7. 알고리즘과 데이터 구조

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

우리는 이전 단원에서 input을 output으로 변환 시키는 과정으로서 알고리즘을 설명하였다. 만약에 데이터가 매우 산만하게 흩어져 있다면, 어질러져 있다면, 그 데이터의 처리를 알고리즘에서 효율적으로 기술하기 힘들 수가 있다. 반면에, 데이터를 의도하는 어떠한 형태로 구조화(structured)시켜놓으면 알고리즘을 효율적으로 만들어 낼 수가 있다. 데이터를 구조화 시켜 놓음으로써 데이터의 효율적인 접근방법과 저장방법을 갖게 됨으로써 효율적인 알고리즘을 기술할 수 있게 한다.

구조화시켜 놓으면 관리와 통제가 쉬워진다.

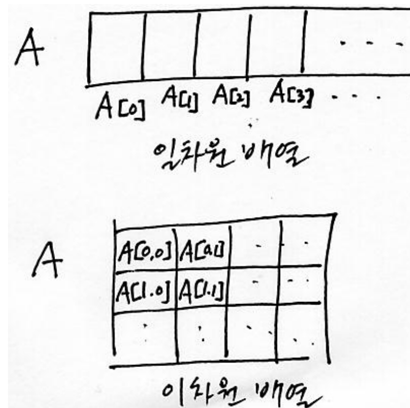
예를 들어, 한 학교를 생각해보자. 학교에는 수천 명의 학생들이 있다 (데이터들!!). 학교는 교실이나 실습실과 같은 한정된 학교시설 내에서 효과적인 수업 계획, 시간표 관리, 성적처리, 학생관리, 입학관리, 졸업지도 등 많은 업무를 수행하여야 한다 (알고리즘들!!). 그를 위하여 '개념적'으로 수천 명의 학생들을 효과적으로 '구조화' 하여 그러한 업무들을 효율적이고 빠르게 수행할 수 있다. 그러한 구조들의 예로서, 학생들에게 학생번호와 반, 학년을 부여하고, 혹은 과목별 성적에 따라 등급을 나누어 관리하기도 하고, 교실에서는 키 순서대로 혹은 이름 순서대로 자기 자리를 정하여 앉게 한다던가 등의 기본적인 구조 체계를 들 수 있다. 어떠한 구조를 사용하느냐에 따라서 앞에서 언급한 학교업무를 얼마나 효율적으로 수행할 수 있는가에 영향을 줄 수가 있다.

우리는 많은 데이터를 computing하고 처리하고자 한다. 이들 데이터를 어떤 구조로 관리하느냐에 따라서 그 데이터를 사용하는 알고리즘이나 프로그램이 얼마나 효율적으로 일을 수행할 수 있느냐가 결정될 수 있다고 할 수 있다. 많은 데이터구조들이 그 동안 연구되어 제공되고 있어서 어떠한 알고리즘개발이나 프로그래밍을 할 때 적절한 데이터구조들을 선택하여 사용할 수 있도록 하고, 혹은 적합한 새로운 데이터구조를 만들어 내어 사용할 수 있도록 할 수 있다.

Array (배열)

여기서는 가장 간단한 형태의 자료구조(데이터구조; data structure)인 array(배열)에 대하여 살펴보도록 하자.

일반적으로 여러 개의 데이터를 기억시키기 위해서는 각각의 데이터에 대한 변수 이름을 필요로 할뿐만 아니라, 이를 위한 많은 기억 장소를 필요로 한다. 따라서 같은 형태의 데이터 집단을 처리하기 위한 경우에는 배열(Array)을 사용하여 처리 방법을 간소화할 수 있을 뿐만 아니라 기억 장소의 낭비도 막을 수 있다.



같은 종류의 데이터들이 많이 있고, 그들을 반복해서 사용해야 하는 경우에는 배열을 사용하면 편리하다. 그들을 모두 배열에 집어 넣고, 배열의 이름과 index만 사용하여 그 데이터에 접근할 수 있다.

우리가 일상에서 사용하고 있는 많은 경우가 바로 이 array(배열)을 사용하고 있는 경우이다. 예를 들어, 한 반의 학생들에게 번호를 매기게 되면, 그 반의 학생들은 1번부터 40번까지 차례대로 번호가 매겨지게 된다. 모든 학생은(데이터는) 고유한 번호를 가지게 된다. 선생님은 그 번호를 가지고 여러 가지 업무(알고리즘!)를 볼 수 있게 된다. 이때 이와 같이 번호를 가지고 반 학생들을 "구조화"시키면 이것은 일차원 배열을 사용한 것이며, 각 학생들에게 부여된 번호는 그 배열의 index가 된다. 반 이름이 "진달래반"이라고 한다면, 그 반 학생들은 일차원 배열을 사용하여 "진달래[1], 진달래[2], 진달래[3],.., 진달래[40]" 과 같이 구조화 될 수 있다.

자 그럼 이번에는, 삼학년에는 7개의 반이 있다고 가정해보자. 한 반에 40명이면 280명의 학생이 있을 것이다. 삼학년이라는 이름의 일차원 배열을 사용하면 "삼학년[1], 삼학년[2], …, 삼학년[280]"과 같이 구조화 할 수 있을 것이다. 좀더 효율적으로 구조화하려면 각 반의 번호와 그 반에서의 학생 번호로 indexing하면 될 것이다. 그러면 삼학년[반번호, 반에서의 학생번호]의 형태로 표현할 수 있다. 두 가지 index가 사용되므로, 이것을 이차원 배열이라고 부른다. "삼학년[1,1], 삼학년[1,2], …, 삼학년[1,40], 삼학년[2,1], …, 삼학년[7,40]" 과 같이 모든 학생들이 indexing 수 있으며, 이것이 효율적인 업무처리 (즉, 알고리즘!)에 더 적합한 구조일 수 있다. 더 나아가, 그 학교 모든 학생을 포함하는 데이터 구조를 만들려면, index를 하나 더 사용하여, 즉 "학년"이라는 index를 더 사용하여 삼차원 배열로 구조화 할 수도 있다. 그렇게 되면, "옥천고등학교[학년, 그 학년에서의 반 번호, 그 반에서의 학생번호]"에 의해 그 학교의 모든 학생(즉, 데이터)들을 indexing할 수 있게 된다.

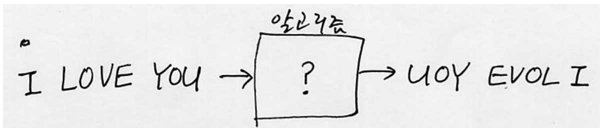
Array(배열)은 데이터를 집어 넣는 하나의 "틀"이라고 (구조의 틀이라고) 생각하자.

[문장을 reverse시키는 문제]

"어떠한 문장이 있을 때, 그 문장의 spell을 거꾸로 보여주는 알고리즘을 만드시오"

음….

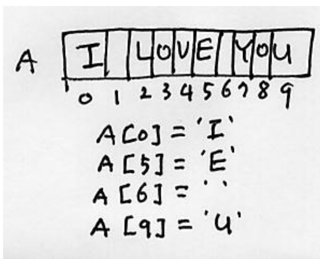
예를 들어, 문장 "*I LOVE YOU*"가 input으로 들어갔을 때 "*UOY EVOL I*"로 출력시키는 알고리즘을 생각해보자.



배열을 사용하지 않았을 경우, 한번 알고리즘을 만들어 보시오. 이 알고리즘은 "어떠한 문장이 들어오더라도 상관없이" 모두 그 문장을 거꾸로 보여줘야 한다. 이러한 일반화된 알고리즘을 데이터 구조 없이 만드는 것은 거의 불가능하다.

배열을 사용했다고 한다면,

Input 문장이 배열(array)에 저장 되어 있다고 가정을 하자. 즉 배열이라는 "구조" A에 input data 가 있다고 가정하자.



이렇게 되면 알고리즘을 기술하기가 좀 더 편리해진다. 먼저 생각나는 대로 알고리즘을 가볍게 기술을 해보자.

① 배열 A의 크기와 같은 크기의 배열 B를 만들고 모두 빈칸으로 채운다.
   배열 A의 크기를 ASize라고 하기로 하자.
② 배열 A의 첫 칸(즉, A[0])에서 시작하여 제일 끝 칸까지 반복하여 다음을 수행한다.
      ②-1 A[칸번호] 데이터를 B[Asize – A의_칸번호]에 넣는다.
③ 배열 B를 출력한다.

이 알고리즘을 좀더 컴퓨터 명확하게 자세하게 표현하면 다음과 같이 기술할 수 있다.

```
Input: array A에 문장의 각 문자가 indexing되어 들어 있다.
Output: array B 에 A에 있는 문자들이 reversed된 순서로 들어 있다.
Algorithm:
Reverse(A) {                                    〈〈알고리즘의_이름(input의_이름)〉〉
  ASize= array A의 size;               〈〈배열 A의 size를 ASize라고 이름 붙이자〉〉
  Create an array B of size ASize;     〈〈배열A와 같은 크기의 배열B를 만들자〉〉
  i=0;                                 〈〈index번호는 i라고 하고 i는 0부터 시작하자〉〉
   while (i 〈 ASize) {                          〈〈index번호 i가 ASize보다 작으면 {과}사이의 명령어들을
                                                   반복적으로 수행하자〉〉
      B[ASize-1-i] = A[i];             〈〈A[i]에 있는 데이터를 B[ASize-1-i]로 옮겨 놓자.〉〉
      i=i+1;                                     〈〈index번호 i의 값을 1 증가시키자.〉〉
      }
  return B;                                      〈〈배열 B를 출력시키자〉〉
}
```

여기서 A와 B는 배열의 이름이고, ASize와 i는 변수(variable)이다.

혹은 다음과 같이 기술할 수도 있다.

```
Reverse(A) {
  ASize = array A의 size;
  i=0;
  while (i 〈 ASize/2) {
     swap(A[i], A[ASize-1-i]);
     i=i+1;
    }
  return A;
  }
```

여기서 우리는 swap( ) 이라는 함수 알고리즘을 다시 정의 해야 한다. 이 swap이라는 알고리즘은 위와 같이 두 개의 데이터를 input으로 받아서 그 값을 맞바꾸는 일을 수행하는 것이라고 하자. (함수에 대한 것은 다음 단원에서 설명될 것임.)

자료구조(data structure)의 선언, 그리고 초기화

앞의 예에서 우리는 array A에 이미 데이터들이 저장되어 있는 것을 가정했다. 실제로는, 데이터 구조 자체도 알고리즘에서 선언을 해줘야 한다. 즉, "*이 알고리즘에서는 array A라는 것을 사용하도록 하겠다*" 라고 선언을 해줘야 한다는 것이다. 그리고 array인 경우에는 그 size도 미리 지정해두어야 한다.

앞서 언급되었듯이 배열(array)는 그 배열 안의 모든 데이터들은 모두 같은 타입(유형)이어야 한다. 즉, 모두 같이 자연수(integer)이거나, 모두 같이 실수(floating-point)이거나, 혹은 모두 같이 문자(character)이거나 하여야 한다. 그래야만 그 자료구조를 사용하는 알고리즘이 복잡해 지지 않기 때문이다. 앞서 예를 든 학교에서의 비유에서도 진달래[i]하는 배열에는 그 반의 학생들만 들어가야지, 진달래[5]에는 학생이, 진달래[8]에는 빗자루가, 진달래[15]에는 담임선생님이 들어가 있다면, 원래 그 자료

구조를 사용하려는 목적, 즉 효율적인 학급학생 관리 업무 (즉, 알고리즘)이 힘들어 질 수 있기 때문이다.

자, 이제 배열을 선언해보자

```
integer A[5];              <<배열 A에 정수 데이터가 들어가며 5칸이 정해져 있다>>
integer StudentId[20];     <<배열 StudentID에 정수 데이터가 들어가며 20칸이 정해져 있다>>
float A[8];
float weight[31];          <<배열 weight에 실수 데이터가 들어가며 31칸이 정해져 있다>>
character A[5];
character Name[25];              <<배열 Name에 문자 데이터가 들어가며 25칸이 정해져 있다>>
string StudentName[4];     <<배열 StudentName에 문장 데이터가 들어가며 4칸이 정해져 있다>>
```

만약에 크기를 처음에 지정해두고 싶지 않다면 그냥 그것을 빈칸으로 두는 것으로 하자. 예를 들어 float weight[ ]; 라고 하면 임의의 크기를 말하는 것이라고 생각하기로 하자.

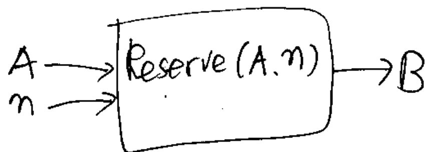
그러면, 배열의 초기값은 어떻게 지정할 것인가?
배열을 처음에 선언할 때, 함께 초기 값을 지정할 수가 있다. 만약 지정하지 않으면 빈칸으로 초기화 된다. 지정 값으로 초기화 하는 예를 보자.

```
integer A[5] = {3, 5, 7, 10, 22};        <<A[0]에는 정수 3이, A[4]에는 정수 22가 들어가 있다>>
float weight[3] = {71.2, 52.3, 63.8};
character MyLastName[3] = {'K', 'I', 'M'};
string StudentName[4] = {"Kim", "Park", "Lee", "Cho"};
```

그러면 다시 "I Love You"문제에 대한 알고리즘을 다시 한번 살펴보자.

```
ILoveYouReverse() {
  Integer A[10] = {'I', ' ', 'L', 'o', 'v', 'e', ' ', 'y', 'o', 'u'};
  Integer B[10];
  ASize= 10;
  i=0;
  while (i < ASize) {
      B[Asize-1-i] = A[i];
      i=i+1;
      }
  return B;
}
```

이번에는 이 문제를 좀 더 일반화 시켜서, 어떠한 문장이 들어오더라도 그 문장을 reverse 시키는 알고리즘으로 표현하여 보자. 이 알고리즘으로의 input은 문장이 들어 있는 배열 A, 그리고 배열 A의 크기 n, 이렇게 둘이고 output은 reversed된 문장이 들어 있는 배열 B라고 하자.




```
Reverse(A, n) {
  Integer B[n];
  Integer i;
  i=0;
  while (i < n) {
```

```
        B[n-1-i] = A[i];
        i=i+1;
        }
    return B;
}
```

끝!!!

데이터를 구조화하는 이유를 다시 한번 정리해보자. 비슷한 특징과 성격을 갖는 data를 일정한 구조로 정의함으로써 data를 효율적으로 다루고 따라서 효율적인 알고리즘을 만들 수가 있다. 문제해결에 있어서 data에 대한 충분한 이해는 필수적이며 data의 구조화는 이러한 충분한 이해를 돕는다.

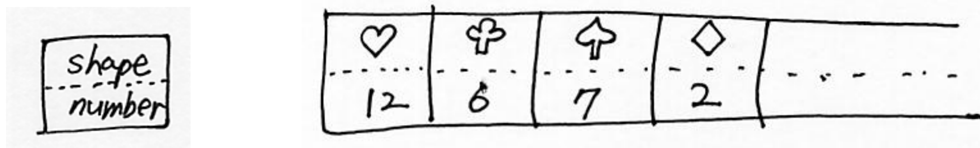
또 다른 종류의 자료구조(data structure)의 예를 보도록 하자.

[카드 게임]
카드게임에 대한 문제가 있다고 하자. 그 문제해결에서 사용하게 될 데이터는 카드에 대한 데이터이다. 카드는 4가지의 그림이 있고 각 그림마다 13개의 다른 번호를 가지고 있다. 따라서 4*13=52장의 카드가 있다 이 52가지의 다른 데이터를 어떻게 구조화 할 수 있을까?

음… 52개의 카드를 배열(array)로 구조화 시키면 되겠구나. 그런데, 배열은 그 안의 모든 데이터들이 모두 같은 타입(즉 integer, float, character, string)를 가져야 한다는 것을 우리는 알고 있다. 어떻게 구조화할까. Card라는 배열을 크기 52로 선언을 하고, 각 칸에는 그 card의 모양과 숫자를 "함께" 쓰면 좋을 텐데.
그럼 그렇게 만들도록 하자! 그런 구조를 만들면 됨!

배열의 각 칸의 원소의 데이터타입을 내가 직접 정의하여 사용해보자!

배열의 한 칸의 데이터 모양이 다음(왼편)과 같이 되게 하면 된다. 그러면 배열의 모양은 아래의 오른편 그림과 같이 될 것이다.



이러한 데이터구조를 record라고 부른다. (A <u>record</u> is a collection of related elements, possibly of different types, having a single name.)

이 레코드의 선언을 어떻게 할 것인가? 컴퓨터 프로그래밍 언어 중에서 많이 사용되는 언어 중의 하나인 "C"언어에서는 다음이 같이 레코드를 선언한다.

```
Struct Card {
        String shape;
        Integer number;
        };
```

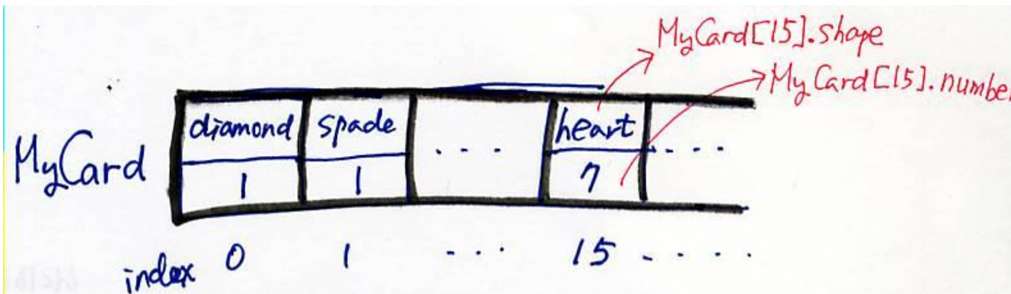
이것을 "구조체"라고 부른다. 즉, 이미 우리가 사용하는 data type인 integer, float, character, string 이외의 다른 data type, 즉 내가 내 맘대로 내가 사용하기 위한 나만의 data type을 만드는 것이다. 그리고 일단 만들면 그것을 마치 다른 data type처럼 사용하기만 하면 되는 것이다.
위에서 Card라는 data type을 만든 것이므로, 우리가 52장의 card를 배열로 선언하고자 한다면 다음과 같이 하면 된다.

Struct Card MyCard[52]; <<배열 MyCard에 52칸을 정해 놓고, 이 배열의 타입은 Card타입으로 한다>>

라고 선언하면 된다.
초기값을 지정하려면 다음과 같이 한다.
Struct Card MyCard[52] = {"spade", 1, "heart", 1, "diamond", 1, "clobber", 1, "spade", 2, … };

알고리즘에서 이 배열을 사용하는 방법은 다음과 같다.
MyCard[7].shape = "diamond"; <<배열 MyCard의 7번째 칸에 shape은 "diamond"이다>>
MyCard[8].number = 11;

즉 그림으로 표시하면 다음과 같이 된다.

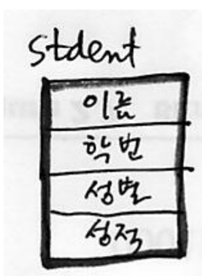


구조체란 data type이 서로 다른 데이터들의 집합을 말한다. 의미상으로 연관성이 있는 여러 가지 자료들을 하나로 묶어 마치 하나의 변수를 사용하듯 활용할 수 있다. (반면에, 배열은 하나의 동일한 자료들의 집합이다)

구조체에 대한 이해를 돕기 위한 예를 하나 더 보도록 하자.

[student의 구조체]
내가 해결해야 하는 문제에는 우리 반 학생들에 대한 data를 사용하여야 한다. 우리 반 학생들은 모두 30명인데, 이들 data를 어떻게 구조화하여야 할까.

"학생"이라는 구조체를 만들자. 학생이라는 변수 속에는 그 학생의 이름, 학번, 성별, 성적 의 data를 포함하도록 하고 싶다. 다음과 같이 구조체를 정의할 수 있다. 즉, student라는 일종의 data type을 만들 수가 있다.




```
Struct student {
        String name;
        Integer StdID;
        Character sex;
        Float gpa;
        }
```

그러면 이제 우리 반 학생들의 배열(array)를 선언할 수 있다.

Struct student MyClassStudent[30];

데이터 구조 -> 좀 더 일반적인 알고리즘

데이터 구조 사용의 장점중의 하나는 데이터 구조를 사용함으로써 데이터와 알고리즘을 분리 시킬 수가 있고, 그럼으로써 좀 더 일반적인 알고리즘을 만들 수가 있다는 것이다. 또한 만들어진 알고리즘을 다른 곳에서 재활용하기가 수월해질 수도 있다. 즉 다시 말하면, 배열과 같은 데이터 구조를 사용하면, 알고리즘은 단지 그 배열의 이름과 index만 가지고 알고리즘을 만들 수가 있

다. 따라서 어떠한 데이터가 들어오더라도 그 데이터를 그 배열 속에만 넣기만 한다면, 알고리즘은 고칠 필요가 없이 새로운 데이터들에 대하여 처리를 할 수가 있을 것이다. 만약 데이터 구조를 사용하지 않는다면 알고리즘은 데이터 자체를 가지고 기술되어야 하며, 데이터가 바뀌면 알고리즘도 다시 새로 기술해야 하는 비효율성을 가지게 될 수가 있다

다음의 문제는 이러한 장점을 잘 보여준다.

[세금 계산해보기]
Input으로 세금이 매겨질 소득 액수가 들어오고, 그 액수에 따라 계산될 세금의 양은 다음과 같은 규칙에 의하여 정해진다고 하자.

① 소득(income)의 처음 $10,000에 대해서는, 세금(tax)은 10%이다.
② $10,000를 초과하는 다음 $10,000에 대해서는 세금은 12%이다.
③ $20,000를 초과하는 다음 $10,000에 대해서는 세금은 15%이다.
④ $30,000를 초과하는 다음 $10,000에 대해서는 세금은 18%이다.
⑤ $40,000를 초과하는 소득에 대해서는 세금은 20%이다.

즉, 다시 이야기를 해보면, 만약에 내 소득이 $64,000이라고 한다면 내가 낼 세금은
($10,000*10%)+($10,000*12%)+($10,000*15%)+($10,000*18%)+($24,000*20%)=$10,300가 된다.

어떤 수입 액수가 input으로 들어오더라도 자동으로 세액을 계산해 줄 수 있는 알고리즘을 만들어 보자.

```
tax = 0;
if (taxable_income == 0) goto EXIT;
if (taxable_income > 10000)
      Tax = tax + 1000;
Else {
      tax = tax + .10 * taxable_income;
      goto EXIT;
}

if (taxable_income > 20000) tax = tax + 1200;
else {
      tax = tax + .12 * (taxable_income - 10000);
      goto EXIT;
}

if (taxable_income > 30000) tax = tax + 1500;
else {
      tax = tax + .15 * (taxable_income - 30000);
      goto EXIT;
}

if (taxable_income < 40000) {
      tax = tax + .18 * (taxable_income - 30000);
      goto EXIT;
} else
      tax = tax + 1800. + .20 * (taxable_income - 40000);
EXIT: ;
```

이 알고리즘은 위의 세금 계산 규칙에 나온 모든 구체적인 액수와 숫자들이 알고리즘 안에 직접 포함되어 기술되어 있다. 이 알고리즘은 불행하게도 매우 비효율적이다. 왜냐하면 우리가 경험하고 있듯이, 세율은 매년 국회에서 새로 정해지고, 만들어지고, 변경되기 때문이다. 그렇게 세율이 변경될 때마다 알고리즘을 매번 직접 고쳐야 한다면 얼마나 비효율적인가?

데이터를 알고리즘과 분리하여 보자. 자주 변경되는 세율은 간단한 형태의 테이블에 적어 놓고, 그 테이블은 데이터 구조로 변경하여 정의하고, 알고리즘에서는 그 데이터 구조만 사용하면 된다. 세율이 변경되어도 그 변경된 값은 데이터 구조에서만 바꿔주면 될 뿐, 알고리즘은 손 댈 필요가 없다.

위의 코드에 나오는 세율계산 방법을 다시 정리하면,

- 수입이 [0 – $10,000]에 속하면 그 액수에 대하여 10% 계산하고
- 수입이 ($10,000 – $20,000]에 속하면 기본(base)으로 $1000을 내고 $10,000넘는 액수에 대해서는 12% 계산하고,
- 수입이 ($20,000 – $30,000]에 속하면 기본(base)으로 $2200을 내고 $20,000넘는 액수에 대해서는 15% 계산하고,
- 수입이 ($30,000 – $40,000]에 속하면 기본(base)으로 $3700을 내고 $30,000넘는 액수에 대해서는 18% 계산하고,
- 수입이 ($40,000 –]에 속하면 기본(base)으로 $5500을 내고 $40,000넘는 액수에 대해서는 15% 계산한다.

이 내용을 테이블로 만들면 다음과 같이 만들 수 있다. 이 테이블에 의하면, 먼저 Bracket이 나오는데 이것은 input수입이 해당되는 칸을 찾기 위한 것이고, 해당 칸을 찾으면 Base의 액수에다가 (input수입 – 해당 bracket 액수)*percent/100 을 더해주면 된다. 예를 들어, input 수입이 $34,000이라면 bracket에서 30,000칸에 해당하고, 따라서 세금은 $3,700 + ($34,000-$30,000)*0.18 이 되는 것이다.

Sample Tax Table

Bracket	Base	Percent
0	0	10
10,000	1000	12
20,000	2200	15
30,000	3700	18
40,000	5500	20

이 테이블은 세 개의 배열, 즉 bracket[ ], base[ ], percent[ ]로 구조화 될 수 있고, 이 데이터구조를 이용하여 알고리즘을 만들면

```
i=0
while (i<5 && income> bracket[i]) {
      i=i+1;
        }
level = i-1;
tax = base[level] + percent[level] * (income – bracket[level]);
```

가 될 수 있다. 예를 가지고 직접 이 알고리즘을 검증해 보시오.
구조화된 데이터를 사용하는 것이 효율적인, 그리고 일반적인 알고리즘을 만드는데 많은 도움이 될 수 있음을 알 수 있다.

이 외에도 많은, 그리고 다양한 데이터 구조가 사용되고 있으며, 여러분도 여러분 나름대로의 창의적인 데이터 구조를 새로 만들어 내어 사용할 수 있다. 서울 지하철 역 지도를 보면 그 수많은 역들의 정보가 매우 정교하게 구조화 됨을 볼 수 있다. 그 구조를 사용하여 우리의 머리 속에서는 많은 알고리즘, 즉 A역에서 B역으로 가는 가장 빠른 방법에 대한 해(solution)을 쉽게 구해 낼 수 있는 것이다. (더 효율적으로 구조화된 지하철 노선도를 생각해 낼 수 있겠는가?)
또한 우리가 문제해결 chapter 에서 본 상태트리, 상태공간, 프레임 등도 모두 효율적인 데이터 구조의 일종이라고 생각해 볼 수 있다.

.끝.