Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

CMPT 706 - ASSIGNMENT 1

Question 1:

Consider the following basic problem. You are given an array A consisting of n integers A[1], A[2], . . . , A[n]. You'd like to output a two-dimensional n-by-n array B in which B[i, j] (for i < j) contains the sum of array entries A[i] through A[j] — that is, the sum A[i] + . . . + A[j]. (The value of array entry B[i, j] is left unspecified whenever i ≥ j, so it doesn't matter what is output for these values.)

Here is a simple algorithm to solve this problem.

```
for i = 1, 2, . . . n do
    for j = i + 1, i + 2, . . . n do
        Add up array entries A[i] through A[j]
        Store the result in B[i, j]
    end for
end for
```

(a) For some function f that should choose, show that the running time of the algorithm on an input of size n is Θ(f(n)).

Solution:

Now, let's analyse why this algorithm:

The algorithm in essence takes **g(n) = n^3, O(g(n)) time.**

Why so?: Because, we add n (not exactly n but still a loop is needed to compute the addition as a 'third for loop' that takes utmost n items) elements everytime we loop over i and j. So, in essence we will be looping over 3 iterations, thus the algorithm takes Big O- O(n^3) running time. And, all other operations performed inside the loops are basic operations that take some constant time say, O(1) (depending on the machine its operated on).

Therefore we can say that, *for g(n) = n^3 and f(n) is the naive algorithm under consideration, The running time of the algorithm is 0 <= f(n) <= k*g(n) for some k>0 (k is a constant) and n > n0, where n0 >= 1.*

Which implies the algorithm can run in Big O of g(n) time => running time of the algorithm = O(g(n)).

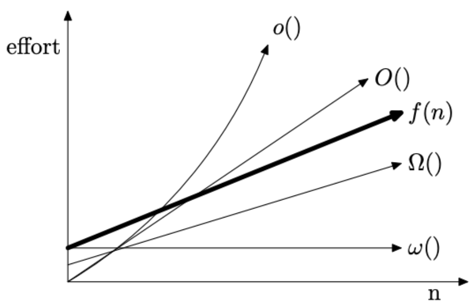
Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

Similarly, we can show that this naive algorithm is *Ω(g(n))*, because the best case scenario in this case also needs g(n) = n^3 iterations or asymptotic complexity.

=> the best case running time of this algorithm or the lower bound is Ω(n^3),

*f(n) is Ω(g(n)) (or f(n) ∈ Ω(g(n))) if there exists a real constant k > 0 and there exists an integer constant n0 ≥ 1 such that f(n) ≥ k * g(n) for every integer n ≥ n0.*

Therefore, **Θ(g(n)) according to its mathematical definition can be considered to be the intersection of O(g(n)) and Ω(g(n)).**



Therefore, the average case run time of this algorithm can be shown to be **Θ(g(n)), where g(n) = n^3,** *that is for exactly (asymptotically exact) f(n) == k * g(n), where k >0 is some constant and n >= n0 and n0>=1.*

(b) Although the algorithm you analyzed in part (a) is the most natural way to solve the problem — after all, it just iterates through the relevant entries of the array B, filling in a value for each — it contains some highly unnecessary sources of inefficiency. Give a different algorithm to solve this problem, with an asymptotically better running time. In other words, you should design an algorithm with running time o(f(n)).

Solution:

<u>New algorithm:</u>

for i = 1, 2, . . . n-1 do

Store A[i] to a temporary variable (or any variable)

for j = i + 1, i + 2, . . . n do

Add temporary variable with A[j] (where temp variable has our A[i] value stored)

Store the result in B[i, j]

end for

end for

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

Now, let's analyse why this algorithm:

The algorithm in essence takes **g(n) = n^2, o(g(n)) time.**

Why so?: Because, we don't add n (not exactly n even in the naive algorithm but still a loop was needed to compute the addition as a third for loop that takes utmost n items) elements everytime we loop over i and j. And, its tightly bound (strictly less than n^2) as essentially we aren't looping over n elements everytime-> the outer for loop (for i= 1 to n-1) loops n-1 times and the inner for loop (j = i+1 to n) loops utmost n-1 times. But, the inner for loop (j) decreases in the number of times it runs as i approaches n (n-1, n-2, n-3.., and so on..). And, all other operations performed inside the loops are basic operations that take some constant time say, O(1) (depending on the machine its operated on).

Therefore we can say that, *for g(n) = n^2 and f(n) is my algorithm under consideration,*

*The running time of the new proposed algorithm is 0 <= f(n) < k*g(n) for some k>0 (k is a constant) and n > n0, where n0 >= 1.*

Which implies the algorithm can run in small o of g(n) time => running time of the algorithm = o(g(n)).

<u>**Question 2:**</u>

The algorithm for computing a b by repeated squaring does not necessarily lead to the minimum number of multiplications. Give an example of b > 10 where the exponentiation can be performed using fewer multiplications.

Solution:

Let's take b=15

Binary representation 15 is => 1111 = 2^0+2^1+2^2+2^3 =15

<u>Repeated squaring method:</u>

a^15 = a^(1+2+4+8)

= **a * a^2 * a^4* a^8**

Now, to compute a^2 = a * a (one multiplication)

a^4 = (a^2)^2 = a^2 * a^2 (one multiplication)

a^8 = (a^4)^2 = a^4 * a^4 (one multiplication)

Then, we need a * a^2= a^ 3 (one multiplication)

a^4 * a^8= a^12 (one multiplication)

a^12 * a^3 = a^15 (one multiplication)

Therefore to **compute a^15 we need 6 multiplication steps by repeated squaring technique**.

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

Alternative method:
Consider the prime factorization of 15 = 1, 3, 5
Thus, a^15 = (a^3)^5 (or vice versa)
= a^(6 + 6 + 3) = a^6 * a^6 * a^3
= **(a^3)^2 * (a^3)^2 * a^3**
Now, to compute a^2 = a * a (one multiplication)
a^3 = a^2 * a (one multiplication)
a^6 = (a^3)^2 = a^3 * a^3 = (one multiplication)
a^6 * a^6 = a^12 (one multiplication)
a^12 * a^3 = a^15 (one multiplication)
Therefore **it can be seen that a^15 can be computed indeed in 5 multiplication steps** by this method, which is one less than (minimum number of steps) that of repeated squares.

Question 3:
In an RSA cryptosystem, p = 11 and q = 13. Find three appropriate pairs of exponents d and e.

Solution:
Given: p=11 and q=13
We know that, n = p*q = 11*13 = 143
And, ϕ(pq) = (p-1)*(q-1) = (11-1)*(13-1) = 10*12 = 120
Now, Let's choose an e value so that we can find d by finding its modulo inverse.
To choose 'e', we must keep in mind that 'e' should be relatively prime or be a co-prime to (p-1) And (q-1).
=> e and ϕ(pq) are relatively prime.
Let's find factors of ϕ(pq)=120:
=> Factors of 120= 1, 2, 3, 4, 5, 6, 8, 10, 12, 15, 20, 24, 30, 40, 60, 120.
For 'e' and 120 to be relatively prime, they should not have any other common factor other than '1'.
=> 'e' can be 7, 17, 23, 101, etc,...

I choose the following 3 values:

1. 'e' = 7:

   To calculate d, we know that:
   ed = 1 (mod ϕ(pq))
   => 7*d = 1 (mod 120) - (a)
   Let's find the inverse modulo of equation a by extended euclidean division algorithm:

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

=> 120 = 7(17) + 1
=> 7 = 1(7) + 0
Let's backpropagate:
1 = 120 - 7(17)
= 120 + 7(-17)
=> therefore the modulo inverse for 7 is -17.
Now, we know that (-17)7d is congruent to (-17)1 (mod 120)
=> d = -17 (mod 120)
=> d = 103
One pair of (e, d) = (7, 103)

2. 'e' = 17
To calculate d, we know that:
ed = 1 (mod ϕ(pq))
=> 17*d = 1 (mod 120) - (b)
Let's find the inverse modulo of equation b by extended euclidean division algorithm:
=> 120 = 17(7) + 1
=> 17 = 1(17) + 0
Let's backpropagate:
1 = 120 - 17(7)
= 120 + 17(-7)
=> therefore the modulo inverse of 17 is -7.
Now, we know that (-7)17d is congruent to (-7)1 (mod 120)
=> d = -7 (mod 120)
=> d = 113
Second pair of (e, d) = (17, 113)

3. 'e' = 101
To calculate d, we know that:
ed = 1 (mod ϕ(pq))
=> 101*d = 1 (mod 120) - (c)
Let's find the inverse modulo of equation a by extended euclidean division algorithm:
=> 120 = 101(1) + 19
=> 101 = 19(5) + 6
=> 19 = 6(3) + 1

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

=> 6 = 1(6) + 0
Let's backpropogate:
1 = 19 - 6(3)
= 19 + 6(-3)
= 19 + (101 - 19(5))(-3)
= 19 + (101 + 19(-5))(-3)
= 120 - 101(1) + (101 + 19(-5))(-3)
= 120 + 101(-1) + (101 + 19(-5))(-3)
= 120 + 101(-1) + 101(-3) + 19(-5)(-3)
= 120 + 101(-4) + 19(-5)(-3)

Question 4:

(a) Insert the key sequence {9, 16, 4, 5, 12, 27, 3, 10, 14, 1330} with hashing by chaining in a hash table with size 11. Please show the final table by using the hash function h(k) = 3k + 1 (mod 11).

Solution:

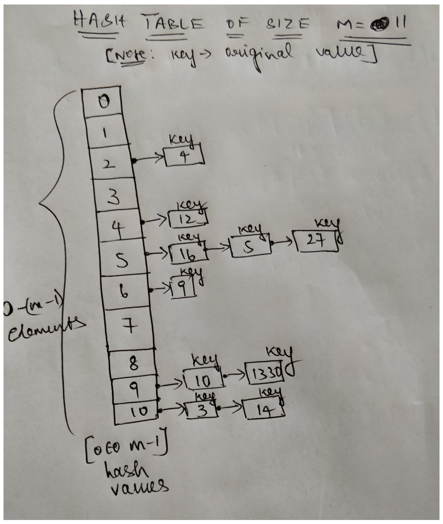
Given h(k) = 3k + 1 (mod 11), S = {9, 16, 4, 5, 12, 27, 3, 10, 14, 1330} and m = 11.

Let's now compute the hash value for each key from the given sequence for the given h(k).

1. K = 9:
   h(9) = 3*9 + 1 (mod 11) = 27 + 1 (mod 11) = 28 (mod 11) = **6**
2. K = 16
   h(16) = 3*16 + 1 (mod 11) = 48+ 1 (mod 11) = 49 (mod 11) = **5**
3. K = 4
   h(4) = 3*4 + 1 (mod 11) = 12 + 1 (mod 11) = 13 (mod 11) = **2**
4. K = 5
   h(5) = 3*5 + 1 (mod 11) = 15 + 1 (mod 11) = 16 (mod 11) = **5**
5. K = 12
   h(12) = 3*12 + 1 (mod 11) = 36 + 1 (mod 11) = 37 (mod 11) = **4**
6. K = 27
   h(27) = 3*27 + 1 (mod 11) = 81 + 1 (mod 11) = 82 (mod 11) = **5**
7. K = 3
   h(3) = 3*3 + 1 (mod 11) = 9 + 1 (mod 11) = 10 (mod 11) = **10**
8. K = 10
   h(10) = 3*10 + 1 (mod 11) = 30 + 1 (mod 11) = 31 (mod 11) = **9**

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

9. K = 14
   h(14) = 3*14 + 1 (mod 11) = 42 + 1 (mod 11) = 43 (mod 11) = **10**
10. K = 1330
    h(1330) = 3*1330 + 1 (mod 11) = 3990 + 1 (mod 11) = 3991 (mod 11) = **9**



(b) Consider a hash table that is an array of length 31 with chaining and the hash function h(n) = (√ n · log n) (mod 31) Is this a good hash function? Assume that the keys to be inserted are integers between 1 and 1000. Hint: You may want to write a short program in order to solve this exercise.

Solution:

Program:

1. Considering that I have taken *log to be base 10*:
   (though the textbook mentions that it's log base 2 generally- unless specified differently, I tried with base 10 as well)

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

```python
# Program to check the whether the given hash function is good or not
import math
import array as arr
import numpy as np

def hash_function(n, m):
    hash_value = int(((math.sqrt(n)) * (math.log(n, 10))) % m)
    #print(hash_value)
    return (hash_value, n)

def main(m, n_max):
    hash_table = np.zeros((1, 31))
    for i in range(n_max):
        key = i + 1
        hashed_value, n = hash_function(key, m)
        hash_table[0][hashed_value] += 1
        print("Integer value: ", n, "hashed to hash value: ", hashed_value)

    count = 0
    print(hash_table[0])

if __name__ == "__main__":
    # Given:
    m = 31
    n_max = 1000
    main(m, n_max)
```

Output:

```
Integer value:  1000 hashed to hash value:  1
[41. 40. 25. 25. 26. 28. 27. 28. 29. 28. 29. 29. 32. 30. 31. 32. 31. 33.
 33. 33. 34. 34. 33. 36. 34. 36. 35. 37. 37. 37. 37.]
```

Where the number at each index (each comma separated number)of the displayed array is the count of keys chained to that particular hash value. The indices are in order- 0 to m-1, which is 0 to 30.

2. Considering that I have taken *log to be base 2 (the general case)*:

```python
# Program to check the whether the given hash function is good or not
import math
import array as arr
import numpy as np

def hash_function(n, m):
    hash_value = int(((math.sqrt(n)) * (math.log(n, 2))) % m)
    #print(hash_value)
    return (hash_value, n)

def main(m, n_max):
    hash_table = np.zeros((1, 31))
    for i in range(n_max):
        key = i + 1
        hashed_value, n = hash_function(key, m)
        hash_table[0][hashed_value] += 1
        print("Integer value: ", n, "hashed to hash value: ", hashed_value)

    count = 0
    print(hash_table[0])

if __name__ == "__main__":
    # Given:
    m = 31
    n_max = 1000
    main(m, n_max)
```

Output:

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

```
integer value:  1000 hashed to hash value:  5
[36. 33. 37. 33. 35. 31. 30. 31. 30. 31. 31. 30. 32. 32. 32. 30. 32. 31.
 33. 30. 34. 32. 33. 32. 32. 32. 35. 31. 35. 30. 34.]
```

Where the number at each index (each comma separated number)of the displayed array is the count of keys chained to that particular hash value. The indices are in order- 0 to m-1, which is 0 to 30.

So, coming to the point *as to if 'h(n) = (√ n · log n) (mod 31)' is a good hash function*?
=> *with log taken as base 2, it's indeed a good (very) hash function*, as we can see that the *keys are getting equally distributed across the hash table, or in other words a key is equally likely to be hashed into any of the m spots in the hash table*. This is in all aspects better than the hash function with log base 10.
=> with log as base 10, I wouldn't call it a poor/bad hash function. I say so, as *there is more divergence of key values to hash values 0 and 1 of the hash table near the ends of the input array (that is near min and max of the array)- making it not so (exactly) uniformly distributed*. But, still the values seem to be almost (variance may be approx 10-15 elements) equally distributed.

(c) We consider universal hashing for the universe U = {0, . . . , 10} of size N = 11. For a hash table of size m = 4, we draw randomly the hash function:

ha,b(x) = ((ax + b) (mod N)) (mod m).

Find the "worst" hash function ha,b for S, meaning the values a and b, so that by hashing with ha,b at least 3 elements of the key sequence S = {1, 5, 8, 9} will be mapped to the same place in the hash table.

Solution:
Given S = {1, 5, 8, 9}, first let's find the factors of these numbers and the difference between each of them from one another.
=> That is-

Factors of 1: 1
Factors of 5: 1,5
Factors of 8: 1,2,4,8
Factors of 9: 1,3,9

=> Also, if we closely ***observe the difference between numbers 1, 5 and 9***, we can see that ***they all are 4 places apart*** from each other (in the sequence 1-5-9).
=> And, *m=4 which is same as the difference observed above between 1, 5 and 9*.
=> Ofcourse, we can use ***Brute Force approach*** as well- that would work anyday!

Savitaa Venkateswaran
MSc Computing Science Specializing in Big Data
SFU Student ID: 301376579

The above analysis proves to be very important in finding a and b such that the chosen hash function hashes at least 3 values from the set 'S' onto the same hash key. *It is important, as now we can multiply 1, 9 and 5 with a number from the universal hash value such that they have common factors -> Let's pick 10* as the multiple that we want to multiply each of three numbers with (a =10) and make b=0. *This approach works as we know that any multiple of the same number modulo a constant value is going to be the same.*

Which yields the has function: (taking a=10 and b=0, there are so many other pairs of a and b values for which this happens)

h10,0(x) = ((10x + 0) (mod 11)) (mod 4).

=>Key = 1

h10,0(1)= ((10*1 + 0) (mod 11)) (mod 4)
= (10 (mod 11))(mod 4)
= 10 (mod 4)
= **2**

=>Key = 5

h10,0(5)= ((10*5 + 0) (mod 11)) (mod 4)
= (50 (mod 11))(mod 4)
= 6 (mod 4)
= **2**

=>Key = 8

h10,0(8)= ((10*8 + 0) (mod 11)) (mod 4)
= (80 (mod 11))(mod 4)
= 3(mod 4)
= **3**

=>Key = 9

h10,0(9)= ((10*9 + 0) (mod 11)) (mod 4)
= (90 (mod 11))(mod 4)
= 2(mod 4)
= **2**

Therefore, it can be clearly seen that keys 1, 5 and 9 hash to the same value 2 for the pair (a,b) = (10, 0)