ECE 198 Fall 2019 Jin Yucheng (yucheng9)

Homework 3 Problem 1-6

Solution:

(1) Show that if m is an integer greater than 1 and $ac \equiv bc (\textbf{mod } m)$, then $a \equiv b (\textbf{mod } m/gcd(c,m))$.

Since $ac \equiv bc (\textbf{mod } m)$, then m divides $(a-b)\cdot c$, which means there is an integer k such that,

$ac - bc = m \cdot k$

$(a-b)\cdot c = m \cdot k$

$(a-b)\cdot \frac{c}{gcd(c,m)} = \frac{m}{gcd(c,m)} \cdot k$

Suppose $gcd(c,m)\cdot s = c$, $gcd(c,m)\cdot f = m$, for some integers s, f such that $gcd(s,f) = 1$.

$(a-b)\cdot s = f \cdot k$

$(a-b) = f \cdot \frac{k}{s}$

$\frac{k}{s}$ must be an integer, since $(a-b)\cdot s = f \cdot k$ and $gcd(s,f) = 1$, k must be a multiple of s. Therefore, if m is an integer greater than 1 and $ac \equiv bc (\textbf{mod } m)$, then $a \equiv b (\textbf{mod } m/gcd(c,m))$.

(2) Show that $F_{n+1}F_{n-1} - F_n^2 = (-1)^n$ when n is a positive integer.

- Basis Step: Since $F_0 = 0, F_1 = 1, F_2 = 0+1 = 1$. For $n = 1$, $F_2F_0 - F_1^2 = -1 = (-1)^1$, which satisfies $F_{n+1}F_{n-1} - F_n^2 = (-1)^n$.
- Inductive Step: Suppose for every positive integer k such that $k \leq n$, $F_{k+1}F_{k-1} - F_k^2 = (-1)^k$. For $n+1$,

  $F_{n+2}F_n - F_{n+1}^2$

  $= (F_{n+1} + F_n)F_n - F_{n+1}^2$ (because $F_n = F_{n+1} + F_{n-1}$ for every positive integer n)

  $= F_nF_{n+1} + F_n^2 - F_{n+1}^2$

  $= F_nF_{n+1} + F_{n+1}F_{n-1} - (-1)^n - F_{n+1}^2$ (by the inductive hypothesis)

  $= (F_n + F_{n-1})F_{n+1} - (-1)^n - F_{n+1}^2$

  $= F_{n+1}^2 - (-1)^n - F_{n+1}^2$ (because $F_n = F_{n+1} + F_{n-1}$ for every positive integer n)

$= -(-1)^n$

$= (-1)^{n+1}$

Therefore, for every positive integer n, $F_{n+1}F_{n-1} - F_n^2 = (-1)^n$.

(3) Give a recursive definition of the function max so that $\max(a_1, a_2, ..., a_n)$ is the maximum of the n numbers $a_1, a_2, ..., a_n$.

max is meaningful when n is a positive integer.

- If n = 1, $\max(a_1) = a_1$.
- If n = 2, if $a_1 > a_2$, $\max(a_1, a_2) = a_1$; else, $\max(a_1, a_2) = a_2$.
- If n > 2, $\max(a_1, a_2, ..., a_n) = \max(\max(a_1, a_2, ..., a_{n-1}), a_n)$.

(4) Give a recursive definition of the set of bit strings that are palindromes.

- The empty string, the string 1 and 0 are palindromes.
  Denote P as the set of palindromes.

$$\epsilon \in P$$
$$1 \in P$$
$$0 \in P$$

- If the string has more than 1 bit, then it is a palindrome if and only if its first bit and last bit are both 1 or both 0, and the rest of the string is a palindrome.
  Denote S as the string. $S \in P$ if and only if,

$$S = 0s0, s \in P$$
or
$$S = 1s1, s \in P$$

(5) Prove that the merge sort algorithm is correct.

- Basis Step: If the input array has 0 element or 1 element, then merge sort is correct since 0 element or 1 element are already sorted.
- Inductive Step: If the input array has more than 1 element, suppose it has n elements, and assume for any array with $k (k < n)$ elements, merge sort is correct.
  Because merge sort first divides the input array into 2 subarrays of roughly equal size, for these two subarrays, they both have elements less than n (at most $\lfloor n/2 \rfloor + 1$ if n is odd, $\lfloor n/2 \rfloor$ if n is even). Because of the inductive hypothesis, for any array with less than n elements, merge sort is correct, so it correctly sort these two subarrays. Then merge sort merges these two subarrays and correctly sorts the input array with n elements.

Therefore, merge sort is correct (actually I assume merge sort can merge two sorted arrays correctly, since it is tedious to prove the correctness of "merge". The proof of the correctness of "merge" can be seen in Jeff Erickson's *Algorithms*, page 28).

(6) Determine the worst-case complexity of the quick sort algorithm in terms of the number of comparisons used.

Quick sort is (from Jeff Erickson's *Algorithms*, page 29),

```
QuickSort(A[1..n]):
  if (n > 1)
      Choose a pivot element A[p]
      r ← Partition(A, p)
      QuickSort(A[1..r−1])     ⟨⟨Recurse!⟩⟩
      QuickSort(A[r+1..n])     ⟨⟨Recurse!⟩⟩
```

```
Partition(A[1..n], p):
  swap A[p] ↔ A[n]
  ℓ ← 0                        ⟨⟨#items < pivot⟩⟩
  for i ← 1 to n−1
      if A[i] < A[n]
          ℓ ← ℓ + 1
          swap A[ℓ] ↔ A[i]
  swap A[n] ↔ A[ℓ + 1]
  return ℓ + 1
```

In QuickSort subroutine, there is one comparison of the array's size, in Partition subroutine, there is one comparison of deciding whether to swap. The worst-case for quick sort is every time the pivot is either the first element or the last element, so it will choose a pivot for $O(n)$ times in the worst-case, and Partition clearly runs in $O(n)$ time, so the quick sort's worst-case time complexity is $O(n^2)$.

■