할 수 있냐 없냐
  = Floyd 는 n^3, 다익스트라는 pq 를 이용하여 O(ElogE)로 나타난다. 알고리즘에서 모든 간선이 1 회씩 검사되고 각 검사된 결과에 따라 최악의 경우 모든 경우에서 pq 에 넣거나 삭제하는 동작을 수행해야한다. 이때 큐에 들어 있을 수 있는 최악의 개수는 E 개이며, 따라서 O(ElogE)의 경우의 수가 나온다. log E <= logV^2 = 2logV 이므로 O(ElogV) 또는 E 가 V^2 에 비해 매우 적은 경우 O(VlogV)로 표기하는 경우도 있다.
11) 프림 알고리즘과 크루스컬 알고리즘의 차이 그리고 시간복잡도는?

프림과 크루스컬 알고리즘은 최소 스패닝 트리를 구하는데 사용되는 알고리즘이다.
- 두가지 모두 Greedy 한 방법으로 현재에서 가장 작은 간선을 선택하는 방식으로 동작한다.

= 크루스컬 알고리즘 : 모든 간선을 가중치의 오름차순으로 정렬한 다음, 간선을 선택하고 해당 간선에 연결된 두 노드가 서로 같은 집합에 포함되어 있는지 여부를 Disjoint set 으로 판단한 다음 포함하거나 제외한다. O(ElogE)
= 프림 알고리즘 : 하나의 시작 노드를 바탕으로 인접한 노드들을 연결하는 간선중에 가장 작은 간선을 선택한다. 전체 알고리즘의 시간복잡도는 O(V^2)이며, 만약 밀집 그래프라면 프림이 더 빠르다. pq 를 이용하면 프림 알고리즘도 O(ElogV)로 구현할 수 있다.

12) Knapsack problem 과 Traveling Salesman Problem 에 대한 설명

- 냅색과 TSP 문제 모두 완전 탐색으로 풀어야 모든 경우의 수를 구할 수 있다.
= 일반적으로 노드의 개수가 적은 경우 냅색은 DP 로 해결한다. TSP 는 모르겠음.
= 완전 탐색의 경우 그 크기가 매우 크므로 두 문제 모두 노드의 수가 많으면 Backtracking 또는 Branch and Bound 로 해결한다.

13) P 와 NP, NP-Completeness 에 대한 설명 - P : Polynomial time 안에 해결할 수 있는 문제
- NP : Polynomial time 안에 답의 존재 여부를 확인할 수 있는 문제
- NP-Hard : 모든 NP 문제를 다항시간내 다른 문제 A 로 환원할 수 있을 때, 이 문제를 NP-Hard 문제라고 한다.
- NP-Complete : NP 이면서 NP-Hard 인 문제로 위의 NP-Hard 의 환원된 문제 A 를 말함. 만약 NP-Complete 문제중 다항시간안에 풀어낼 수 있다면 P=NP 임을 증명할 수 있다. (TSP, 해밀토니안)

OS 데드락 문제
퀵 소트 알고리즘
+ 프로젝트 이야기

<창의면접>

- 버블정렬

인접한 두개의 원소를 비교하여 자리를 교환하는 방식이다. 첫 번째 원소부터 마지막 원소까지 반복하여 한 단계가 끝나면 가장 큰 원소가 마지막 자리로 정렬됨. 첫 번째 원소부터 인접한 원소끼리 계속 자리를 교환하면서 맨 마지막 자리까지 교환한다. 1 회전을 수행하고 나면 가장 큰 자료가 맨 뒤로 이동하므로 2 회전에서는 맨 끝에 있는 자료는 정렬에서 제외되고, 2 회전을 수행하고 나면 끝에서 두 번째 자료까지는 정렬에서 제외된다. 이렇게 정렬을 1 회전 수행할 때마다 정렬에서 제외되는 데이터가 하나씩 늘어난다. 일반적으로 자료의 교환 작업(SWAP)이 자료의 이동 작업(MOVE)보다 더 복잡하기 때문에 버블 정렬은 단순성에도 불구하고 거의 쓰이지 않는다. O(N^2)

```java
public class BubbleSort {

    public static void bubbleSort(int[] arr) {
        final int length = arr.length;
        for (int i = 0; i < length; i++) {
            for (int j = 0; j < length - i; j++) {
                if (j + 1 < length && arr[j] > arr[j + 1]) {
                    arr[j] = arr[j] + arr[j + 1];
                    arr[j + 1] = arr[j] - arr[j + 1];
                    arr[j] = arr[j] - arr[j + 1];
                }
            }
        }
    }

```

- 선택정렬

제자리 정렬(in-place sorting) 알고리즘의 하나로, 해당 순서에 원소를 넣을 위치는 이미 정해져 있고, 어떤 원소를 넣을지 선택하는 알고리즘이다. 즉, 전체 원소들 중에서 기준 위치에 맞는 원소를 선택하여 자리를 교환하는 방식이다. 전체 원소 중에서 가장 작은 원소를 찾아 선택하여 첫 번째 원소와 자리를 교환한다. 그 다음 두번째로 작은 원소를 찾아서 선택하여 두번째원소와 자리를 교환한다. 그 다음에는 세번째로 작은 원소를 찾아 선택하여 세번째 원소와 자리를 교환한다. 이 과정을 반복하면서 정렬이 완성된다. 자료 이동 횟수가 미리 정해진다는 장점이 있지만, 값이 같은 레코드가 있는 경우에 상대적인 위치가 변경될 수 있다는 단점이 있다. O(N^2)

```
public class SelectionSort {

    public static void selectionSort(int[] arr) {
        final int length = arr.length;
        for (int i = 0; i < length; i++) {
            int min = i;

            for (int j = i + 1; j < length; j++) {
                if (arr[j] < arr[min]) {
                    min = j;
                }
            }

            if (min == i) {
                continue;
            }

            arr[min] = arr[min] + arr[i];
            arr[i] = arr[min] - arr[i];
            arr[min] = arr[min] - arr[i];
        }
    }

}
```

- 삽입정렬

정렬되어 있는 부분집합에 정렬할 새로운 원소의 위치를 삽입하는 방법이다. 즉, 자료 배열의 모든 요소를 앞에서부터 차례대로 이미 정렬된 배열 부분과 비교 하여, 자신의 위치를 찾아 삽입함으로써 정렬을 완성하는 알고리즘이다. 매 순서마다 해당 원소를 삽입할 수 있는 위치를 찾아 해당 위치에 넣는다. 자료를 두 개의 부분집합 S 와 U 로 가정한다. S 와 U 를 각각 정렬된 앞 부분의 원소들과 아직 정렬되지 않은 나머지 원소들이라고 가정하자. (보통 U 의 부분집합은 두번째 값부터 시작한다.)정렬되지 않은 부분집합 U 의 원소를 하나씩 꺼내서 이미 정렬되어 있는 부분집합 S 의 마지막 원소부터 비교하면서 위치를 찾아 삽입한다. 삽입 정렬을 반복하면서 부분집합 S 의 원소는 하나씩 늘리고 부분집합 U 의 원소는 하나씩 감소하게 된다. 부분집합 U 가 공집합이 되면 정렬이 완성된다. 이미 정렬되어 있는 자료의 경우에는 삽입 정렬이 가장 빠르고 효율적이다. 바로 앞자리 원소와 한번만 비교하면 되기 때문이다. 단 레코드 수가 많고 레코드의 크기가 클 경우 비효율적이다. 최선의 경우 한번의 비교로 이루어지는 O(N)의 시간 복잡도를 갖지만 최악의 경우 O(N^2)이다.

```
1  public class InsertionSort {
2
3      public static void insertionSort(int[] arr) {
4          final int length = arr.length;
5          for (int i = 1; i < length; i++) {
6              final int key = arr[i];
7              int position = i;
8
9              while (position > 0 && key < arr[position - 1]) {
10                 arr[position] = arr[position - 1];
11                 position--;
12             }
13
14             arr[position] = key;
15         }
16     }
17
```

- 퀵정렬

정렬할 전체 원소에 대해 정렬을 수행하지 않고 기준 값을 중심으로 왼쪽 부분 집합 과 오른쪽 부분 집합으로 분할하여 정렬하는 방법이다. 왼쪽 부분 집합에는 기준 값(피봇)보다 작은 원소들을 이동시키고 오른쪽 부분 집합에는 기준 값보다 큰 원소들을 이동시킨다. 피벗을 제외한 왼쪽 리스트와 오른쪽 리스트를 다시 정렬한다. 그리고 분할된 부분 리스트에 대하여 순환 호출 을 이용하여 정렬을 반복한다. 부분 리스트에서도 다시 피벗을 정하고 피벗을 기준으로 2 개의 부분 리스트로 나누는 과정을 반복한다. 분할 정복 (Devide and Conquer)의 알고리즘을 사용하여 원소들을 정렬한다. 부분집합의 크기가 1 이하로 충분히 작을때까지 재귀 호출을 이용하여 분할한다. 순환 호출이 한번 진행될 때마다 최소한 하나의 원소(피벗)는 최종적으로 위치가 정해지므로, 이 알고리즘은 반드시 끝난다는 것을 보장할 수 있다. 퀵정렬은 평균적으로 매우 빠른 수행 속도를 자랑하는 정렬 방법이다. 순환 호출의 깊이가 LogN 이고 각 순환 호출 단계의 연산이 N 번 이므로 평균적으로 O(NlogN)의 시간복잡도를 보인다. 퀵 정렬이 불필요한 데이터의 이동을 줄이고 먼 거리의 데이터를 교환할 뿐만 아니라, 한 번 결정된 피벗들이 추후 연산에서 제외되는 특성 때문이다. 이미 정렬된 리스트를 정렬하는최악의 경우 O(N^2)이다. 리스트를 불균등하게 분할한다는점에서 병합정렬과 차이점을 보인다.

```java
public class QuickSort {

    public static void quickSort(int[] arr) {
        quickSort(arr, 0, arr.length -1);
    }

    private static void quickSort(int[] arr, int begin, int end) {
        if (begin >= end) {
            return;
        }

        int middle = begin + (end - begin) / 2;
        int pivot = arr[middle];
        int left = begin;
        int right = end;

        while (left <= right) {
            while (arr[left] < pivot) {
                left++;
            }

            while (arr[right] > pivot) {
                right--;
            }

            if (left <= right) {
                arr[left] = arr[left] + arr[right];
                arr[right] = arr[left] - arr[right];
                arr[left] = arr[left] - arr[right];

                left++;
                right--;
            }
        }

        if (begin < right) {
            quickSort(arr, begin, right);
        }

        if (end > left) {
            quickSort(arr, left, end);
        }
    }
}
```

- 힙정렬

자료구조 '힙'은 완전 이진 트리의 일종으로 우선순위 큐를 위하여 만들어진 자료구조로, 최댓값, 최솟값을 쉽게 추출할 수 있는 자료구조이다. 최대 힙 트리나 최소 힙 트리를 구성해 정렬을 하는 방법으로 내림차순 정렬을 위해서는 최대 힙을 구성하고 오름차순 정렬을 위해서는 최소힙을 구성하면 된다. N 개의 노드에 대한 완전 이진 트리를 구성한다. 이때 루트 노드부터 부모 노드, 왼쪽 자식노드, 오른쪽 자식 노드 순으로 구성한다. 최대 힙을 구성한다. (최대 힙 : 부모노드가 자식노드보다 큰 트리) 큰 수(루트에 위치) 최대 힙에서 최댓값은 루트 노드이므로 루트 노드가 삭제된다. 그리고 삭제된 루트 노드에는 힙의 마지막 노드를 가져오고 힙을 재구성한다. 이 과정을 반복하며 정렬을 수행한다. 힙 트리의 전체 높이가 거의 $\log_2 N$(완전 이진 트리이므로)이므로 하나의 요소를 힙에 삽입하거나 삭제할 때 힙을 재정비하는 시간이 $\log_2 n$ 만큼 소요된다. 그리고 원소의 개수가 n 개 이므로 전체적으로 O(NlogN)의 시간이 걸린다. 힙정렬이 가장 유용한 경우는 전체 자료를 정렬하는 것이 아니라 가장 큰 값 몇개만 필요할 때 이다. O(NlogN)

```
public class HeapSort {

    public static void heapSort(int[] arr) {
        final int length = arr.length;
        final Heap heap = new Heap(length);
        for  (int element : arr) {
            heap.insertHeap(element);
        }

        for (int i = length - 1; i >= 0; --i) {
            arr[i] = heap.deleteHeap();
        }
    }

    private static class Heap {
        private int heapSize;
        private final int[] items;

        private Heap(int count) {
            heapSize = 0;
            items = new int[count + 1];
        }

        private void insertHeap(int item) {
            int i = ++heapSize;
            while (i != 1 && item > items[i / 2]) {
                items[i] = items[i / 2];
                i /= 2;
            }

            items[i] = item;
        }

        private int deleteHeap() {
            int item = items[1];
            int temp = items[heapSize--];
            int parent = 1;
            int child = 2;

            while (child <= heapSize) {
                if (child < heapSize && items[child] <
                    child++;
                }

                if (temp >= items[child]) {
                    break;
                }

                items[parent] = items[child];
                parent = child;
                child *= 2;
            }

            items[parent] = temp;
            return item;
        }
    }

}
```

- 병합(합병) 정렬

2 개 이상의 자료를 오름차순이나 내림차순으로 재배열하는 것이다. 여러 개의 정렬된 자료의 집합을 병합하여 한 개의 정렬된 집합으로 정렬하는 것으로써 부분집합으로 분할(divide)하고 각 부분집합에 대해서 정렬 작업을 완성(conquer) 한 후에 정렬된 부분집합들을 다시 결합(combine) 하는 분할 정복(divide and conquer) 기법을 사용한다. 리스트의 길이가 0 또는 1 이면 이미 정렬된 것으로 본다. 그렇지 않은 경우에는 정렬되지 않은 리스트를 절반으로 잘라 비슷한 크기의 두 부분 리스트로 나눈다. 각 부분 리스트를 재귀적으로 합병 정렬을 이용해 정렬한다.두 부분 리스트를 다시 하나의 정렬된 리스트로 합병한다. 부분집합의 크기가 충분히 작지 않으면 순환 호출을 이용하여 다시 분할 정복 기법을 적용한다. 결합단계에서는 정렬된 부분집합들을 하나의 집합으로 결합한다. 합병 정렬에서 실제로 정렬이 이루어지는 시점은 2 개의 리스트를 합병(merge)하는 단계 이다. 그리고 이 방법을 반복 수행하면서 정렬을 완성시킨다. 만약 레코드를 배열(Array)로 구성하면, 임시 배열이 필요하다. 하지만 만약 레코드를 연결 리스트(Linked List)로 구성하면, 링크 인덱스만 변경되므로 데이터의 이동은 무시할 수 있을 정도로 작아진다. 순환 호출의 깊이는 $O(\log_2 N)$이고 합병 단계에서는 최대 N 번의 비교연산을 수행한다. $O(N\log_2 N)$

```java
public class MergeSort {
    private static int[] sorted;

    public static void mergeSort(int[] arr) {
        sorted = new int[arr.length];
        mergeSort(arr, 0, arr.length - 1);
    }

    private static void mergeSort(int[] arr, int begin, int end) {
        if (begin < end) {
            final int middle = (begin + end) / 2;
            mergeSort(arr, begin, middle);
            mergeSort(arr, middle + 1, end);
            merge(arr, begin, middle, end);
        }
    }

    private static void merge(int[] arr, int begin, int middle, int end) {
        int i = begin;
        int j = middle + 1;
        int k = begin;

        while (i <= middle && j <= end) {
            if (arr[i] <= arr[j]) {
                sorted[k] = arr[i++];
            } else {
                sorted[k] = arr[j++];
            }

            k++;
        }

        int t;
        if (i > middle) {
            for (t = j; t <= end; t++, k++) {
                sorted[k] = arr[t];
            }
        } else {
            for (t = i; t <= middle; t++, k++) {
                sorted[k] = arr[t];
            }
        }

        for (t = begin; t <= end; t++) {
            arr[t] = sorted[t];
        }
    }
}
```

- 분할 정복

문제를 더 작은 문제로 분할하는 과정(devide), 각 문제에 대해 구한 답을 원래 무제에 대한 답으로 병합하는 과정(merge), 더이상 답을 분할하지 않고 곧장 풀 수 있는 매우 작은 문제(base case). 비슷한 decrease and conquer 방법으로 이분탐색 알고리즘도 있음.
문제를 나눔으로써 어려운 문제를 해결할 수 있고 병렬적으로 문제를 해결하는데 큰 강점이 있다. 그러나 함수를 재귀적으로 호출한다는 점에서 함수 호출로 인한 오버헤드가 발생하며, 스택에 다양한 데이터를 보관하고 있어야 하므로 스택 오버플로우가 발생하거나 과도한 메모리 사용을 하게 되는 단점이 있음. 이분탐색(binary-search) 이 대표적인 분할 정복의 알고리즘.

- 그리디 알고리즘

결정해야 할 때, 그 순간에 가장 좋다고 생각하는 것을 선택하면서 답을 찾아가는 알고리즘. 그 때 그때는 최적일지도 모르지만,최종적으로는 답이 최적이 아닐 수도 있다. (거스름돈 문제) 크루스칼, 프림 알고리즘

- 다이나믹 프로그래밍

큰 문제를 작은 문제로 나눠서 푸는 알고리즘. 두가지 속성을 만족해야 다이나믹 프로그래밍으로 문제를 풀 수 있다.
1) Overlapping Subproblem(겹치는 부분문제. 피보나치 수열과 같이 큰 문제와 작은 문제를 같은 방법으로 풀 수 있는지, 그리고 문제를 작은 문제로 쪼갤 수 있는지.)
2). Optimal Substructure. (문제의 정답을 작은 문제의 정답에서 구할 수 있다. 예를 들어 서울에서 부산을 가는 가장 빠른 길이 대전과 대구를 순서대로 거쳐야한다면 대전에서 부산을 가는 가장 빠른 길은 대구를 거쳐야한다. 피부나치 수열의 경우 문제의 정답을 작은 문제의 정답을 합하는 것으로 구할 수 있다. 즉 Optimal Substructure 를 만족한다면 문제으 크기에 상관없이 어떤 한 문제의 정답은 일정하다.)

따라서 다이나믹 프로그래밍에서 각 문제는 한번만 풀어야한다. Optimal Substructure 를 만족하기 때문에 같은 문제는 구할 때마다 정답이 같다. 따라서 정답을 한 번 구했으면 정답을 어딘가에 메모해놓는다. 이런 메모하는 것을 코드의 구현에서는 배열에 저장하는 것으로 할 수 있다. => memoization.

다이나믹 프로그래밍을 푸는 두가지 방법이 있다.
1) Top-down : 문제를 작은 문제로 나누고 작은 문제를 푼 후 전체 문제를 푸는 순서로 진행(재귀 호출)
2)Bottom-up : 문제 크기가 작은 문제부터 작은 문제로 푸는 방식. 문제의 크기를 조금씩 크게 만들면서 진행.
Ex) 플로이드 와샬 알고리즘. 분할정복 알고리즘과의 차이점은 분할정복의 하부 문제들은 보다 독립적이라는 점. 동적 프로그래밍에서 하부 문제들은 서로 겹침.