MIT 코스웨어
C11 | 프로그램의 효율성 (PT2)

Constant Complexity / O(1)

constant complexity의 경우,
데이터 인풋 사이즈와 상관없이 시간이 동일
루프나 재귀가 가능하지만, 인풋이 달라지지 않는 선에서

Logarithmic Complexity / O(logN)

Logarithmic Complexity의 경우,
인풋 사이즈의 로그에 따라 복잡성이 증가한다,
대표적인 예시는 바이섹션 서치와 리스트의 바이너리 서치

Bisection Search

바이섹션 서치는 어떤 리스트 속의 검색을 할때,
1. 먼저 리스트 전체를 반으로 나누는 인덱스를 찾고
2. 그 인덱스가 찾고자 하는 요소인지를 체크한 다음에
3. 아니면 검색 대상이 그 인덱스 이상 이하인지를 체크하고
4. 해당하지 않는 범위는 전부 무시하는 방식이다.

이 경우 찾고자 하는 input의 갯수가 반으로 줄어드므로,
1 = n/2^i -> i = log n -> O(log n)이 된다.

```
def bisect_search1 (L, e):
  # 만약 L이 빈 리스트이면 False (없다)
  if L == []:
    return False
  # 만약 L이 한 개면 True (있다)
  elif len(L) == 1:
    return L[0] == e:
  else:
    # half라는 임의의 포인트를 설정
    half = len(L) // 2
    # 만약 하프에 위치한 값이 e보다 높으면 작은쪽 반만
    # recursive
    if L[half] > e:
      return bisect_search1(L[:half],e)
    # 만약 하프에 위치한 값이 e보다 낮으면 큰쪽 반만
    # recursive
    else:
      return bisect_search1(L[half:],e)

def bisect_search2 (L, e):
  # 리스트, 타겟, low, high
  def bisect_helper(L,e, low, high):
  # high, low가 같으면 있다는 의미!
    if high == low:
      return L[low] == e
  # mid 변수로 새로 생성한다.
    mid = (low + high) // 2
     # 만약 e가 mid 인덱스값에 있으면 True
    if L[mid] == e:
      return True
    # 만약 e가 mid 인덱스값보다 작으면
    elif L[mid] > e:
      # 만약 mid가 low까지 갔는데도 없으면 False
      if low == mid:
        return False
    # 아니면 lower half를 분석
    else:
      return bisect_helper(L,e,low,mid-1)
  # 리스트 길이가 0이면 False
  if len(L) == 0:
    return False
  # 아니면 전체 리스트에 대해 low, high 설정 후 실시
  else:
    return bisect_helper(L,e,0,len(L)-1)
```

Linear Complexity / O(N)

인풋의 증가량 비율에 따라 연산량도 비례
즉, 연산량이 5개에서 10개로 늘어난다면 시간도 x2
주로 리스트 안에 특정 요소가 있는지를 찾는 in이나,
iterative loops같은 코드에서 보임

Iterative Factorial

```
def fact_iter(n):
  prod = 1
  # 리스트를 훑어가므로 연산량이 길이에 비례
  for i in range(1, n+1):
    prod *= i
  return prod
```

Log-Linear Complexity / O(NlogN)

대부분의 알고리즘이 Log-Linear에 해당한다.
가장 대표적인 log-linear는 merge sort이다.

Polynomial Complexity O(N^C)

quadratic한 알고리즘이 대표적이다.
nested loops이나 recursive function call

Exponential Complexity O(C^N)

사이즈가 커질때마다 재귀가 늘어나는 경우
대표적인 예시는 하노이의 탑이다.
많은 중요한 문제들은 대부분 Exponential하다.

하노이의 탑

```
def printMove(fr, to):
  print(str(fr) -> str(to))

# 링의 개수, 1/2/3의 바
def Towers(n, fr, to, spare):
  # 종료 조건
  # 링이 한개 밖에 없으면 그냥 from->to
  if n == 1:
    printMove(fr, to)
  # 만약 링이 1개 이상이면
  # n-1개의 링을 from에서 spare로 옮기고
  # 1개의 링을 from에서 to로 옮기고
  # 나머지 n-1개의 링을 spare에서 to로 옮긴다
  else:
    Towers(n-1, fr, spare, to)
    Towers(1, fr, to, spare)
    Towers(n-1, spare, to, fr)
```

All Possible Subsets

```python
# 만약 n개의 subset을 만들어야 한다면,
# n-1개의 subset를 만들고
# 이거에 전체적으로 n번째 element를 붙이고
# 붙인 것과 오리지널을 합치는 것

def gensubsets(L):
  # 만약 L 길이가 0이면 그냥 그대로
  if len(L) == 0:
    return [[]]
  # smaller는 마지막 빼고 subset
  smaller = genSubsets(L[:-1])
  # extra는 마지막 요소
  extra = L[-1:]
  new = []
  # smaller의 subset에 extra를 더해서 new
  for small in smaller:
    new.append(small+extra)
  return smaller+new
```

재귀와 속도의 관계

피보나치를 구하는 두 가지 방법에 대해서,
반복은 변수를 창출해야 하지만 (메모리 사용) O(n)이고,
재귀는 변수 정의가 필요없지만 worst가 O(2^n)이다.
즉, 재귀는 항상 빠른 속도를 보장하지는 않는다.

대표적 기능의 시간 복잡도

index: O(1) -> WC: O(n) -> AC: O(1)
store: O(1) -> WC: O(n) -> AC: O(1)
length: O(1) -> WC: O(n)
append: O(1) -> WC: O(n)
==: O(1)
delete: O(1) (AC)
remove: O(n)
copy: O(n)
reverse: O(n)
iteration: O(n) -> WC: O(n)
in list: O(n)