CS 58000\_01/02I Algorithm Design Analysis & Implementation(3 cr.)

Assignment As\_01

Student Name: Harsh Sharma

**Problem I [80 points] Algorithms Comparison**

Given two algorithms “function divide(x, y)” in my lecture note Ch 00\_02\_IntroFoundation\_ProgCorrectionLec.pptx, (posted in the Purdue.brightspace.com), the first one is iterative and the second one is recursive. The operators used are: “shift right one bit”, “shift left one bit”, copy, add (+), compare two contents of given variables (such as ), assign (:=), and call statement (such as a recursive call statement).

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

**(I.a) for the algorithm Ia**

The input specification are:

Two n-bit integers x and y, where y ≥ 1 and x > 0

Output Specification are:

The quotient (q) and remainder (r) of x divided by y.

**(I.b) for the algorithm Ib**

The input specification are:

Two n-bit integers x and y, where y ≥ 1 and x > 0

Output Specification are:

The quotient (q) and remainder (r) of x divided by y.

Note: **The input and output specification is the same as both the algorithms are performing to give us the same result, it’s just their methodology is different.**

(I.cd) What is the input size for:

(I.c) for the algorithm Ia?

Since both x and y are n-bit integers, the space required to store two n-bits is **2n.**

(I.d) for the algorithm Ib?

Since both x and y are n-bit integers, the space required to store two n-bits is **2n.**

(I.ef) What is the basic operation for :

(I.e) for the algorithm Ia?

**Iteration** is the basic operation where we perform **subtraction**. We perform this in the following pseudocode:

r := r - y

This is executed in the while loop.

(I.f) for the algorithm Ib?

**Recursion** is the basic operation where we perform **right shift operation** since it requires n-bits right shift. This is performed when calculating:

(q, r) := divide(└x/2┘, y ).

(I.g) In algorithm Ib, state the functionality for the following statements?

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};

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

This operation performs the left shift operation on q and r. This is achieved by multiplying by 2.

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

This is a conditional statement which checks if x is odd. If it is odd then add it by 1 to make it even. This is done because when we perform a binary right shift on an odd number, the least significant bit is effectively shifted out.

if (r ≥ y) then

Again, this is a condtional statement. If this condition is met, it indicates that y can be subtracted from r without making r negative which means r has at least y in its binary representation.

{ r := r – y; q := q + 1};

Since the above condition was true, y is subtracted from r. The quotient q is also incremented by 1 to keep track of how many times y has been subtracted from r. This step simulates a binary division operation.

(I.h) Analyze and derive the algorithm’s time and space efficiency for Algorithm Ia.

(Hint: express time efficiency in terms of summation )

Given algorithm Ia,

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

q := 0; r := x;

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

{ q := q + 1;

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

return (q, r);

Solution:

**Time Complexity**

At first we initiate q as 0 and r as x which are O(1) operation. Inside of while loop we subtract y from r, which is the remainder. The result will be at most ‘n’ bit since x is n bit. The operation:

r:=r + ( -y ),

takes fixed amount of time and hence can be considered as constant time of c1. And so the total running time for addition would be: c0 + c1n.

The constants can be ignored giving us the time complexity of O(n)

The algorithm produces the result (x – i \* y) < y after i iterations (represented as q = i). During each iteration, it involves 2(c0 + c1 n) additions/subtractions. Subsequently, x < (i + 1) \* y, which can be expressed as 𝑥/𝑦 < 𝑖+1 for exiting the iteration. If x consists of n bits, its maximum value lies within the range 0 < x ≤ 2n -1 < 2n.

In scenarios where x >> y and the minimum value of y is 1, the ratio 𝑥/𝑦 will grow to "2n" (provided that both x ≥ 0 and y > 0 are integers). Consequently, i will approximately reach "2n" in the worst-case scenario when y equals 1.

For the execution of these operations { r := r – y; q := q + 1; }, the algorithm's time complexity can be represented as ∑2^1 to "2n" of "2(c0 + c1 n)", which simplifies to "2n" \* ("2(c0 + c1 n))"= O(n \* 2n). This indicates an exponential time complexity of 2n to complete these operations. Now the time efficiency after ignoring the constants would be **O(n2n).**

**Space Complexity**

There are in total of four variables required to run the complete algorithm. And these four values will always be stored and updated irrespective of the value x and y. Therefor the space complexity will be O(4) which could be written as O(1).

(I.i) Analyze and derive the algorithm’s time and space efficiency for algorithm Ib.

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

Given algorithm Ib,

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);

Solution:

**Time Complexity**

x and y are both integers consisting of 2 ‘n’ bits.

Let's set n equal to 2k.

Consequently, the time efficiency can be expressed as follows: T(n) = T(2k)

= T(2k-1) + 2k

= T(2k-2) + 2k-1 + 2k

= T(2k-i) + (2k-i+1) + (2k-i+2) + ... + (2k-3) + (2k-2) + (2k-1) + (2k) = T(2k-k) + (2k-k+1) + (2k-k+2) + ... + (2k-3) + (2k-2) + (2k-1) + (2k),

If there are 'i' iterations where k = I,

= T(2k-k) + (2k-k+1) + (2k-k+2) + ... + (2k-3) + (2k-2) + (2k-1) + (2k),

= 1 + (21) + (22) + ... + (2k-3) + (2k-2) + (2k-1) + (2k),

= (2k+1 - 1),

= 2n - 1,

which is equivalent to O(n) for each recursive call.

Since the algorithm is recursive, it incurs n recursive calls.

Therefore, the time complexity is **O(n2)**

The **space efficiency** for the various variables assignment such as ‘q’, ’r’, ‘x’, ‘y’ are constant. Since this is a recursive algorithm and the respective variables are called recursively until x = 0, in the memory heap, there are n allocations made. Therefore, the efficiency can be 4\*O(n) or simply written as **O(n).**

(I.j) Can these two algorithms Ia and Ib be improved? Justify your answer.

1. for the algorithm Ia?

Solution:

For first algorithm we are only storing value of four constants irrespective of value of x or y or in other words the size of ‘n’ doesn’t affect. It has O(1) space complexity which is the best possible.

Whereas the time complexity of this algorithm is O(n2n), which definitely can be improved to O(n) or log(n). We can use bit manipulation to improve the time complexity.

1. for the algorithm Ib?

The space complexity is O(n) which is not the best. As we have seen in the first algorithm it could be achieved in 0(1) by keeping the track of quotient and remainder.

Regarding time complexity, algorithm 'b' currently operates at O(n^2). To enhance this algorithm's efficiency, we aim to reduce its time complexity to either O(n) or O(log n). Achieving this goal involves modifying the algorithm to incorporate bit manipulation techniques, resulting in a time complexity of O(log n).

(I.k) Compare these two algorithms Ia and Ib:

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

Solution:

Time Complexity: The time complexity of first algorithm is 2 power n, which is an exponential growth. For n = 2, algorithm will take only 4 but as we increase n to be 8 the time increases drastically to 256. Whereas the time complexity of second algorithm is n square, which means for n = 8, it will take only 64. Clearly this is less than 256 and hence the second algorithm performs well in regards to time complexity.

Space Complexity: As for time complexity the first algorithm outperforms the second one. It only needs four constant variable which makes the space complexity to be O(1). The time complexity of second algorithm is O(n). It uses recursion and stores the value of q and r in each iteration. And so the first algorithm is better performing than the second algorithm in respect to space complexity.

NOTE: First algorithm has better space complexity while the second algorithm has better time complexity. Depending on what the use case is we need to select the trade off.

**Problem II [30 points]: Polynomial function**

Given the following modular exponentiation called function modexp(x, y, N)

function modexp(x, y, N)

//Compute xy mod N

Input: Two n-bits integers x and N, an integer exponent y.

Output: xy mod N.

if (y == 0) then return 1;

z = modexp(x, └y/2┘, N); // z = x└ y/2 ┘ mod N

if (y is even) then return z2 mod N;

else return x \* z2 mod N;

(II.1) Apply function modexp(x, y, N) to compute mod 13. Show step by step. Complete the following table to show the execution of the algorithm modexp(15, 7, 13).

|  |  |  |
| --- | --- | --- |
| x, y, N | z = modexp(x, └y/2┘, N); | if (y is even) then return z2 mod N;  else return x \* z2 mod N; |
| 15, 7,13 | z = modexp(15, 7, 13). | if (y is even) then return z2 mod N;  else return x \* z2 mod N; |
| 15, 3,13 | z = modexp(15, 3, 13). | if (y is even) then return z2 mod N;  else return x \* z2 mod N; |
| 15, 1, 13 | z = modeexp(15, 1, 13) | if (y is even) then return z2 mod N;  else return x \* z2 mod N; |
| 15, 0, 13 | (Base Case: Return 1) |  |
| 15, 1, 13 | Returned: 1 |  |
| 15, 3, 13 | Returned: 1 |  |
| 15, 7, 13 | Returned: 8 |  |
|  |  |  |

Solution:

Given x = 15, y = 7, and N = 13.

Step 1:

As y is neither 0 nor even:

x \* z2 mod N = 15 \* z2 mod 13

Calculate z = modexp(x, └y/2┘, N)

z = modexp(15, └7/2┘, 13)

z = modexp(15, 3, 13)

Step 2:

Now, with z = modexp(15, 3, 13):

Given x = 15, y = 3, and N = 13.

Since y is neither 0 nor even:

x \* z2 mod N = 15 \* z2 mod 13

Calculate z = modexp(x, └y/2┘, N)

z = modexp(15, └3/2┘, 13)

z = modexp(15, 1, 13)

Step 3:

Now, with z = modexp(15, 1, 13):

Given x = 15, y = 1, and N = 13.

Since y is neither 0 nor even:

x \* z2 mod N = 15 \* z2 mod 13

Calculate z = modexp(x, └y/2┘, N)

z = modexp(15, └1/2┘, 13)

z = modexp(15, 0, 13)

Step 4:

Now, with z = modexp(15, 0, 13):

Given x = 15, y = 0, and N = 13.

Since y is 0, return 1

z = 15^0 mod 13 = 1 mod 13

z = 1

Substitute z=1 in step 3:

x \* z2 mod N = 15 \* 12 mod 13 = 15 mod 13 = 2

Hence, z=2 in step 2, and substitute the value of z in step 2:

x \* z2 mod N = 15 \* 22 mod 13 = 60 mod 13 = 8

Hence, z=8 in step 1, and substitute the value of z in step 1:

x \* z2 mod N = 15 \* 82 mod 13 = 960 mod 13 = 11

(II.2) How many times for execute the recursive calls modexp(x, └y/2┘, N)?

We start making recursive calls for y = 7, the second recursive call will be for 3, the third recursive call for 1, and the last recursive call for 0 which then returns the base case. So in total 4 recursive calls.

(II.3) Use the slide of the example to compute mod 13.

Let’s break down 7 to power of two’s = 4 + 2 + 1

= 22 + 21 + 20

Let’s calculate 152k for k = 4, 2, 1

151 mod 13 = 2

152 mod 13 = 4

154 mod 13 = 2252 mod 13 = 3

Now, 157 can be written as = 154 \* 152 \* 151

So, 157 mod 13 = ((154 mod 13) (152 mod 13) (151 mod 13))mod 13

= (2\*4\*3) mod 13

= 11

**Problem III(40 points) : Cutting a rope:**

A rope n inches long needs to be cut into n pieces 1 inch long.

(III.a) Outline/design an algorithm unitCut (do not give me any program code) that performs this task with the minimum number of cuts if several pieces of the rope can be cut at the same time.

Solution:

Since multiple ropes could be cut together, we can use the previously cut ropes together to give them a cut, cutting multiple ropes. We can start by cutting the ropes in middle and then cut the two new halves again and we can continue doing this until we get to length of 1.

The problem with this approach is that what if the length of rope is odd. Then we can’t cut the rope in half in a whole number.

In order to deal with the problem we can initially check if the length is odd or not. If it is, then cut the present rope for 1-inch giving us even length ropes. Then again we can continue cutting the ropes in halves.

Pseudocode for the same:

while (N > 1): //N is the length of rope

if N is odd:

N = N – 1 //if the length is odd then just perform one single cut.

N = N/2 // performing binary cut.

Case I: We run the algorithm until the length of rope becomes less than 1. If the length of rope of is 1 then the loop never runs and it handles the edge case.

Case II: When the length of the rope is odd then the conditional statement ‘if’ runs and we perform one single cut to make the length to be even.

Case III: When the length of rope is even then we simply reduce the length of rope in half, essentially performing binary cut.

(III.b) Also, give a formula for the minimum number of cuts.

Formula: **log2(N)**

Explanation: We are performing cuts in halves, bisecting the whole rope in two equal halves. Since these two halves could again be cut in half, we are essentially reducing the space in log of base 2. When the length of rope is odd then we execute ‘if’ condition which takes a constant K time. For sufficiently large length of rope, N, the formula would be: **log2(N)**

(III.c) If n is 50000 inches, for getting 50000 1-inch pieces, where is the first cut? And how many cuts are to be performed.

Solution: Since 50000 is an even number we simply cut the rope in half. Therefore, the first cut would be at 25000.

The number of cuts performed would be log2 (50000)

= 15.61

~ 16 cuts.

(III.d) What is the input size and time efficiency of your algorithm in (I.a)?

There’s only one input, which is the length of rope. Let it be N. Then the input size would be the space required to store N which is **└ log2 *N* ┘ + 1 bits.**

The time efficiency will be O(**log2 *N***) **.** The reason is that the size of the rope decreases in halves with each iteration.