**CS566 Homework-1**

**Due – Jan 30, 6:00 PM, at the class beginning**

Student’s Name: Kevin Lopez Sepulveda\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Upload copy of your homework and any supporting materials (source code and test cases) online to Assignment1 in the CS566\_A3 course site on the blackboard.

**No extensions or late submissions for anything** other than major emergency.

(1)   [15 pts.]

Suppose Algorithm-1 does f(n) = *n*\*\*2 + 3*n* steps in the worst case, and Algorithm-2 does *g*(*n*) = 14n\*\*1.3 + 33 steps in the worst case, for inputs of size *n*. For what input sizes is Algorithm-1 faster than Algorithm-2 (in the worst case)?

First we want to create the inequality f(n) < g(n). After simplifying by getting both terms on one side and having it equal to zero.. After finding the zeros we plugged in on each side where the function was positive or negative. After testing, we determine that Algorithm-1 is faster for values of n approximately between n=43 and n=44. Beyond this, Algorithm-2 becomes more efficient in the worst case.

(2)  [15 pts.]

Prove or disprove: T(n) = SUM{*i*\*\*3} = theta(*n*\*\*3),

where *i* changes from *i*=1 to *i*=*n*.

For T(n) = SUM{i\*\*3) we know that the formula is equal to ((n/2)(n+1)\*\*2) which simplified becomes (n\*\*2/4)((n+1)\*\*2). After expanding we learn that the dominant term is n\*\*4. This translates to T(n)=Sum{i\*\*3} = theta(n\*\*4) which does not equal theta(n\*\*3). The statement is false.

(3)  [25 pts.]

Write out the algorithm (pseudocode) to find *K* in the **ordered** array by the method that compares *K* to every m-th entry (1< m is less than array size n) until *K* itself or an entry larger than *K* is found, and then, in the latter case, searches for *K* among the preceding m. How many comparisons does your algorithm do in the worst case?

Test your algorithm for m=3,4,5 … and come up with the general formula.

I saw that this is similar to jump search. Below is the pseudo code:

FindK(arr, n, K, m):

i = 0

// Step 1: Jump through the array in steps of m

while i < n and arr[i] < K:

i = i + m

// Step 2: Perform linear search in the previous m elements

start = max(0, i - m) // Ensure index is within bounds

end = min(i, n) // Avoid going out of bounds

for j from start to end:

if arr[j] == K: return j // Found K at index j

return -1 // K not found

This algorithm has both worst case scenarios n/m(Jump Phase) and m(Linear) where n=sqrt(m). if we sum these up then we get O(sqrt(n)) as the common worst case scenario.

Example 1: m=3

1. Jump Phase: Check elements at indices 3, 6, 9, ..., 99.
   * This requires 100/3≈34 comparisons.
2. Linear Search Phase: If K is within the last block (e.g., between 97 and 100), we scan up to 2 more elements.
   * Total comparisons: 34 + 2 = 36.

Example 2: m=4

1. Jump Phase: Check elements at indices 4, 8, 12, ..., 100.
   * This requires 100/4=25 comparisons.
2. Linear Search Phase: If K is within the last block (e.g., between 97 and 100), we scan up to 3 more elements.
   * Total comparisons: 25 + 3 = 28.

Example 3: m=5

1. Jump Phase: Check elements at indices 5, 10, 15, ..., 100.
   * This requires 100/5=20 comparisons.
2. Linear Search Phase: If K is within the last block (e.g., between 96 and 100), we scan up to 4 more elements.
   * Total comparisons: 20 + 4 = 24.

The general formula based on the above 3 cases are C(m) =(n/m) + (m-1). It follows the logic of the above where there is a base number of comparisons based on total elements(n) divided by partition(m) plus the element required being in the last element in that partition(m-1).

(4)  [25 pts.] Compare two algorithms to raise an integer to an integer power

assume in both cases that n, the exponent, is a power of 2:

Algorithm 1

***X\*\*n = X\* X\*\*(n-1)***

***X\*\*0 = 1***

Algorithm 2

***n* = 2\*\**m***

**X\*\**n* = ((X\*\*2)\*\*2)\*\*2…, etc. [NOTE: the symbol of power (\*\*) is used *m* times here, i.e., X\*\*8 = ((X\*\*2)\*\*2)\*\*2, because 8 = 2\*\*3].**

Which algorithm is more efficient with respect to the number of multiplications?

**Number of Multiplications:**

**For Algorithm 1:**  
Each step reduces the exponent by **1**, requiring n multiplications in the worst case. Since n=2\*\*m we learn that algorithm 1 is O(n).

**For Algorithm 2:**

Each step squares the result, reducing the exponent size exponentially.  
Since n=2\*\*m, we perform m squaring operations, where m=log2​(n). This results in a logarithmic complexity O(log(n))

Algorithm 2 (Exponentiation by Squaring) is exponentially faster than Algorithm 1 and should always be preferred for exponentiation when n is a power of 2. It is log2(n) better for n=2^m.

(5) [20 pts]

You are given 12 bags of gold coins. One bag contains all false coins that weigh several grams less. What is the minimum number of weighting required to identify false coins bag? Does the answer depend on the balance type: digital or not?

**Method 1: Using a Digital Scale (Known Weight)**If we know the weight of a genuine gold coin, we can determine the bag with the false coins in just one weighing. We take 1 coin from Bag 1, 2 coins from Bag 2, 3 coins from Bag 3, and so on, until we take 12 coins from Bag 12. The expected weight of all these coins, assuming they are all genuine, would be (1+2+3+…+12)⋅w=78w, where w is the weight of a genuine coin. If the bag containing false coins is Bag k (where the coins weigh w−d), the actual weight will be 78w−d. By weighing the total and calculating the difference from the expected weight, we can directly determine the bag number k with false coins.

**Method 2: Using a Balance Scale (Unknown Weight)**If the weight of a genuine gold coin is unknown, we can still find the bag containing false coins using a balance scale with a systematic approach in a maximum of three weighing. First, we split the 12 bags into three groups of 4 bags each: Group A (Bags 1, 2, 3, 4), Group B (Bags 5, 6, 7, 8), and Group C (Bags 9, 10, 11, 12). We weigh Group A against Group B. If they balance, the false coins are in Group C; if one side is lighter, the false coins are in the lighter group. Next, we take the suspected group of 4 bags and weigh 2 of them against 2 bags from a group we know is genuine. The outcomes will help us narrow down to 2 suspected bags. Finally, we weigh one of these suspected bags against a known real bag to identify the bag containing the false coins.

Conclusion:  
In summary, if we know the weight of a genuine gold coin, we can identify the false bag in one weighing using a digital scale. However, if we do not know the weight of the coins, we can still find the false bag using a balance scale in three weighing by employing a systematic divide-and-conquer strategy.