Austin Stau

Lab0x03

October 27, 2021

CSC 482

Lab0x03

**Algorithm A**

Looking at this algorithm I was seeing how many loops it would iterate when using only 1 test case. I put counts in each loop to see. I had a max N of only 16 to keep it small.

A picture containing text

Description automatically generated

So, in the loops we can see that the A count and B count are for the if else statement A being the if and B being the else. Looking at it, every A and B count always equal up to the cumulative while count. So, this functions like a standard for loop. My algorithm is also taking random numbers as well and comparing them. A/B counts change, but the while counts never change. Taking a look at the breakdown of the algorithm.

Text

Description automatically generated with medium confidence

C – overhead cost of everything

(n x a) – first loop

(n x b) – second loop

\*\*C would also be considered for the counters\*\*

We have an either-or situation though, these both affect the loop, but only one of them. They either iterate the a or decrement the b, not both, so this follows the same route of a standard for loop.

So, for this it is T(n)~c\*N time. Here is an actual test:

A picture containing text

Description automatically generated

The algorithm is showing that it is doing T(n)~c\*N time by the doubling ratio.

**Algorithm B**

For this one I broke down the algorithm into pieces to see what each one had. I did the inner loop first with. The comparison for the most part will just be considered C or overhead. The comparison doesn’t affect the loop but does add some time because of the working pieces. I’m more concerned with the two loops. Breaking down the first loop inside:

|  |  |  |
| --- | --- | --- |
| N | Count | Log2(n) |
| 2 | 1 | 1 |
| 4 | 1 | 2 |
| 8 | 2 | 3 |
| 16 | 2 | 4 |
| 32 | 3 | 5 |
| 64 | 4 | 6 |
| 128 | 5 | 7 |
| 256 | 6 | 8 |
| 512 | 7 | 9 |
| 1024 | 8 | 10 |

This is the inner loop, and the count is how many times it went through the loop per N times. This looks to be 2log(n), though at first, I thought it might be log(n)\*log(n) or log(n)2 or log(n)\*log(n-2), I show the time test for that later.

Here is the second outer loop broken down:

|  |  |  |
| --- | --- | --- |
| N | Count | Log2(n) |
| 2 | 1 | 1 |
| 4 | 2 | 2 |
| 8 | 3 | 3 |
| 16 | 4 | 4 |
| 32 | 5 | 5 |
| 64 | 6 | 6 |
| 128 | 7 | 7 |

We can see this follow log2(n) times fairly easily, so we know the outer loop is log(n).

Here is the time test for that log(n)2:

A picture containing calendar

Description automatically generated

Then I thought that maybe the algorithm was running log(n)\*log(n-2), here’s the time test for that:

A picture containing table

Description automatically generated

This was not the solution, then I took a closer look and that’s where I got my answer from the table on the other page where this is probably 2log(n) \* log(n).

Here’s the time test for this

Table

Description automatically generated

So, I conclude that this algorithm is running at T(n)~c\* 2log(n) \* log(n). Even if my actual algorithm isn’t that fast, it might be for the overhead C.

**Algorithm C**

Before I did the master theorem, I wanted to take a look at what was coming out of each phase.

I commented out all the recursion to see what kind of timing they were giving and got this table:

|  |  |  |
| --- | --- | --- |
| N | Count | Log2(n) |
| 2 | 1 | 1 |
| 4 | 2 | 2 |
| 8 | 4 | 3 |
| 16 | 8 | 4 |
| 32 | 16 | 5 |
| 64 | 32 | 6 |
| 128 | 64 | 7 |

Here is the test:

A picture containing calendar

Description automatically generated

This was set to 1 run so the Alg call is only ever going to be 1, but the first loop count and second loop counter both equal n/2. Since these both get called, I’m saying this can just be consider n for both.

The count here is the number of times the loop was iterated.

So, each of the for loops was giving me that same number of loops at N/2 time. Now doing this with just one recursion I get this:

Here’s the run for just running 1 recursion:

A picture containing table

Description automatically generated

There you can see that each Alg call is the same as log(n), each recursion produces the same results.

|  |  |  |
| --- | --- | --- |
| N | Count | Log2(n) |
| 2 | 1 | 1 |
| 4 | 2 | 2 |
| 8 | 3 | 3 |
| 16 | 4 | 4 |
| 32 | 5 | 5 |
| 64 | 6 | 6 |
| 128 | 7 | 7 |

The count here is the number of times the algorithm was called.

So, we have a log(n) time for each recursion every time we call it.

So, theoretically we have (n/2+n/2) \* log(n)3

Using the master theorem, we have:

A = 3 – we have 3 recursion so I’m using this

B = 3 – The output I was seeing was that this was splitting this operation into 4 separate parts, such as at N=8 and N=16 there was a difference of x3 algorithm calls.

Log3(3) = 1 = ccrit = c so case 2.

3T(N/3) + k\*W so f(n) = T(n) = O(nlogb(a)log(n)) we have T(n) ~ c\*n\*log(n)

Here are the final results for that:

A picture containing text

Description automatically generated

It looks like the doubling ratio and the expected ratio seem to be close.

**Algorithm D**

I’m thinking that master theorem cannot be used here. I know this is a Fibonacci sequence where we use the last N or N-1 and second last of N-2 and add them together to get our new number.

So we have:

T(n) = T(n-1)+T(n-2) + constant

T(n) = is going to equal some form of 2^2c+2^3c+2^4c+2^5c…2^{n}-1\*c, with c here being some constant.

So we’ll get something like T(n) = c \* 2^n – 1 the c and -1 can go away because in the big picture they do not do much, so were left with 2^n.

For this sequence I’ll give it T(n) = c \* 2n

Here are the results:

A picture containing text, computer

Description automatically generated