2018 2월 10일 스터디 내용 정리

노트북:	컴퓨터 프로그램의 구조와 해석		
만든 날짜:	2018-02-10 오전 12:24	**수정한 날짜:**	2018-02-11 오전 1:10
작성자:	haidn1994@gmail.com		
URL:	https://en.wikipedia.org/wiki/List_of_mathematical_symbols		

>>> 2월 10일 스터디

다음 세가지 프로세스의 차이점을 알아보자.

- 반복하는 프로세스
  - 따로 상태변수를 가진다.
  - 반복하는 식의 맨 왼쪽에 자기 자신을 호출한다.
  - 시간 복잡도는 이론적으로 선형 재귀 프로세스랑 같다.
    하지만 실제로는 하드웨어적인 특성을 고려해야하고, 이를 고려하면 선형 재귀 프로세스에 비해서 실제로 걸리는 시간은 더(훨씬) 적다.
    이런 특성을 극대화한 알고리즘/자료구조가 바로 비트맵이다. 참고하라.
  - 공간 복잡도는 O(1)이므로 가장 효율적인 메모리 사용률을 자랑한다.
- 되도는 프로세스
  - 선형 재귀 프로세스
    - 상태변수 없이 스택에 이를 저장한다.
    - 반복하는 식의 맨 왼쪽에 자기 자신을 호출하지 않고 다른 함수를 호출하여 이를 스택에 쌓는다.
    - 시간 복잡도는 반복하는 프로세스와 마찬가지로 O(n)이지만 하드웨어적인 특성을 고려했을 때 실제로 걸리는 시간은 더(훨씬) 많다.
    - 공간 복잡도는 O(n)으로 반복하는 프로세스보다 좋지 않다. 하지만 준수한 편이다.
  - 여러 갈래로 되도는 프로세스
    - 반복하는 식에 자기 자신이 2번이상 호출되면(물론 동시에 맨 왼쪽에 다 여러 갈래로 되돌고 있다고 할 수 있다.
    - 시간 복잡도는 좋지 않다. 대략 1.6^n 정도로 시간 복잡도는 지수적으로 증가한다.
      사실상 못써먹을 알고리즘이기 때문에 개선이 필요하다. 피보나치 수열을 효과적으로 개선하는 방법은 다이나믹 프로그래밍이다.
    - 공간 복잡도는 다양하게 나타날 수 있다. 예를 들어 피보나치 수열의 경우 시간 복잡도는 1.6^n으로 아주 안좋지만, 공간 복잡도는 O(n)으로 괜찮은 편이다.

즉, 어떤 프로세스가 반복하는지, 선형 재귀 프로세스인지, 그리고 여러 갈래로 되도는 프로세스인지를 판별하는 기준은 내부적으로 계산하는 방법이 어떻게 되느냐에 달려있다.

- 반복하는 프로세스, 선형 재귀 프로세스, 여러 갈래로 도는 프로세스는 모두 되도는 프로시저로 만들어진다.
  - 반복하는 프로세스는 상태변수를 두고, 상태변수의 상태를 계속 바꾸는 식으로 필요한 계산을 수행한다.
    시간 복잡도는 보통 O(n)으로 나오고, 공간 복잡도는 보통 O(1)로 나온다. 하지만 절대적이지는 않다.

  - 되도는 프로세스는 상태변수를 두는 스택을 쌓고 내리는 과정을 통해서 필요한 계산을 수행하는 프로세스를 뜻한다.
    보통 함수가 재귀함수로 구현되어 있고 반복 조건에서 자기자신을 한번씩만 호출된다면(한번에 스택을 한번씩만 쌓는다면) 되도는 프로세스라 할 수 있다.
    시간 복잡도는 보통 O(n)으로 나로고, 공간 복잡도는 보통 O(1)로 나온다. 하지만 절대적이지는 않다.

  - 여러 갈래로 되도는 프로세스인지를 판별하는 기준은 반복 조건에서 자기 자신을 한번에 두번 이상씩 호출한다면 여러 갈래로 되돈다고 할 수 있다.
    시간 복잡도는 보통 지수복잡도를 가진다. 밑은 다양하게 나온다. 따라서 보통 다이나믹 프로그래밍을 사용해서 개선한다. 이때는 시간 복잡도가 O(N)으로 감소한다.
    공간 복잡도는 다양하게 나타난다. 예를 들어 피보나치 수열의 공간 복잡도는 O(n)이다. 하지만 여러 갈래로 되도는 프로세스는 보통 다이나믹 프로그래밍을 통해서 시간 복잡도를 개선하려고 하기 때문에, 기본적으로 사용하는 스택을 제외하고 또 다른 메모리를 더 사용하게 된다.

자바스크립트?

자바스크립트도 마찬가지로 이해하면 좋다.
하지만 올바른 재귀함수가 되기 위해서는 다음 2가지 조건을 만족해야 한다.

1. return에 자기 자신을 호출할 것(반복 조건)
2. 올바른 기저사례(base case)를 적을 것(탈출 조건)

그래야 스택을 터뜨리지 않고(그 유명한 Stack Overflow를 일으키지 않고!) 자신이 맡은 바를 수행하는 재귀함수를 만들 수 있다.

반복하는 프로세스와 선형으로 반복하는 프로세스의 이론적인 시간 복잡도는 같지만, 실제로는 좀 차이가 있는데 왜 그럴까?

- 메모리에 배정하는 명령어는 상당히 비싸기 때문에 스택을 계속 쌓아올리는 되도는 프로세스는 비록 이론적인 시간 복잡도가 같을지라도 하드웨어적인 차이 때문에 비용의 차이가 발생한다.
- 즉, 변수 자체가 달라진다고 이해해도 좋다.
- O(n) 은 O(k)와 다르고, n < k라고 이해하면 좋다.

추상과 패러다임

미리 올려놓은 추상, 추상화의 뜻과 패러다임의 뜻, 그리고 프로그래밍을 공부하면서 만나는 패러다임의 2가지 예시들을 알아보자.
다음 링크를 통해 공부하면 된다.

- 추상: 어떤 객체(인식할 수 있는 모든 대상 중의 하나)에서 복잡하고 관심없는 특성들은 배제하고, 특정한 관점(View)에서 관심있는 특성들만 골라 하나의 집합으로 파악하는 것
- 패러다임(으뜸꼴, 표준꼴): 다음 링크를 참고하자. > https://github.com/haidn1994/SICP_Study/wiki/IT-%EA%B8%B0%EB%B3%B8-%EA%B0%9C%EB%85%90#%ED%8C%A8%EB%9F%AC%EB%8B%A4%EC%9E%84paradigm-%EC%9C%BC%EB%9C%B8%EA%BC%B4-%ED%91%9C%EC%A4%80%EA%BC%B4
  본래 수사학에서 유래했으며, 무엇보다도 멋있는 말이기 때문에 다양한 분야에서 다양한 용도로 사용된다.

알고리즘 설계 패러다임 - 분할 정복법

프로그래밍을 공부하다 보면 알고리즘을 공부할 필요성을 느끼게 되고, 알고리즘을 공부할 때는 여러 패러다임을 만나게 되는데
그 중에 하나가 바로 분할 정복법(divide and conquer)이다. 분할 정복법은 한번에 해결하기 어려운 큰 문제를 작은 문제로 쪼개고,
쪼개는 과정에서 문제가 절로 해결되거나(쪼개는 과정 자체가 문제를 해결하는 것), 또는 작은 문제에 적용할 수 있는 해법을 적용할 수 있는 크기까지 문제를 쪼갠다/분할한다.(divide)
그렇게 얻은 여러 해(solution)를 병합(merge)하고, 그렇게 병합하면 문제를 해결/정복(conquer)할 수 있게 된다.

즉, 분할 정복 패러다임을 적용해서 문제를 풀어나가는 과정을 정리하면 다음과 같다.

- 문제를 해결 될때 까지 분할 또는 작은 문제에 적용할 수 있는 해법을 적용 가능한 크기까지 분할(divide)
- 그렇게 얻은 해집합을 병합한다.(merge)
- 문제를 정복/해결한다.(conquer)

이를 통해서 1.17번 문제를 이해하고 해결할 수 있는 능력을 키울 수 있다.

참고 링크: https://en.wikipedia.org/wiki/Divide_and_conquer_algorithm

로그 표기법

로그는 지수함수의 역함수로 큰 수를 계산하거나 할 때 굉장히 유용하게 사용할 수 있는 일종의 함수다.
로그는 3가지 구성 요소를 가지고 있다.

- 로그 본체(log)
- 밑(base)
- 진수(영어로 부르는 명칭은 없는 것 같다.)

그리고 유명한 밑이 있는데, 보통 2, 자연상수, 10 3가지로 구성된다.
자세한 사항은 다음 링크를 참고하자. > https://en.wikipedia.org/wiki/Logarithm#Particular_bases

컴퓨터 프로그래밍에서는 보통 밑이 2인 로그를 매우 자주 쓴다.

덧셈(addition), 곱셈(multiplication), 거듭제곱(exponentiation)

> 주의: 여기서 나오는 모든 수는 모두 0을 포함하는 자연수다.

예를 들어 3이란 무엇일까? 가장 직관적인 정의로는 1이 3번 더한것을 3이라고 할 수 있다.
즉, 자연수(정수(?))의 단위가 되는 1을 필요한 만큼 **더한다**는 것으로 이를 정의할 수 있다.

이를 통해서 덧셈의 정의를 유추하면 n + m이 있다면 1을 n만큼 더하고, 다시 1을 m만큼 더한것을 서로 더하면 n + m을 얻을 수 있다.
컴퓨터 프로그래밍에서 루프를 돌리는 것을 생각하면 좋다.

```
(1 + 1 + 1 + .... + 1) <- 서로 n번 1을 더한다. = n
(1 + 1 + .... + 1) <- 서로 m번 1을 더한다. = m

n + m = (1 + 1 + 1 + .... + 1) <- n번 + (1 + 1 + .... + 1) <- m번
```

그렇다면 곱셈은? 예를 들어 n 곱하기 m이 정의되었다고 하자. 이것은 어떻게 정의할 것인가?
답은 n을 m번 곱한다고 이해하면 좋다. 즉, 이것도 덧셈과 마찬가지로 루프를 돌린다고 생각하면 좋다.
즉, 덧셈의 정의를 가지고 곱셈을 정의 할 수 있다. 다음을 보자.

```
m*n = (m + m + m + ... + m) <- 이 연산을 n번 반복한다.
그러면 새로운 정수 k를 얻을 수 있다. 즉, m*n = k
```

덧셈과 곱셈의 관계처럼, 곱셈과 거듭제곱의 관계도 비슷하다.
예를 들어서, a^k라는 지수표현식을 보자. 여기서 밑(base)은 a가 되고, k는 지수(exponent)가 된다.

여기서 주의해야 할것은 여태까지 해왔던 덧셈과 곱셈의 경우에는 어떤 항이 먼저 오든 관계가 없었지만(교환법칙 성립) 거듭제곱의 경우는 이 순서가 관련이 있다. 그래서 밑과 지수를 엄격하게 구분하는 표기법을 고안하여 사용하는 것이다.

다시 돌아가서, 거듭제곱의 정의는 밑을 지수번만큼 곱하는 것이다. 이를 수식으로 표현하면 다음과 같다.

```
(a * a * a * .... * a * a) = a^k <- a를 k번 곱한다. 순서는 엄격하게 지켜져야 한다.
```

이를 덧셈의 직관적인 정의를 통해서 곱셈을 정의하고, 덧셈을 통한 곱셈의 정의를 통해 거듭제곱의 정의도 알아보았다.

- 참고 링크들
  - 덧셈: https://en.wikipedia.org/wiki/Addition
  - 곱셈: https://en.wikipedia.org/wiki/Multiplication
  - 거듭제곱: https://en.wikipedia.org/wiki/Exponentiation

항등원(identity element)

특히 덧셈이나 곱셈이 많이 일어나는 컴퓨터 환경에서는 항등원의 존재에 대해서 명확하게 인식하고 있어야 한다.
어떤수(예를 들어, a라 하자.)를 해당 연산을 했을 때, 항상 어떤수(a)가 나온다는 것을 보장하는 숫자를 항등원이라고 한다.

- 덧셈의 항등원은 0이다. 어떤수에 0을 아무리 더해도 언제나 그 결과는 어떤수다.
- 곱셈의 항등원은 1이다. 어떤수에 1을 아무리 곱해도 언제나 그 결과는 어떤수다.
- 덧셈과 곱셈 둘 다 교환법칙이 성립하기 어떤수와 더하고자 하는 수, 또는 곱하고자 하는 수를 덧셈 뒤에 쓰든, 앞에 쓰건 상관없고, 곱셈도 마찬가지다.
- 하지만 위에서 언급했듯이 거듭제곱은 좀 다르다. 항상 밑을 지수번 만큼 반복해서 곱한다는 것을 알아야 한다.(교환법칙이 성립하지 않는다.)

- 참고링크: https://en.wikipedia.org/wiki/Identity_element

첨수 표기법을 통한 수열의 합과 곱 그리고 첨수 표기법을 통한 임의의 집합들의 교집합과 합집합

먼저 첨수란 무엇인지 알아보자. > https://en.wikipedia.org/wiki/Indexed_family

표기법(이때 a, b, i는 0을 포함하는 자연수)	이름	설명 및 참고 링크
Σ from i = a	해당 문자는 시그마(Sigma)라고 읽는다.	주어진 수열을 모두 합하고 이를 결과로 내놓는다. 즉, 수열의 합을 뜻한다. link: https://en.wikipedia.org/wiki/Summation#Capital-sigma_notation

to b	Capital-sigma notation	
$\prod$ from i = a to b	해당 문자는 파이(PI)라고 읽는다. Capital Pi notation	주어진 수열을 모두 곱하고 이를 결과로 내놓는다. 즉, 수열의 곱을 뜻한다. link: https://en.wikipedia.org/wiki/Multiplication#Products_of_sequences
$\bigcup$ from i = a to b	해당 문자는 유니온(Union)라고 읽는다. Arbitrary unions	주어진 집합을 모두 합집합(Union)해서 얻은 새로운 집합을 결과로 내놓는다. link: https://en.wikipedia.org/wiki/Union_(set_theory)#Arbitrary_unions
$\bigcap$ from i = a to b	해당 문자는 인터섹션(Intersection)라고 읽는다. Arbitrary Intersections	주어진 집합을 모두 교집합(Intersection)해서 얻은 새로운 집합을 결과로 내놓는다. link: https://en.wikipedia.org/wiki/Intersection_(set_theory)#Arbitrary_intersections