Please enter your name and uID below.

Name: Devin White

uID:u0775759

Collaborators, if any, and how you collaborated:

Submission notes

- Due at 11:59 pm (midnight) on Tuesday, Sept 15, 2021.
- Solutions must be typeset using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for this problem set, you are allowed to collaborate in detail with your peers, as long as you cite them. However, you must write up your own solution, alone, from memory. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Parameter passing)

   Or more simply....

   The Mergesort algorithm works by continually dividing the parameter array into 2 equal(or relatively) halves. It then divides those 2 halfs in half, then those 1/4ths in half to 1/8.....until each array is divided into a single element, at which point it begins to compare and combine these elements backwards into larger arrays. Due to this, the **general** Reccurence can be summed up as: 2T(each call produces 2 "branches") + and $\frac{n}{2}$(each "branch" or new array is approximately $\frac{1}{2}$ of the parent array. + O(n) because to merge it must go through all n elements This gives an overall **general** recurrence relation of $T(n) = 2T(n) + O(n)$ We can guess just by looking at the tree that the general run time of Mergesort is $n \log n.$, because for each division by 2, you make a new level of leafs until it eventually gets to 1 after $\log_2 n$ divisions

   We can figure out the cost of parameters by adding them to the end of the generalized equation:

   (a) When the array is passed by pointer, the equation for the recurrence is $T(n) = 2T(\frac{n}{2}) + cn + parameter = 2T(\frac{n}{2}) + \theta(n) + 2$
   We can see immediately that this will have the same run time as the generalized equation, mentioned above, which makes sense, as it is passing by pointer, which is a constant, and doesn't alter how the algorithm runs from our "reference" this gives us a run time of $O(n \log(n))$ However, with the master theorem, where a = 2, b = 2, gives us the case where $n^{\log_b a} = n^{\log_2 2 = n^1} which with is the case of \theta(n \log n)$

   (b) By copying the contents of the array upon every recursion, rather than passing a reference, we have to do re-do bunch of the work every single recursion call, Essentially, if the cost of passing a parameter is N, then each recursion will incur a cost of 2N, one array for each "split". we can think of the recurrence as , $T(n) = T(R) + T(N)$ (r for recurrence work, N for passing parameters) or $T(n) = 2T(\frac{n}{2}) + O(n) + 2N$. Looking at the tree, we can see that work is increasing upon each recursion call, the general equation narrows down to $\sum_{i=0}^{\log_i = 0^n - 1} * cn + N * \sum_{i=0}^{\log_i = 0^n - 1} 2^i = cn \log n + nN - N - \theta(nN)$ This gives us a running time of (n^2)

   (c) When the subarray is passed, is looks similar to passing the original N array in the above. However, since we are only copying over the smaller n/2 array each time, and this is decreasing work with every recursive call we get a recurrence similar to the running time of the pointer, but with the added n work of coppying the $\frac{n}{2}$ arrays. Looking at the tree, we get a basic relation of $T(n) = 2T(\frac{n}{2}) + O(n) + \frac{2n}{2}$, we're doing n amount of work copying each array, and $n/2^k$ work on each recursion, which translate to $\log n$ which comes out ultimately to a bound of $\theta n \log n$
   This gives us a running time of $O(NlogN)$

2. (Index fixed point)

   Known Facts:

   - Array is Sorted
   - each item in array is distinct
   - **Therefore:** $A[i-1] < A[i] < A[i+1]$ and each must differ by at least 1 ($A[i-1]$ must be AT LEAST 2 less than $A[i+1]$)
   - The algorithm returns a boolean True or False

   Using the known facts above, we know that if $A[i] > i$, then $A[i+1]$ is greater than $i+1$, conversely, if $A[i] < i$, then $A[i-1]$ is less than $i-1$. The proposed algorithm goes as follows:

   - Pick the middle index of the initial array of size n ($A[\frac{n}{2}]$)
   - There are then 3 possibilities
     (a) if $A[\frac{n}{2}] = \frac{n}{2}$ (or rather, A[i] = i)Then the program terminates immediately and returns true. This is the base case
     (b) if $A[\frac{n}{2}] > \frac{n}{2}$ then we know that $A[(\frac{n}{2})+1] > ((\frac{n}{2})+1$ and $A[(\frac{n}{2})+2] > (\frac{n}{2})+2$ ..... $A[(\frac{n}{2})+n] > (\frac{n}{2})+n$ Therefore, the right half($i > \frac{n}{2}$) can be totally ignored. We can then use recursion on the left half of the array($i < (\frac{n}{2})$) and repeat these 3 possibilities with recursion as needed, with the result returned from the recursion(Return(IndexFixedpoint(A[0...i]))
     (c) if $A[\frac{n}{2}] < \frac{n}{2}$ then we know that $A[(\frac{n}{2})-1] < ((\frac{n}{2})-1$ and $A[(\frac{n}{2})-2] < (\frac{n}{2})-2$ ..... $A[(\frac{n}{2})-n] < (\frac{n}{2})-n$ Therefore, the left half($i < \frac{n}{2}$) can be totally ignored. We can then use recursion on the right half of the array($i > (\frac{n}{2})$) and repeat these 3 possibilities with recursion as needed, with the result returned from the recursion(Return(IndexFixedpoint(A[i...n]))

   Eventually, after the above is complete, true or false is returned depending on if A[i] = i was found in any of these sub-arrays during recursion.

   Give a recurrence for the worst case running time of your algorithm, and derive a tight upper bound on the closed form solution of the recurrence.

   Upon each iteration, this algorithm(binary search) can discard one half of the size n array, and pass the remaining half as the Parameter for the next recursion. In essence, it makes a single recursive call $1T$,which then splits the array in half $\frac{n}{2}$ there is no recombining needed, just cutting the array shorter and shorter, until(or if) it finds a point where A[i] = i so additional work comes to be constant. Giving us the recurrence relation for the algorithm $T(n) = T(n/2) + O(1)$
   The tree should look something like:

   - $T(n) = T(n/2) + O(1)$ 1 branch
   - $T(\frac{n}{2}) + 1$ 1 branch
   - $T(\frac{n}{2}) = T(\frac{n}{4}) + 1$ 1 branch
   - $T(\frac{n}{4}) = T(\frac{n}{8}) + 1$ 1 branch
   - .....
   - $T(\frac{n}{k}) = T(\frac{2}{2^k}) + 1$ 1 branch
   - Since this is a relatively simple "divide and conquer" algorithm, we can see that from the tree steps above, $2^k = 1$ which should give us an asymptotic bound of $\theta(\log n)$

3. (Weighted median)

   (a) The algorithm is as follows:

   - we input our 2 Arrays, S[0..n] and W[0..n], and the median(input/2) as parameters
   - If S[0..n] and W[0..n] sizes do not match. Return
   - If S[0..n] and W[0..n] are size 1. Return S[0] (Base case)
   - If S[0...n] and W[0...n] are ¡ some arbitrary number, brute force a median by sorting both S[] and W[] and taking the middle index.(other base case)
   - Otherwise, find the absolute median of the array with Quickselect, this will be our pivot.
   - Use a partitioning subroutine on the pivot. Put the weight W[i] of the pivot S[i] in a variable, which we will call P as a reference. The partitioning subroutine will split S[0...n] into 2 sides, with the left side being values less than S[i] and the right being those greater than S[i], making sure to move the weight W[n] alongside its matching S[n] value. We will call these 2 partitions L and G for lesser and Greater
   - Upon returning from partitioning, sum the weights by adding L + P + R. This is the total weight, we will call Tw
   - sum up the weights in array L, which we will call Lw, subtract this from Tw, this is the middle goal weight, or Gw.
   - if Lw is greater than W[]/2
     - Add p to Gw, Recurse back to step 1 with the lesser array now using Gw for comparison.
   - if Gw is greater than W[]/2
     - Add p to Lw, Recurse back to step 1 with the greater array now using lW for comparison.
   - if Both sides are equal to Gw, then return that number, that is the median

   (b) This Algorithm runs at approximately O(n) if the pivot is optimally picked, which, while not able to be guaranteed, can be more consistently met with several pivot picking strategies such as Median-of-medians, etc. When an optimal pivot is picked for this algorithm, then half of the parent array can be discarded upon each recursion, reducing the total amount of work done per level by searching smaller and smaller portions of the input arrays essentially giving us a tree of $n- > \frac{n}{2}- > \frac{n}{4}- > \frac{n}{8}.....\frac{n}{2^k}$ which comes out to 2n and a $\theta(n)$