CS170–Fall 2015 — Homework 3 Solutions

Nguyet Minh Duong, SID 24444722

Collaborators: Henry Kwan & James Carlson

1. Two Sorted List

1. Main Idea
   We look at the value at $\lfloor \frac{k}{2} \rfloor$ in both lists given. Remove all elements up to to the left of the position $\lfloor \frac{k}{2} \rfloor$ and the element at position $\lfloor \frac{k}{2} \rfloor$ in the list that has the smaller value at position $\lfloor \frac{k}{2} \rfloor$. Repeat the algorithm with a new k such that k = k - number of values removed from the list, until $k = 1$. At this point, we return the smaller value in the first position of both lists.

2. Pseudocode

```
// if any values is null when trying to look at it
// automatically return infinity
Procedure GetKSmallest(array A[..], array B[..], smallest value k)
  if k = 1:
    return the smaller value between A[1] and B[1]
  else:
    // assume floor is a function
    // that gives the rounds the value down and takes O(1) time
    // if either A[..] or B[..] does not have element at index floor(k/2)
    // return negative infinity as a result
    value_of_a := A[ floor(k/2) ]
    value_of_b := B[ floor(k/2) ]
    if value_of_a >= value_of_b:
      remove all values in B[..] up to index floor(k/2), inclusive
      c := the number of elements that were removed from B[..]
      return getKSmallest(A, B, K-c)
    else:
      remove all values in A[..] up to index floor(k/2), inclusive
      c := the number of elements that were removed from A[..]
      return getKSmallest(A, B, K-c)
```

3. Proof of Correctness
   We are going to begin to prove the base case will return the correct values, and then argue that correct values are called if the previous recursive calls return the correct value.

Given $k = 1$ (the 1st smallest element), the smallest element has to either be the first element in the first list (call this l_1) or the second list (call this l_2) because both lists are in ascending order, meaning their smallest element will be in the first index. We can just return whichever first element is smaller – the one in l_1 or the one in l_2.

Now consider the two arrays, if we go to index $\lfloor \frac{k}{2} \rfloor$ for both l_1 and l_2, we compare the two element there. Whichever element is smaller is the element we want to discard because there's no way our k-th smallest element will be in that position since it is only half and all values before that element should be obmitted because it cannot be the k-th smallest element due to the fact that it is in ascending order. This means the element at index $\lfloor \frac{k}{2} \rfloor$ must be either bigger or equal to all the previous elements and it is OK to obmit and remove it from the list that contains the smaller element out of the two that was previous compared.

We can now repeat the process on the revised list and the list that was not changed with a new k value, where $k = k - \lfloor \frac{k}{2} \rfloor$. Eventually we should reach $k = 1$, where we would reach our base case and we can decide on our smallest value.

The reason this works because you will cut off the smaller values that leads up to the k-th smallest value, and that's why we reduce the k because we have taken the smaller values out of consideration.

4. Running Time Algorithm
   This algorithm takes $\Theta(log(k))$ because we are consistently dividing the list into $\frac{k}{2}$. $\Theta(logk) = O(logm + logn)$ because $k < m + n$. In this case, $logk = log(n + m) = log(n) + log(m)$ hence our run time can be $O(logm + logn)$.

2. Hidden Surface Removal

1. Main Idea
   We are going to implement this similar to merge sort such that we are going to split the list of lines in half until it goes to the base case of 1. Under a new procedure it will merges elements together, andl compare to see which line is "visible" by checking the variables $a\&b$ where $l_i = a_ix + b_i$. What we want to do is check for areas of overlap, and record all the lines such that we can have its starting point and ending point such that no other lines conflict with between the starting and ending point, and that the starting point is always less than the ending point.

2. Pseudocode

```
// neg-inf := negative infinity
// inf := positive infinity
// assume the lines they give us contains variables a & b such that line = ax + b
Procedure VisibleLines(list l):
  if length of l <= 1:
    // returns a list with the line, its valid lowest and highest x-value
    return list(l, neg-inf, inf)
  right := empty list
  left := empty list
  middle :=  (length of l) / 2
  for each x in l before middle:
    put x in list left
  for each x in l after or equal to middle:
    put x in list right
  left = VisibleLines(left)
  right = VisibleLines(right)

  return Compare(left, right)

Procedure Compare(list L, list R):
  // takes in list L and R such that it its elements inside the lists are:
  // ((line1, x1-initial, x1-final), (line2, x2-intial, x2-final) ...)
  // such that x-initial and x-final represents areas that they are visible
  result := empty list
  while L.isNotEmpty and R.isNotEmpty:
    L_e := first  element of list L
    R_e := first element of list R
    L_xi := getX-Initial(L_e)
    L_xf := getX-Final(L_e)
    R_xi := getX-Initial(R_e)
    R_xf := getX-Final(R_e)
    L_aVar := getVarA(L_e)
    L_bVar := getVarB(L_e)
    R_aVar := getVarA(R_e)
    R_bVar := getVarB(R_e)
```

```
if L_xi >= R_xf or L_xf <= R_xi:
  if L_aVar > R_aVar:
    result = result append(L_e)
    remove L_e from L
  else if L_aVar < R_aVar:
    result = result append(R_e)
    remove R_e from R
  else if L_bVar > R_bVar:
    result = result append(L_e)
    remove L_e from L
  else if L_bVar < R_bVar:
    result = result append(R_e)
    remove R_e from R
cross := the value x where line(L) and line(R) intercepts
if L_aVar > R_aVar:
  L_xi := cross // assume these changes its direct value in the L_e
  R_xf := cross // assume these changes its direct value in the R_e
  if L_xi > L_xf:
    remove L_e from L
  else if R_xi > R_xf:
    remove R_e from R
    result = result append(L_e)
    remove L_e from L
  else:
    result = result append(L_e)
    remove L_e from L
else if L_aVar < R_aVar:
  R_xi := cross
  L_xf := cross
  if R_xi > R_xf:
    remove R_e from R
  if L_xi > L_xf:
    remove L_e from L
    result = result append (R_e)
    remove R_e from R
  else:
    result = result append (R_e)
    remove R_e from R
else if L_bVar > R_bVar:
  if L_xi > L_xf:
    remove L_e from L
  else if R_xi > R_xf:
    remove R_e from R
    result = result append(L_e)
    remove L_e from L
  else:
    result = result append(L_e)
    remove L_e from L
```

```
      else if L_bVar < R_bVar:
        R_xi := cross
        L_xf := cross
        if R_xi > R_xf:
          remove R_e from R
        if L_xi > L_xf:
          remove L_e from L
          result = result append (R_e)
          remove R_e from R
        else:
          result = result append (R_e)
          remove R_e from R
    if R.isEmpty:
      result = append all of L to result
    else if L.isEmpty:
      result = append all of R to result
    return result
```

3. Proof of Correctness
We are going to prove this by proving in an induction based proof, such that if we prove the simplest case (size of list of Lines = 1), we can prove that it works for all bigger cases.

Our base case in this situation is $|Lines| = 1$. In this case, we know that since there is only one line in the list, this line must be the line that is visible on the x-axis from negative infinity to positive infinity because there are no other lines in the list to cover it up. To go one step higher when $|Lines| = 2$, we have to result to comparison to come up with new areas of visibility (if possible) because part of one line may cover up another part of the other line. We do this by checking if the there is an overlap between its subproblem when split apart as $|Lines| = 1$ in the x-axis. If there is an overlap, we can fix this by finding the interception point (if there are any). Our interception point is where we will change the end of the x-axis of the line that's considered to be "smaller" (meaning it's variable A is less than the other line's, or if they're both equal/does not exist, we compare the B similarly), and change the beginning of the x-axis of the line that's considered to be "bigger". However if in either line, if the initial x is bigger than the final x, that line is no longer valid and should be removed from the list since there does not exist an area on the x-axis such that it is visible. We always add the bigger line first into our resulting list.

Our hypothesis is that we assume our initial base case, and all cases up to $|Lines| < n$ will work accordingly. It should return to us the correct list of lines such that all of these lines in the list have a point on the x-axis that it is visible.

Then for our induction step, consider that $|Lines| = n$. If we split the list into twos in the middle and run our algorithm on it, it should work because $\frac{n}{2} < n$. Similarly, when we get back up to the merge process, we repeat a similar algorithm to our base case of 2 but with all the elements in the two sides. Eventually we would have weeded out all the lines that does not have an area of visibility, and all the ones returned will be visible with the area of x-axis to let us know exactly where it is visible.

4. Running Time Analysis
   $O(nlogn)$ because we split our problem down to half, and at every sub level, we compare all the values to each other (assuming the comparison takes $O(1)$). Meaning we went did n comparisons on every level, and we had $logn$ levels because we split it down the middle. As a result, the total amount of comparisons we did was $nlogn$.

   A better way to think of it is as: $T(n) = 2T(n/2) + O(n)$ where $T(1) = O(1)$. Because we split it in half and applied the analysis on all values it contains on every level. Using Master Theorem, we see that this results into a run time of $O(nlogn)$.

3. The Hadamard Matrices

1. Main Idea
   We are going to split the matrix into 4 quadrants until it results in a H_0 matrix. From there, we are only going to calculate the values of upper left and right quadrants because the bottom quadrants are the same as the upper quadrants, with the exception that the bottom right quadrant is the negative of upper right. We can do this because the left half of H multiplies the same values in v, and the right half of H multiplies the same values in v. As we recursve back to the bigger matrix, we store the values in its original position, duplicate it across the board then add it all together to result in the final vector answer.

2. Pseudocode

```
Procedure NewMatrix(matrix H, vector V):
  if H is a 1 x 1 matrix:
    H[1] := 1 * V[1]
    return H
  middle := (size of V) / 2
  V_left := (1 x mid) matrix, contains values from V[1] to V[mid]
  V_right := (1 x mid) matrix, contains values from V[mid+1] to last element in V
  H_topLeft := top left quadrant of matrix H of size 2^(k-1) x 2^(k-1)
  H_topRight := top right quadrant of matrix H of size 2^(k-1) x 2^(k-1)

  H_topLeft := NewMatrix(H_topLeft, V_left)
  H_topRight := NewMatrix(H_topRight, V_right)
  matrix K :=
      [H_topLeft, H_topRight]
      [H_topLeft, -H_topRight]
  result := 1 x (size of V ) matrix,
    where the value of row1 is the addition of all values in row1 of K
  return result
```

3. Proof of Correctness
   This is an induction proof, where we will prove the first case where H_0 is a 1 x 1 matrix. If it is a 1 x 1 matrix, we can just quickly multiply it against v where it is a 1 x 1 vector. This will result in a new vector that is the answer to the multiplication of Hv.

   Now we can notice that the matrix H_0 is [1]. And by definition, our matrix

$$H_1 = \begin{bmatrix} H_0 & H_0 \\ H_0 & -H_0 \end{bmatrix}$$

   Where ultimately our pattern is this:

$$H_n = \begin{bmatrix} H_{n-1} & H_{n-1} \\ H_{n-1} & -H_{n-1} \end{bmatrix}$$

   It is key to notice that it is all going to be the same pattern of 1's, except for the left-right quadrant where it is the negative values of all the 1's. It is also key to realize everything on the left half will end up multiplying the same values (the top half of the vector v). And because it is the same pattern and values, the top half and the bottom half will ultimately result in the same numbers. So we actually only have to multiply the top-left quadrant with the top half of the vector v, then we can duplicate the resulting matrix to the bottom-left quadrant.

The same concept applies for the right side of our H matrix and the bottom half of the vector v. Instead, the bottom right quadrant of the matrix will have the copy of the top-right quadrant but with all of its values multiply by -1. Now we have sucessfully created the new matrix with all of the values if we were to multiply them normally but have not added them together. Hence why resulting, we need to add the values across to get the correct new matrix.

As we reduce the size, we will be doing the same thing for all the other parts we place the algorithm on. Eventually we will recurse such that the matrix will be 2 x $|v|$, as we have finished calculating all the values on the top left and top right side, duplicate it appropriately to the bottom. Which leaves us with only the horizontal adidtion to finalize our answer.

4. Running Time Analysis
The run time of this is $O(nlogn)$ because of the following: $T(n) = 2T(n)+O(n)$ where $T(1) = 1$. Because we are effectively dividing the matrix into halfs each time, and in each division, we add and multiply for all the values in the step. This is why we have $O(n)$. From the Master Theorem, this gives a run time of $O(nlogn)$, which is what was requested.

4. Modular Arithmetic FT

(a) Through brute force, I found that $\omega = 3$ works such that $(\omega + \omega^2 + \omega^3 + \omega^4 + \omega^5 + \omega^6) \pmod 7) = 0 \rightarrow 3 + 9 + 27 + 81 + 243 + 729 = 1092 \pmod 7) \rightarrow 0$

(b) So given what we have...

$$M_6(\omega) = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 6 & 4 & 5 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 5 & 4 & 6 & 2 & 3 \end{bmatrix}$$

And we want to transform the sequence $(0, 1, 1, 1, 5, 2)$. This means we multiply the two together:

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 3 & 2 & 6 & 4 & 5 \\ 1 & 2 & 4 & 1 & 2 & 4 \\ 1 & 6 & 1 & 6 & 1 & 6 \\ 1 & 4 & 2 & 1 & 4 & 2 \\ 1 & 5 & 4 & 6 & 2 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 5 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 6 \\ 4 \\ 2 \\ 3 \\ 3 \end{bmatrix}$$

All of the intermediate steps are not shown. The resulting matrix is our answer.

(c) We will invert $M_6(\omega)$ and it will result in this:

$$M_6(\omega)^{-1} = \begin{bmatrix} 6 & 6 & 6 & 6 & 6 & 6 \\ 6 & 2 & 3 & 1 & 5 & 4 \\ 6 & 3 & 5 & 6 & 3 & 5 \\ 6 & 1 & 6 & 1 & 6 & 1 \\ 6 & 5 & 4 & 6 & 5 & 3 \\ 6 & 4 & 6 & 1 & 3 & 2 \end{bmatrix}$$

Now to check if this is the correct inverse, we will multiply it by the answer we got in part B, and see if it results in original sequence:

$$\begin{bmatrix} 6 & 6 & 6 & 6 & 6 & 6 \\ 6 & 2 & 3 & 1 & 5 & 4 \\ 6 & 3 & 5 & 6 & 3 & 5 \\ 6 & 1 & 6 & 1 & 6 & 1 \\ 6 & 5 & 4 & 6 & 5 & 3 \\ 6 & 4 & 6 & 1 & 3 & 2 \end{bmatrix} \begin{bmatrix} 3 \\ 6 \\ 4 \\ 2 \\ 3 \\ 3 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 5 \\ 2 \end{bmatrix}$$

Since it did result in the same sequence, we successfully found the inverse. Intermediate steps were not shown.

(d) To multiply these polynomials, we need to convert both into sequences. So $x^2 + x + 1 = (1, 1, 1, 0, 0, 0)$ and $x^3 + 2x - 1 = (6, 2, 0, 1, 0, 0)$. Then we multiply these sequences against $M_6(\omega)$.

$$\begin{bmatrix}1&1&1&1&1&1\\1&3&2&6&4&5\\1&2&4&1&2&4\\1&6&1&6&1&6\\1&4&2&1&4&2\\1&5&4&6&2&3\end{bmatrix}\begin{bmatrix}1\\1\\1\\0\\0\\0\end{bmatrix}=\begin{bmatrix}3\\6\\0\\1\\0\\3\end{bmatrix}$$

and for the other sequence:

$$\begin{bmatrix}1&1&1&1&1&1\\1&3&2&6&4&5\\1&2&4&1&2&4\\1&6&1&6&1&6\\1&4&2&1&4&2\\1&5&4&6&2&3\end{bmatrix}\begin{bmatrix}6\\2\\0\\1\\0\\0\end{bmatrix}=\begin{bmatrix}2\\4\\4\\3\\1\\1\end{bmatrix}$$

Now we multiply the resulting answers:

$$\begin{bmatrix}6\\2\\0\\1\\0\\0\end{bmatrix}*\begin{bmatrix}2\\4\\4\\3\\1\\1\end{bmatrix}=\begin{bmatrix}6\\3\\0\\3\\0\\3\end{bmatrix}$$

And to find the answer to this problem, we need to multiply the answer we got above to our inverse matrix.

$$\begin{bmatrix}6&6&6&6&6&6\\6&2&3&1&5&4\\6&3&5&6&3&5\\6&1&6&1&6&1\\6&5&4&6&5&3\\6&4&6&1&3&2\end{bmatrix}\begin{bmatrix}6\\3\\0\\3\\0\\3\end{bmatrix}=\begin{bmatrix}6\\1\\1\\3\\1\\1\end{bmatrix}$$

And when we convert the sequence to the polynomial it will be: $x^5+x^4+3x^3+x^2+x+6 \pmod 7$, which is what we get if we manually multipled the two values together. All intermediate steps are not shown.

5. Fast Pattern Matching

(a) 1. Main Idea
We manually check every value in s_1 to the last 4 bits of s_2. If it matches with equal or less than k errors, we add the index in which the beginning of the occurance appears in.

2. Pseudocode

```
Procedure Occurance(bits S1, bits S2, leniency k):
  appeared := initially list null, the list of index where pattern appears
  S2Pos := 1
  while S2 is not empty:
    compareBits := the sizeOf(S1) right most bits of S2
    numOfErrors := initially 0, number of how many mismatch bits

    while compareBits is not empty:
      for x from 1 to sizeOf(S1):
        if S1[x] not equal to firstBit(compareBits):
          numOfErrors = numOfErrors + 1
        remove firstBit(compareBits) from compareBits

     if numOfErrors <= k:
        appeared = S2Pos
     numOfErrors = 0
     S2Pos = S2Pos + 1

     remove firstBit(S2) from S2

   return appeared
```

3. Proof of Correctness
This will work because we are literally checking every single value of m to every possible set size of m of n values. If we check every single value, we can manually record how many times it errors. And if the number of errors is less than or equal to the leniency, k, then it is considered to be correct. Hence we just have a list that keeps track of all the location of where its beginning is correct

4. Running Time Analysis
$O(mn)$ because we are literally comparing every single value in S1 (length m), to every single value in S2 (length n). The run time would result in the time it takes for each m to compare to n. And it becomes a multiple of each other, hence our run time is $O(mn)$.

(b) 1. Main Idea
We are going to use Fourier Transformation so we can quickly multiply the polynomial we are going to create given the bit values. From there, we just need to run through the coefficent, checking its value. As long as the value is greater or equal $m - 2k$, that counts as one occurance bit set S2.

2. Pseudocode

```
Procedure Occurance(bits S1, bits S2, leniency k):
  appeared := initially list null, the list of index where pattern appears
```

```
turn all bits that are 0 in S1 to -1
turn all bits that are 0 in S2 to -1
v1 := turn bits in S1 into a vector where
    v1[1] is S1[1], v2[2] is S1[2] ... v1[m] is S1[m]
v2 := turn all bits in S2 into a vector where
    v2[1] is S2[1], v2[2] is S2[2] ... v2[n] is S2[n]
v3 := use Fourier Transformation on v1 & v2
    to find its vector that represents the polynomial
    such that if we were to turn S1 and S2 into polynomial
    and multiplied them together
for all element in v3:
  if element <= m - 2k:
     appeared = append indexOf(element)
return appeared
```

3. Proof of Correctness
   We change all the values of 0's to -1's in our case so we can realize when two different values are multiply each other (hence it is not the same). Because -1 * -1 and 1 * 1 results in the same value, 1, we know that it is the same. However -1 * 1 will result in -1. Now we have a way to differentiate between when it is the same value and when it isn't.

   All of these values goes in to affect the values in our vector (v_3,), which are the coefficents in the new polynomial we have created. And the coeffients in the new polynomial will add up to m if all the bits in S1 matches up with bits in S2. However whenever we get an error, it's -1 rather than 0. This is why at the end, we have to compare it to $m - 2k$ since we not only didn't account for it, we subtracted it.
4. Running Time Analysis
   The running time of this is $O(nlogn)$. The reason for this is because we already know the run time it takes to multiply and find the new vector because we're just FT. This run time is $O(nlogn)$. And we know that it takes $O(n)$ to go through all the bits and switch all the 0's into -1's for S1, and $O(m)$ to do the same thing in S2. Then at the end, we go through all of the elements, which is $m + n$ to compare the value, this will also take $O(m + n)$. As a result, we have our total run time as: $O(nlogn) + O(n) + O(m) + O(m + n)$. And since we only care for the one that will affect our run time the most, we pick $O(nlogn)$ out and decide it as our run time.