**Assignment 1**

Advanced Algorithms 1 (7081)

**Group 13**

Jianfeng Zeng (zengjg)

Xin Li (li4x7)

Alex Stewart (stewaaw)

Nathan Daughety (daughenl)

Written Part [50pts]

1. *[10pts] Page 22. Exercise 1.1 (a), (d).*

Compute binary representation of 123 obtaining 1111011.

Compute binary representation of 711 obtaining 1011000111.

1. *[10pts] Page 22, Exercise 1.8.*
2. (24, 108)
3. (23, 108)
4. (89, 144)
5. (1953, 1937)
6. *[10pts] Page 22, Exercise 1.9. Prove that the formula you have given holds.*

**Proof:** let ,

for some x and for some y.

is divisible by a and b by the definition of lcm.

Since x and y are coprime by the definition of gcd;

therefore, we have .

1. *[10pts] Page 24, Exercise 1.19.*

Instead of evaluating a polynomial, say using Horner’s method directly, we can divide into one sub polynomial that consists of indeterminates with only even degree, and one sub polynomial that consists of indeterminates with only odd degree, . Then we have,

We can then evaluate even polynomials and odd polynomials separately using Horner’s method with multiplications and additions. We need to do multiplications because we multiply the coefficients with instead of . To evaluate , we simply add the results from and . On the other hand, to evaluate , we simply subtract from , since -v can only affect the indeterminates with odd degree. Thus, we simply need to do one more addition to evaluate .

1. *[10pts] Design an efficient algorithm for computing the majority element in a list L[0:n – 1] if it exists or determining that it doesn’t exist. An element is a majority element if it is repeated more than n/2 times. Don’t settle for straightforward algorithm. Try to design a linear-time algorithm. Analyze the algorithm you have designed in a), i.e., determine B(n) and W(n).*

**function** findMajorityElement(list[0:n])

**input:** list[0:n] (an array of elements)

**output:** -1 (not found) or i (element index)

HashTable {}

Max 0 (used to track the current max value in the table)

**for** i 0 **to** n - 1 **do**

**if** list[i] **not** in table **then**

HashTable [list[i]] 1

**else**

HashTable [list[i]] +1

**endif**

**if** HashTable [list[i]] > Max **then**

Max HashTable [list[i]]

**if** Max > (n - 1) / 2 **then**

**return** i

**endif**

**if** ((n - 1) / 2) > ((n – 1) - i + Max) **then**

**return** -1

**endif**

**endif**

**endfor**

**return** -1

**end** findMajorityElement

**Analysis:** The algorithm iterates through the input list and uses a look up table to keep tracking the number of times an element has encountered. The table stores an element into the table initialized beforehand with value set for 1 if the element is not found in the table or increments the value by 1 otherwise. A variable Max is also initialized for 0 at the start of the algorithm to store the current max value in the table. The iteration stops and returns the element index when the max value is larger than one half of the list length. This max value is used to stop the iteration when one half of the list length is larger than the list length minus the current index plus the max value. In another words, the iteration stops and returns -1 when there is no way the current max value will exceed one half of the list length after iterating through the rest of the list. The algorithm at the end returns -1 if no return statement is hit inside the list iteration. The time complexity for this algorithm is **O(n)** as it iterates through the entire list. The space complexity for this algorithm is also **O(n)** as it creates a key-value relation in the look up table for each element encountered in the input list. The best case, **B(n)** for this algorithm is because it stops whenever the current max value is larger than one half of the list. On the other hand, the worst case, **W(n)** for this algorithm is because the majority element could be at the very end of the list.

Programming Project [50pts]