CS 58000\_01 Algorithm Design, Analysis & Implementation(3 cr.)

Assignment As\_01

Student Name: Truc Huynh

80 – 28 = 52

Why is your filename an ….CS486\_As01\_\_\_36?????

If this was your solution for CS 486, then I would say that you did not improve the analysis of algorithms.

I: [80 points]: each of I.1 through I.8 is 10 points.

**(I.1) Give the input and output specifications for :**

1. the algorithm a?
2. the algorithm b?

Answer:

|  |  |  |
| --- | --- | --- |
|  | algorithm a | algorithm b |
| Input | Two n-bit integer x and y, where y ≥ 1 | Two n-bit integer x and y, where y ≥ 1 |
| Output | The quotient and reminder of x divided by y | The quotient and reminder of x divided by y |

-4

Answer: For both a and b, their input and output specifications are as follows:

Input specification: {x, y | x and y are two integers, where **x ≥ 0, y ≥ 1**}.

Output specification: {q, r | x = q\*y + r, where **0 r < y**}. q and r are the quotient and remainder for y dividing x.

**(I.2) What is the input size for:**

1. the algorithm a?
2. the algorithm b?

Answer:

|  |  |
| --- | --- |
|  | algorithm a & algorithm b |
| Input Size | Algorithm a & algorithm b will have the same input size as they take x and y for division  The input x or y will be an integer n. The input size for each of them will be└ log2 n ┘ + 1 bits  Define the input size as the number of symbols (in this case bits) used for encoding a positive integer n. |

Answer: For both (a) and (b), their input size is:

The number of bits for representing the value of x, say m. m = └ log2 x┘ + 1.

The number of bits for representing the value of y, say n. n = └ log2 y┘ + 1.

Not n but x and y. -3

**(I.3) What is the basic operation for:**

1. the algorithm a?
2. the algorithm b?

Answer:

|  |  |  |
| --- | --- | --- |
|  | algorithm a | algorithm b |
| Basic Operation | Right shift (bit additions) | Left shift (bit multiplication)  Right shift (bit additions) |

For (a): The basic operation is the **compare operation ( r ≥ y),** or “either + or :=”

in “(r := r + -y); or q := q + 1;”.

For (b): The basic operation could be: compare operation ( x == 0) or **integer**

**division └ x/2 ┘** (shift x right by one bit) for each recursive call. Both can be used to define the number of recursive calls.

-4

**(I.4) In algorithm b, what is the functionality of the following segment of statements?** (i.e., give the reasons for writing each of the statements. Why double the q and r? Why increase r by one when x is odd? Why reduce r by y and increase q by one when r ≥ y?)

q := 2 \* q; r := 2 \* r;

if (x is odd) then r := r + 1;

if (r ≥ y) then

{ r := r – y; q := q + 1};

Answer:

|  |  |
| --- | --- |
| Why double the q and r? | In the recursive call, we divide x by 2. There we must double q and r to make it even. It can be done by shift left one bit |
| Why increase r by one when x is odd? | Because we want to get the top division (not the floor division) when using integer division. We want to change it to an even number that can divide by 2. Ex: 7/2 = 3 (floor division) but that is not what we want to get. Therefore, we add (7+1)/2 = 4 |
| Why reduce r by y and increase q by one when r ≥ y?) | * Reduce r by y to get the remaining value of x * Adding 1 to q because we just subtract y from (reduce) the value of r. Thus, we prove that previous r (before the subtraction) can divide one more y -> Increase q by 1 |

-6

Suppose that preparing for the first recursive call,

**└x/2┘ = q0 \* y + r0** …..…. (1)

Then, where x is even, └x/2┘ = **x/2 = q0 \* y + r0,** since x is divisible by 2.

**x = 2q0 \* y + 2r0 ;** …….(2)

where x is old, └x/2┘ = **(x-1)/2 = q0 \* y + r0**

**x = 2q0 \* y + 2r0 + 1** …….(2)

From (2), q1 = 2q0  and r1  = 2r0, then qi+1 = 2qi  and ri+1  = 2ri;

**these lead to “q := 2 \* q, r := 2 \* r;”,**

Preparing for the ith recursive call, (2) implies (2’),

x = 2qi-1 \* y + 2ri-1 , where x is even,

or x = 2qi-1 \* y + 2ri-1 + 1, where x is old. …….(2’)

That is, after the first recursive call,

r1  = 2r0 , where x is even,

or r1 = 2r0 + 1, where x is old. …………..(3)

(3) implies (3’):

That is, after ith recursive call,

ri  = 2ri-1 , where x is even,

or ri = 2ri-1 + 1, where x is old. …………..(3’)

This leads to “if (x is odd) then r := r + 1; ”, where x is odd.

However, cases would arise: either if (r1 < y)

or, if (r1 ≥ y).

According to if (r ≥ y) then

{ r := r – y; q := q + 1};

For if (r1 < y) then no action “{ r := r – y; q := q + 1};” is needed to be taken.

For if (r1 ≥ y), this implies that 2y is r0. Since (r1 ≥ y) then

{ r2 := r1 – y; q2 := q1 + 1};

implying that if (ri ≥ y) then

{ ri+1 := ri – y; qi+1 := qi + 1}; ….. (4)

For if (r1 ≥ y) then execute “{ r := r – y; q := q + 1};”; reduce r1 by one times of y to generate r2 = r1 – y; 0 r2 < r1 . and q2 := q1 + 1. That means x = q2 \* y + r2 . That is, r2 = x – q2\*y.

This leads to “there exists an integer i ≥ 1, such that 0 ri – i\*y < y.

Equivalently, there exists an integer i ≥ 1, such that 0 i\*y ri < (i + 1)y.

Assume that these are true for i, “there exists an integer i ≥ 1, such that 0 ri – i\*y < y.”

if (ri < y) is true, no further execution of “{ r := r – y; q := q + 1};” is needed. Otherwise,

if (ri ≥ y) is true, then ri+1 = ri - y, and qi+1 := qi + 1, such that x = qi+1 \* y + ri+1 .

if x = qi+1 \* y + ri+1 , then x = (qi + 1) \* y + (ri – y ) using (4).

x = (qi \*y + 1 \*y) + (ri – y ) = qi \*y + ri

Therefore, if ( ri ≥ y ) then ri+1 = ri - y; and to keep the equations (2) hold, qi+1 := qi + 1 for each of the subtract y from r. That is, this leads to

“if (r ≥ y) then { r := r – y; q := q + 1};”

**(I.5) Analyze and derive the algorithm’s time and space efficiency for Algorithm a**.

(Hint: express time efficiency in terms of summation )

Given algorithm a,

if x = 0, then return (q, r) := (0, 0);

q := 0; r := x;

while (r ≥ y) do // takes n iterations for the worse case.

{ q := q + 1;

r := r – y}; // O(n) for each r – y, where y is n bits long.

return (q, r);

Answer:

|  |  |
| --- | --- |
|  | algorithm a |
| Time Efficiency | Takes 2n iterations for the worse case, which says x/1 where x is n bits long with a max value 2n – 1. Since this takes linear time O(n) for each iteration. The worse case would be O(n2n) |
| Space Efficiency | Take 4 spaces: x, y, q, r |

Consider the worst-case, name y = 1. Assume that x is n + 1 bits long. The position of bit at n is a sign bit. The maximum value of x would be 2n – 1. Then the value of x will take (2n – 1) time reducing the value of x by 1 to get down to 0. That is, (2n – 1) number of times executing (r ≥ y ). For a large n, (2n – 1) 2n. That means it requires 2n time for executing the ‘+’ addition operation for the statement r := r + (-y); and likewise 2n time for executing the ‘+’ addition operation for the statement q := q + 1;

Since each addition operation requires O(n), the number of times for an addition operation could be n 2n for executing r = r – y in the **worse case; which is exponential, O(n 2n ) = O( 2n )**

Likewise for the best case, either x = 0, or x 0 and x < y, or if x 0 and x ≥ y, where x = y, then **Ω(1), for the best case.**

For **the average case, it is O( 2n ).**

For space efficiency,

it requires space for each of the variables x, y, q, and r. It requires a constant c space. Or it requires cn bits, where n is the maximum bits for the max{x, y, q, r} bits. **That is, space efficiency is ϴ(1).**

-4

**(I.6) Analyze and derive the algorithm’s time and space efficiency for algorithm b.**

(Hint: for time efficiency, use recurrence relation and solve for the recurrence relation system)

Given algorithm b,

if (x = 0) then return (q, r):=(0, 0);

(q, r) := divide(└x/2┘, y ) //requires n-bits right shift

q := 2 \* q, r := 2 \* r; // shift left one bit.

if (x is odd) then r := r + 1; // needs c\*n-bits

if (r ≥ y) then // additions

{ r := r – y; q := q + 1};

return (q, r);

Answer:

|  |  |
| --- | --- |
|  | algorithm a |
| Time Efficiency | * Takes n call, so the worse case would be O(n2) |
| Space Efficiency | * Take 4 spaces: x, y, q, r |

For time efficiency

**Assume x ≥ y where x is n bits long and n = └ log2 x┘ + 1.**

If x = 0, then one compare operator (=) and two := operations

If x 0 then it has n = └ log2 x┘ + 1 number of recursive calls. For each call, there is a shift right one bit on x. **That means, there will be n numbers of recursive calls**. Upon return from each recursive call, it requires two := operations for (q, r).

For each recursive, upon each of the returns, it needs two numbers of left shift on q and r. two := operations from q := **2 \* q**, r := **2 \* r;** Since it has n times of recursive call, then it requires 2 n shift left of q and r, and 2 n of := operation. **That means these require each O(n) for each recursive call.**

**if (x is odd) then r := r + 1;** To determine whether (x is odd) by examine the rightmost digit of x whether is 1 (for x is odd) or 0 (for x is even). And, if x is odd, then it requires one + and one := operations. Since it has n times of recursive call, each of them at most would be executed n times. **That is,** **this requires O(n) for each recursive call.**

Then one r ≥ y compare operation, and if r ≥ y, then execute two + and two := operations **for { r := r – y; q := q +1} statements**. We can assume that **they require O(n) for each recursive call.**

**Since there is an n number of the recursive call, and upon each return, it requires O(n) for executing the rest of the statements:**

**n(O(n)) =** **O(n2), where n = └ log2 x┘ + 1.**

For space efficiency,

It requires space for x, y, q, r, and space cn for the return address after each recursive call. It has **ϴ(n).** space for a return address. Otherwise, it is a constant **ϴ(1).**

-3

**(I.7) Can these two algorithms a and b be improved? Justify your answer.**

1. for the algorithm a?
2. for the algorithm b?

Answer:

|  |  |  |
| --- | --- | --- |
|  | algorithm a | algorithm b |
| Improvement | * No improvement if the use of iteration is applied. * Can only improve by using recursive (algorithm b) because time complexity is huge compared to algorithm b (O(n2n) > O(n2)) * x must be ≥ 0 for the input (precondition). | * Pretty optimal already (in terms of time and space complexity). * x must be ≥ 0 for the input (precondition). |

Answer:

1. Since **algorithm b is an improvement when compared to algorithm a in terms of time**. The algorithm b **reduces an exponent time** from algorithm a to **quadrative time,** despite the algorithm b having O(n) space complexity compared to the O(1) for algorithm a.
2. For some cases, **algorithm b could be reduced to O(1) space complexity**.

-2

**(I.8) Compare these two algorithms a and b:**

Which is a better algorithm in terms of time and space efficiency? Justify your answer.

Answer:

|  |  |  |
| --- | --- | --- |
|  | algorithm a | algorithm b |
| Time efficiency | * O(n2n) is slower algorithm b | * O(n2) is faster than algorithm a |
| Space efficiency | * Take 4 spaces: x, y, q, r. The calculation between them is stored in the computer register (the number of registers depends on the computer). Thus, the same efficiency as algorithm b. | * Take 4 spaces: x, y, q, r. Thus, the same efficiency as algorithm a |

Answer

Let n = = └ log2 x┘ + 1 bits. For the **worse case**, algorithm a requires O(n2n) that grows exponentially, but algorithm b requires only O(n2). In the **best case**, both algorithms could be achieved by constant time O(1). For the **average case,** algorithm a is ϴ(2n) and algorithm b is ϴ(n2)

-2

Reference:

Purdue slides Ch000\_02\_IntroFoundation\_ProgCorrectionLec.pdf from CS 58000\_01 Algorithm Design, Analysis & Implementation