CSCI 5454: Algorithms: Homework 1

Ashutosh Gandhi

February 2, 2023

Problem 1

1.1 Part A

To prove the loop invariant $a = sa_0 + tb_0$ and $b = \hat{s}a_0 + \hat{t}b_0$

Base Case At the first encounter of the loop head we have $s = 1$, $\hat{s} = 0$, $t = 0$, $\hat{t} = 1$ and we also have $a = a_0$ and $b = b_0$

substituting these values into our loop invariant equations we get
$a = 1 * a_0 + 0 * b_0 \qquad b = 0 * a_0 + 1 * b_0$
$a = a_0 \qquad b = b_0$
Thus the invariant is trivially true

Induction step Let the relation $a = sa_0 + tb_0$ and $b = \hat{s}a_0 + \hat{t}b_0$ hold true for the i^{th} iteration. Then for the $i + 1^{th}$ iteration we have the variables $a', b', s', t', \hat{s}', \hat{t}'$

The relation here is
$a' = b, b' = a\%b, s' = \hat{s}, t' = \hat{t}, \hat{s}' = s - (\lfloor a/b \rfloor * \hat{s}), \hat{t}' = t - (\lfloor a/b \rfloor * \hat{t}) \; -- > ①$
The goal is to establish $a' = s'a_0 + t'b_0 \quad and \quad b' = \hat{s}'a_0 + \hat{t}'b_0$

$$\begin{aligned} s'a_0 + t'b_0 &= \hat{s}a_0 + \hat{t}b_0 \quad [from ① \quad s' = \hat{s}, t' = \hat{t}] \\ &= b \quad [from \quad induction \quad step] \\ &= a' \quad [from ① \quad a' = b] \end{aligned}$$

$$\begin{aligned} \hat{s}'a_0 + \hat{t}'b_0 &= (s - \lfloor a/b \rfloor * \hat{s})a_0 + (t - (\lfloor a/b \rfloor * \hat{t})b_0 \quad [from ①] \\ &= sa_0 + tb_0 - \lfloor a/b \rfloor(\hat{s}a_0 + \hat{t}b_0) \quad simplifying \\ &= a - \lfloor a/b \rfloor * b \quad [from \quad induction \quad step] \\ &= a\%b \quad [because \quad a = quo * b + rem; \quad rem = a - quo * b] \\ &= b' \quad [from ① \quad b' = a\%b] \end{aligned}$$

Thus, we have proven the desired loop invariant by establishing our goal $a' = s'a_0 + t'b_0 \quad and \quad b' = \hat{s}'a_0 + \hat{t}'b_0$

1.2 Part B

To prove gcd(a,b) is the smallest positive number that can be written in the form $sa + tb$ i.e. $gcd(a, b) = sa + tb$

Consider the loop invariant from the Part-A, we proved $a = sa_0 + tb_0$
after the last iteration (when $b = 0$), a would be equal to $gcd(a, b)$
hence we get $gcd(a, b) = sa_0 + tb_0$ where $a_0 \& b_0$ are the starting values a,b

now to prove that gcd(a,b) is the smallest positive number in $sa+tb$ let's consider a positive number of the form $la + ub$
let's call $gcd(a, b) = g$
Since g divides both a and b so it would also divide $la + ub$ because la/g and ub/g would be positive numbers. so g must be less than or equal to $la + ub$. In other words, g is the smallest positive number in the form $la + ub$. Hence Proved.

1.3 Part C

Since n and p are relatively prime the $gcd(n, p) = 1$
From Bezout's Lemma we have $gcd(n, p) = xn + yp = 1$ where x and y are integers

$$xn + yp = 1$$
$$xn = 1 - yp \quad \text{[taking (mod p) on both sides]}$$
$$xn(\text{mod p}) = 1 - yp(\text{mod p})$$
$$xn(\text{mod p}) = 1 \text{ mod p} \quad [as \text{ yp (mod p)=0}]$$
$$\text{Let x (mod p) be a natural number m, such that } 0 <= m < p$$
$$n * m = 1 \text{ mod p} \quad \text{[where m is a unique number x (mod p)]}$$

1.4 Part D

In Part C we proved if n and p are relatively prime numbers then $n * m = 1 \quad mod \quad p$ where m is a unique number $x(mod \quad p)$ that is the modulo inverse, x being the value returned from extended Euclid's algorithm.

$(g, x, y) = Extended\_gcd(n, p)$ substituting $n = 13113 \quad and \quad p = 2133555512$ we get
$g = 1 \quad \& \quad x = 572234785 \quad \& \quad y = -3517$
$m = 572234785(mod \quad 2133555512)$
$m = 572234785$

Problem 2

2.1 Part A

procedure TRADESUM(Arr, i, j)
#precond : $0 <= i, j <= n$ and $i <= j$
  $n \leftarrow len(Arr)$ # initialize n to length
  $sumArr \leftarrow [0] * n$
  $sumArr[0] \leftarrow Arr[0]$
  for $k : 1 \rightarrow n - 1$ **do** #calculate cumulative sum
    $sumArr[k] \leftarrow sumArr[k-1] + Arr[k]$
  if i is 0 **then return** $sumArr[j]$
  return $sumArr[j] - sumArr[i-1]$ #return the range sum

In the pre-processing step of the above algorithm, we store the cumulative sum at each trade. This is done so that the queries can be done in O(1) time as it would just be a lookup from the array. As the j^{th} index will have a sum of the first j trades and $i - 1^{th}$ index would have the sum of the first i-1 trades, subtracting the two would be the range sum. In the pre-processing step, 2 array lookups are done at each iteration and then added, this is done for all the n trades in the Array.

2.2 Part B

procedure TRADESUM(Arr, i, j)
#precond : $0 <= i, j <= n$ and $i <= j$
  $n \leftarrow len(Arr)$ # initialize n to length
  $k \leftarrow \lceil \sqrt{n} \rceil$ # ceiling of sqrt(n)
  $sumArr \leftarrow [0] * k$
  for $u : 0 \rightarrow n$ **do** #calculate cumulative sum
    $ind \leftarrow \lfloor u/k \rfloor$ # find the slice to store the sum to
    $sumArr[ind] \leftarrow sumArr[ind] + Arr[u]$
  $sum \leftarrow 0$
  while $(i+1)\%k! = 0\&\&i <= j$ **do**
    $sum \leftarrow sum + Arr[i]$
    $i \leftarrow i + 1$
  while $i + k <= j$ **do**
    $sum \leftarrow sum + sumArr\lfloor(i/k)\rfloor$
    $i \leftarrow i + k$
  while $i <= j$ **do**
    $sum \leftarrow sum + Arr[i]$
    $i \leftarrow i + 1$
  return sum

To only use space of $O(\sqrt{n})$ we create $\sqrt{n}$ number of slices of the array sized n and store the sum of each of these slices. This way space used is at max $\sqrt{n} + 1 \quad or \quad O(\sqrt{n})$ and the pre-processing time is $O(n)$ as to calculate each slice sum all elements would have to be scanned at least once. Next, for the query, we would need a $O(\sqrt{n})$ time to return the sum. the first scan is from I to the end of its slice, then each slice of $i+k$ till j is directly added, and then the remaining j elements in the last slice. In the worst case, we would have to iterate over all the elements in our summArr and hence a time complexity of $O(\sqrt{n})$.

2.3 Part C

```
procedure findLargestIndex(A, x)
assert ( A[0] <= x )
    n ← len(A)
    if x >= A[n − 1] then return n − 1
    l ← 0
    u ← n − 1
#Loop Invariant : 0 <= l < u < n   and   A[l] <= x < A[u]
    while l < u − 1 do
        mid ← ⌊(a + b)/2⌋
        if A[mid] = x then return mid
        else if A[mid] < x then
            l ← mid
        else
            u ← mid
    return l
```

We start with the lower pointer at 0 and the upper pointer at 1 and find the mid, if the mid is lesser than our search we update low to mid else high is updated to mid, this way we are eliminating half our search space at each iteration. To satisfy the loop invariant we are not updating to low and upper to $mid-1$ and $mid+1$ as doing so would cause the $x < A[u]$ to fail at the end of the last iteration of the loop. Rather the condition in the while loop is $l < u-1$ so that it does terminate and no element is missed to be checked.

2.4 Part D

```
procedure negativeTrade(Arr, i, j)
#precond : 0 <= i, j <= n and i <= j)
    n ← len(Arr)    # initialize n to length
    negInd ← [−1] ∗ n
    if Arr[n − 1] < 0 then
        negInd[n − 1] ← n − 1
    for k : n − 2 → 0 do    # build the pre-processing array
        if Arr[k] < 0 then
            negInd[k] ← k    # store the index of negative element
        else
            negInd[k] ← negInd[k + 1]    # else store same as right neighbour
```

if $negInd[i] \leq j$ **then return** $negInd[i]$
return "No" # no negative trades

In the pre-processing step, the algorithm iterates from right to left and stores the index, if a negative number is found else, it keeps the same value as its right neighbor. This way to check if a negative number is present for a given range all we check is that the value in the negInd array at the i^{th} index is less than or equal to the value at j^{th} index, then we know for sure that there is at least one negative number in that range.