CSCI 3104 Spring 2017 **Problem Set 5**

Matthew Niemiec (06/24)
Mehmet Karaoglu (07/30)

1. (50 pts total) Recall that Huffman's algorithm takes as input a vector `f` of length n, containing the frequencies of the input symbol set Σ and returns a Huffman encoding tree. We can make the process of encoding real strings faster by instead returning a "codebook," which is a dictionary ADT T that contains n key-value pairs (x_i, y_i), where $x_i \in \Sigma$ and y_i is the Huffman binary encoding of x_i. Let $T(x_i) = y_i$. Hence, the encoding of $\texttt{x} = x_1x_2x_3 \dots x_\ell$ is simply $\texttt{y} = T(x_1)T(x_2)T(x_3) \dots T(x_\ell)$.

   In this problem, we will implement and apply three functions.

   (i) `string2freq` takes as input an ASCII string `x` and returns (a) a vector `S`, which contains the n unique symbols of `x` in lexicographic order, and (b) a vector `f` containing the frequencies of those symbols, in the same order. Here is pseudo-code:

```
string2freq(x) { // x is a string of symbols from alphabet S
   S = list of unique symbols in x, in lexicographic order
   f = histogram of frequencies of symbols in x, same order
   return S, f
}
```

   (ii) `huffmanEncode` takes as input the vectors `S` and `f`, and returns a dictionary `T`, that represents the codebook:

```
huffmanEncode(S,f) { // f is vector of symbol frequencies
   initialize H.empty()
   for i = 1 to n {  H.insert(i,f[i])  }
   for k=n+1 to 2n-1 {
      i = H.deletemin(), j = H.deletemin()
      create a node numbered k with children i,j
      f[k] = f[i] + f[j]
      H.insert(k,f[k])
   }
   create and return codebook T
}
```

(iii) `encodeString` takes an input ASCII string `x` and a codebook `T`, and returns a string `y`, which is the binary encoding of `x` using `T`:

```
encodeString(x,T) {
   y = empty string
   for i = 1 to x.len() { y += T[x[i]] }  // encode each symbol of x
   return y
}
```

If we then call `y=encodeString( x , huffmanEncode( string2freq(x) ) )`, we can obtain an optimal prefix-free binary encoding `y` of an input string `x`.

(a) From scratch, implement the `string2freq`, `huffmanEncode`, and `encodeString` functions. For Huffman's algorithm, you must use a priority queue data structure (or equivalent), in which `insert` and `deletemin` operations cost $O(\log n)$. You may not use any library that implements a priority queue (or equivalent) data structure—the goal is for you to implement it yourself, using only basic language features. Within the priority queue, ties should be broken uniformly at random.

Submit your code implementation, with code comments.

Hint: a min-heap is a good choice for a priority queue here (Chapter 6 in CLRS).

```
import math, inspect, random

F = open("csci3104_S2017_PS5_data.txt", "r")
poem = F.read()
F.close()

def string2freq(x):
    f = []
    S  = list(set(x)) #list out each symbol as single element and without duplicates
    S.remove("\n")
    xs = sorted(S)    #sort the symbols in lexicographic order
    for c in xs:
        count = x.count(c)  #count each occurence of every single symbol
        f.append(count)   #populate f list with number of occurences
    return xs,f
```

```
def huffmanEncode(S,f):

    class HuffmanNode():
        def __init__(self,left, right, freq, char):
            self.left = left
            self.right = right
            self.freq = freq
            self.char = char

    for i in range(0, len(f)):
        f[i] = (HuffmanNode(None, None, f[i], S[i]))

    def minHeapify(H,i,heapsize):  # this method maintains heap property
        l = 2*i + 1
        r = 2*i + 2
        if (l <= heapsize) and (H[l].freq< H[i].freq):
            smallest = l
        else:
            smallest = i
        if (r <= heapsize) and (H[r].freq< H[smallest].freq):
            smallest = r
        if smallest != i:
            temp = H[i]
            H[i] = H[smallest]
            H[smallest] = temp
            minHeapify(H,smallest,heapsize)

    def buildMinHeap(A):    #building a MinHeap data structure
        heapsize = len(A)-1
        for i in range(int(math.floor(heapsize/2)), -1, -1):
            minHeapify(A, i, heapsize)
        return heapsize

    def heapExtractMin(H, heapsize):   #remove and return the Min node from heap and
        if heapsize < 0:
            print ("heap underflow")
            return
        minimum = H[0]
        H[0] = H[heapsize]
```

```
    heapsize -= 1
    H.pop()
    minHeapify(H, 0, heapsize)
    return minimum, heapsize

def parent(i):
    return int(math.floor(i/2))

def heapDecreaseKey(H, i, node):  #decreasing the key of a particular node in hea
    if node.freq > H[i].freq:
        print ("new key is larger than current key")
        return
    H[i] = node
    while (i > 0) and (H[parent(i)].freq > H[i].freq):
        temp = H[i]
        H[i] = H[parent(i)]
        H[parent(i)] = temp
        i = parent(i)

def minHeapInsert(H, node, heapsize): #inserting new node in Heap
    heapsize += 1
    newNode = HuffmanNode(None, None, float("inf"), None)
    H.append(newNode)
    heapDecreaseKey(H, heapsize, node)
    return heapsize

hs = buildMinHeap(f)

for i in range(0, len(S)-1):  #building the Huffman tree to be used for encoding
    l = heapExtractMin(f, hs)
    hs = l[1]
    r = heapExtractMin(f, hs)
    hs = r[1]
    z = HuffmanNode(l[0], r[0], l[0].freq + r[0].freq, None) #create new node and
    hs = minHeapInsert(f, z, hs)

root = heapExtractMin(f, hs)[0]
current = root

T = {}
```

```
def generateHuffmanDict(node, code):  #find encoding for leaf nodes and build dic

    if (node.left == None) and (node.right == None) :
        T[node.char] = code
    if(node.left != None):
        generateHuffmanDict(node.left, code + "0")
    if(node.right != None):
        generateHuffmanDict(node.right, code + "1")

generateHuffmanDict(root, "")
return T

def encodeString(x, T): #use the dictionary to encode each character of the strin
    y = ""
    for c in range(0, len(x) - 1):
        y += T[x[c]]
    return y
```

(b) Using asymptotic analysis, determine the running time of the call
`y=encodeString( x , huffmanEncode( string2freq(x) ) )`
for an input string `x` where $\forall_i\ x_i \in \Sigma$ and $|\Sigma| = n$. Justify your answer.

In order to bound the total running time $T(n)$ of the call `y=encodeString(x, huffmanEncode(string2freq(x)))`, we can bound the running times of the functions `string2freq` ($T_1(n)$), `huffmanEncode` ($T_2(n)$), and `encodeString` ($T_3(n)$) individually first.

`string2freq`: The built-in Python functions `list()` and `set()` used to convert the inputted string to a list and to list them out (as characters) without any duplicate elements, respectively, both depend on the length of the string, and thus both take $O(n)$ time. The `sorted()` Python function used to sort the characters in lexicographic order takes $O(n \log n)$ time for the average case. Next, there is a `for()` loop that runs exactly n times and inside the loop we use the Python function `count()` which runs in $O(n)$ times at worst, giving us $O(n^2)$ time in total. All the rest of the lines run in $O(1)$ time. We can just get rid of the lower order terms and say this function is $O(n^2)$.

`huffmanEncode`: The `huffmanEncode` function runs in $O(n \log n)$ time. We get this bound by analyzing the Huffman prefix code building procedure. This procedure contains a `for()` loop which runs $n - 1$ times and each of the Min-Heap

procedures that we call take $O(logn)$ time. Multiplying these gives us a running time of $O(n \log n)$.

`encodeString`: Our last function, `encodeString`, has a goes through each of the symbols in the string and tries to math it with a code taken from the dictionary T. This gives a worst case of $O(n^2)$.

Using the information above, we can conclude that our algorithm takes $O(n^2)$ time at its worst case.

(c) The data file for PS5 (see class Moodle) contains the text of a famous poem by Robert Frost. The text is given as a string `x` containing $\ell = 761$ symbols drawn from an alphabet Σ containing 31 ASCII symbols (24 alphabetical characters, 6 punctuation marks and 1 space character).
A plaintext ASCII character normally takes 8 bits of space. How many bits does `x` require when encoded in ASCII?

When encoded in ASCII, `x` requires $761 * 8 = 6088$ bits in total.

(d) Claude Shannon famously proved that the lower bound on the number of encoded bits per input symbol for a prefix-free binary encoding of a string `x` is given by the *entropy* function, defined as

$$H = -\sum_{i=1}^{|\Sigma|} \left(\frac{f_i}{\ell}\right) \log_2 \left(\frac{f_i}{\ell}\right) \quad , \tag{1}$$

where f_i is the frequency in `x` of the ith symbol of Σ, and ℓ is the length of `x`. Because we take $\log_2$, H has units of "bits." Using Eq. (1), compute (i) the predicted number of bits per symbol needed to optimally encode the Frost poem, and (ii) the predicted number of bits needed to encode the entire poem.

I don't know.

(e) Now, encode `x` using your Huffman encoder from part (1a) and report the number of bits in the encoded string. Compare this number to the lower bound obtained

in part (1d), and comment on the comparison.

When we encode the Huffman encoder by doing:

```
len(encodeString(poem, T))
```

We find out that the number of bits in the encoded string is 3174 bits, which is smaller than the amount of bits calculated by the entropy function.

(f) (5 pts extra credit) Transmitting an optimally encoded message minimizes the transmission cost, but without the codebook, the receiver cannot decode it. Oops. How many additional bits would be required to write down the codebook? Could this number be made any smaller?

I don't know.

2. (35 pts) Here, you will modify and then use your Huffman encoder from part (1) to perform a *numerical experiment* to show that your encoder uses $\Theta(n \log n)$ time on an input of size n. The deliverables here are (i) a nice figure showing this result and (ii) with a brief description of how you modified your code and ran your experiment.

   First, implement a new function `makeHuffmanInput`, which takes an argument n, the size of the symbol set Σ, and returns a vector of counts `f` of length n, in which each f_i is a positive integer chosen uniformly at random from the interval $[1, 100]$:

```
makeHuffmanInput(n) { // n is the number of symbols
   for i = 1 to n {
      f[i] = rand-int(1,100)  // random frequency from 1..100
   }
   return f
}
```

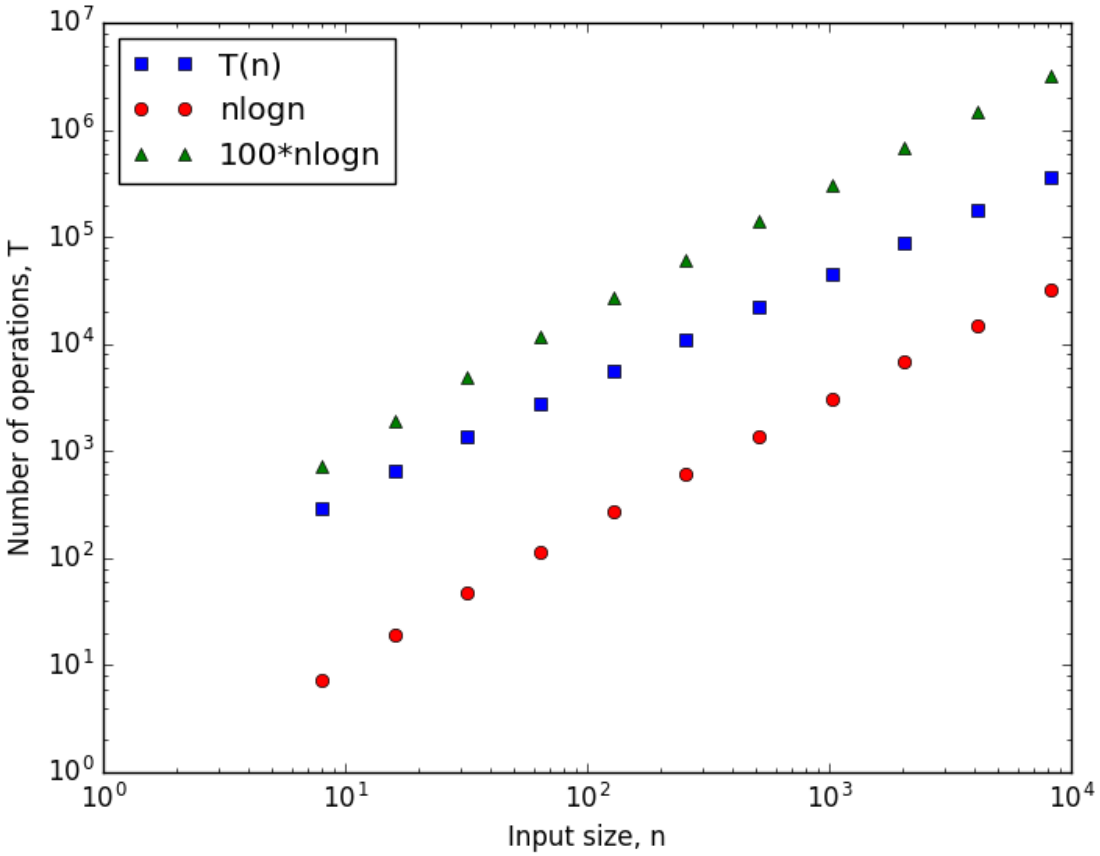
   Second, modify your `huffmanEncode` function by adding *measurement code* that counts the number of atomic operations it performs before returning. You do not need to count all atomic operations, only the ones whose number depends on the input size n. (Hint: think about how you would analyze your code's asymptotic running time.) When your modified function returns, it should return this count as well.

   Now, perform the numerical experiment, collect the data, plot the results, and write your explanation.

   To run the experiment, for each value of $n \in \{2^3, 2^4, 2^5, \ldots, 2^{13}\}$, generate several inputs using `makeHuffmanInput`, and run each input through your modified `huffmanEncode` to obtain a set of operation counts for each value of n. Then, compute the average number of operations $T(n)$ your algorithm used for each value of n, and visualize the function $T(n)$ on a "log-log" plot (where both the x- and y-axes are logarithmic), like the one below.

   Include two trend lines on your figure of the form $c \times n \log n$, which bound your results from above and below. Label your axes and trend line. Give a clear and concise description (1-2 paragraphs) of exactly how you modified your code and ran your experiment.

   No credit if axes or trend lines are not labeled, or if your plot is not on log-log axes.



For our numerical experiment, we first wrote a new function called `makeHuffmanInput(n)` that gives us a random list of numbers which we use as input for our modified Huffman encoding function. The Huffman encoding function that we use for this experiment is a bit different than the function we used for 1a). We added counter variables throughout our function and the helper functions inside the function to count the number of atomic operations. We didn't count every single operation in our function, only the operations which affect the running time as n grows substantially larger. Our function returns the number of operators counted, which we then use to estimate the running time $T(n)$.

Then, we ran simulations of Huffman encoding for input sizes of $n \in \{2^3, 2^4, 2^5, \ldots, 2^{13}\}$. For each input n, we ran 5 separate simulations and then we used the average of those

5 simulations for our y-value for that particular n. In the end, we plotted our function $T(n)$ on a log-log scale on both axes. We were able to find two functions $100 * n \log n$ and $n \log n$ which bounded our function from above and below, respectively, sandwiching it between them. By looking at this plot, we were also able to see that $n \log n$ provides a tight bound for $T(n)$.

3. (10 pts) Ginerva Weasley is playing with the network given below. Help her calculate the number of paths from node 1 to node 14.

   Hint: assume a "path" must have at least one edge in it to be well defined, and use dynamic programming to fill in a table that counts number of paths from each node j to 14, starting from 14 down to 1.

   Since the graph is directional, the number of paths to any given node is just the sum of the number of paths to all of the connected nodes. Let $P(x)$ be the number of paths to node x.

Node	1st connection	2nd connection	3rd connection	total
1	N/A	N/A	N/A	1
2	P(1)=1	N/A	N/A	1
3	P(2)=1	N/A	N/A	1
4	P(3)=1	N/A	N/A	1
5	P(4)=1	N/A	N/A	1
6	P(5)=1	N/A	N/A	1
7	P(1)=1	P(2)=1	N/A	2
8	P(2)=1	P(3)=1	P(7)=2	4
9	P(3)=1	P(8)=4	N/A	5
10	P(5)=1	P(9)=5	N/A	6
11	P(6)=1	P(10)=6	N/A	7
12	P(9)=5	N/A	N/A	5
13	P(10)=6	P(12)=5	N/A	11
14	P(11)=7	P(13)=11	N/A	18

   Thus the total number of paths from node 1 to node 14 is **18**.

4. (30 pts total) Ron and Hermione are having a competition to see who can compute the nth Lucas number L_n more quickly, without resorting to magic. Recall that the nth Lucas number is defined as $L_n = L_{n-1} + L_{n-2}$ for $n > 1$ with base cases $L_0 = 2$ and $L_1 = 1$. Ron opens with the classic recursive algorithm:

```
Luc(n) :
   if n == 0 { return 2 }
   else if n == 1 { return 1 }
   else { return Luc(n-1) + Luc(n-2) }
```

which he claims takes $R(n) = R(n-1) + R(n-2) + c = O(\phi^n)$ time.

(a) Hermione counters with a dynamic programming approach that "memoizes" (a.k.a. memorizes) the intermediate Lucas numbers by storing them in an array `L[n]`. She claims this allows an algorithm to compute larger Lucas numbers more quickly, and writes down the following algorithm.
[1]

```
MemLuc(n) {
   if n == 0 { return 2 } else if n == 1 { return 1 }
   else {
      if (L[n] == undefined) { L[n] = MemLuc(n-1) + MemLuc(n-2) }
      return L[n]
   }
}
```

i. Describe the behavior of `MemLuc(n)` in terms of a traversal of a computation tree. Describe how the array `L` is filled.
The first algorithm calculates every Lucas number twice. For example, when finding the Lucas number at $n = 5$, the algorithm calculates $Luc(4)$, but in doing so calculates $Luc(3)$ and $Luc(2)$. This is good, but when the process reaches the first call and goes to calculate $Luc(n-2)$, it calculates $luc(3)$ all over again! The second algorithm doesn't do this. Instead, it keeps all of the numbers in a (presumably) global array, so that way instead of going to calculate $MemLuc(3)$, it just references $L[3]$, which is much much faster for large numbers.

[1] Ron briefly whines about Hermione's `L[n]=undefined` trick ("an unallocated array!"), but she point out that `MemLuc(n)` can simply be wrapped within a second function that first allocates an array of size n, initializes each entry to `undefined`, and then calls `MemLuc(n)` as given.

ii. Determine the asymptotic running time of `MemLuc`. Prove your claim is correct by induction on the contents of the array.

The recurrence relation for the MemLuc algorithm is $T(n) = T(n-1) + O(1)$, and the runtime is $O(n)$ because the function makes a total of $2n$ calls, each of which takes constant time. For example, when calling $MemLuc(5)$, it calculates $MemLuc(4)$ and $MemLuc(3)$, but by the time the second call runs the value has already been stored in the array, so it returns that value rather than recalculating it, so it makes no more recursive calls. Thus there are 2 calls for all 1 to n. We prove this using induction by creating base cases, $MemLuc(0) = MemLuc(1) = 1$. Then our hypothesis is that $T(n+1) \leq n+1$. We see that since the time it takes to call the $n+1$ number is a constant addition and it takes $\leq n$ time to make the base calls,

$$T(n+1) \leq T(n) + T_{call}(n+1) \tag{2}$$
$$\leq n+1 \tag{3}$$

(b) Ron then claims that he can beat Hermione's dynamic programming algorithm in both time and space with another dynamic programming algorithm, which eliminates the recursion completely and instead builds up directly to the final solution by filling the L array in order. Ron's new algorithm is [2]

```
DynLuc(n) :
   L[0] = 2,   L[1] = 1
   for i = 2 to n {  L[i] = L[i-1] + L[i-2]  }
   return L[n]
```

Determine the time and space usage of `DynLuc(n)`. Justify your answers and compare them to the answers in part (4a).

This algorithm runs through a for loop. Inside there are constant operations and the loop runs $n-1$ times. Therefore, the runtime is $O(n)$. The memory allocates an array of size n as well as and integer i, so the memory efficiency is $O(n+1) = O(n)$. This is the same asymptotic runtime as and memory usage as part 4(a), though technically there isn't the overhead of the recursive calls.

[2] Ron is now using Hermione's undefined array trick; assume he also uses her solution of wrapping this function within another that correctly allocates the array.

(c) With a gleam in her eye, Hermione tells Ron that she can do everything he can do better: she can compute the nth Lucas number even faster because intermediate results do not need to be stored. Over Ron's pathetic cries, Hermione says

```
FasterLuc(n) :
   a = 2,  b = 1
   for i = 2 to n
      c = a + b
      a = b
      b = c
   end
   return a
```

Ron giggles and says that Hermione has a bug in her algorithm. Determine the error, give its correction, and then determine the time and space usage of `FasterLuc(n)`. Justify your claims.

The bug is that the algorithm returns a instead of b. a represents the first Lucas Number in the two it keeps track of at any given time, however, after running through the loop n-2 times, the n^{th} Lucas number will be in the second spot, not the first. This is also contingent upon the fact that the loop does, in fact, run $n-2$ times, which means that the loop runs the n^{th} time in the loop, and doesn't stop after $n-1$. This implies that $Luc(2) = 3$, since $L[0] = 2$

(d) In a table, list each of the four algorithms as rows and for each give its asymptotic time and space requirements, along with the implied or explicit data structures that each requires. Briefly discuss how these different approaches compare, and where the improvements come from. (Hint: what data structure do all recursive algorithms implicitly use?)

Algorithm	Runtime	Space Efficiency
`Luc`	$O(\phi^n)$	$O(\phi^n)$
`MemLuc`	$O(n)$	$O(n)$
`DynLuc`	$O(n)$	$O(n)$
`FasterLuc`	$O(n)$	$O(1)$

Luc uses a binary tree of recursive, because at each recursion two more recursive calls are made. MemLuc technically makes a binary tree since every other call makes two more calls. However, it ends up looking more like a linked list with an extra node sticking out of each node on the list, or like a severely unbalanced tree.

This is good in this case because it still has depth n, but is not a full tree. DynLuc uses an array L, as well as an integer i to iterate through the for loop. This uses less than half the space as MemLuc, since there are twice as many recursions as spaces in the array, and each recursive call takes up more memory than an integer. FasterLuc uses just 4 integers - 2 to keep track of the current Lucas numbers, one temp variable c, and i, which really isn't ever used as a variable, just a way to iterate through the loop a specific number of times.

(e) (5 pts extra credit) Implement `FasterLuc` and then compute L_n where n is the four-digit number representing your MMDD birthday, and report the first five digits of L_n. Now, assuming that it takes one nanosecond per operation, estimate the number of years required to compute L_n using Ron's classic recursive algorithm and compare that to the clock time required to compute L_n using `FasterLuc`.

Again, assuming that $Luc(2) = 3$, my birthday is June 24th, so $Luc(624) = 25602...67682$. We know that `FasterLuc` takes $O(n)$ time to run the program. Specifically, there are no addition or multiplication operations outside of the loop, and there is one addition inside of the loop, so we can expect $n - 1 = 624 - 1 = 623$ operations. The original algorithm is trickier. If the algorithm were a perfect tree (called "return Luc(n-1) + Luc(n-1)"), then we could calculate the number of operations with the following algorithm:

```
totalOps = 0
for i=1 to n
    totalOps = totalOps + 2**i
return totalOps
```

And it is far more difficult to count the actual number, so we will just count the upper bound as 2^n, which is an upper bound for all n. However, for a closer approximation we can use 1.53^n, which is an upper bound after $n = 27$. So we know that the operations will be no more than $1.53^{624} = 1.768\text{x}10^{115}$ operations. To get the time in nanoseconds, we simply divide each number by 10^9, which results in $624\text{x}10^{-9} = 6.24\text{x}10^{-7}$ seconds to execute. The original takes $1.768\text{x}10^{106}$ seconds to execute. We know that there are 365 days/year x 24 hours/day x 60 minutes/hour x 60 seconds/minute $= 31,536,000$ seconds/year. This means that the algorithm will take on the order of $5.6\text{x}10^{98}$ years to run. We, as humans, really have a hard time fathoming how many years that is. Even if the algorithm isn't near that slow, the heat death of the universe will occur long before the algorithm finishes running. And the `FasterLuc` takes a very small fraction of a second.

5. (25 pts extra credit) Professor Dumbledore hands you a set X of $n > 0$ intervals on the real line and asks you to find a subset of these intervals $Y \subseteq X$, called a *tiling cover*, where the intervals in Y cover the intervals in X, that is, every real value contained within some interval in X is also contained in some interval Y. The *size* of a tiling cover is just the number of intervals $|Y|$. To satisfy Dumbledore, you must return the minimum cover $Y_{\min}$: the tiling cover with the smallest possible size.

   For the following, assume that Dumbledore gives you an input consisting of two arrays $X_L[1..n]$ and $X_R[1..n]$, representing the left and right endpoints of the intervals in X.

   (a) In pseudo-code, give a *greedy* algorithm that computes the minimum cover $Y_{\min}$ in $O(n \log n)$ time. Justify your answer. (Hint: Starting with the left-most tile.)

   First what you want to do is sort the two arrays based on the first array. This just means that we need to sort the first array, but every operation we do in the process we do to the second array. If there is a tie, we sort by the second array. The idea is that we start at a currentIndex=1, and then we go through and find which one of the intervals with the current index reaches the farthest, then when we find it we append the left, then the right to the list of tiles that we're going to return. Then we go through and find out if there's an interval that starts to the left of the end of our interval and ends to the right. If there is, we find the one that ends farthest to the right, thus being *greedy*. Then if there's not such an interval, we find the interval that starts closest to the interval that we were just one, which we have to do since we must cover all intervals, and then start the process over again. It is $O(n \lg n)$ because the longest part of the algorithm is the Merge Sort, which takes $O(n \lg n)$ time.

```
someSorcery(magic)
    modifiedMergeSort(Xl, Xr)
    tiles = []
    currentIndex = 1
    numOfTiles = 0
    while Xr[currentIndex] != Xr[n]
        numOfTiles++
        temp = currentIndex
        currentLeft = Xl[temp]
        longestInt = Xr[temp]
        longestIndex = temp
        temp++
        while Xl[temp] == xl[currentIndex]
```

```
            if xl[temp] > longestInt
                longestIndex = temp
                longestInt = Xl[temp]
        tiles.append(Xl[longestIndex], Xr[longestIndex])

        // Case when there is NOT a gap between the end
        // of the current interval and the start of the next
        longestTile = longestInt
        i = longestIndex
        while Xl[i] <= Xr[longestIndex]
            if Xr[i] > Xr[longestIndex] and Xr[i] > longestIndex
                longestIndex = i
                longestTile = Xr[i]
            i++
        if longestTile != longestInt
            currentIndex = longestIndex
            continue  // Skips the next part of the code

        // Case where there is a gap between the end
        // of the current interval and the start of the next
        i = longestIndex
        while Xl[i] <= Xr[currentIndex]
            i++
        currentIndex = i + 1
    return tiles
```

(b) Prove that your algorithm (i) covers each component and (ii) produces the minimal cover for each component. Explain why these conditions are necessary and sufficient for the correctness of the algorithm.

The algorithm covers every component because we always search for an interval that doesn't create a gap in between intervals, but if that isn't possible, then we go immediately to the next smallest-starting interval. Thus every interval gets covered. Now for optimization, let's assume that there is a more optimal solution. If there were an interval that lasted longer, then it would have been found by the algorithm. Likewise, if there were more than one interval that start at the same place after the end of the current interval, then it would have been found in the sorting at the start of the main while loop. Also, any interval that overlaps with an interval that ends before another given interval also overlaps with that given interval, so it cannot be more efficient.