Dara Lim

CS 3130

Homework #1

A.

1. (a) Find the gcd(213486, 5423) by applying Euclid’s algorithm.

gcd(213486, 5423)

gcd(5423, 1989)

gcd(1989, 1445)

gcd(1445, 544)

gcd(544, 357)

gcd(357, 187)

gcd(187, 170)

gcd(170, 17)

gcd(17, 0) = 17

Therefore, 17 is the greatest common divisor of 213486 and 5423.

There are 9 iterations or divisions. Mod (%) is just an integer division where the return is

the remainder.

(b). Estimate approximately how many times faster it will be.

Consecutive integer checking will take 5423 iterations (actually it takes 10846 divisions).

Euclid’s algorithm is 5423 / 9 = 602 times faster in terms of iterations or Euclid’s algorithm

is 10846 / 9 = 1205 times faster in terms of divisions.

2. Prove the formula, on which the Euclid’s algorithm is based:

gcd(m, n)=gcd(n, m mod n) for every pair of positive integers m and n.

- Assume d is any integer. If two integers x and y are divisible by d, then x + y and x - y are

also divisible by d. If an integer u is divisible by d, then the integer multiple of u is also divisible by d.

- Using the Euclid’s algorithm for gcd(a, b), if a >= b > 0 then a can be written as

a = qb + remainder. For the gcd(m, n), m = qn + m mod n. The remainder can be written as

(m mod n) = m - qn.

- Assume an integer d that divides both m and n. Now, apply the Euclid’s algorithm for GCD

of m, n.

m = (q\*n)+(m mod n), where m is the dividend, n is the divisor, q is the quotient and m mod

n is the remainder.

- The GCD of m, n is computed by replacing the pair (m, n) by (n, m mod n). The process

continues until the remainder becomes 0. For any two positive integers m and n, if d divides both m, n then n and m mod n are also divisible by d. There are finite set of common divisor of m and n that are same as the set of common divisors of n and m mod n.

- Consider one of the m, n is greater than 0. The gcd(m, n) divides m, n and m - qn (remainder). Thus, gcd(m, n) <= gcd(n, m mod n). The gcd(n, m mod n) divides n, m mod n and m. Therefore, gcd(n, m mod n) <= gcd(m, n).

- Thus, gcd(m, n) = gcd(n, m mod n).

B.

1. Suppose arrays A[1....n] and B[1.....n] are two integer arrays to store in two n element number. Each of these arrays can store n bits.

- Suppose array C[1....n+1] to store the binary addition of A and B array. The length of this array is 1 more than A and B array because it may possible the addition can give one more value in the array.

**Logic**:

- Start fetching the elements of both A and B array from last index to 0th index.

- Add ith bits (that is ith element) of both array A and B. When ith bit of both array are 1 then place 0 at the i+1th position in C array and put 1 in the array forward.

- When ith bits of both A and B array are added then it is also necessary to check whether there is any carry to ith bit from i+1 bit.

**Pseudo code:**

Binary-Addition (A[1....n], B[1.....n])

- Declare an array C[1.....n+1] to store binary addition

- carry = 0

- for i = n-1 to 0

- sum = A[i]+B[i]+carry

- C[i+1]=sum%2

- carry=sum/2

- C[0]=carry

-return C

**Analyzing the running-time:**

The for loop in above Binary-Addition() function iterate n time to fetch n bit of A and B array. Therefore, the worst case running time of this algorithm is O(n).

2. **PsuedoCode**:

for i=1 to n

minIndex=i

for j = i+1 to n

if A[j] < A[minIndex]

minIndex = j

if minIndex != i

temp = A[i]

A[i] = A[minIndex]

A[minIndex] = temp

**Loop invariant:**

After each iteration, the element at the position index is properly placed. The array is divided into two subarrays.

The first subarray A[1....index] will contain the index smallest elements arranged in increasing order.

The second subarray A[index+1.....n] will contain the element to be sorted.

**Reason to run only for first n-1 elements:**

In the first iteration, the smallest element in the array A[1....n] is found and is exchanged with the element at first position.

In the second iteration, the smallest element in the array A[2....n] is found and exchanged with the element at second position.

In the third iteration, the smallest element in the array A[3....n] is found and exchanged with the element at the third position.

The process continues for all the elements until there are only 2 elements left to be sorted.

In the n-1 iteration, there will be only two elements in the array. The smallest element in the array A[n-1, n] is found and is exchanged with the element at n-1 position.

The last element A[n] is already placed in its proper position. Therefore, there is no necessity for executing the loop.

Thus, the loop runs for first n-1 elements rather than for all n elements.

**Best-case and worst-case running times:**

In selection sort algorithm, the entire unsorted array must be searched to find the minimum element.

In the ith iteration, the inner loop executes exactly (n-i) times, and on each such execution exactly one array comparison is performed.

Thus, in all cases

3. Modify the fragment of pseudocode on page 40 named BUBBLESORT(A) so that it will i mplement bubble sorting algorithm with counting swaps.

initialize numSwap to 0

for i = 1 to A.length - 1

for j = A.length down to i+1

if A[j]<A[j-1]

exchange A[j] with A[j-1]

numSwap = numSwap + 1

6. Problem 2.1 -3, page 22

**Input**: A sequence of n numbers A= <a1, a2,.....an > and a value v.

**Output**: An index i such that v=A[i] or the special value NIL if v does not appear in A.

**Linear Search Pseudocode:**

- for i=1 to A.length , where A.length = n

- { if A[i]=v

return i

}

- if i > A.length

return NIL

**Loop invariant and the correctness of linear search:**

**.** Initialization: It will start by initializing the from 1 to length of the array A.

. Maintenance: Each iteration maintains the loop invariant. Informally, the body of the loop works by searching each element whether the searched element v present in the loop or not. Each time loop checks for the next element. Thus, the second property satisfies.

. Termination: Finally, for linear search, the for loop ends, when the searched element v finds at the i index and returns i or i exceed A.length. If i exceed A.length then it returns Nil. Hence, either the searched element found or the algorithm returns NIL. After for loop completion either the loop index i gets returned if the element v present in the array A or it returns NIL. Thus, it shows the correctness of the algorithm.

7. For the algorithm from the problem 6 (Linear Search), describe the average case performance by counting the average number of comparisons used. Assume that the integer v is in the list and it is equally likely that v is in any position.

**Average case:** occurs when the element to be searched is found in the array. In such a case, the linear search will search through half of the array. Therefore, the number of elements that are searched on an average is n/2.