1. Time Complexity of Algorithms

1.1 Time Complexity

알고리즘(Algorithm)은 수학과 컴퓨터 과학, 언어학 또는 엮인 분야에서 어떠한 문제를 해결하기 위한 정해진 일련의 절차나 방법이다. 계산을 실행하기 위한 단계적 절차를 의미하며 프로그램 명령어의 집합을 의미하기도 한다.

시간 복잡도(Time Complexity)는 문제를 해결하는데 걸리는 시간과 입력의 함수 관계를 가리킨다. 문제를 해결하기 위한 알고리즘의 로직을 코드로 구현할 때, 입력값의 변화에 따라 연산을 실행할 때 연산 횟수에 비해 시간이 얼마나 걸리는가를 의미한다. 즉, 효율적인 알고리즘을 구현한다는 것은 입력값이 커짐에 따라 증가하는 시간의 비율을 최소화한 알고리즘을 구성했다는 이야기다.

```python
# 최댓값 산출 알고리즘

test_values = [4, 3, 5, 6, 2, 1]

def maximum(values):
    answer = None

    for value in values :
        if (answer == None) or (value > answer) :
            answer = value

    return answer

max_value = maximum(test_values)
```

알고리즘의 시간복잡도는 기본적으로 **최악 시간 복잡도(Worst-case Time Complexity)**를 기준으로 계산한다. 대부분의 경우 데이터가 다양한 소스로부터 추출되어 정돈되지 않았기 때문에 시행떄마다 최소시간과 최대시간이 제각각 발생할 수 있기 때문이다. 최악 시간 복잡도를 기준으로 산출할 때 코드의 길이가 길면 길수록 정확한 시간복잡도를 측정하기란 쉽지 않다. 따라서 **빅-오 표기법(Big-O)**에 따라 입력데이터에 따른 차원의 정도만 확인한다.

1.2 Evaluating the execution time

알고리즘이나 코드의 실행속도를 확인하는 함수는 time 패키지와 timit 패키지에 존재한다.

```python
import time
start = time.time()
f(algorithm)
end = time.time()
runtime = end-start

import timeit
runtime = timeit.timeit(f(algorithm), runtime = 100)
```

2. Constant Time Complexity

2.1 Constant time complexity

상수 시간 복잡도(Constant Time Complexity)는 입력값에 관계없이 코드의 수행시간이 변하지 않는 시간 복잡도를 의미한다. Big-O 표기법상 상수 시간 복잡도는 $O(1)$로 표기한다. 상수 시간 복잡도를 가지고 있는 연산은 다음과 같다.

- 변수의 값을 단일 메모리 위치에 할당하는 것
- 단일 메모리 위치에서 값을 읽고 쓰는 것
- 고정 비트 길이의 숫자를 이용한 기본적인 연산
- 단일 조건문을 확인하는 것
- 문자를 출력하는 것

2.2 Amortized Time Complexity

리스트에 원소를 추가할 때 컴퓨터는 새로운 원소가 늘어난 길이만큼의 저장소 크기를 찾아 저장하게 된다. 좀더 효율적으로 값을 추가하기 위해 리스트의 원소가 가득 차게되면 해당 길이만큼 여유 공간을 갖는 리스트를 생성하는 방식을 채택한다고 하자.

```python
def append_cost(array_length, list_length):
    if array_length == list_length:
        return array_length
    return 1

def append_N_list_cost(N) :
    array_length = 1
    list_length = 0
    total_cost = 0
    for i in range(N) :
        cost = append_cost(array_length, list_length)
        total_cost += cost
        if array_length == list_length :
            array_length *= 2
        list_length += 1
    return total_cost

costs = []
for N in range(5000) :
    cost = append_N_list_cost(N)
    costs.append(cost)
```

효율적인 append()의 시간복잡도 그래프는 시간 복잡도가 크게 증가하는 부분이 있지만 대체적으로 선형 주세를 가지고 있으므로 O(N)이라고 생각할 수 있다. 하지만 연산중에서 증가하는 부분은 매우 드물기 때문에 최악 시간 복잡도에 따라 O(N)이라고 판단하는 것은 손해일 수 있다. 따라서 평균값을 적용한 $O(1) = \frac{O(N)}{N}$ 으로 시간복잡도로 간주한다. 이를 **분할 상환 시간 복잡도(Amotized Time Complexity)**라고 한다.

3. Logarithmic Time Complexity

3.1 Binary Search Algorithm

이진 건색 알고리즘(Binary Search Algorithm)은 오름차순으로 정렬된 리스트에서 특정한 값의 위치를 찾는 알고리즘이다. 처음 선택한 중앙값이 찾는 값보다 크면, 그 값은 새로운 최댓값이 되며, 작으면 그 값은 새로운 최솟값이 된다. 검색 원리상 정렬된 리스트에만 사용할 수 있다는 단점이 있지만, 검색이 반복될 때마다 목표값을 찾을 확률은 두 배가 되므로 속도가 빠르다는 장점이 있다.

```python
def binary_search(target) :
    range_start = 0
    range_end = 63
    while range_start < range_end :
        range_middle = (range_start + range_end) // 2
        value = ask(range_middle)

        if value == target :
            return middle_range
        elif value < target :
            # Discard the first half of the range
            range_start = range_middle + 1
        else :
            # Discard the second half of the range
            range_end = range_middle - 1
    # At this point range_start = range_end
    if values[range_start] != target :
        # return -1 if there isn't target value in values
        return -1
    return range_start
```

3.2 Logarithmic TIme Compleixty

이진 검색 알고리즘의 최악 시간 복잡도는 while loop가 반복될 때마다 확인해야 하는 데이터의 값은 절반으로 줄어든다. 따라서 수학적 정의에 따라 시간복잡도는 **로그 시간 복잡도(Logarithmic Time Complexity)**를 따른다.

4. Sorting Algorithms

4.1 Selection sort

정렬 알고리즘은 문제가 이해하기 쉽거나 추상적이지 않으며, 여러가지 솔루션이 필요한 복잡한 문제일 경우 사용된다. Python 내부의 sorted(), list.sort() 함수가 존재한다. **선택정렬(Selection Sort)** 효율적이지는 않지만 단순한 알고리즘으로 최솟값의 인덱스를 앞에서부터 하나씩 변경한다. 선택 정렬의 시간복잡도는 두개의 for loop가 중첩되어 있기 때문에 $O(N^2)$의 시간 복잡도를 따른다.

```python
def swap(values, i, j) :
    temp = values[i]
    values[i] = values[j]
    values[j] = temp

def select_minimum_index_in_range(values, range_start) :
    minimum = None
    minimum_index = None
    N = len(values)
    for i in range(range_start, N) :
        if minimum == None or values[i] < minimum :
            minimum = values[i]
            minimum_index = i
    return minimum_index

def selection_sort(values) :
    N = len(values)

    for range_start in range(N) :
        min_index = select_minimum_index_in_range(values, range_start)
        swap(values, range_start, min_index)
```

5. Space Complexity

공간 복잡도(Space Complexity)는 프로그램을 실행시킨 후 완료하는데 필요한 자원 공간의 양으로 알고리즘에 해당되는 메모리의 크기(변수, 리스트)를 의미한다. 공간복잡도와 시간복잡도는 서로 반비례의 관계를 가지고 있는데, 시간 복잡도를 증가시키게 되면 공간 복잡도의 크기가 줄어들고, 공간 복잡도를 증가시키게 되면 시간 복잡도의 크기가 감소한다는 특징이 있다. 이를 적절히 균형을 맞추어 효율적인 메모리와 빠른 실행시간을 갖는 알고리즘을 설계할 수 있다.

```python
# No preprocessing algorithm for the sums of pairs problem:
def find_sums(values, target_sums):
    sums = {}
    for target in target_sums:
        sums[target] = False
        for i in range(len(values)):
            for j in range(i, len(values)):
                if values[i] + values[j] == target:
                    sums[target] = True
    return sums

# Full preprocessing algorithm:
def find_sums_precompute(values, target_sums):
    possible_sums = set()
    for i in range(len(values)):
        for j in range(i, len(values)):
            possible_sums.add(values[i] + values[j])
    sums = {}
    for target in target_sums:
        sums[target] = target in possible_sums
    return sums

# Balanced algorithm:
def find_sums_balanced(values, target_sums):
    value_set = set(values)
    sums = {}
    for target in target_sums:
        for value1 in values:
            value2 = target - value1
            sums[target] = value2 in value_set
    return sums
```