

# 큰 수 곱셈 구현

## Big Number Multiplication

System Programming Assignment

100,000 × 100,000 연산 구현 및 16진수 출력

학과: 모바일시스템공학과

학번: 32240633

이름: 김민준

제출일: 2025년 11월 8일

GitHub: [github.com/codeminjun/big-mult](https://github.com/codeminjun/big-mult)

## Contents

<b>1</b>	<b>서론</b>	<b>2</b>
1.1	과제 개요	2
1.2	문제 정의	2
<b>2</b>	<b>구현 방법</b>	<b>2</b>
2.1	자료구조 설계	2
2.2	알고리즘 설계	2
2.3	핵심 함수 구현	3
2.3.1	big_mult() 함수	3
<b>3</b>	<b>실행 결과 및 검증</b>	<b>3</b>
3.1	메인 과제 결과	3
3.2	추가 테스트 케이스	4
3.3	실행 스크린샷	4
<b>4</b>	<b>코드 분석 및 평가</b>	<b>4</b>
4.1	장점	4
4.2	개선 가능 사항	5
4.3	향후 개선 방안	5
<b>5</b>	<b>결론</b>	<b>5</b>
<b>A</b>	<b>전체 소스 코드</b>	<b>6</b>

# 1 서론

## 1.1 과제 개요

본 과제는 일반적인 32비트 정수형 변수로는 표현할 수 없는 큰 수의 곱셈을 구현하는 것이다. 구체적으로  $100,000 \times 100,000 = 10,000,000,000$  (100억)을 계산하고 결과를 16진수로 출력하는 프로그램을 작성한다.

## 1.2 문제 정의

- 32비트 정수(int)의 최대값:  $2^{31} - 1 = 2,147,483,647$  (약 21억)
- 목표 계산:  $100,000 \times 100,000 = 10,000,000,000$  (100억)
- 문제: 100억은 32비트 정수 범위를 초과하여 오버플로우 발생
- 해결책: 자체 큰 수 연산 라이브러리 구현

# 2 구현 방법

## 2.1 자료구조 설계

큰 수를 저장하기 위해 다음과 같은 구조체를 설계하였다:

```

1 #define MAX_DIGITS 20
2
3 typedef struct {
4     int digits[MAX_DIGITS];    // Store each digit
5     int length;                // Actual number of digits
6 } BigInt;

```

Listing 1: BigInt Structure Definition

설계 특징:

- `digits` 배열: 각 자리수를 역순으로 저장 (`digits[0]`이 일의 자리)
- `length`: 유효 자릿수를 추적하여 불필요한 연산 방지
- 역순 저장 이유: 낮은 자리부터 계산하므로 인덱스 접근이 용이

## 2.2 알고리즘 설계

초등학교에서 배운 필산 곱셈 방식을 구현하였다. 예를 들어  $123 \times 456$ 의 경우:

$$\begin{array}{r}
 & 123 \\
 & \times 456 \\
 \hline
 & 738 \quad (123 \times 6) \\
 & 6150 \quad (123 \times 5 \times 10) \\
 & 49200 \quad (123 \times 4 \times 100) \\
 \hline
 & 56088
 \end{array}$$

## 2.3 핵심 함수 구현

### 2.3.1 big\_mult() 함수

```

1 void big_mult(long long a, long long b, BigInt *result) {
2     BigInt num1, num2;
3     int_to_bigint(a, &num1);
4     int_to_bigint(b, &num2);
5     init_bigint(result);
6
7     for (int i = 0; i < num1.length; i++) {
8         int carry = 0;
9         for (int j = 0; j < num2.length; j++) {
10            int pos = i + j;
11            int product = num1.digits[i] * num2.digits[j]
12                           + result->digits[pos] + carry;
13            result->digits[pos] = product % 10;
14            carry = product / 10;
15        }
16
17        // Handle remaining carry
18        int pos = i + num2.length;
19        while (carry > 0) {
20            result->digits[pos] += carry;
21            carry = result->digits[pos] / 10;
22            result->digits[pos] %= 10;
23            pos++;
24        }
25    }
26 }
```

Listing 2: Big Number Multiplication Function

#### 알고리즘 핵심:

1. 각 자리수끼리 곱셈 수행
2. 현재 위치( $pos = i + j$ )에 결과 누적
3. 자리올림(carry) 처리
4. 시간복잡도:  $O(n \times m)$  ( $n, m$ 은 각 숫자의 자릿수)

## 3 실행 결과 및 검증

### 3.1 메인 과제 결과

프로그램을 실행한 결과는 다음과 같다:

==== 큰 수 곱셈 프로그램 ===

계산:  $100000 \times 100000$

결과:

10진수: 100000000000

10진수 결과: 100000000000

16진수 결과: 0x2540BE400

결과 검증:

- $100,000 \times 100,000 = 10,000,000,000 \checkmark$
- 16진수 변환:  $10,000,000,000_{10} = 2540BE400_{16} \checkmark$
- Python 검증: `hex(100000 * 100000) = '0x2540be400'`  $\checkmark$

## 3.2 추가 테스트 케이스

프로그램의 정확성을 검증하기 위해 다양한 테스트 케이스를 수행하였다:

Table 1: 테스트 케이스 결과

A	B	결과 (10진수)	결과 (16진수)	검증
12	34	408	0x198	$\checkmark$
123	456	56,088	0xDB18	$\checkmark$
999	999	998,001	0xF3A71	$\checkmark$
12,345	6,789	83,810,205	0x4FED79D	$\checkmark$
50,000	50,000	2,500,000,000	0x9502F900	$\checkmark$

모든 테스트 케이스가 정확하게 통과하였다.

## 3.3 실행 스크린샷

그림 1은 프로그램의 실제 실행 결과를 보여준다.

## 4 코드 분석 및 평가

### 4.1 장점

- 명확한 알고리즘: 초등학교 곱셈 방식으로 이해하기 쉬움
- 높은 정확도: 모든 테스트 케이스 통과
- 확장 가능성: MAX\_DIGITS를 조정하여 더 큰 수 처리 가능
- 디버깅 용이: 단계별 결과 확인 가능

```

==== Large Number Multiplication Program ====

Calculation: 100000 × 100000

Result:
Decimal: 10000000000
Decimal result: 10000000000
Hexadecimal result: 0x2540BE400

==== Additional Tests ===

12 × 34 = 408 (0x198) Correct!
123 × 456 = 56088 (0xDB18) Correct!
999 × 999 = 998001 (0xF3A71) Correct!
12345 × 6789 = 83810205 (0x4FED79D) Correct!
50000 × 50000 = 2500000000 (0x9502F900) Correct!
○ ubuntu@VM-dfe10dd7-44b6-4286-8081-121c40a675b5:~/v

```

Figure 1: 프로그램 실행 결과 스크린샷

## 4.2 개선 가능 사항

- 성능: 현재  $O(n^2)$  알고리즘은 매우 큰 수에 비효율적
- 메모리: 고정 크기 배열 사용으로 메모리 낭비 가능

## 4.3 향후 개선 방안

1. Karatsuba 알고리즘 적용:  $O(n^{1.585})$  시간복잡도
2. FFT 기반 곱셈:  $O(n \log n)$  시간복잡도
3. 동적 메모리 할당으로 메모리 효율성 개선
4. 뺄셈, 나눗셈 등 추가 연산 구현

## 5 결론

본 과제를 통해 일반적인 정수 타입의 한계를 극복하고, 자체 큰 수 연산 라이브러리를 성공적으로 구현하였다.  $100,000 \times 100,000 = 10,000,000,000$ 을 정확히 계산하고 16진수 결과 0x2540BE400을 출력하였다.

이 과정에서 다음과 같은 것들을 학습하였다:

- 적절한 자료구조 설계의 중요성
- 알고리즘의 이론과 실제 구현의 차이

- 자리올림(carry) 처리와 같은 세밀한 구현 기술
- 체계적인 테스트의 필요성

향후 성능 최적화와 기능 확장을 통해 더욱 완성도 높은 큰 수 연산 라이브러리로 발전 시킬 수 있을 것이다.

## A 전체 소스 코드

전체 소스 코드는 GitHub 저장소에서 확인할 수 있다:

<https://github.com/codeminjun/system-programming>

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4
5 #define MAX_DIGITS 20 // Maximum digits for 100,000 x 100,000
6
7 // Structure to store large numbers
8 typedef struct {
9     int digits[MAX_DIGITS]; // Store each digit (reversed)
10    int length;             // Actual number of digits
11 } BigInt;
12
13 // Initialize BigInt
14 void init_bigint(BigInt *num) {
15     memset(num->digits, 0, sizeof(num->digits));
16     num->length = 0;
17 }
18
19 // Convert integer to BigInt
20 void int_to_bigint(long long n, BigInt *result) {
21     init_bigint(result);
22
23     if (n == 0) {
24         result->digits[0] = 0;
25         result->length = 1;
26         return;
27     }
28
29     int i = 0;
30     while (n > 0) {
31         result->digits[i] = n % 10;
32         n /= 10;
33         i++;
34     }
35     result->length = i;
36 }
37

```

```

38 // Print BigInt in hexadecimal
39 void print_hex(BigInt *num) {
40     long long decimal_value = 0;
41     long long multiplier = 1;
42
43     for (int i = 0; i < num->length; i++) {
44         decimal_value += num->digits[i] * multiplier;
45         multiplier *= 10;
46     }
47
48     printf("Decimal result: %lld\n", decimal_value);
49     printf("Hexadecimal result: 0x%llx\n", decimal_value);
50 }
51
52 // BigInt multiplication function
53 void big_mult(long long a, long long b, BigInt *result) {
54     BigInt num1, num2;
55     int_to_bigint(a, &num1);
56     int_to_bigint(b, &num2);
57
58     init_bigint(result);
59
60     // Multiply each digit
61     for (int i = 0; i < num1.length; i++) {
62         int carry = 0;
63         for (int j = 0; j < num2.length; j++) {
64             int pos = i + j;
65             int product = num1.digits[i] * num2.digits[j]
66                         + result->digits[pos] + carry;
67             result->digits[pos] = product % 10;
68             carry = product / 10;
69         }
70
71         // Handle remaining carry
72         int pos = i + num2.length;
73         while (carry > 0) {
74             result->digits[pos] += carry;
75             carry = result->digits[pos] / 10;
76             result->digits[pos] %= 10;
77             pos++;
78         }
79     }
80
81     // Calculate actual length
82     result->length = num1.length + num2.length;
83     while (result->length > 1 &&
84             result->digits[result->length - 1] == 0) {
85         result->length--;
86     }

```

```
87 }
88
89 // Print BigInt in decimal (for debugging)
90 void print_decimal(BigInt *num) {
91     printf("Decimal: ");
92     for (int i = num->length - 1; i >= 0; i--) {
93         printf("%d", num->digits[i]);
94     }
95     printf("\n");
96 }
97
98 int main() {
99     printf("== Large Number Multiplication Program ==\n\n");
100
101    long long num1 = 100000;
102    long long num2 = 100000;
103
104    printf("Calculation: %lld x %lld\n\n", num1, num2);
105
106    BigInt result;
107    big_mult(num1, num2, &result);
108
109    printf("Result:\n");
110    print_decimal(&result);
111    print_hex(&result);
112
113    printf("\n== Additional Tests ==\n\n");
114
115 // Test cases
116 long long test_cases[][2] = {
117     {12, 34},
118     {123, 456},
119     {999, 999},
120     {12345, 6789},
121     {50000, 50000}
122 };
123
124    int num_tests = sizeof(test_cases) / sizeof(test_cases[0]);
125
126    for (int i = 0; i < num_tests; i++) {
127        printf("%lld x %lld = ",
128               test_cases[i][0], test_cases[i][1]);
129        big_mult(test_cases[i][0], test_cases[i][1], &result);
130
131        long long expected = test_cases[i][0] * test_cases[i][1];
132        long long calculated = 0;
133        long long multiplier = 1;
134
135        for (int j = 0; j < result.length; j++) {
```

```
136         calculated += result.digits[j] * multiplier;
137         multiplier *= 10;
138     }
139
140     printf("%lld (0x%llx) ", calculated, calculated);
141
142     if (calculated == expected) {
143         printf("Correct!\n");
144     } else {
145         printf("Error! (Expected: %lld)\n", expected);
146     }
147 }
148
149 return 0;
150 }
```

Listing 3: big\_mult.c - 전체 소스 코드