퍼온 곳(풀버전 있는 곳): https://seamless.tistory.com/96
이 글은 마이크로소프트웨어 2002년 기사 중에서 알고리즘 부분을 발췌한 것입니다.

프로그래밍과 알고리즘 공부 방법 - 김창준

"*우리 프로그래머들은 항상 공부해야 합니다. 우리는 지식을 중요하게 여깁니다. 하지만 지식에 대한 지식, 즉 내가 그 지식을 얻은 과정이나 방법 같은 것은 소홀히 여기기 쉽습니다. 따라서 지식의 축적과 공유는 있어도 방법론의 축적과 공유는 매우 드문 편입니다. 저는 평소에 이런 생각에서 학교 후배들을 위해 제 자신의 공부 경험을 짬짬이 글로 옮겨놓았고, 이번 기회에 그 글들을 취합, 정리하게 되었습니다. 그 결실이 바로 이 글입니다.*"

이 글은 공부하는 방법과 과정에 관한 글입니다. 이 글은 제가 공부한 성공/실패 경험을 기본 토대로 했고, 지난 몇 년간 주변에서 저보다 먼저 공부한 사람들의 경험을 관찰, 분석한 것에 제가 다시 직접 실험한 것과 그밖에 오랫동안 꾸준히 모아온 자료들을 더했습니다. '만약 다시 공부한다면' 저는 이렇게 공부할 것입니다.

부디 독자 제현께서 이 글을 씨앗으로 삼아 자신만의 나무를 키우고 거기서 열매를 얻고, 또 그 열매의 씨앗이 다시 누군가에게 전해질 수 있다면 더 이상 바랄 것이 없겠습니다.

이 글은 특정 주제들의 학습/교수법에 대한 문제점과 제가 경험한 좋은 공부법을 소개하는 식으로 구성됐습니다. 여기에 선택된 과목은 리팩토링, 알고리즘·자료구조, 디자인패턴, 익스트림 프로그래밍(Extreme Programming 혹은 XP) 네 가지입니다.

이 네 가지가 선택된 이유는 필자가 관심 있게 공부했던 것이기 때문만은 아니고, 모든 프로그래머에게 어떻게든 널리 도움이 될만한 교양과목이라 생각하여 선택한 것입니다. 그런데 이 네 가지의 순서가 겉보기와는 달리 어떤 단계적 발전을 함의하는 것은 아닙니다. 수신(修身)이 끝나면 더 이상 수신은 하지 않고 제가(齊家)를 한다는 것이 어불성설인 것과 같습니다.

원래는 글 후미에 일반론으로서의 공부 패턴들을 쓰려고 했습니다. 하지만 지면의 제약도 있고, 독자 스스로 이 글에서 그런 패턴을 추출하는 것도 의미가 있을 것이기에 생략했습니다. 그런 일반론이 여기저기 숨어있기 때문에 알고리즘 공부에 나온 방법 대부분이 리팩토링 공부에도 적용할 수 있고, 적용되어야 한다는 점을 꼭 기억해 주셨으면 합니다. 다음에 기회가 닿는다면 제가 평소 사용하는 (컴퓨터) 공부 패턴들을 소개하겠습니다.

알고리즘·자료구조 학습에서의 문제

우리는 알고리즘 카탈로그를 배웁니다. 이미 그러한 해법이 존재하고, 그것이 최고이며, 따라서 그것을 달달 외우고 이해해야 합니다. 좀 똑똑한 친구들은 종종 "이야 이거 정말 기가 막힌 해법이군!"하고 감탄할지도 모릅니다. 대부분의 나머지 학생들은 그 해법을 이해하려고 머리를 쥐어짜고 한참을 씨름한 후에야 어렴풋이 왜 이 해법이 그 문제를 해결하는지 납득하게 됩니다.

그리고는 그 '증명'은 책 속에 덮어두고 까맣게 사라져버립니다. 앞으로는 그냥 '사용'하면 되는 것입니다. 더 많은 대다수의 학생은 이 과정이 무의미하다는 것을 알기 때문에 왜 이 해법이 이 문제를 문제없이 해결하는지의 증명은 간단히 건너뜁니다.

이런 학생들은 이미 주어진 알고리즘을 사용하는 일종의 객관식 혹은 문제 출제자가 존재하는 시험장 상황에서는 뛰어난 성적을 보일 것임은 자명합니다. 하지만 스스로가 문제와 해답을 모두 만들어내야 하는 상황이라면, 또는 해답이 존재하지 않을 가능성이 있는 상황이라면, 혹은 최적해를 구하는 것이 불가능에 가깝다면, 혹은 알고리즘을 완전히 새로 고안해내야 하거나 기존 알고리즘을 변형해야 하는 상황이라면 어떨까요?

교육은 물고기를 잡는 방법을 가르쳐야 합니다. 어떤 알고리즘을 배운다면 그 알고리즘을 고안해낸 사람이 어떤 사고 과정을 거쳐 그 해법에 도달했는지를 구경할 수 있어야 하고, 학생은 각자 스스로만의 해법을 차근차근 '구성'(construct)할 수 있어야 합니다(이를 교육철학에서 구성주의라고 합니다. 교육철학자 삐아제(Jean Piaget)의 제자이자, 마빈 민스키와 함께 MIT 미디어랩의 선구자인 세이머 페퍼트 박사가 주창했습니다). 전문가가 하는 것을 배우지 말고, 그들이 어떻게 전문가가 되었는지를 배우고 흉내 내야 합니다.

결국은 소크라테스적인 대화법입니다. 해답을 가르쳐 주지 않으면서도 초등학교 학생이 자신이 가진 지식만으로 스스로 퀵소트를 유도할 수 있도록 옆에서 도와줄 수 있습니까? 이것이 우리 스스로와 교사들에게 물어야 할 질문입니다.

왜 우리는 학교에서 '프로그래밍을 하는 과정'이나 '디자인 과정'(소프트웨어 공학에서 말하는 개발 프로세스가 아니라 몇 시간이나 몇십 분 단위의, 개인적인 차원의 사고 과정 등을 일컫습니다)을 명시적으로 배운 적이 없을까요? 왜 해답에 이르는 과정을 가르쳐주는 사람이 없나요? 우리가 보는 것은 모조리 이미 훌륭히 완성된, 종적 상태의 결과물로서의 프로그램뿐입니다. 어느 날 문득 하늘에서 완성된 프로그램이 뚝 떨어지는 경우는 없는데 말입니다.

교수가 어떤 알고리즘 문제의 해답을 가르칠 때, "교수님, 교수님께서는 어떤 사고 과정을 거쳐, 그리고 어떤 디자인 과정과 프로그래밍 과정을 거쳐서 그 프로그램을 만드셨습니까?"하고 물어봅시다. 만약 여기에 어떤 체계적인 답변도 할 수 없는 분이라면 그분은 자신의 사고에 대해 '사고'해 본 적이 없거나 문제해결에 어떤 효율적 체계를 갖추지 못한 분이며, 따라서 아직 남을 가르칠 준비가 되어있지 않은 분일 것입니다. 만약 정말 그렇다면 우리는 어떻게 해야 할까요?

자료구조와 알고리즘 공부

제가 생각건대, 교육적인 목적에서는 자료구조나 알고리즘을 처음 공부할 때 우선은 특정 언어로 구현된 것을 보지 않는 것이 좋을 때가 많습니다. 대신 말로 된 설명이나 의사코드(pseudo-code) 등으로 그 개념까지만 이해하는 것이죠. 그 아이디어를 절차형(C, 어셈블리어)이나 함수형(LISP, Scheme, Haskell), 객체지향(자바, 스몰토크) 언어 등으로 직접 구현해 보는 겁니다. 그다음에는 다른 사람이나 다른 책의 코드와 비교합니다. 이 경험을 애초에 박탈당한 사람은 귀중한 배움과 깨달음의 기회를 잃은 셈입니다.

만약 여러 사람이 함께 공부한다면 각자 동일한 아이디어를 같은 언어로 혹은 다른 언어로 어떻게 다르게 표현했는지를 서로 비교해 보면 배우는 것이 무척 많습니다.

우리가 자료구조나 알고리즘을 공부하는 이유는, 특정 '실세계의 문제'를 어떠한 '수학적 아이디어'로 매핑시켜 해결할 수 있는지, 그것이 효율적인지, 또 이를 컴퓨터에 어떻게 효율적으로 구현할 수 있는지 따지고, 그것을 실제로 구현하기 위해서입니다. 따라서 이 과정에 있어 실세계의 문제를 수학 문제로, 그리고 수학적 개념을 프로그래밍 언어로 효율적으로 표현해내는 것은 아주 중요한 능력이 됩니다.

알고리즘 공부에서 중요한 것

개별 알고리즘의 목록을 이해, 암기하며 익히는 것도 중요하지만 더 중요한 것은 다음 네 가지입니다.

1. 알고리즘을 스스로 생각해낼 수 있는 능력
2. 다른 알고리즘과 효율을 비교할 수 있는 능력
3. 알고리즘을 컴퓨터와 다른 사람이 이해할 수 있는 언어로 표현해낼 수 있는 능력
4. 이것의 정상작동(correctness) 여부를 검증해 내는 능력

첫 번째가 제대로 훈련되지 못한 사람은 알고리즘 목록의 스테레오 타입에만 길들여져 있어서 모든 문제를 자신이 아는 알고리즘 목록에 끼워 맞추려고 합니다. 디자인패턴을 잘못 공부한 사람과 비슷합니다. 이런 사람들은 마치 과거에 수학 정석만 수십 번 공부해 문제를 하나 던져주기만 하면, 생각해 보지도 않고 자신이 풀었던 문제들의 패턴 중 가장 비슷한 것 하나를 기계적·무의식적으로 풀어제끼는 문제 풀이 기계와 비슷합니다. 그들에게 도중에 물어보십시오. "너 지금 무슨 문제 풀고 있는 거니?" 열심히 연습장에 뭔가 풀어나가고는 있지만, 그들은 자신이 뭘 풀고 있는지도 제대로 인식하지 못하는 경우가 많습니다.

머리가 푸는 게 아니고 손이 푸는 것이죠. 이렇게 되면 도구에 종속되는 '망치의 오류'에 빠지기 쉽습니다. 새로운 알고리즘을 고안해야 하는 상황에서도 기존 알고리즘에 계속 매달릴 뿐입니다. 알고리즘을 새로 고안해 내건 혹은 기존의 것을 조합하건 스스로 생각해내는 훈련이 필요합니다.

두 번째가 제대로 훈련되지 못한 사람은 일일이 구현해 보고 실험해 봐야만 알고리즘 간의 효율을 비교할 수 있습니다. 특히 자신이 가진 카탈로그를 벗어난 알고리즘을 만나면 이 문제가 생깁니다. 이건 상당한 대가를 치르게 합니다.

세 번째가 제대로 훈련되지 못한 사람은, 문제를 보면 "아, 이건 이렇게 저렇게 해결하면 됩니다" 하는 말은 곧잘 할 수 있지만, 막상 컴퓨터 앞에 앉혀 놓으면 아무것도 하지 못합니다. 심지어 자신이 생각해낸 그 구체적 알고리즘을 남에게 설명해 줄 수는 있지만, 그걸 '컴퓨터에게' 설명하는 데는 실패합니다. 뭔가 생각해낼 수 있다는 것과 그걸 컴퓨터가 이해할 수 있게 설명할 수 있다는 것은 다른 차원의 능력을 필요로 합니다.

네 번째가 제대로 훈련되지 못한 사람은, 알고리즘을 특정 언어로 구현해도, 그것이 옳다는 확신을 할 수 없습니다. 임시변통(ad hoc)의 아슬아슬한 코드가 되거나 이것저것 덧붙인 누더기 코드가 되기 쉽습니다. 이걸 피하려면 두 가지 훈련이 필요합니다.

하나는 수학적·논리학적 증명의 훈련이고, 다른 하나는 테스트 훈련입니다. 전자가 이론적이라면 후자는 실용적인 면이 강합니다. 양자는 상보적인 관계입니다. 특수한 경우들을 개별적으로 테스트해서는 검증해야 할 것이 너무 많고, 또 모든 경우에 대해 확신할 수 없습니다. 테스트가 버그의 부재를 보장할 수는 없습니다. 하지만 수학적 증명을 통하면 그것이 가능합니다. 또, 어떤 경우에는 수학적 증명을 굳이 할 필요 없이 단순히 테스트 케이스 몇 개만으로도 충분히 안정성이 보장되는 경우가 있습니다. 이럴 때는 그냥 테스트만으로 만족할 수 있습니다.

실질적이고 구체적인 문제를 함께 다루라

자료구조와 알고리즘 공부를 할 때에는 가능하면 실질적이고 구체적인 실세계의 문제를 함께 다루는 것이 큰 도움이 됩니다. 모든 학습에 있어 이는 똑같이 적용됩니다. 인류의 지성사를 봐도, 구상(concrete) 다음에 추상(abstract)이 옵니다. 인간 개체 하나의 성장을 봐도 그러합니다. 'be-동사 더하기 to-부정사'가 예정으로 해석될 수 있다는 룰만 외우는 것보다 다양한 예문을 실제 문맥 속에서 여러 번 보는 것이 훨씬 나을 것은 자명합니다. 알고리즘과 자료구조를 공부할 때 여러 친구들과 함께 연습문제(특히 우리가 경험하는 실세계의 대상들과 관련이 있는 것)를 풀어보기도 하고, ACM의 ICPC(International Collegiate Programming Contest: 세계 대학생 프로그래밍 경진대회) 등의 프로그래밍 경진대회 문제 중 해당 알고리즘·자료구조가 사용될 수 있는 문제를 같이 풀어보는 것도 아주 좋습니다. 이게 가능하려면 "이 알고리즘이 쓰이는 문제는 이거다"하고 가이드를 해줄 사람이 있으면 좋겠죠. 이것은 그 구체적 알고리즘·자료구조를 훈련하는 것이고, 이와 동시에 어떤 알고리즘을 써야 할지 선택, 조합하는 것과 새로운 알고리즘을 만들어내는 훈련도 무척 중요합니다.

알고리즘 디자인 과정의 중요성

알고리즘을 좀더 수월하게, 또 잘 만들려면 알고리즘 디자인 과정에 대해 생각해 봐야 합니다. 그냥 밑도 끝도 없이 문제를 쳐다본다고 해서 알고리즘이 튀어나오진 않습니다. 체계적이고 효율적인 접근법을 사용해야 합니다. 대표적인 것으로 다익스트라(E. W. Dijkstra)와 워스(N. Wirth)의 '조금씩 개선하기'(Stepwise Refinement)가 있습니다. 워스의 「Program Development by Stepwise Refinement」(1971, CACM 14.4, http://www.acm.org/classics/dec95)를 꼭 읽어보길 바랍니다. 여기 소개된 조금씩 개선하기는 구조적 프로그래밍에서 핵심적 역할을 했습니다(구조적 프로그래밍을 'goto 문 제거' 정도로 생각하면 안 됩니다). 다익스트라의 「Stepwise Program Construction」 (Selected Writings on Computing: A Personal Perspective, Springer-Verlag, 1982, http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD227.PDF) 추천합니다.

알고리즘 검증은 알고리즘 디자인과 함께 갑니다. 새로운 알고리즘을 고안할 때 검증해 가면서 디자인하기 때문입니다. 물론 가장 큰 역할은 고안이 끝났을 때의 검증입니다. 알고리즘 검증에는 루프 불변식(loop invariant) 같은 것이 아주 유용합니다. 아주 막강한 무기입니다. 익혀 두면 두고두고 가치를 발휘할 것입니다. 맨버(Udi Manber)의 알고리즘 서적(『Introduction to Algorithms: A Creative Approach』)이 알고리즘 검증과 디자인이 함께 진행해 가는 예로 자주 추천됩니다. 많은 계발을 얻을 것입니다. 고전으로는 다익스트라의 『A Discipline of Programming』과 그라이스(Gries)의 『The Science of Programming』이 있습니다. 특히 전자를 추천합니다. 프로그래밍에 대한 관을 뒤흔들어 놓을 것입니다.

알고리즘과 패러다임

알고리즘을 공부하면 큰 줄기들을 알아야 합니다. 개별 테크닉도 중요하지만 '패러다임'이라고 할 만한 것들을 알아야 합니다. 이것에 대해서는 튜링상을 수상한 로버트 플로이드(Robert Floyd)의 튜링상 수상 강연(The Paradigms of Programming, 1978)을 추천합니다. 패러다임을 알아야 알고리즘을 상황에 맞게 마음대로 변통할 수 있습니다. 그리고 자신만의 분류법을 만들어야 합니다. 구체적인 문제들을 케이스-바이-케이스로 여럿 접하는 동안 그냥 지나쳐 버리면 개별자는 영원히 개별자로 남을 뿐입니다. 비슷한 문제들을 서로 묶어서 일반화해야 합니다.

이런 패러다임을 발견하려면 '다시 하기'가 아주 좋습니다. 다른 것들과 마찬가지로, 이 다시 하기는 알고리즘에서만이 아니고 모든 공부에 적용할 수 있습니다. 같은 것을 다시 해보는 것에서 우리는 얼마나 많은 것을 배울 수 있을까요. 대단히 많습니다. 왜 동일한 문제를 여러 번 풀고, 왜 같은 내용의 세미나에 또다시 참석하고, 같은 프로그램을 거듭 작성할까요? 훨씬 더 많이 배울 수 있기 때문입니다. 화술 교육에서는 같은 주제에 대해 한번 말해본 연사와 두 번 말해본 연사는 천지 차이가 있다고 말합니다. 같은 일에 대해 두 번의 기회가 주어지면 두 번째에는 첫 번째보다 잘할 기회가 있습니다. 게다가 첫 번째 경험했던 것을 '터널을 벗어나서' 다소 객관적으로 볼 수 있게 됩니다. 왜 자신이 저번에 이걸 잘 못 했고, 저걸 잘했는지 알게 되고, 어떻게 하면 그걸 더

잘할 수 있을는지 깨닫게 됩니다. 저는 똑같은 문제를 여러 번 풀더라도 매번 조금씩 다른 해답을 얻습니다. 그러면서 정말 엄청나게 많은 것을 배웁니다. '비슷한 문제'를 모두 풀 능력이 생깁니다.

제가 개인적으로 존경하는 전산학자 로버트 플로이드(Robert W. Floyd)는 1978년도 튜링상 강연에서 다음과 같은 말을 합니다.

제가 어려운 알고리즘을 디자인하는 경험을 생각해 볼 때, 제 능력을 넓히는 데 가장 도움이 되는 특정한 테크닉이 있습니다. 어려운 문제를 푼 후에, 저는 그것을 처음부터 완전히 새로 풉니다. 좀 전에 얻은 해법의 통찰(insight)만을 유지하면서 말이죠. 해법이 제가 희망하는 만큼 명료하고 직접적인 것이 될 때까지 반복합니다. 그런 다음, 비슷한 문제들을 공략할 어떤 일반적인 룰을 찾습니다. 아까 주어진 문제를 아예 처음부터 최고로 효율적인 방향에서 접근하도록 이끌어줬을 그런 룰을 찾는 거죠. 많은 경우에 그런 룰은 불변의 가치가 있습니다. … 포트란의 룰은 몇 시간 내에 배울 수 있습니다. 하지만 관련 패러다임은 훨씬 더 많은 시간이 걸립니다. 배우거나(learn) 배운 것을 잊거나(unlearn) 하는 데 모두.

수학자와 프로그래머를 포함한 모든 문제 해결자들의 고전이 된 죠지 폴리야(George Polya)의 『How to Solve it』에는 이런 말이 있습니다:

심지어는 꽤나 훌륭한 학생들도, 문제의 해법을 얻고 논증을 깨끗하게 적은 다음에는 책을 덮어버리고 뭔가 다른 것을 찾는다. 그렇게 하는 동안 그들은 그 노력의 중요하고도 교육적인 측면을 잃어버리게 된다. … 훌륭한 선생은 어떠한 문제이건 간에 완전히 바닥이 드러나는 경우는 없다는 관점을 스스로 이해하고 또 학생들에게 명심시켜야 한다.

저는 ACM의 ICPC 문제 중에 어떤 문제를 이제까지 열 번도 넘게 풀었습니다. 대부분 짝 프로그래밍이나 세미나를 통해 프로그래밍 시연을 했던 것인데, 제 세미나에 여러 번 참석한 분이 농담조로 웃으며 물었습니다. "신기해요. 창준 씨는 그 문제를 풀 때마다 다른 프로그램을 짜는 것 같아요. 설마 준비를 안 해 와서 그냥 내키는 대로 하는 건 아니죠?" 저는 카오스 시스템과 비슷하게 초기치 민감도가 프로그래밍에도 작용하는 것 같다고 대답했습니다. 저 스스로 다른 해법을 시도하고 싶은 마음이 있으면 출발이 조금 다르고, 또 거기서 나오는 진행 방향도 다르게 됩니다. 그런데 중요한 것은 이렇게 같은 문제를 매번 다르게 풀면서 배우는 것이 엄청나게 많다는 점입니다. 저는 매번, 전보다 개선할 것을 찾아내게 되고, 또 새로운 것을 배웁니다. 마치 마르지 않는 샘물처럼 계속 생각할 거리를 준다는 점이 참 놀랍습니다.

알고리즘 개론 교재로는 CLR(Introduction to Algorithms, Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest)을 추천합니다. 이와 함께 혹은 이 책을 읽기 전에 존 벤틀리(Jon Bentley)의 『Programming Pearls』도 강력 추천합니다. 세계적인 짱짱한 프로그래머와 전산학자들이 함께 꼽은 위대한 책 목록에서 몇 손가락 안에 드는 책입니다. 아직 이 책을 본 적이 없는 사람은 축하합니다. 아마 몇 주간은 감동 속에 하루하루를 보내게 될 겁니다.

-- 이하, 리팩토링, 디자인패턴, XP에 대한 이야기는 생략합니다 --

Refactor Me

이 글에 소개된 제 공부론은 어찌 보면 상당히 진부해 보이기도 할 것입니다. 하지만 저는 이런 상식적이고 일상적이며 심지어는 소소해 보이는 것들에서 많은 감동을 받아왔습니다. 이 글도 사실 제 감동의 개인사입니다. 저는 "만약 오늘 어떤 것에라도 감동한 것이 없었다면, 오늘은 뭔가 잘못 산 것이다"라는 신조를 갖고 있습니다. 그것이 컴퓨터이건 대화이건 상관없이 말이죠. 저는 날마다 감동하며 살려고 노력합니다. 그러나 이 감동에 뭔가 꼭 대단한 것이 필요한 것은 아닙니다. 저는 오히려 감동 받기 위해 스스로 대단해져야 할 필요를 느끼기도 합니다. '감동'이라는 것은 주어지는 것이 아닙니다. 나와 타자가 공조하여 만드는 대화입니다.

감동해야 체득할 수 있다고 생각합니다. 그리고 이 감동은 개인적 삶 속에서 자기가, 자신의 몸으로, 직접 얻는 것입니다. 工夫 열심히 합시다.