ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO ĐỒ ÁN

Đề tài: **Xây dựng kiểu dữ liệu số nguyên lớn có dấu BigInt và thư viện các hàm chức năng**

Lớp 20CTT1TN – Nhóm 10

Huỳnh Đức Nhâm – MSSV: 20120020

Từ Văn Quí – MSSV: 20120561

GVHD: Võ Hoài Việt

Môn học: Kỹ thuật lập trình

Thành phố Hồ Chí Minh – 2021

MỤC LỤC

PHẦN 1: GIỚI THIỆU

1.1 Giới thiệu về đồ án:

Lập trình gắn với nhu cầu thao tác và xử lý trên nhiều kiểu dữ liệu khác nhau. Để giải quyết nhu cầu cơ bản đó, những kiểu dữ liệu chuẩn ra đời, cho phép chúng ta làm việc với hầu hết những loại dữ liệu cần thiết nhất: từ số nguyên, số thập phân cho đến chuỗi ký tự…

Tuy nhiên, để tối ưu bộ nhớ và tốc độ xử lý, những kiểu dữ liệu này thường chỉ thao tác được trên một tập giá trị xác định. Và để giải quyết nhu cầu thao tác và xử lý số nguyên nằm ngoài những tập giá trị như thế, ta cần thiết phải xây dựng một kiểu dữ liệu số nguyên lớn có dấu BigInt và một thư viện những hàm thực hiện chức năng liên quan.

Đó cũng là nội dung của đồ án môn học Kỹ thuật lập trình của nhóm chúng em.

1.2 Yêu cầu và mục tiêu đặt ra:

Dựa trên yêu cầu mà GV đề ra, kết hợp cùng những yếu tố thực tế phát sinh, chúng em thống nhất đưa ra những mục tiêu cần đạt được ở sản phẩm đồ án như sau:

- Bộ xử lý tập lệnh input/output từ file, theo đúng cú pháp yêu cầu
- Cấu trúc BigInt có thể lưu và xử lý số nguyên âm
- Phép gán = và phép khởi tạo từ giá trị kiểu int
- Các phép toán +, -, *, /, % và các phép toán trên bit &, |, ^, ~, >>, <<
- Hàm readBit, setBit tại một vị trí nào đó của số BigInt
- Chuyển đổi BigInt giữa hệ thập phân và hệ nhị phân
- Các phép so sánh >, >=, <, <=, ==, !=
- Các hàm tính trị tuyệt đối abs, tính số lượng ký tự số hệ thập phân digits, tìm min, max giữa 2 số hạng, kiểm tra tính nguyên tố
- Xuất ra dạng base58, base32, base64
- Tính toán thời gian, dung lượng bộ nhớ sử dụng khi tính toán, xử lý BigInt

PHẦN 2: NỘI DUNG BÁO CÁO

2.1 Cấu trúc của BigInt. Cách lưu dữ liệu và cơ chế tự giải phóng bộ nhớ:

Định nghĩa: byte = unsigned char

❖ Một cấu trúc BigInt sẽ gồm có:

- Mảng bytes động 1 chiều chứa các byte đủ để biểu diễn dãy bit của số nguyên cần lưu
- Biến byteCount kiểu số nguyên uint16_t lưu số lượng phần tử trong mảng bytes phía trên.
- Biến isHasSign kiểu bool thể hiện xem số đang lưu có dấu hay không:
  + Nếu là true thì cấu trúc này biểu diễn 1 số âm và bit có giá trị 1 cuối cùng trong dãy bit là bit dấu
  + Nếu là false thì cấu trúc này biểu diễn 1 số dương và không có bit dấu trong dãy bit

Với cách lưu dữ liệu sử dụng mảng byte biểu diễn dãy bit của số, việc xử lý các phép bit sẽ thuận tiện hơn đồng thời giúp tối ưu bộ nhớ hơn và tiếp cận gần với cách lưu dữ liệu của các kiểu dữ liệu chuẩn như int, long, …

Ta nhận thấy, mỗi phần tử trong mảng bytes đều có giá trị từ 0 => 255 biểu diễn một phần (8 bit) của số nguyên được lưu. Vì vậy có thể coi mảng bytes là dạng biểu diễn của số nguyên ở cơ số 256 với chữ số hàng thứ 1, chữ số hàng thứ 2, … là phần tử thứ 0, phần tử thứ 1, ... của mảng. Công thức đổi:

$$<\text{Số nguyên}>_{10} = <\text{Giá trị phần tử thứ n}>*256^n + \ldots + <\text{Giá trị phần tử thứ 2}>*256^2 + <\text{Giá trị phần tử thứ 1}>*256^1 + <\text{Giá trị phần tử thứ 0}>*256^0$$

Từ đây, chúng ta có thể chuyển đổi một số kiểu int sang dạng BigInt và ngược lại bằng cách chuyển đổi qua lại giữa hệ 10 và hệ 256.

❖ Phép gán BigInt = BigInt:

Nếu dùng phép gán thông thường với cấu trúc thì BigInt được gán sẽ không chuẩn xác, khi đó nó sẽ có 2 biến byteCount và isHasSign có giá trị giống với BigInt gán, nhưng vẫn là 2 biến khác biệt với 2 biến trong BigInt gán. Và vấn đề ở đây chính là mảng bytes, vì chúng ta lưu vào cấu trúc BigInt một con trỏ bytes nên sau khi gán, cả BigInt được gán và BigInt gán sẽ dùng chung một mảng bytes (2 biến con trỏ mang địa chỉ cùng trỏ tới 1 vùng nhớ).

Để giải quyết vấn đề này, chúng ta cần can thiệp vào phép gán bằng operator = và trong hàm này, chúng ta sẽ khởi tạo cho BigInt được gán 1 vùng nhớ riêng chứa dữ liệu y hệt như mảng bytes của BigInt gán.

❖ Giải phóng bộ nhớ BigInt tự động:

Khi làm việc với BigInt, chắc chắn chúng ta sẽ khởi tạo, gán các biến BigInt rất nhiều lần, và như chúng ta đã thấy ở trên, mỗi lần như vậy sẽ có một vùng nhớ heap mới được cấp phát. Để tránh việc giải phóng sót, rò rỉ bộ nhớ, chúng ta sẽ cho BigInt tự giải phóng bộ nhớ bytes khi nó được hệ điều hành thu hồi (Biến cấu trúc được khởi tạo bình thường vẫn được hệ điều hành quản lý trong vùng nhớ stack), bằng cách cho chạy lệnh giải phóng bộ nhớ (free) trong hàm hủy (Destructor). Hàm hủy là hàm được gọi ngay trước lúc cấu trúc bị thu hồi bởi hệ điều hành.

2.2 Thuật toán xử lý tập lệnh input:

Như đã mô tả trong hướng dẫn sử dụng, mỗi dòng trong file input là một câu lệnh tính toán.

Vì vậy ý tưởng ở đây chính là duyệt từng dòng trong file input và đối chiếu dòng đó với **bộ cấu trúc lệnh** của chương trình để biết được đó là lệnh gì, tham số nhập vào là gì để từ đó thực thi.

Mỗi dòng lệnh gồm các thành phần tham số được tách nhau bởi dấu cách, ta định nghĩa một kiểu cấu trúc CmdParam và hàm chuyển một dòng text sang cấu trúc này (sử dụng hàm strtok):

```
struct CmdParam {
   char** params;
   uint8_t paramCount;
};
```

Như vậy trước hết, chương trình phải có **bộ cấu trúc lệnh** – Một mảng các **lệnh mẫu.**

Và một cấu trúc **lệnh mẫu** Command sẽ là:

```
typedef bool (*CommandFunction)(CmdParam cmdParam, FILE* outFile);

struct Command {
   CommandFunction cmdFunc; // Hành thực thi lệnh
   CmdParam param; // Các tham số để đối chiếu
};
```

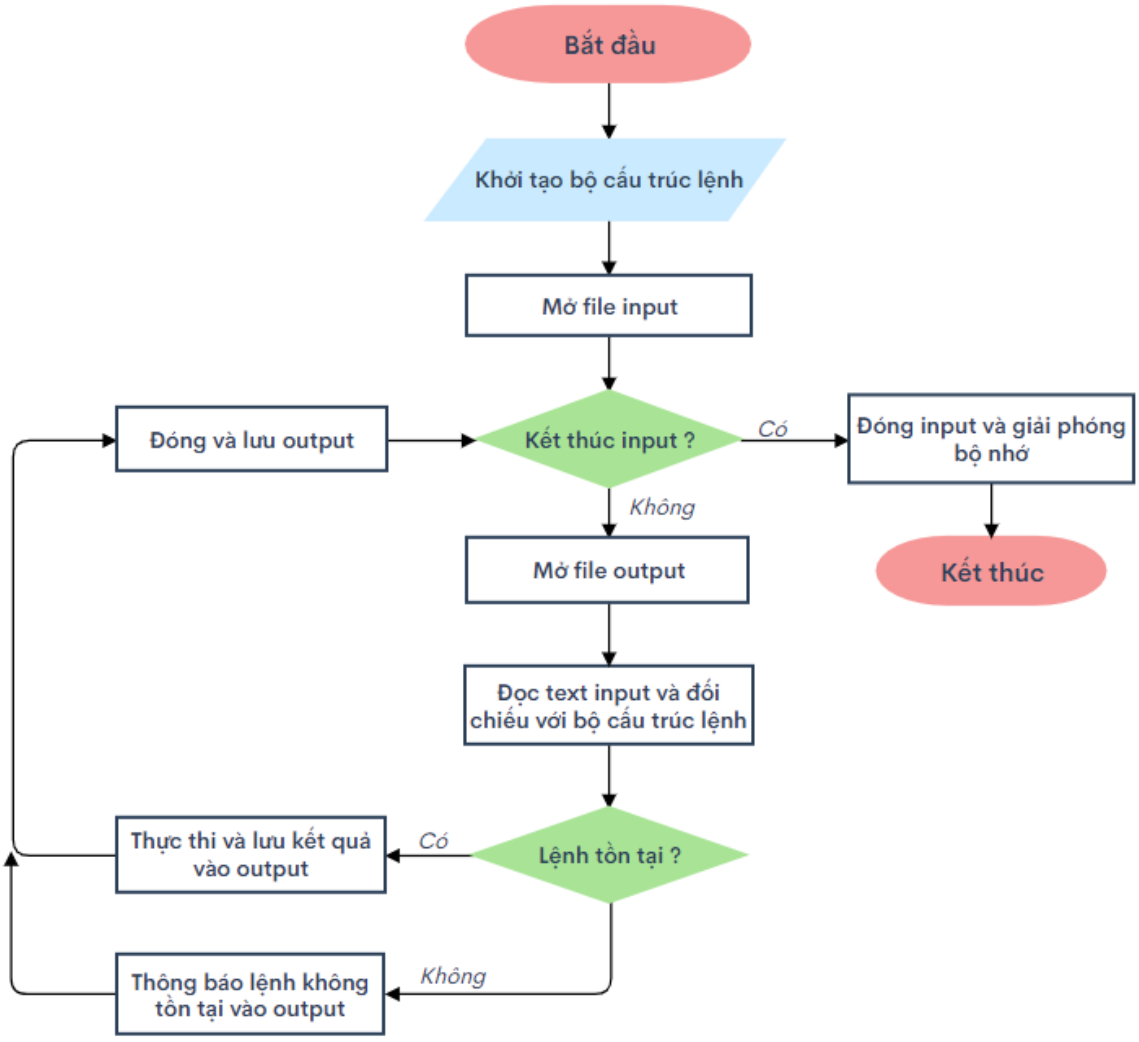
Khi đối chiếu một lệnh input dạng CmdParam với **lệnh mẫu,** chúng ta sẽ so sánh lần lượt từng tham số và bỏ qua những tham số tùy chọn do người dùng nhập (Bỏ

qua những tham số "<>" trong CmdParam của **lệnh mẫu**). Khi đã khớp hết thì gọi hàm cmdFunc với CmdParam lấy từ input.

Ví dụ về một dòng **lệnh mẫu**: "10 <> + <>"

NÓI THÊM: Với các phép toán, việc xử lý sẽ được thực hiện trên giá trị tuyệt đối của các số hạng và xử lý phần dấu sau để giúp đơn giản quá trình xử lý hơn.

Như vậy, ta đã hiểu được cách mà chương trình xử lý file input thành tập lệnh để thực thi. Từ đây, ta phát triển tiếp một vòng đời của chương trình sẽ như sau:



2.3 **Các phép toán cơ bản và các phép toán trên bit:**

❖ Phép cộng :

Ta xét phép cộng BigInt: res = a + b

Để có thể thực hiện tính toán, 2 số hạng BigInt a và b phải được lưu và biểu diễn (bằng phương pháp bù 2) trên cùng một số lượng byte nhất định, ở đây ta chọn là byteCount tối đa của a và b: maxByte.

Ta thấy ví dụ: $5_{10} = 0101_2$ trên 4 bit $-3_{10} = 101_2$ trên 3 bit

$= 00000101_2$ trên 8 bit $= 11111101_2$ trên 8 bit

Như vậy, bằng cách sửa những bit vô nghĩa phía trước bit dấu thành chính bit dấu, ta sẽ biểu diễn được 2 số hạng trên cùng một số lượng byte: maxByte.

Từ đây, ta thực hiện cộng lần lượt các phần tử trong mảng bytes của a và b theo thứ tự trong mảng và lưu kết quả vào byte có thứ tự tương ứng của *res*. Nếu 2 phần tử tương ứng có tổng lớn hơn 255, tức là bị "tràn" 1 bit khi lưu kết quả vào chỉ 1 byte tương ứng thứ tự trên *res*, ta thực hiện "nhớ 1" bằng biến bitOverFlow và cộng dồn vào phần tử byte liền sau trên res. Dễ thấy, quá tình thực hiện phép cộng như trên giống hệt cách chúng ta thực hiện "phép cộng tiểu học" !

Việc xét dấu cho kết quả phép cộng được thực hiện như sau:

- Nếu a và b cùng dấu: kết quả mang dấu của a và b

- Nếu a và b khác dấu: ta xét đến biến bitOverFlow của cặp phần tử cuối cùng thực hiện cộng:

+ bitOverFlow = 0 : kết quả mang dấu âm

+ bitOverFlow = 1 : kết quả mang dấu dương, bỏ đi bit bị "tràn" cuối cùng trong kết quả

Kết thúc tính toán, ta rút gọn *a, b* và *res* về dạng biểu diễn với số byte tối thiểu của chúng trước khi trả về kết quả.

❖ Phép trừ:

Ta có thể hiểu phép trừ: res = a – b như là phép cộng số bị trừ với số đối của số trừ: res = a + (-b)

Trong đó, hàm lấy số đối được xây dựng rất đơn giản theo phương pháp bù 2 :

–b = ~(b) + 1

Áp dụng cách làm này, ta vừa tận dụng được hàm cộng đã được xây dựng, vừa đảm bảo tính ngắn gọn, nhanh chóng của phép toán.

❖ Phép nhân:

Ta xét phép nhân BigInt: res = a * b

Sử dụng thuật toán Russian Peasant:

1) Gán giá trị ban đầu cho res = 0

2) Lặp những lệnh bên dưới với b > 0

a) Nếu b là lẻ thì cộng thêm a vào res

b) Nhân đôi a (Dịch bit sang trái 1 bit) và chia đôi b (Dịch bit sang phải 1 bit)

3) Trả về kết quả res

Để tối ưu thời gian thực hiện thì nên sắp xếp để b là số nhỏ hơn a. Đồng thời cải tiến bước 2b: Với b lẻ thì dịch bit 1 lần, nhưng với b chẵn thì chúng ta sẽ tính số bit 0 liên tiếp ở đầu dãy bit của số b (gọi là n) và thực hiện dịch bit n bit (Tức là thay vì nhân/chia đôi từng lần 1, chúng ta sẽ nhân/chia 2^n trong 1 lần).

❖ Phép chia lấy nguyên & chia lấy dư:

Ta xét Q = a / b và R = a % b

Nếu a < b thì Q = 0 và R = a

Nếu a >= b thì ta có thể phân tách a thành:

$a = b*2^{n1} + b*2^{n2} + b*2^{n3} + \dots + b*2^{nk} + R$ (*)

Lần lượt xác định n1 => nk

Với n1 là lớn nhất sao cho $a >= b*2^{n1}$

n2 là lớn nhất sao cho $a - b*2^{n1} >= b*2^{n2}$

n3 là lớn nhất sao cho $a – (b*2^{n1} + b*2^{n2}) >= b*2^{n3}$

……..

nk là lớn nhất sao cho $a – (b*2^{n1} + b*2^{n2} + b*2^{n3} + \dots + b*2^{n(k-1)}) >= b*2^{nk}$

Cuối cùng ta tìm được số R < a để thỏa biểu thức (*). Và kết của phép chia lấy nguyên là:

$Q = 2^{n1} + 2^{n2} + 2^{n3} + \dots + 2^{nk}$

Sở dĩ chúng em chọn cách này là vì việc xác định 1 số n **tương đối gần** lớn nhất để $A >= B*2^n$ có thể thực hiện dễ dàng qua việc tính chênh lệch độ dài dãy bit giữa 2 số A và B, sau đó trừ đi 1. Xuất phát từ cơ sở nhân một số B cho 2^n thì ta chỉ cần

dịch bit số B qua trái n bit. Việc trừ đi 1 ở trên là vì khi 2 số A và B có độ dài dãy bit bằng nhau thì ta không thể xác định được rằng là A < B hay A > B hay A = B, nên số n được gọi là **tương đối gần** lớn nhất.

❖ Các phép toán bit & | ^ ~ :

Đối với các phép & | ^ thực hiện giữa 2 số hạng thì đầu tiên chúng ta cần làm cho số byte giữa 2 số hạng bằng nhau: Thêm các bit 1 (với số âm/có dấu) hoặc bit 0 (với số dương/không dấu) vào bên trái dãy bit của số hạng có ít byte hơn cho đến khi cả 2 số hạng có số byte (số bit) bằng nhau. Tiếp theo, ta chỉ việc thực hiện các phép bit & | ^ với từng cặp byte tương ứng giữa 2 số hạng.

Đối với phép ~ thực hiện với 1 số hạng, nếu số hạng này âm thì cần thêm bit 1 (bit dấu) vào dãy bit sao cho byte cuối cùng của số hạng được lấp đầy với bit dấu. Tiếp theo, ta thực hiện phép ~ trên từng byte và đồng thời lấy NOT (!) cho biến isHasSign.

❖ Phép dịch bit << >> :

Được mô phỏng lại giống hệt với cách dịch bit của kiểu dữ liệu chuẩn: Khi dịch phải, bit dấu (1 với số âm và 0 với số dương) được thêm vào bên trái dãy bit; Khi dịch trái, bit 0 được thêm vào bên phải dãy bit.

2.4 Các phép so sánh:

Khi thực hiện một phép so sánh giữa 2 số A và B, chúng em sẽ chuyển đổi lại và thực hiện một phép so sánh tương đương: (A – B) và 0. Khi đó việc so sánh giữa một số với số 0 sẽ dễ dàng thực hiện hơn rất nhiều.

2.5 Các hàm chuyển đổi:

❖ Các hàm binStrToBigInt và bigIntToBinStr:

Thao tác nhập và xuất BigInt dưới dạng chuỗi nhị phân được thực hiện khá đơn giản bằng cách xử lý trên từng ký tự trong chuỗi, tương ứng với một bit trong BigInt, áp dụng các hàm readbit và setbit đã xây dựng.

❖ Hàm decStrToBigInt:

Hàm thực hiện chuyển đổi từ chuỗi số thập phân đầu vào (str) sang dữ liệu lưu trên máy tính theo cấu trúc BigInt (res) như sau:

Thực hiện lặp :

-Lấy giá trị 1 chữ số (từ hàng đơn vị đến hàng lớn nhất của str) và chuyển thành một số BigInt đơn giản *num*.

-Nhân *num* với giá trị BigInt 10^n rồi cộng dồn vào kết quả *res*. (n tăng 1 dần từ 0)

Kết thúc lặp, ta được *res* là một BigInt dương mang giá trị tuyệt đối của số đầu vào. Nếu str[0] là dấu ' - ', trả về số đối của res. Nếu không, ta trả về *res*.

❖ Hàm bigIntToDecStr:

Ngược lại với hàm trên, ta thực hiện chuyển một số BigInt a thành một số thập phân dưới dạng chuỗi ký tự *res*.

Ta thao tác trên một BigInt b là giá trị tuyệt đối của a.

Thực hiện lặp khi b != 0 :

- Chia lấy dư b cho BigInt 10 thu được một BigInt num, chia b cho 10.
- Chuyển BigInt num thành một ký tự chữ số c có cùng giá trị.
- Thêm ký tự c vào đầu chuỗi *res*.

Nếu a mang dấu dương, trả về chuỗi *res*. Nếu a mang dấu âm, thêm dấu '-' vào đầu chuỗi *res* và trả về.

2.6 **Các hàm chức năng khác:**

❖ Hàm kiểm tra số nguyên tố:

Sử dụng thuật toán Miller(một biến thể của thuật toán Rabin-Miller):

Xét số lẻ n>1, để kiểm tra xem n có phải là số nguyên tố hay không, ta làm như sau:

- Phân tích n−1 thành dạng $2^s \times d$ (s và d là số nguyên dương, d lẻ)
- Xét mỗi số nguyên a trong khoảng [2, min(n−1, ⌊2 x $(\ln(n))^2$⌋)]
  - Nếu (a^d % n) != 1 và (r % n) != (n-1) với mọi r từ 0 đến s−1 thì n **không phải là số nguyên tố**.
  - Nếu n vượt qua được tất cả các lần thử với a ở trên thì n **là số nguyên tố**.

Xét một cách **tương đối đủ chặt** thì ta có thể chỉ cần xét đến 3-5 số a trong khoảng phía trên là đủ để kết luận số nguyên tố.

❖ Mã hóa base32, base64:

Khi mã hóa base32 hay 64 cho một mảng bytes, ta sắp các byte trong mảng từ trái qua phải nhưng vẫn phải giữ cho thứ tự 8 bit của 1 byte sắp đúng thứ tự biểu diễn nhị phân từ phải qua trái. Sau đó, ta nhóm từng cụm 5 bit (với base32) hoặc 6 bit (với base64) từ trái qua phải để quy đổi ra ký tự sau mã hóa dựa theo giá trị của nhóm bit đó. Thực hiện tương tự đến hết dãy byte, ta được một chuỗi các ký tự sau mã hóa.

Bộ ký tự sau mã hóa của base64:

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/
Padding		=									

Bộ ký tự sau mã hóa của base32:

Value	Symbol	Value	Symbol	Value	Symbol	Value	Symbol
0	0	9	9	18	I	27	R
1	1	10	A	19	J	28	S
2	2	11	B	20	K	29	T
3	3	12	C	21	L	30	U
4	4	13	D	22	M	31	V
5	5	14	E	23	N		
6	6	15	F	24	O		
7	7	16	G	25	P		
8	8	17	H	26	Q	*pad*	=

Vì nhóm mỗi 5 bit (với base32) hoặc 6 bit (với base64), nên để chẵn thì chúng ta cần thêm các bit dấu vào dãy bit của số nguyên để sao cho số byte chia hết cho 5 (với base32) hoặc chia hết cho 3 (với base64). Các nhóm bit mang giá trị 0 dư ra sau cùng sẽ được gọi là padding và thể hiện trong chuỗi mã hóa là ký tự '='.

Ví dụ: Số nguyên 952395 có dạng nhị phân là $(00001110\ 10001000\ 01001011)_2$, gồm 3 byte

Đổi sang base32:

Cần thêm 2 byte để đủ 5 byte, đồng thời sắp lại theo yêu cầu phía trên, ta được:

01001011 10001000 00001110 00000000 00000000

Tách thành từng nhóm 5 bit:

01001 01110 00100 00000 11100 00000 00000 00000

Đối chiếu với bảng mã được chuỗi sau mã hóa: 9E40S===

Đổi sang base64:

Số byte chia hết cho 3 nên không cần thêm byte padding, đồng thời sắp lại theo yêu cầu phía trên, ta được:

01001011 10001000 00001110

Tách thành từng nhóm 6 bit:

010010 111000 100000 001110

Đối chiếu với bảng mã được chuỗi sau mã hóa: S4gO

❖ Mã hóa base58:

Không giống base32 và base64, base58 không mã hóa bằng cách gộp từng cụm bit. Thực tế, ta mã hóa base58 một số thập phân bằng cách biểu diễn số đó dưới hệ cơ số 58 và áp dụng bộ ký tự mã hóa sau:

Value	Character	Value	Character	Value	Character	Value	Character
0	1	1	2	2	3	3	4
4	5	5	6	6	7	7	8
8	9	9	A	10	B	11	C
12	D	13	E	14	F	15	G
16	H	17	J	18	K	19	L
20	M	21	N	22	P	23	Q
24	R	25	S	26	T	27	U
28	V	29	W	30	X	31	Y
32	Z	33	a	34	b	35	c
36	d	37	e	38	f	39	g
40	h	41	i	42	j	43	k
44	m	45	n	46	o	47	p
48	q	49	r	50	s	51	t
52	u	53	v	54	w	55	x
56	y	57	z				

Cách chuyển đổi tương tự như ta đã sử dụng trong hàm bigIntToDecStr (thực tế chính là hàm to_base10)

Ví dụ: $3456801_{10} = 17x58^3 + 41x58^2 + 34x58^1 + 1x58^0$

=> Đối chiếu với bộ ký tự mã hóa và biểu diễn theo base58: Jib2

❖ Hàm pow:

Xét res = pow(a,e)

Có nhiều thuật toán để tính lũy thừa của một số. Tuy nhiên, hàm pow() trong chương trình sử dụng đệ quy, kỹ thuật chia để trị:

Trường hợp cơ bản: e = 0: trả về BigInt 1

Còn lại, với n = pow(a , e << 1) :

- Nếu e là số chẵn, trả về: n * n

- Nếu e là số lẻ, trả về: n * n * a

Cách làm trên cho độ phức tạp thuật toán $O(\log_2(e))$ là rất tốt, lại sử dụng được hàm nhân và hàm dịch bit có sẵn, cho tốc độ xử lý nhanh.

❖ Hàm digits:

Hàm này trả về số lượng ký tự số của giá trị, biểu diễn trên hệ thập phân. Như vậy, ta thực hiện chia giá trị a cho số BigInt 10 cho đến khi a = 0. Mỗi lần chia ứng với một ký tự số. Ta đếm và trả về số lượng ký số đó.

PHẦN 3: TỔNG KẾT ĐỒ ÁN

Đồ án của nhóm chúng em đến đây đã cơ bản hoàn thành. Qua quá trình kiểm thử và sửa lỗi, chúng em nhận thấy rằng nhìn chung chương trình đã đạt đầy đủ những mục tiêu đề ra, và nhóm xin tự nhận xét những ưu khuyết của sản phẩm:

❖ Ưu điểm:

- Có đầy đủ các phép toán trên bit, các phép toán, phép gán, phép so sánh cơ bản
- Sử dụng hàm operator, giải phóng bộ nhớ tự động, giúp việc sử dụng BigInt đơn giản và gần gũi như dùng các kiểu dữ liệu khác
- Bộ xử lý lệnh input mở, giúp việc tạo thêm lệnh input mới nhanh và dễ dàng

❖ Nhược điểm:

- Tốc độ xử lý, tính toán còn chậm ở một số hàm, đặc biệt là hàm is_prime

Trên đây chính là báo cáo đồ án, bản nộp chính thức cho đồ án môn học Kỹ thuật lập trình của nhóm 10 - lớp 20CTT1TN kính gửi đến thầy. Những tài liệu cần thiết: hướng dẫn sử dụng, bảng phân công, video demo ... sẽ được gửi kèm như theo quy định.