Essential Information For Using a2n.c and main.c

To compile a2n.c, enter the command line "gcc -fsanitize=undefined -g a2n.c -o a2n" in your terminal. This command line includes a flag for undefined integer behaviour, which can happen when we left-shift 1 by more than 30 bits. You can also compile with just "gcc a2n.c -o a2n", but there will be no warning if an n>30 is inputted. To run a2n.c, in the next command line, enter "./a2n a n", with a and n corresponding to a*2^n. For example, to calculate 3*2^6, enter "./a2n 3 6".

To compile main.c with the flag, similarly enter "gcc -fsanitize=undefined -g main.c -o main" in the terminal. To compile without the flag for undefined integer behaviour, enter "gcc main.c -o main". In the next command line, enter "./main n m" with n corresponding to 2^n and m being the number of times you would like to run each method of computation. For example, to calculate 2^5 and iterate each method of computation 1000 times, enter "./main 5 1000".

Ensure for both programs that n is between 1 and 30, inclusive.

Test Cases for Calculating a*2^n Using Bit Manipulation

n in 2^n	binary	number	compile status
29	01…(followed by twenty-nine 0's).	536870912 (correct value: 536870912)	completed
30	01…(followed by thirty 0's).	1073741824 (correct value: 1073741824)	completed
31	01…(followed by thirty-one 0's).	-2147483648 (correct value: 2147483648)	flagged
32	01…(followed by thirty-two 0's).	1 (correct value: 4294967296)	flagged

Table 1: Computing 2^n for n between 29 and 32, inclusive, with the purpose of finding the n value at which calculating 2^n using bit-shift manipulation is no longer accurate.

Questions:

1. **What is the command line that you used to compile the program?**

I typed in “gcc a2n.c -o a2n” in my terminal to compile this program, because the name of the file is a2n.c, “gcc” is the command for using the GNU compiler, and “-o a2n” is what names my executable file “a2n”. Furthermore, to run the program, I typed “./a2n a n” in the next command line, where a and n are the actual numbers that I want a and n to be (not just the letters “a” and “n”).

2. **Beyond what n are the results incorrect?**

   Starting from n=31, which is beyond n=30, the results of calculating 2^n using my_a2n(1, n) are incorrect.

3. **Why are the results incorrect from a certain n? Please be specific with numbers supporting your answer.**

   They are incorrect starting from n=31 because 2^31 is equal to 2147483648, but the upper bound for a 32-bit signed integer is ((2^31) – 1) = 2147483647. The actual value of 2^31 is too big for a 32-bit signed integer to hold. The function my_a2n(1,31) does give us a result of 1 with thirty-one 0’s after it, but this is instead interpreted using the two’s complement system. For a 32-bit integer, after 2147483647 (thirty-one 1’s in binary), the next binary number (1 with thirty-one 0’s after it) is interpreted as -2147483648 instead of 2147483648. The next binary representation (1 followed by thirty 0’s and then one 1) would be interpreted as -2147483647 instead of 2147483649. The next binary representation after that (1, then twenty-nine 0’s, and finally 10). Following this increasing pattern, the binary representation for 2^32 – 1 will be interpreted as -1. The interpretation of the binary representation for 2^32, however, is not 0. With 32-bits, we only have enough permutations to show numbers from 0 to (2^32 – 1), inclusive. The binary representation of 2^32 would require 33 bits (a 1 with thirty-two 0’s after it). What happens instead when we assign 1<<32 to a 32-bit data type is that the “1” is brought back to the rightmost bit (2^0 place). So, we have thirty-one 0’s followed by a 1, which is just 1 in both binary and base ten. The results we get for my_a2n(1, n) with n in range [0,31] will essentially line up with the results for n in range [32, 63] (both intervals are inclusive).

4. **Is there any compiler flag we can use to give us a warning avoiding such problems?**

We can type in “gcc -fsanitize=undefined -g a2n.c -o a2n” as our first command line. The “-fsanitize=undefined -g” is what will flag the undefined integer behaviour if we enter “./a2n a n” with n>30 in our next command line. The “-g” tells the compiler to include debugging information and the “-fsanitize=undefined” specifically is what completes runtime checks for any undefined behaviour.

Case Study for CPU Time Required to Calculate a*2^n Using Three Different Methods

m	bit-shift	pow(2,n)	2^n with a loop
10	0.000016	0.000003	0.000001
100	0.000019	0.000004	0.000004
10000	0.000042	0.000106	0.000104
100000000	0.158044	0.676815	0.669956
1000000000	1.560896	6.827214	6.721497

Table 2: CPU time when n=2 in seconds.

m	bit-shift	pow(2,n)	2^n with a loop
10	0.000020	0.000004	0.000003
100	0.000017	0.000003	0.000009
10000	0.000049	0.000108	0.000712
100000000	0.158655	0.677752	4.688755
1000000000	1.546937	6.775313	46.785259

Table 3: CPU time when n=28 in seconds.

Please note that the “gcc main.c -o main” was used to compile the program here (without the compiler flag).

Questions:

1. **Why does the ratio of CPU times between different methods vary when we change the number of times (m) the computation is repeated? In other words, which one is more reliable between m = 10^1 and m = 10^8.**

   For each time we complete a computation using a certain method, the CPU time can differ slightly due to random factors like background processes or other small system activities. It’s kind of like running an experiment: the more times you repeat it, the more accurate and consistent your overall results will be. So, when m = 10^1, there’s more room for randomness to affect the result. But with m = 10^8, those random fluctuations tend to even out, making the timing more reliable, which will

also make the ratio between the CPU times for the different methods more accurate. That's why a larger m gives a better comparison between different methods.

2. **Why does the CPU time ratio differ when the input n is changed? Which n gives us better case study results?**

   The bigger the n, the better our case study results will be. Let's say the time complexity of the bit-shift method is f(n), for-loop method is g(n), and the pow function method is h(n). To compare the time complexity of the bit shift method to the for-loop method for example, we can take the limit of f(n)/g(n) as n approaches infinity. If the bit-shift method is slower, this limit will not exist (f(n)/g(n) will approach infinity). If the g(n) is slower, then this limit will be 0. If f(n) and g(n) are of the same order, then this limit will approach a non-zero finite constant. We can use the same reasoning for g(n)/h(n) and f(n)/h(n), or the reciprocals of any of these ratios. Since taking these limits as n approaches infinity, bigger n's will definitely give us better approximations for the CPU time ratios, since CPU time matters more as n increases.

   To concretely explain why the CPU time ratio changes as n changes, let's say f(n) is an element of O(1) and g(n) is an element of O(n). Then, f(n)/g(n) will be much larger when n=2 than when n=28. This is because g(n) is of higher order and therefore grows much faster (will have far more steps to do the same task) than f(n). This is also exactly why bigger n's give us a better idea of which algorithm is faster or slower.

Appendix

Here are all lines of code in a2n.c and main.c.

a2n.c

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

void my_a2n(int a, int n);

int main(int argc, char *argv[])
```

```
{
    if (argc != 3)
    { // argv[0] = "./a2n", argv[1] = "a", argv[2] = "n". argc =
sizeOf(argv)/sizeOf(argv[0])
        printf("Incorrect number of inputs! Expected two numbers separated by a
space.\n");
        return 1;
    }

    int a = atoi(argv[1]);
    int n = atoi(argv[2]);
    if (n < 0)
    {
        printf("Please ensure n is non-negative.\n");
        return 1;
    }
    my_a2n(a, n);
}

void my_a2n(int a, int n)
{
    int result = a << n;
    printf("Calculation result using my_a2n: %d\n", result);
}
```

main.c

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

void my_a2n(int a, int n, int m);
void a2n_with_pow(int a, int n, int m);
void a2n_with_loop(int a, int n, int m);

int main(int argc, char *argv[])
{

    clock_t start, end;
    double my_a2n_time, a2n_with_pow_time, a2n_with_loop_time;

    if (argc != 3)
    { // argv[0] = "./main", argv[1] = "n", argv[2] = "m". argc =
sizeOf(argv)/sizeOf(argv[0])
```

```
        printf("Incorrect number of inputs! Expected two numbers separated by a
space.\n");
        return 1;
    }

    int n = atoi(argv[1]);
    int m = atoi(argv[2]);

    if (n < 0 || m < 0)
    {
        printf("Please ensure n and m are non-negative.\n");
        return 1;
    }

    start = clock();
    my_a2n(1, n, m);
    end = clock();
    my_a2n_time = (double)(end - start) / CLOCKS_PER_SEC;
    printf("CPU time for my_a2n: %lf seconds\n", my_a2n_time);

    start = clock();
    a2n_with_pow(1, n, m);
    end = clock();
    a2n_with_pow_time = (double)(end - start) / CLOCKS_PER_SEC;
    printf("CPU time for a2n_with_pow: %lf seconds\n", a2n_with_pow_time);

    start = clock();
    a2n_with_loop(1, n, m);
    end = clock();
    a2n_with_loop_time = (double)(end - start) / CLOCKS_PER_SEC;
    printf("CPU time for a2n_with_loop: %lf seconds\n", a2n_with_loop_time);
}

void my_a2n(int a, int n, int m)
{
    int result = 0;

    for (int i = 0; i < m; i++)
    {
        result = a << n;
    }

    printf("Calculation result using my_a2n: %d\n", result);
}

void a2n_with_pow(int a, int n, int m)
{
    int result = 0;
```

```
    for (int i = 0; i < m; i++)
    {
        result = a * pow(2, n);
    }

    printf("Calculation result using a2n_with_pow: %d\n", result);
}

void a2n_with_loop(int a, int n, int m)
{
    int result = a;
    for (int i = 0; i < m; i++)
    {
        result = a;
        for (int j = 0; j < n; j++)
        {
            result *= 2;
        }
    }
    printf("Calculation result using a2n_with_loop: %d\n", result);
}
```