به نام آنکِ جان را فکرت آموخت

امیر ارسلان یاوری - ۴۰۲۲۰۳۴۹۷

امتحان درس سیستم‌های تحمل‌پذیر اشکال دانشگاه صنعتی شریف — استاد: دکتر اجلالی

پاسخ سوال اول — نیمسال اول ۱۴۰۴-۱۴۰۳

---

۱) یک روش کشف اشکال که تعمیمی بر روش DWC است به این صورت عمل می‌کند که به جای اینکه از یک واحد F دو عدد داشته باشیم و ورودی یکسان x را به هر دو واحد داده و دو خروجی y (که به صورت y=F(x) هستند) را مقایسه کنیم، از واحد F واحد F^{-1} که عمل معکوس آن را انجام می‌دهد داریم. سپس بعد از محاسبه‌ی y=F(x) اقدام به محاسبه‌ی معکوس آن یعنی $x=F^{-1}(y)$ نموده و سپس دو مقدار x را مقایسه می‌کنیم. مزایای این روش این است که اولاً diversity دارد و ثانیاً برخی مواقع سربار پیاده‌سازی F^{-1} از F کمتر است که آن را گزینه‌ی مناسبی می‌کند. اما عیب آن این است که موجب افزونگی زمانی می‌گردد. بر اساس این روش فقط یکی از دو مورد زیر را انتخاب نموده و طراحی مربوطه را انجام دهید:

ب) قطعه کدی به زبان C که عمل تقسیم چندجمله‌ای‌ها را انجام می‌دهد توسعه دهید و مکانیسم کشف خطای فوق را برای آن بکار بگیرید (فقط زبان C قابل قبول است). این برنامه دو چندجمله‌ای با ضرایب ثابت را در ورودی دریافت نموده و سپس خارج قسمت و باقی‌مانده را در خروجی ارائه می‌دهد. سربار زمانی ناشی از اعمال این روش کشف خطا را اندازه‌گیری نموده و گزارش کنید. در گزارش خود ذکر کنید که برای اجرای قطعه کد مذکور از چه سیستمی استفاده کرده‌اید (با گزارش مشخصات سخت‌افزاری سیستم مورد استفاده و سیستم‌عامل آن).

برای حل این سوال یک کد C نوشته‌ام که در ادامه آن را توضیح خواهم داد اما به طور مفصل مقسوم و مقسوم‌الیه در کد به صورت hardcode نوشته می‌شوند. سپس یک تابع برای pare کردن آنها دارم. درواقع این تابع هر چند جمله‌ای به فرمتی که در ادامه آن را توضیح خواهم داد را در یک استراکت نگه داری می‌کند به صورتی که ضرایب جملات آن و در درجه‌ی آن را نگه می‌دارد. سپس یک تابع برای تقسیم مقسوم و مقسوم‌الیه دارم. همچنین یک تابع برای ضرب باقی‌مانده در خارج‌قسمت دارم که در همین تابع باقی‌مانده را هم با حاصل ضرب جمع می‌کنم که درواقع پیاده‌سازی تابع F^-1 است. یک تابع دیگر برای مقایسه‌ی دو چند جمله‌ای پیاده‌سازی کرده‌ام و یک تابع هم برای چاپ یک چنده‌جمله‌ای بر اساس استراکت‌ش دارم. در بدنه‌ی تابع main هم توابع مذکور صدا زده شده‌اند و زمان هر یک را هم اندازه‌گیری کرده‌ام که در گزارشات پایانی خود آنها را تحلیل کنم. در ادامه به توضیح کد خواهیم پرداخت. کد تحمل‌پذیر اشکال ما در صورتی که مشکلی وجود نداشته باشد CORRECT و در غیر اینصورت INCORRECT بر خواهد گرداند.

همچنین شمای کلی ساختاری کد نوشته شده به صورت تصویر زیر است:

```c
Q1 > answer.c > multiplyPolynomials(Polynomial, Polynomial)
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <math.h>
#include <time.h>

#define MAX_DEGREE 100
#define EPSILON 1e-9
char flag = 0;

const char *dividend_str = "1000.00000001x^3 - 0.25x^2 + 3x + 4.5";
const char *divisor_str = "3x + 0.5";
// Example polynomials with very precise floating-point coefficients
// const char *dividend_str = "0.0000000x^3 + 0.000000x^2 + 0.000x + 1";
// const char *divisor_str = "0.5";

typedef struct
{
    int degree;
    long double coefficients[MAX_DEGREE + 1];
} Polynomial;

Polynomial parsePolynomial(const char *polyStr)
{ ...

Polynomial multiplyPolynomials(Polynomial poly1, Polynomial poly2) ...

int printPolynomial(Polynomial poly)
{ ...

int comparePolynomials(Polynomial poly1, Polynomial poly2)
{ ...

Polynomial polynomialDivision(Polynomial dividend, Polynomial divisor, Polynomial *remainder)
{ ...

int main()
{
    struct timespec start, end;
    long parsePolyTime, divisionPolyTime, multiplyPolyTime, comparePolyTime;
    clock_gettime(CLOCK_MONOTONIC, &start);
```

مقادیر MAX_DEGREE و EPSILON به ترتیب مشخص‌کننده‌ی ماکسیمم درجه‌ی چندجمله‌ای‌ها و مینیمم دقت در محاسبات می‌باشند. وجود EPSILON ضروری‌ست چرا که اعداد ممیز شناور دارای خطا هستند و برای همین اگر از یک اپسیلونی استفاده نکنیم در مواردی با اینکه چندجمله‌ای‌ها برابرند اما به دلیل مشکل یاد شده نتیجه‌ی INCOORECT برگردانده خواهد شد.

همچنین بر اساس متغیرهای dividend_str و divisor_str فرمت مشخص کردن مقسوم و مقسوم‌الیه در کد مشخص شده است.

استراکت Polynomial دارای یک عدد صحیح degree که درجه‌ی چند جمله‌ای را نگه می‌دارد و یک آرایه‌ای از ضرایب به اسم coefficients است.

تابع parsePolynomial وظیفه‌ی این را دارد که یک جمله‌ی چندجمله‌ای (برای مثال dividend یا divisor) را در استراکت یاد شده ذخیره کند. به طور مفصل در این تابع کاراکتر به کاراکتر جمله خوانده می‌شود، در صورتی که کاراکتر اسپیس باشد از آن صرف نظر می‌شود؛ در صورتی که کاراکتر + یا - باشد علامت جمله‌ی چند جمله‌ای مشخص می‌شود (علامت هر یک از ضرایب جملات چند جمله‌ای)؛ در صورتی که . باشد مشخص کننده‌ای این است که عدد آمده در ادامه‌ی جمله قسمت شناور ضریب‌ست؛ در صورتی که x باشد با استفاده از آن درجه‌ی چند جمله‌ای را خواهم فهمید که بعد از x و به صورت n^ آمده است. یک بار جمله خوانده شده و با استخراج موارد گفته شده در ادامه‌ی این تابع ضریب جملات هربار محاسبه و در خانه‌ی متناظر آن در آرایه‌ی ضرایب استراکت مربوطه‌ش نوشته می‌شود و درجه‌ی آن هم در پایان به عنوان بزرگ‌ترین درجه در قسمت degree استراکت ذخیره می‌شود. شمای کلی این تابع به صورت زیر است:

```
24  Polynomial parsePolynomial(const char *polyStr)
25  {
26      Polynomial poly = {0};
27      int strLen = strlen(polyStr);
28      int coefficientSign = 1;
29      long double coefficient = 0;
30      long double fractionalPart = 0;
31      long double fractionalMultiplier = 0.1;
32      int currentDegree = 0;
33      int readingCoeff = 1;
34      int readingFraction = 0;
35      int coeffSpecified = 0;
36      int fractionDigits = 0;
37      int highestNonZeroDegree = -1;
38
39      for (int i = 0; i < strLen; i++)
40      {
41          if (polyStr[i] == ' ')
42 >        {…
45
46          if (polyStr[i] == '+' || polyStr[i] == '-')
47 >        {…
81
82          if (isdigit(polyStr[i]))
83 >        {…
99
100         else if (polyStr[i] == '.')
101 >       {…
106
107         else if (polyStr[i] == 'x')
108 >       {…
134     }
135
136     if ((coeffSpecified > 0 || currentDegree > 0) && (fabsl(coefficient) >= EPSILON || fabsl(fractionalPart) >= EPSILON))
137 >   {…
150
151     if (highestNonZeroDegree == -1)
152 >   {…
154     }
155     else
156     {
157         poly.degree = highestNonZeroDegree;
158     }
159     return poly;
160 }
```

در ادامه تابع ضرب دو چندجمله‌ای را داریم که به صورت زیر است:

```
161
162 Polynomial multiplyPolynomials(Polynomial poly1, Polynomial poly2)
163 {
164     Polynomial result = {0};
165
166     for (int i = 0; i <= poly1.degree; i++)
167     {
168         for (int j = 0; j <= poly2.degree; j++)
169         {
170             result.coefficients[i + j] += poly1.coefficients[i] * poly2.coefficients[j];
171         }
172     }
173
174     result.degree = poly1.degree + poly2.degree;
175     while (fabsl(result.coefficients[result.degree]) < EPSILON && result.degree > 0)
176     {
177         result.degree--;
178     }
179
180     return result;
181 }
```

این تابع دو استراکت که شامل چندجمله‌ای‌های اول و دوم هستند را گرفته و یک استراکت جدید (که در ابتدا تمام صفر است) را بر اساس ضرب آن دو ساخته و بر می‌گرداند به صورتی که درجه‌ی آن می‌شود مجموع درجات دو چند جمله‌ای (در صورتی

که صفر نباشد) و ضرایب جملات هم می‌شود ضرب تک تک ضرایب جملات چند جمله‌ای دوم در تک تک جملات چند جمله‌ای اول که در خانه‌ی جمع توان‌های آنها مقدار مذکور با مقدار قبلی بروزرسانی می‌شود.

```c
int printPolynomial(Polynomial poly)
{
    flag = 1;
    for (int i = poly.degree; i >= 0; i--)
    {
        if (fabsl(poly.coefficients[i]) < EPSILON)
        {
            continue;
        }

        printf("%+.8Lf", poly.coefficients[i]);
        if (i > 0)
        {
            printf("x");
            if (i > 1)
            {
                printf("^%d", i);
            }
        }

        flag = 0;
    }
    if (flag)
    {
        flag = 0;
        printf("0");
    }
    printf("\n");
    return 0;
}
```

تابع printPolynomial هم که تصویر آن در عکس روبرو آمده، بر روی تمامی خانه‌های آرایه‌ی ضرایب استراکت گرفته شده در ورودی تابع حلقه زده و ضرایب را چاپ و پس از آن یک X به همراه درجه‌ی آن چاپ می‌نماید.

```c
int comparePolynomials(Polynomial poly1, Polynomial poly2)
{
    if (poly1.degree != poly2.degree)
    {
        return 0;
    }

    for (int i = 0; i <= poly1.degree; i++)
    {
        if (fabsl(poly1.coefficients[i] - poly2.coefficients[i]) > EPSILON)
        {
            return 0;
        }
    }

    return 1;
}
```

در ادامه تابع مقایسه‌ی دو چندجمله‌ای را داریم که در ابتدا درجات آن مقایسه شده و سپس یک به یک ضرایب آنها مقایسه شده و در صورتی که مغایرتی نبود ۱ و در غیر اینصورت ۰ بر می‌گرداند که به معنای نامساوی بودن آنهاست.

در پایان هم تابع تقسیم دو چندجمله‌ای را داریم که چندجمله‌ای خارج قسمت و باقی‌مانده را بر اساس تقسیم دو چندجمله‌ای مقسوم و مقسوم‌الیه انجام می‌دهد. به دلیل اینکه در زبان برنامه‌نویسی C امکان بازگرداندن دو متغیر وجود ندارد باقی‌مانده به صورت call by reference در ورودی تابع دریافت می‌شود و خارج قسمت به عنوان return تابع بازگردانده می‌شود. عملیات تقسیم هم به طور کلی به این صورت است که ضرایب از تقسیم ضرایب مقسوم به مقسوم‌الیه بدست آمده و درجه‌ی آن نیز از تفریق درجات بدست می‌آید. کد به صورت clean نوشته شده است که با خواندن آن (که در تصویر زیر آمده) ریز جزئیات آن نیز مشخص خواهد شد.

```
Polynomial polynomialDivision(Polynomial dividend, Polynomial divisor, Polynomial *remainder)
{
    Polynomial quotient = {0};

    while (dividend.degree >= divisor.degree)
    {
        if (flag == 1)
        {
            flag = 0;
            break;
        }
        if (divisor.degree == 0 && dividend.degree == 0)
        {
            flag = 1;
        }

        long double coefficient_ = dividend.coefficients[dividend.degree] / divisor.coefficients[divisor.degree];
        int degree_ = dividend.degree - divisor.degree;

        quotient.coefficients[degree_] = coefficient_;
        if (degree_ > quotient.degree)
        {
            quotient.degree = degree_;
        }

        for (int i = 0; i <= divisor.degree; i++)
        {
            dividend.coefficients[dividend.degree - i] -= coefficient_ * divisor.coefficients[divisor.degree - i];
        }

        while (dividend.degree > 0 && fabsl(dividend.coefficients[dividend.degree]) < EPSILON)
        {
            dividend.degree--;
        }
    }

    *remainder = dividend;

    return quotient;
}
```

در پایان نیز تابع main را داریم که تصویر آن را قرار نداده‌ام (در تصویر جا نمی‌شود) اما در ابتدا از کتابخانه‌ی time.h دو استراکت شروع و پایان ساخته‌ام که زمان شروع و پایان را بتوانم اندازه‌گیری کنم؛ همچنین مقادیر را در متغییرهای مربوطه با نام‌های **parsePolyTime, divisionPolyTime, multiplyPolyTime, comparePolyTime** ذخیره می‌کنم. پس در ابتدا تایم را شروع و تابع parser را برای مقسوم و مقسوم‌الیه صدا زده و زمان انجام آنها در متغیر مربوطه ذخیره می‌شود سپس یکبار دیگر تایم استارت و تابع تقسیم مقسوم بر مقسوم‌الیه صدا زده می‌شود و زمان اندازه‌گیری می‌شود. یک بار دیگر زمان شروع مشخص و تابع ضرب صدا زده شده و زمان اندازه گیری و در نهایت هم، زمان استارت زده شده و عمل مقایسه انجام می‌شود و زمان آن نیز ثبت می‌شود. بدیهی‌است که تابع F شامل دو زمان اول و تابع F^-1 شامل زمان سوم است (برای F^-1 باید عمل parse مقسوم‌الیه هم محاسبه شود که از آن صرف نظر کردم اما می‌توانیم آن را یا دقیقا اندازه بگیریم یا اغماض کنیم و به صورت حدودی نصف زمان اولیه که ثبت کردیم در نظر بگیریم؛ من کلا صرف نظر کردم چون یک بار محاسبه شده و دیگه واقعا نیازی نیست محاسبش کنیم). نتایج بدست آمده را در ادامه قرار داده‌ام که یک تحلیل بر اساس آنها هم در ادامه خواهم داشت.

```
andre  Adrian   ~/code/uni/fault/midterm/Q1                    main   12/08/24
❯ ./answer
Dividend: +1000.00000000x^3+3.00000000x+4.00000000
Divisor: +3.00000000x
Quotient: +333.33333333x^2+1.00000000
Remainder: +4.00000000

Reconstructed Dividend: +1000.00000000x^3+3.00000000x+4.00000000
Verification: Polynomial division is CORRECT!

Parsing dividend and divisor time:                          90524 ns
Division dividend to divisor time:                          9289 ns
Multiplay quotient to divisor time:                         617 ns
Compare reconstructed polynomial with dividend time:        5498 ns
```

با توجه به زمان‌های اندازه‌گیری شده زمان parse بیشتر از همه طول کشیده شده اما برای محاسبه‌ی تقسیم به آن نیاز داشته‌ایم پس می‌توان آن را با زمان تقسیم با هم جمع نموده و به عنوان زمان اجرای فرآیند گزارش کرد. همانطور که در متن سوال هم گفته شده بود عملیات F^-1 کمتر از F طول کشید و همچنین عملیات محاسبه‌ی تقسیم (فرآیند مورد انتظار) که جمع پارسر و تقسیم دو چند جمله‌ای است هم بسیار بیشتر از عملیات کشف خطا که شامل F^-1 و مقایسه‌ی چندجمله‌ای بازسازی شده با مقسوم است می‌باشد. این بدان معنی‌است که اورهد کشف خطا کم است و ارزش استفاده از این روش کشف خطا وجود دارد. درواقع با پرداخت هزینه‌ی کمتر به نسبت هزینه‌ی انجام فرآیند می‌توانیم مطمئن باشیم که فرآیند دارای خطا بوده است یا خیر.

زمان اجرای فرآیند برابر 99813 نانوثانیه

زمان محاسبه‌ی تقسیم (تابع F) معادل 9289 نانوثانیه

زمان محاسبه‌ی F^-1 معادل 617 نانوثانیه

و زمان فرآیند تشخیص اشکال برابر 6115 نانوثانیه است.

مشخصات سیستم استفاده شده:

```
andre  Adrian  ~
 ❯ cat /etc/os-release
PRETTY_NAME="Debian GNU/Linux 12 (bookworm)"
NAME="Debian GNU/Linux"
VERSION_ID="12"
VERSION="12 (bookworm)"
VERSION_CODENAME=bookworm
ID=debian
HOME_URL="https://www.debian.org/"
SUPPORT_URL="https://www.debian.org/support"
BUG_REPORT_URL="https://bugs.debian.org/"

andre  Adrian  ~
 ❯ gcc --version
gcc (Debian 12.2.0-14) 12.2.0
Copyright (C) 2022 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.


andre  Adrian  ~
 ❯ lscpu | grep "Model name"
Model name:                           11th Gen Intel(R) Core(TM) i7-11370H @ 3.30GHz

andre  Adrian  ~
 ❯ free -h | grep "Mem"
Mem:            15Gi        10Gi       327Mi       900Mi       5.6Gi       4.7Gi

andre  Adrian  ~
 ❯ uname -a
Linux Adrian 6.1.0-23-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.1.99-1 (2024-07-15) x86_64 GNU/Linux

andre  Adrian  ~
 ❯
```

مشخصات سخت‌افزاری به تفصیل در فایل system_info.txt در کنار PDF قرار داده شده است.

همچنین کد نوشته شده در فایل answer.c در کنار PDF قرار داده شده است.

با تشکر :)