9. 기타 알고리즘

2010 데이터로 표현하는 세상 요약본
고려대학교 김현철 교수
hkim64@gmail.com

More algorithms

같은 문제에 대하여 알고리즘은 여러 가지 방법으로 만들어 질 수 있다. 우리가 지금까지 학습한 여러 기법들이 어떻게 문제를 해결하는 알고리즘에 적용될 수 있는지 다음 두 가지 문제를 가지고 살펴 보도록 하자.

문제1] 최적 값을 찾는 문제
최적화(optimization): the problems where we try to find the best value for a parameter in some situation.

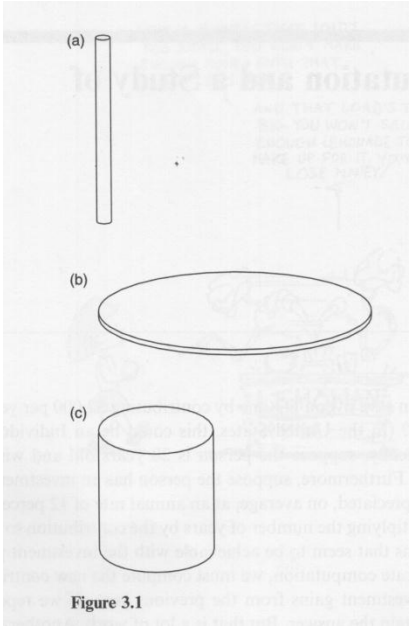


Figure 3.1

알루미늄 판이 정확하게 1000cm^{2}를 사용하여 깡통을 만들려고 한다. 1000cm^{2}를 사용하여 위의 그림과 같이 여러 가지 모양의 깡통을 만들 수가 있으나, 우리는 같은 양의 알루미늄을 사용하여 가장 부피가 큰 깡통을 만들어야 한다. 가장 큰 부피의 깡통을 만들기 위하여서는 깡통의 반지름과 높이를 어떻게 정하여야 할까.

[1분간 당신의 아이디어를 정리하여 보시오]

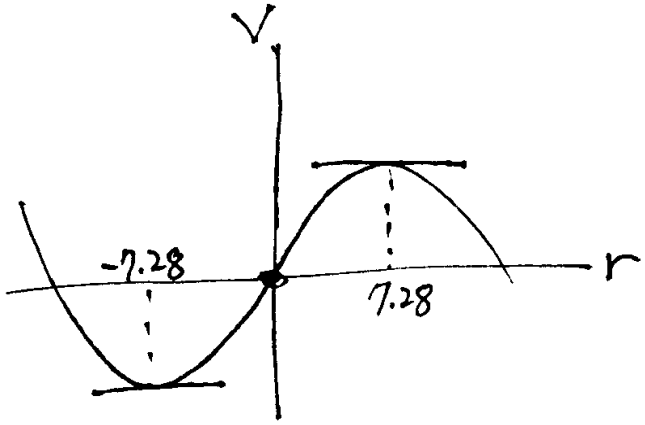
먼저 우리가 알고 있는 수학의 지식을 사용하여 위의 문제를 추상화하여 보자.
깡통은 원형 모양의 뚜껑 2개와 직사각형 모양의 몸통 하나로 이루어져 있다. 뚜껑의 반지름을 r이라고 하고, 몸통의 높이를 h라고 하면, 하나의 뚜껑 면적은 pi*r^2 이고 그 뚜껑의 둘레길이는 2*pi*r 이 된다. 따라서 면적은 2*pi*r^2 + 2*pi*r*h 가 되고 이것은 1000이다.
즉, 1000 = 2*pi*r^2 + 2*pi*r*h
즉, 500= pi*r^2 + pi*r*h (식1)
그리고 부피 V= pi*r^2*h 가 된다. (식2)

우리는 부피 V가 최대가 되게 하는 r과 h를 구하고자 하는 것이다.

Parameter의 개수를 줄여서 문제를 좀 더 간단하게 만들어 보자.
(식1)을 h에 대하여 풀면, h = (500- pi*r^2)/ (pi*r) 이 된다.
그리고 이 h를 (식2)에 대입하면
V = 500r - pi*r^3 이 된다. (식3)

자 이제 부피 V는 하나의 parameter r에 대한 식으로 바뀌었다. 이제 V를 최대로 하게 하는 r를 구해보자. (그림에서도 알 수 있듯이, r이 너무 커도 부피는 최소가 되고 r이 너무 작아도 부피는 최소가 된다.)

[고등학교 수학에서 미적분을 배운 학생은 수학적으로 바로 답을 구할 수 있을 것이다]
즉, V의 식에 미분을 취하고 0으로 놓으면, V를 최대 혹은 최소로 하게 하는 r을 구할 수 있다.
즉, 0 = 500 - 3*pi*r^2
즉, 3*pi*r^2 = 500
즉, r=+- 500/(3*pi) = +- 7.28
즉, r=7.28 일 때 V가 최대가 됨을 알 수 있다.



[미분을 모른다고 고민하지 말라. 우리는 알고리즘으로 해결할 것이다]

V = 500r + pi*r^3 에서
r의 범위는 정수로 1에서 12까지라고 하자. (실제로 r이 12가 넘어가면 알루미늄 면적 1000cm^{2}를 넘어가 버린다. 편의상 r은 정수만 사용하기로 하자.

〈〈12개의 candidate중에서 하나를 고르는 문제로 정의 된 것이다. 그 12개의 candidates가 어떠한 구조를 가지고 자리잡고 있으면 반복적 알고리즘을 통하여 구할 수 있다.〉〉

자 우리의 전략을 세워보자!
그러면 우리가 해 볼 수 있는 가장 쉬운 방법은
r을 1부터 12까지 하나씩 증가하면서 V를 각각 계산하고, 그 중에서 가장 큰 V값을 만든 r을 찾으면 된다. (간단! 이것은 이전에 수업시간에 했던 array에서 max값을 찾는 문제와 같다.)

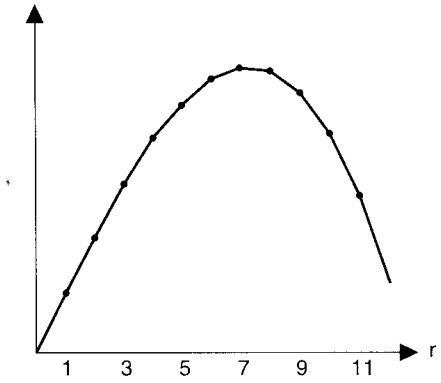
알고리즘을 만들어 보면,

```
MaximumVolume( ) {

maxV=0;
maxR=0;
pi=3.14159
for (r=1; r<=12; r=r+1) {
  V=500r + pi*r^3;
  If (V>maxV) then {
      maxV=V;
      maxR=r;
      }
return maxR;
}
```

모든 r에 대한 V값을 그림으로 나타내면 다음과 같다.

r	V
1	497
2	975
3	1415
4	1799
5	2107
6	2321
7	2422
8	2392
9	2210
10	1858
11	1319



즉 r=7 일 때 V가 최대값이 됨을 알 수 있다.

만약, 우리가 이렇게 V가 r=1..12의 범위 내에서 간단한 curve(즉 올라갔다가 내려가는 curve)를 나타내는 것을 이미 알고 있다면 정상까지 올라 갔다가, 다시 내려가기 시작하면 그 다음은 계산할 필요가 없을 것이다. 그래서 다음과 같은 "전략"을 짠 사람도 있을 것이다.
즉, r을 제일 적은 값에서 시작하여 계속 증가시켜 나가면서 V를 계산하여 현재의 V값과 바로 직전의 V값을 비교한다. 어느 순간에 현재의 V값이 직전의 V값보다 작아진 순간을 만나면, 그것은 이제부터 V가 하향하는 것이므로 바로 직전의 V가 최대 V값이 되는 것이고, 그 최대 V값의 r값을 return하면 된다.

이 전략을 알고리즘으로 만들어 보자.

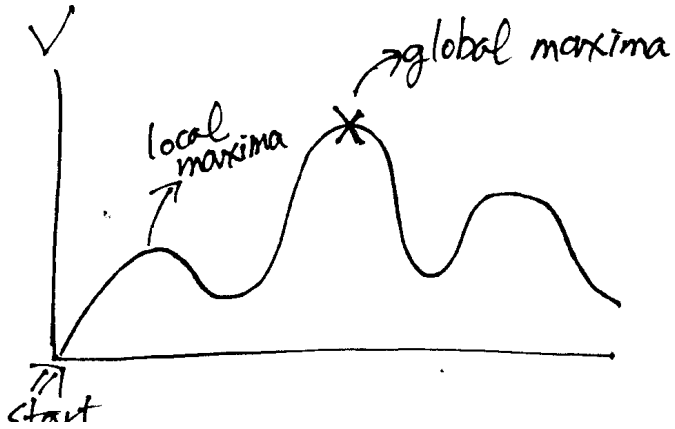
```
MaximumVolume2( ) {

prevV=0;
pi=3.14159
r=1;
V=500r + pi*r^3;
while (V>=prevV) {
    r=r+1;
    prevV=V;
    V=500r + pi*r^3;
}
return r-1;                    /* maximum V는 prevV 가 된다 */
}
```

조금 더, efficient 해졌는가?

이 두 번째 전략을 조금 더 자세히 살펴본다면, 지금보다 바로 옆을 바라봐서 조금 더 나으면, 그리로 이동하고, 그 곳에서 다시 바로 옆을 봐서 조금 더 나으면 그리로 이동하는 것을 반복하다가, 바로 옆을 바라봐도 지금보다 더 좋은 곳이 없으면, 지금의 그곳이 제일 좋은 곳인 줄 알고 그곳에 자리잡는다.
이러한 전략을 greedy algorithm이라고 하며 (탐욕적 알고리즘, 욕심쟁이 알고리즘), hill-climbing algorithm이라고 한다.
A greedy algorithm always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution.

하지만, curve가 위와 같이 간단하고 예상할 수 있는 경우에는 이 "전략"이 유효하지만, 만약에 curve가 복잡하고 그 모양을 우리가 미리 예상하지 못한다면 위와 같은 방법은 틀린 방법이 된다. 예를 들어, curve가 다음과 같이 생겼다고 가정해보자.



우리가 사용했던 "욕심쟁이 전략"(greedy algorithm)은 현재의 위치에서 바로 옆만 보면서 더 좋은 곳으로 가다가 "local minima"에서 멈추게 된다. 그리고 그는 그것이 가장 좋은 solution이라고 생각하고 남은 인생을 마치게 된다. Global maxima가 있을 줄 어찌 알까. **이것은 우리의 인생과 같다**. 인생을 greedy전략으로 살면 그렇게 불쌍한 인생을 살게 된다. 멀리 내다보라. Local maxima에서도 global maxima가 있음을 알고 계속 움직여야 한다. Global을 위해 local을 떠날 때는 현재의 local maxima로서 누렸던 기득권을 잃어야 하는 "아픔"이 있다. 손해가 있다. 당장은 손해를 보는 것이지만, 결국은 마지막에는 더 많은 것을 얻고 정상에 서게 되는 것이다. Greedy한 인생을 살지 말라.

자, 이제 global maxima를 찾을 전략을 세워보자. 그리고 그 알고리즘을 만들어 보자.
제일 저음의 전략대로 r값을 차례대로 하나씩 증가시켜 나가면서 끝까지 가보고 그 다음에 인생을 뒤돌아 보면서 그때 그것이 제일 좋았던 것이라고 말할 텐가. (결혼 배우자도 그렇게 정할 것인가?) 끝까지 가보기도 전에 당신의 인생은 끝날 것이다. 인생은 짧다. 그러면 어떤 전략을 세울 것인가.

[정리]
우리는 알고리즘을 설계하는 몇 가지 예를 살펴보았다.
(1) 수학적 추상화를 통하여, 아주 간단한 수학공식을 만들어 계산하는 것.
(2) 컴퓨터의 빠른 무식한 힘만 믿고, "반복적 계산의 힘"으로 밀어 붙이는 것. Brute Force 방법이라고 한다. 우리가 배운 procedural한 step by step에 의하여 세상의 문제를 해결하려고 한다. 하지만, 세상에는 그런 방법으로 해결하지 못하는 많은 문제들이 있다. Ex) Traveling Salesman Problem. NP 문제. NP complete 문제 발생 가능.
(3) 문제 도메인에 대하여 약간 알고 있는 지식(혹은 힌트)을 이용하여, computing의 시간을 줄이는 것. Heuristic algorithm 이라고 한다.
(4) Randomness를 사용하기도 한다.

.끝.