TI DSP, MCU 및 Xilinx Zynq FPGA 프로그래밍 전문가 과정

(알고리즘)

강사 - 이상훈

gcccompil3r@gmail.com

장성환

redmk1025@gmail.com

알고리즘 시작 - 랜덤한 순열 생성

```
8
9   int main(void) {
10
11      int random;
12      int arr[10]; // 0~9
13      srand(time(NULL));
14      for (int i = 0; i < sizeof(arr)/sizeof(int); i++) {
15          random = rand()%10+1;
16          arr[i] = random;
17
18          for (int j = 0; j < i; j++) {
19              if (arr[i] == arr[j]) {
20                  i--;
21                  break;
22              }
23          }
24      }
25      for (int i = 0; i < 10; i++) {
26          printf("%d\n", arr[i]);
27      }
28      return 0;
29  }
30
31
```

해당 코드는 랜덤한 값 (1 ~ 10)을 생성하며, 같은 값이 나오지 않도록 j를 활용하는 형태의 코드이다. 최대 $O(N^2)$ 반복 이므로 효율이 좋지 않다. 위와 같은 코드를 개량하면, 다음과 같다.

```
#define N 20

int main(void) {

    int i, j, d, a[N + 1];
    srand(time(NULL));

    for (i = 1; i <= N; i++) { //1 ~ 20
        a[i] = i; // 1 ~ 20 까지 저장됨 인덱스는 0은 없음.
    }

    for (i = N; i > 1; i--) { // 20 ~ 2
        j = rand() % i + 1; // 1~20 -> 1~19 -> 1~18 -> ....-> 1~2
        d = a[i];
        a[i] = a[j];
        a[j] = d;
    }

    for (i = 1; i <= N; i++) {
        printf("%d\n", a[i]);
    }


    return 0;
}
```

원하는 숫자의 구간을 미리 생성한 뒤에 배열 순서를 바꾸는 방법으로 O(N)으로 바뀐다.

매핑

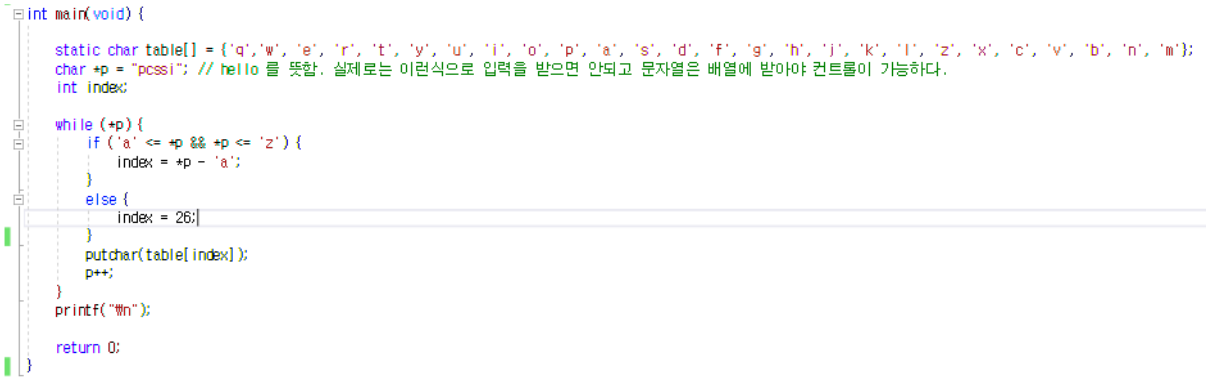
매핑의 의미는 어떤 데이터의 범위를 다른 데이터의 범위로 변환하는 것이다. 예를 들면, 0~100까지의 데이터들을 0~10까지의 값을 가지도록 매핑한다는 것이다.

```c
int main(void) {

    int histo[11] = { 0 };
    static int arr[] = {10,66,46,12,58,100,97,33,4,0,12,56,77,20,91};
    int tmp;

    for (int i = 0; i < sizeof(arr) / sizeof(int); i++) {
        tmp = arr[i] / 10;
        histo[tmp]++;
    }

    for (int i = 0; i < sizeof(histo) / sizeof(int); i++) {
        printf("%3d : %3d\n", i, histo[i]);
    }

    return 0;
}
```

해당 코드는 0~100까지의 데이터를 10으로 나누어 데이터의 범위를 0~10으로 제한하고, 히스토그램 형식으로 만드는 과정이다.

```c
int main(void) {

    char p[]="hello my wolrld"; // hello
    int index=0;

    scanf("%s", p);

    while (p[index]) {

        if (p[index] == 'z') {
            p[index] -= 1;
            continue;
        }
        p[index] += 1;
        index++;
    }
    printf("encrytion str : %s\n",p);

    return 0;
}
```

해당 코드는 시저암호 형식으로 ASCII 코드의 값에 일정한 수를 더한 값으로 암호 문자를 생성한다.



```c
int main(void) {

    static char table[] = {'q','w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', 'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'z', 'x', 'c', 'v', 'b', 'n', 'm'};
    char *p = "pcssi"; // hello 를 뜻함. 실제로는 이런식으로 입력을 받으면 안되고 문자열은 배열에 받아야 컨트롤이 가능하다.
    int index;

    while (*p) {
        if ('a' <= *p && *p <= 'z') {
            index = *p - 'a';
        }
        else {
            index = 26;
        }
        putchar(table[index]);
        p++;
    }
    printf("\n");

    return 0;
}
```

해당 코드는 테이블에 ASCII 코드를 만들어 두고, 입력된 값을 기준으로 시저 암호의 역순 방법을 이용하여 테이블의 값을 추출하여 암호를 해독하는 방법이다.

몬테카를로 (PI 구하기)

몬테카를로 방법은 어떠한 문제를 수치 계산이 아니라 확률(난수)를 활용하여 문제를 해결하는 방법이다.

파이를 구하는 공식은 다음과 같다.

$$\frac{\pi}{4} : 1 = a : a + b \; [\frac{1}{4} \textit{인 원 넓이 대} \; 1 \textit{인 정사각형 넓이} = \textit{원에 발생한 난수} : \textit{전체 발생한 난수}]$$

따라서 $a = \frac{\pi(a+b)}{4}$ *이므로* $\pi = \frac{4*a}{(a+b)}$ 이다.

```
17  int main(void) {
18      double x, y;
19      srand(time(NULL));
20      double b=0;
21      double a=0;
22
23      for (int i = 0; i < 1000000; i++) {
24          x = (double)rand( ) /RAND_MAX;
25          y = (double)rand( ) / RAND_MAX;
26
27          if ((double)sqrt((double)pow(x, 2) + (double)pow(y, 2)) > 1.0000) {
28              b++;
29          }
30          else {
31              a++;
32          }
33      }
34
35      printf("%lf\n", (4*a) / (a+b));
36      return 0;
37  }
```

해당 코드의 계산 값은 3.142064 가 나온다. (물론 실행시마다 값이 다르다. 어느 정도의 정확도를 보장하는 것이고, 완벽한 값을 출력해 주진 않는다.)

원 내부에 발생한 난수의 경우 a의 값을 상승시키고, 원 외부에 발생한 난수의 경우에는 b의 값을 상승시킨다.

이러한 비율을 사용하여 위에 계산한 파이 계산 공식을 활용하여 PI를 출력하게 된다.

순위 매기기

숫자 배열이 저장되어 있고, 같은 크기의 rank 배열을 선언한다.

숫자 배열에서 모두 탐색을 해가면서 자신보다 큰 숫자가 있을경우, 1로 초기화 된 랭크 배열을 +1 씩 해준다. 랭크 배열을 출력시, 해당 숫자의 랭크를 알 수 있다.

```
11  int main(void) {
12
13      static int a[] = { 56,25,67,88,100,61,55,67,76,56 };
14      int rank[sizeof(a) / sizeof(int)];
15      int i, j;
16
17      for (i = 0; i < sizeof(a) / sizeof(int); i++) {
18          rank[i] = 1;
19          for (j = 0; j < sizeof(a) / sizeof(int); j++) {
20              if (a[i] < a[j])
21                  rank[i]++;
22          }
23      }
24
25      for (i = 0; i < sizeof(a) / sizeof(int); i++) {
26          printf("%d\n", rank[i]);
27      }
28
29      return 0;
30  }
31
```

코드는 간단한 형태이다. rank를 1로 초기화 되어있고, 어떠한 i값을 기준으로 다른 숫자 배열이 크면 같은 자리에 있는 rank 배열의 count가 1씩 늘어나게 된다.

따라서 제일 큰 값은 1이 되고 제일 작은 숫자는 숫자 배열의 개수만큼이 된다.

$O(N^2)$ 이므로 효율성이 좋지 않다.

순위 매기기 방식에서 코드를 개량한 버전은 다음과 같다.

순위 매기기+

```c
int main(void) {

    int arr[] = { 11,23,32,44,53,12,34,32,90,100 }; //0~9
    int rank[101] = { 0 };  // 0~100
    int size = sizeof(rank) / sizeof(int); // size = 101
    rank[100] = 1;
    int j=0;

    for (int i = 0; i < sizeof(arr) / sizeof(int); i++) {
        rank[(arr[i])-1] = 1;
    }

    for (int i = 0; i < size-1; i++) { // i = 0 ~ 99
        rank[size - i - 2] += rank[size - i - 1];
    }

    while (1) {
        if (j == 10)
            break;
        printf("%d\n", rank[arr[j]]);
        j++;
    }
    return 0;
}
```

해당 숫자배열의 값에 해당하는 rank의 인덱스의 값을 1로 초기화 한 후,

rank의 index가 큰 수부터 계속 더해간다. 따라서 숫자 배열의 값에 위치한 인덱스부터 +1씩 된다.

간단히 예를 들면 rank가 11개의 배열이고 arr이 {3,5} 라면

rank의 값은 00010100001 에서 33332211111 의 형태가 된다. 따라서 이를 토대로 순위 매기기가 완성된다.

이전 코드와 달리 O(N)이 되어 효율성이 매우 증가 하였다.

점화식 (조합 nCr 구하기)

조합을 구하는 식은 다음과 같다.

nCr = $\frac{n!}{r!(n-r)!}$ (n개에서 r개를 고르는 가짓 수)

```
8   int factorial(int n) {
9       if (n == 1)
10          return 1;
11      return n * factorial(n - 1);
12  }
13
14  int nCr(int n, int r, int(*factorial)(int n)) {
15      int upper;
16      int bottom;
17
18      upper = factorial(n);
19      bottom = factorial(r)*factorial(n - r);
20
21      return upper / bottom;
22  }
23
24  int main(void) {
25
26      int n = 0;
27      int r = 0;
28
29      scanf("%d %d", &n, &r);
30      printf("result is %d₩n", nCr(n, r, factorial));
31
32      return 0;
33  }
```

factorial이라는 함수(n!)를 재귀적으로 구성하고, nCr을 구하는 코드를 함수로 구현하였다.

분모와 분자를 따로 구하여 해당 값을 리턴하는 형식이다. nCr 함수는 위의 조합을 구하는 식과 동일하다는 것을 알 수 있다.

점화식 (다항식 호너법)

호너법이란? $f(x) = a_n * x^n + a_{n-1} * x^{n-1} + \cdots + a_{n-2} * x^{n-2} + a_1 * x^1 + a_0$ 와 같은 형태의 다항식 풀이법을 의미한다.

예를 들어 $f(x) = 3x^2 + 2x + 4$ 라고 할 때,

$f_2 = f_1 x + 4,\ f_1 = f_0 x + 2,\ f_0 = 3$ 이런 형태로 나타 낼 수 있다.

따라서 이를 재귀적으로 구현한다.

```
18  double getRes(int arr[], int size, double x) {
19      if (size == 1)
20          return arr[size-1];
21
22      return getRes(arr,size-1,x)*x+arr[size-1];
23  }
24
25
26  int main(void) {
27
28      int *arr = NULL;
29      int size;
30      double x;
31      double res;
32
33      printf("input size₩n");
34      scanf("%d", &size);
35
36      arr = (int *)malloc(sizeof(int)*size);
37
38      for (int i = 0; i < size; i++) {
39          printf("input h (x의%d승의 항)₩n",size-i-1);
40          scanf("%d", &(arr[i]));
41      }
42      printf("input x₩n");
43      scanf("%lf", &x);
44
45      res = getRes(arr, size, x);
46
47      printf("result is %lf₩n", res);
48
49      return 0;
50  }
51
```

size로 항의 차수가 몇인지 입력하고, 해당하는 배열을 생성한다. 배열의 인덱스는 차수를 나타낸다. 각각의 항을 arr 배열에 삽입 한 뒤에, x의 값을 입력받는다.

해당 x와 arr 배열을 이용하여 getRes 라는 재귀함수를 통하여 값을 출력한다.

최대공약수 (유클리드 호제법)

유클리드 호제법을 사용하여 최대 공약수를 구하는 알고리즘이다.

해당 코드는 다음과 같다.

```c
#include <stdio.h>

int main(void) {

    int m, n;

    scanf("%d %d", &m, &n);
    printf("select number is %d %d\n", m, n);

    while (m != n) {
        if (m > n) {
            m = m - n;
        }
        else {
            n = n - m;
        }
    }

    printf("result is %d\n", m);

    return 0;
}
```

두 값 m과 n을 입력받아 두 수의 최대 공약수를 유클리드 호제법으로 구한다.

m과 n이 같아 질 때까지 m - n 혹은 n - m 을 한다. (m이 큰지 n이 큰지에 따라서 수행한다.)

m과 n이 같아질 때, m을 출력하면 해당 수가 최대 공약수가 된다.

일양성의 검정

선형합동법으로 생성한 일양난수(각 값이 균일한 빈도로 나타나는 난수)를 히스토그램의 높이 및 분산 계산을 통하여 생성된 난수의 분포가 일정한지 검정하는 코드이다.

```
#define N 1000 //난수 발생횟수
#define M 10 //정수 난수의 범위
#define F (N/M) // 기대값
#define SCALE (40.0/F) // 히스토그램의 높이(자동 스케일)

unsigned rndnum = 13;
unsigned irnd(void);
double rnd(void);

int main(void) {
    int i, j, rank, hist[M + 1];
    double e = 0.0;

    for (i = 0; i <= M; i++) //hist[] 초기화 총 11개의 배열
        hist[i] = 0;

    for (i = 0; i < N; i++) { //1 ~M의 난수를 하나 발생
        rank = (int)(M*rnd() + 1);
        hist[rank]++; //rank는 1~10까지의 수 즉, 몇개가 나왔지는 알기 위하여 hist[]를 이용한다.
    }

    for (i = 1; i <= M; i++) {
        printf("%3d:%3d ", i, hist[i]);
        for (j = 0; j < hist[i]*SCALE; j++) { // 히스토그램 표시, SCALE은 *이 너무 많기에 스케일링을 해준 것이다(거진 1/2)
            printf("*");
        }

        printf("₩n");
        /*************************************************************************************************/
        e = e + (double)(hist[i] - F)*(hist[i] - F) / F; //분산 계산
        /*************************************************************************************************/
    }

    printf("variance = %f₩n", e);
    return 0;
}

unsigned irnd(void) {
    rndnum = (rndnum * 109 + 1021) % 32768;
    return rndnum;
}

double rnd(void) {
    return irnd() / 32767.1;

}
```

이전 순위매기기 알고리즘을 차용하였다. hist[]라는 어떠한 값이 몇번 나왔는지 나오는 히스토그램을 생성하고 해당 값을 출력한다.

분산은 다음과 같이 계산된다. $V(X) = E(X^2) - \{E(X^2)\} = \sum_{i=1}^{M} \frac{(f_i - E(X))^2}{E(X)}$ 이다.

따라서 분산식 e = e+(double)(hist[i]-F)* (hist[i]-F) / F; 와 같이 계산 할 수 있다.

Box-Muller의 정규난수 변환

일양난수가 아닌 정규난수(정규분포$N(m,\sigma^2)$를 따르는 난수) 해당 정규분포의 식은 다음과 같다.

$$N(m,\sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-m)^2}{2\sigma^2}}$$

m:평균, σ:표준편차를 나타낸다.

m-σ~m+σ 의 확률은 68.27%, m-2σ~m+2σ 의 확률은 95.45%, m-3σ~m+3σ 의 확률은 99.73%

해당 난수는 두개의 일양난수와 Box-Muller 정규난수 변환공식을 사용한다. 해당 공식은

$x = \sigma\sqrt{-2\log r_1}\cos(2\pi r_2) + m$, $y = \sigma\sqrt{-2\log r_1}\sin(2\pi r_2) + m$ 으로 변환하여 사용한다.

```
5   void brnd(double, double, double *, double *);
6   int main(void) {
7       int i, j, hist[100];
8       double x, y;
9       for (i = 0; i <100; i++) //hist[] 초기화 총 11개의 배열
10          hist[i] = 0;
11      for (i = 0; i < 1000; i++) { //1 ~M의 난수를 하나 발생
12          brnd(2.5, 10.0, &x, &y);
13          hist[(int)x]++;
14          hist[(int)y]++;
15      }
16      for (i = 1; i <= 20; i++) {
17          printf("%3d : ",i);
18          for (j = 0; j < hist[i]/10; j++) { // 히스토그램 표시, SCALE은 *이 너무 많기에 스케일링을 해준 것이다(거진 1/2)
19              printf("*");
20          }
21          printf("\n");
22      }
23      return 0;
24  }
25  void brnd(double sig, double m, double *x, double *y) {
26      double r1, r2;
27      r1 = (double)rand( ) / RAND_MAX;
28      r2 = (double)rand( ) / RAND_MAX;
29      *x = sig * sqrt(-2 * log(r1))*cos(2 * 3.14159*r2) + m;
30      *y = sig * sqrt(-2 * log(r1))*cos(2 * 3.14159*r2) + m;
31  }
32
```

이전 일양성의 검정 코드와 같다.

하지만 난수를 발생시키는 함수가 다르다. 위의 변환 공식을 사용하여 x와 y를 만든다.

분산 sig와 평균 m을 함수에 입력하여 원하는 정규분표를 생성 가능하다.

사다리꼴 공식에 의한 정적분

정적분의 개념은 미소 크기 증가마다의 f(x)를 구하여 미소크기*f(x)를 이용하여 넓이를 무한대로 더하면 원하는 구간의 넓이를 구한다는 것이다.

```c
#include <stdio.h>
#include <math.h>

#define f(x) (sqrt(4-(x)*(x))) //피적분 함수. 구간설정을 잘못할 경우 에러발생 (음수 구간 때문에)

int main(void) {
    int k;
    double a, b, n, h, x, s, sum;

    printf("적분구간 A,B ? ");
    scanf("%lf %lf", &a, &b);

    n = 1000;
    h = (b - a) / n;
    x = a; s = 0;

    for (k = 1; k < n; k++) { // 1 ~ 1000 까지
        x = x + h; //x는 변수 x 미소크기만큼 증가.
        s = s + f(x); //s는 미소구간에 대하여 함수의 f(x)의 합. (각각에 미소구간 h를 곱해야 넓이가 된다.)
    }

    sum = h * ((f(a) + f(b)) / 2 + s); // 초반 f(a)와 f(b)가 빠졌으므로, 추가해 주어야 한다.

    printf("    /%f\n", b);
    printf("    |   sqrt(4-x*x) = %f\n", sum);
    printf("    /%f\n", a);

    return 0;
}
```

각각에 f(x+h) 에 미소구간 h를 곱하는 작업을 한다면, 연산량이 많아져 비 효율적이다.

위의 코드처럼

모든 f(a)+f(a+h)+(fa+2h)+.....+f(b) 를 구한뒤에 미소구간 h를 곱하는 방식이 되어야 연산량을 줄일 수 있다.

테일러 전개 (cos x)

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \pm ? \frac{e_{(k-2)} * x^2}{((k-1)((k-1)+1))!} \pm ? \frac{e_{(k-1)} * x^2}{(k(k+1))!} \pm ? \cdots$$

cos(x)에 대한 테일러 전개는 위의 식과 같다.

위의 식은 무한급수가 되지만, 실제 계산은 유한한 횟수로 제한해야 한다.

k-1 항 까지의 합을 d, k 항 까지의 합을 s라고 할 때, 중단 조건은 다음과 같이 설정한다.

$$\frac{|s-d|}{|d|} < EPS$$

$|s-d|$는 중단 오차, $\frac{|s-d|}{|d|}$는 상대 중단오차라 한다. EPS는 요구되는 정확도에 따라 적절하게 설정한다.

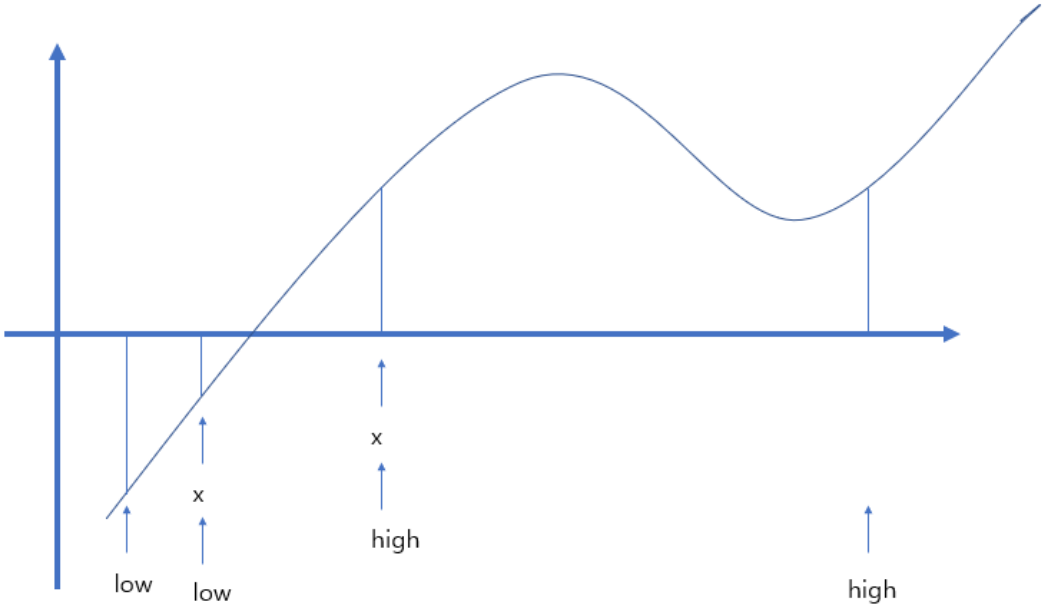
```c
#include <stdio.h>
#include <math.h>

double mycos(double);
int main(void) {
    double x, rd = 3.141592 / 180;
    printf("    x       mycos(x)       cos(x)\n");
    for (x = 0; x <= 180; x = x + 10) {
        printf("%5.1f%14.6f%14.6f\n", x, mycos(x*rd), cos(x*rd));
    }
    return 0;
}
double mycos(double x) {
    double EPS = 1e-08;
    double s = 1.0, e = 1.0, d = 1.0;
    int k;
    x = fmod(x, 2 * 3.141592);
    for (k = 1; k <= 200; k = k + 2) {
        d = s;
        e = -e * x*x / (k*(k + 1));
        s = s + e;
        if (fabs(s - d) < EPS*fabs(d))
            return s;
    }
    return 0;
}
```

해당 식을 점화식 형태로 나타내면 위 코드와 같다.

d와 s를 구하여 요구 정확도를 만족하면 s를 리턴하는 방식이다.

비선형 방정식의 해법 (이분법)

방정식 f(x) = 0 일때의 근 x를 이분법으로 구하는 방법이다.



a 와 b의 구간에서 단 한번 x축과 교차하고 f(a)<0, f(b)>0 이라고 하자.

1. 근의 좌우에 있는 두 점 a, b를 low와 high의 초기값으로 한다.

2. low와 high의 중점 x를 x = (low+high)/2로 구한다.

3. f(x) > 0 이면 근은 x보다 왼쪽에 있으므로 high = x라 해서 상한을 반으로 줄인다.

   f(x) < 0 이면 근은 x보다 오른쪽에 있으므로 low = x라 해서 상한을 반으로 줄인다.

4. f(x) 가 0 or $\frac{|\text{high}-\text{low}|}{|low|} < \text{EPS}$ 될때, 구하는 x값이 근이 된다. EPS는 수렴판정값으로 적절한 정확도를 선택하면 된다.

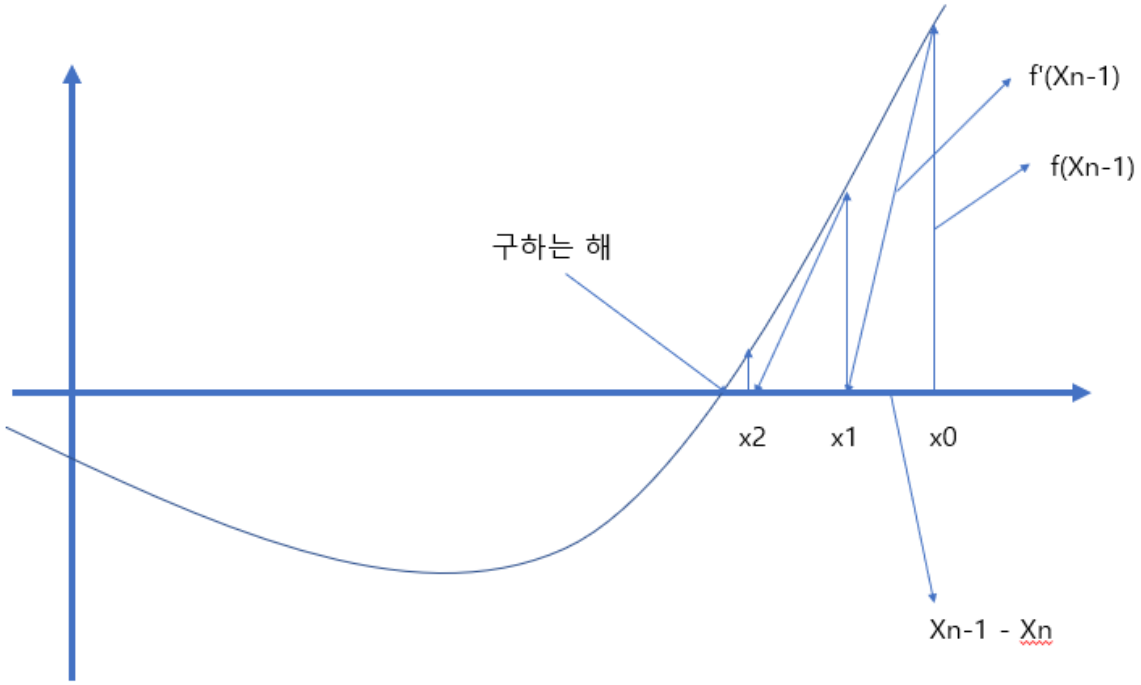
즉, 이분법 방식은 데이터 범위를 반으로 나누어 근이 어디있는지 조사하여 조사 범위를 근을 향하여 점차 좁혀가는 방식이다.

코드는 다음과 같다.

```c
#include <stdio.h>
#include <math.h>
#define f(x) ((x)*(x)*(x)-(x)+1)
#define EPS 1e-8
#define LIMIT 50
int main(void) {
    double low, high, x;
    int k = 1;
    low = -2.0; high = 2.0;
    for (k = 1; k <= LIMIT; k++) {
        x = (low + high) / 2;
        if (f(x) > 0) // f(low)>0 f(high)<0인 반대의 경우도 포함시키려면, f(low)*f(x)로 조건문을 쓴다.
            high = x;
        else
            low = x;
        if (f(x) == 0 || fabs(high - low) < fabs(low)*EPS) {
            printf("x=%f\n", x);
            break;
        }
    }
    if (k > LIMIT)
        printf("수렴하지 않는다.\n");
    return 0;
}
```

비선형 방정식의 해법 (뉴턴법)

방정식 f(x) = 0 일때의 근 x를 뉴턴법으로 구하는 방법이다.



1. 근에 가까운 적당한 값 x_0을 초기값으로 한다.

2. y=f(x), x=x_0에서 접선을 긋고, x축과 교차하는 점을 x_1이라고 하고, 같은 방법으로 $x_2, x_3 \dots x_n$을 구한다.

3. f(x) 가 0 or $\frac{|x_n - x_{n-1}|}{|x_{n-1}|} < \text{EPS}$ 될때, 구하는 x_n값이 근이 된다. 그렇지 않을 경우에는 2번을 반복한다. EPS는 수렴판정값으로 적절한 정확도를 선택하면 된다.

x_n을 구하기 위하여 다음과 같은 관계를 이용한다.

$$x_n = x_{n-1} - \frac{\mathrm{f}(x_n - 1)}{f'(x_n - 1)}$$

코드는 다음과 같다.

```c
#include <stdio.h>
#include <math.h>

#define f(x) ((x)*(x)*(x)-(x)+1)
#define g(x) (3*(x)*(x)-1)
#define EPS 1e-8
#define LIMIT 50

int main(void) {

    double x=-2.0, dx;
    int k = 1;

    for (k = 1; k <= LIMIT; k++) {
        dx = x;
        x = x - f(x) / g(x); // x1= x0 - f(x0)/f'(x0)
        if (fabs(x - dx) < fabs(dx)*EPS) {
            printf("x=%f\n", x);
            break;
        }
    }

    if (k > LIMIT)
        printf("수렴하지 않는다.\n");

    return 0;
}
```

f(x)는 원함수, g(x)는 도함수를 나타낸다.

기존에 위에서 관계식을 그대로 코드로 옮겼다.

보간법 (라그랑쥐 보간)

보간은 몇 쌍의 x, y 데이터가 주어졌을 때, 이 점들을 지나는 보간 다항식을 다그랑쥐 보간으로 구하고 주어진 점 이외의 점 데이터를 구하는 것이다.

즉, 부족한 데이터를 활용하여 더 정확한 데이터를 만들어 내는 방법으로 볼 수 있다.

$(x_0, y_0), (x_1, y_1), (x_2, y_2), \dots, (x_{n-1}, y_{n-1})$ 와 같이 점 n개가 주어졌을 때, 이 점들을 지나는 함수 f(x)는 다음과 같이 구할 수 있다.

$$\mathrm{f(x)} = \frac{(x-x_1)(x-x_2)\cdots(x-x_{n-1})}{(x_0-x_1)(x_0-x_2)\cdots(x_0-x_{n-1})}y_0 + \frac{(x-x_0)(x-x_2)\cdots(x-x_{n-1})}{(x_1-x_1)(x_1-x_2)\cdots(x_1-x_{n-1})}y_1 + \cdots + \frac{(x-x_0)(x-x_1)\cdots(x-x_{n-2})}{(x_{n-1}-x_0)(x_{n-1}-x_1)\cdots(x_{n-1}-x_{n-2})}y_{n-1}$$

$$= \sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\left(\frac{x-x_j}{x_i-x_j}\right)y_i$$

이것을 라그랑쥐 보간 다항식이라 하며 n-1차의 다항식이 된다. 따라서 주어진 데이터 외의 점은 이 다항식을 이용해서 계산이 가능하다.

```c
#include <stdio.h>
#include <math.h>
double lagrange(double [], double [], int, double);
int main(void) {

    static double x[] = { 0.0,1.0,3.0,6.0,7.0 };
    static double y[] = { 0.8,3.1,4.5,3.9,2.8 };

    double t;

    printf("      x      y₩n");
    for (t = 0.0; t <= 7.0; t = t + 0.5) {
        printf("%7.2f%7.2f₩n", t, lagrange(x, y, 5, t));
    }
    return 0;
}
double lagrange(double x[], double y[], int n, double t) {
    //오리지날 데이타 xy, 오리지날 데이타 갯수 n, 구간을 분할한 값
    int i, j;
    double s, p;
    s = 0.0;
    for (i = 0; i < n; i++) {
        p = y[i];
        for (j = 0; j < n; j++) {
            if(i!=j)
                p = p * (t - x[j]) / (x[i] - x[j]);
        }
        s = s + p;
    }
    return s;
}
```

오리지널 데이터는 (0, 0.8), (1, 3.1), (3, 4.5), (6, 4.9), (7, 2.8) 이지만, 보간법을 이용하여 복원한 데이터는 (0, 0.8), (0.5, 2.15), (1, 3.1), (1.5, 3.74), (2, 4.14), …. , (6, 3.9), (6.5, 3.46), (7, 2.80)으로 0.5 time 간격으로 복원이 되었다.

-연속.