Pset2

Problem 0: Reccurences (20%)

Give upper (O(·)) asymptotic bounds for the following recurrences. You may assume a O(1) base case for small n. Justify your answer by some combination of the following: deriving how much total work is done at an arbitrary level k, how many levels there are, and how much work is required to merge (function body). For each recurrence, state whether or not it is top-heavy, bottom-heavy, or even work. Answers that only cite the Master theorem will not receive full credit.

Solutions

1. $T(n) = 2T(\frac{n}{2}) + O(n)$ a=2, b=2, d=1
    1. Work Done at level k: $O((n^1)(\frac{2}{2^1}))^k$ = $O(n)^k$
    2. Number of Levels: there are $log_2 n$ levels, each with 2^k subproblems
    3. Work Required to Merge: $O(n)$
    4. TH,BH,EW?: EW
    5. This equation shows that we create two new subproblems each itteration; each of these subproblems being half the size of the previous problem. The total of this takes O(n) work at the top. Because these are creating 2 problems of half the work, there is equal work throughout the algorithm.
2. $T(n) = 2T(\frac{n}{2}) + O(1)$ a=2, b=2, d=0
    1. Work Done at level k: $O(n^0)(\frac{2}{2^0})^k$ = $O(2^k)$
    2. Number of Levels: there are $log_2 n$ levels, each with 2^k subproblems
    3. Work Required to Merge: O(1)
    4. TH,BH,EW?: BH
    5. This equation shows we create two new subproblems, each requiring half the work of the previous subproblem. Since d = 0, we know that the problem itself would have had a constant runtime, which means this is a bottom heavy algorithm as the original problem only takes O(1) time to solve and would likely not be best to solve with Divide and Conquor.
3. $T(n) = 7T(\frac{n}{2}) + O(n^3)$ a=7, b=2, d=3
    1. Work Done at level k: $O(n^3)(\frac{7}{2^3})^k$
    2. Number of Levels: there are $log_2 n$ levels, each with 7^k subproblems
    3. Work Required to Merge: $O(n^3)$
    4. TH,BH,EW?: TH
    5. This equation shows we create 7 new subproblems each only half of the size of the previous subproblems. This is exponential creation of new problems, and we multiply the number of new problems by a greater factor than we reduce their work, meaning the problem becomes harder

to solve as we get deeper, despite this, the ammount of work we need to do at the top is n^3, which is greater than the work we do at the k'th level $n^{log_2 7}$. This would mean the algoritm is Top Heavy.

4. $T(n) = 7T(\frac{n}{2}) + O(n^2)$ a=7, b=2, d=2
    1. Work Done at level k: $O(n^2)(\frac{7}{2^2})^k$
    2. Number of Levels: there are $log_2 n$ levels, each with 7^k subproblems
    3. Work Required to Merge: $O(n^2)$
    4. TH,BH,EW?: BH
    5. In this equation, we produce 7 new subproblems, each of half the size. Our original problem took $O(n^2)$, meaning the algoritm would be bottom heavy as its simliar to problem 3, but has a lighter top at n^2 instead of n^3 , because we are making 7 new problems, but they are only half as difficult to solve. Since our top number is $O(n^2)$, its less than the work done at the k'th level making this bottom heavy rather than top heavy.
5. $T(n) = 4T(\frac{n}{2}) + O(n^2 * \sqrt{n})$ a=4, b=2, d=$\frac{5}{2}$
    1. Work Done at level k: $O(n^{\frac{5}{2}})(\frac{4}{2^{\frac{5}{2}}})^k$
    2. Number of Levels: there are $log_2 n$ levels, each with 4^k subproblems
    3. Work Required to Merge: $O(n^2 * \sqrt{n})$
    4. TH,BH,EW?: TH
    5. In this equation, we produce 4 new subproblems, each half the size of the previous subproblem. Our top level work takes $O(n^2 * \sqrt{n})$ and our bottom level takes $n^{log_2 4}$ or n^2. Since our top level takes more work than our bottom level, this is a top heavy algorithim.
6. $T(n) = 7T(\frac{n}{2}) + O(nlog_2(n))$ a=7, b=2, d=
    1. Work Done at level k: $O(n^{nlog_2(n)})(\frac{7}{2^{nlog_2(n)}})^k$
    2. Number of Levels: there are $log_2 n$ levels, each with 7^k subproblems
    3. Work Required to Merge: $O(nlog_2(n))$
    4. TH,BH,EW?: BH
    5. This is a bottom heavy algorithm. We are producing 7 new subproblems of half the original problems size for each itteration. Our top level takes $O(nlog_2(n))$ and our bottom takes $n^{log_2 7}$, making our bottom level harder to compute than our top level.

Problem 1: Covering a chess Board (20%)

You are given a 2k×2k board of squares (e.g. a chess board) with the top left square removed. Prove, by giving a divide-and-conquer algorithm or argument, that you can exactly cover then entire board with L-shaped pieces (each covering 3 squares).

Solutions

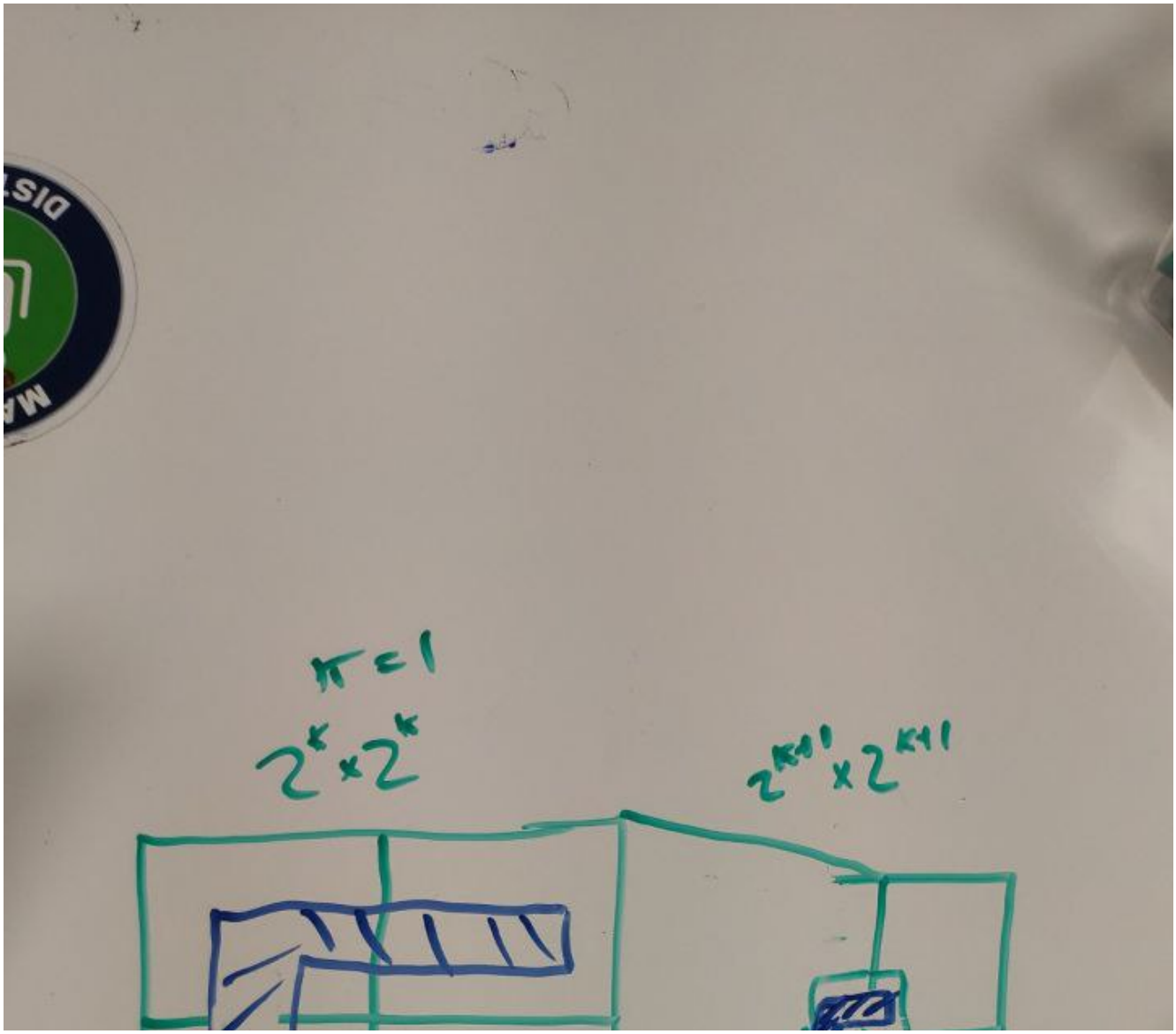
Let P(n) be the proposition that a 2^nx2^n can be tiled using L-shaped peices of size 3. I will now prove $P(n)$ is true for all positive integers n

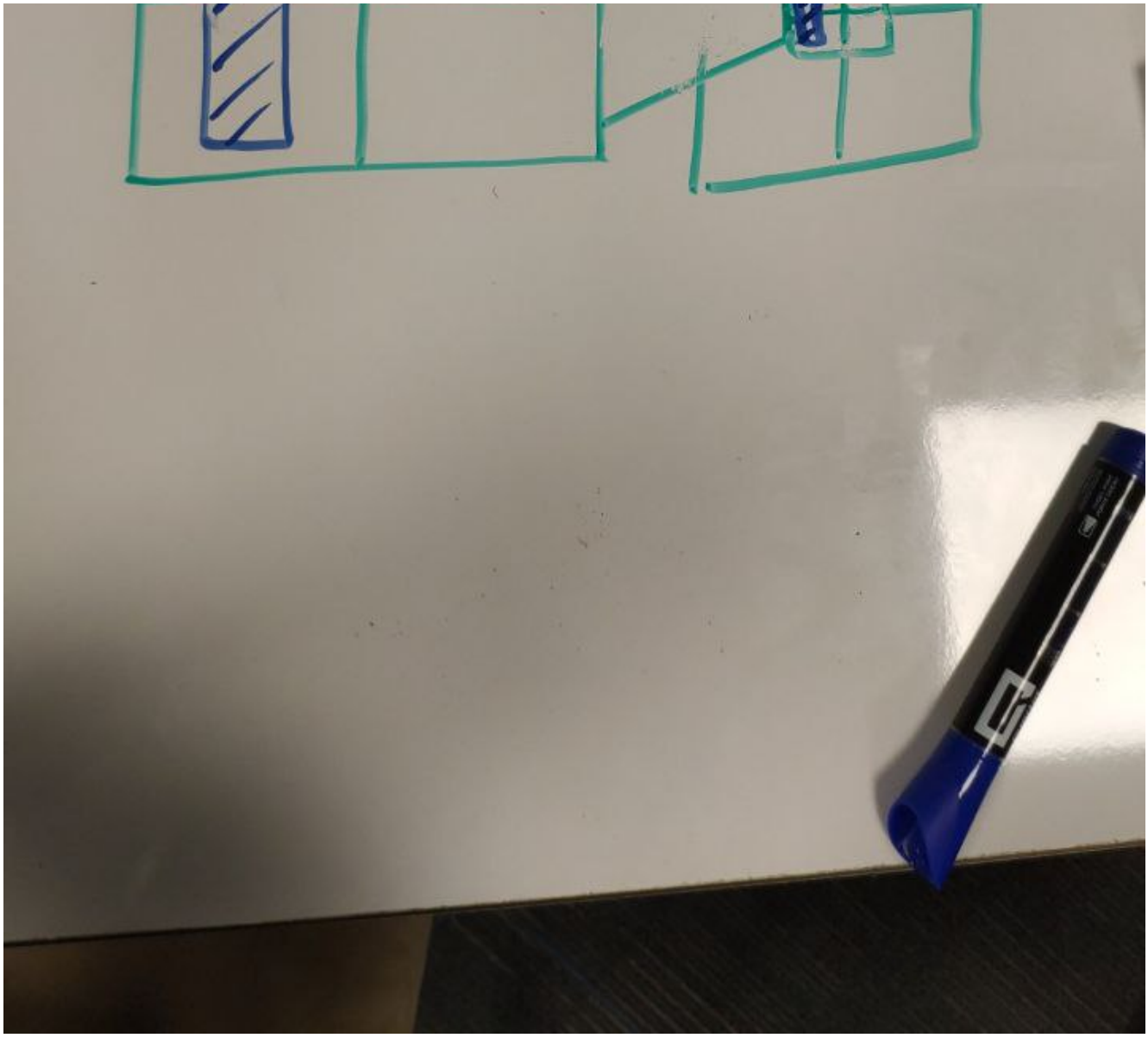
Case 1: P(1), A 2x2 Chessboad

- Solution is Trivial
- 2x2 grid is filled with one peice missing

Case 2: P(k), 2^kx 2^k size chess board can be tiled with L-shaped peices of size 3.

- Induction: Prove $P(k+1)$ must be true
- Consider a checkerboard of size 2^{k+1}x2^{k+1}, this checkerboard could be cut into 4 equal peices of size 2^kx2^k
  - By tiling these subproblems, it leaves us with a tiling that leaves 1 extra open spot on each of the quadrents
  - The if this spot is situated towards the middle, a third L3 peice could tile the middle portion leaving one peice left
- this shows we can tile a checkerboard of size 2^{k+1}x2^{k+1} if we can tile a checkerboard of size 2^k
  - By induction, we have proved that P(k+1) given P(k)
- Therefore, by mathmeatical induction, P(n) is true for all positive integers *n*, where n is a positive integer with one square removed





Problem 2: Counting inverted Pairs (20%)

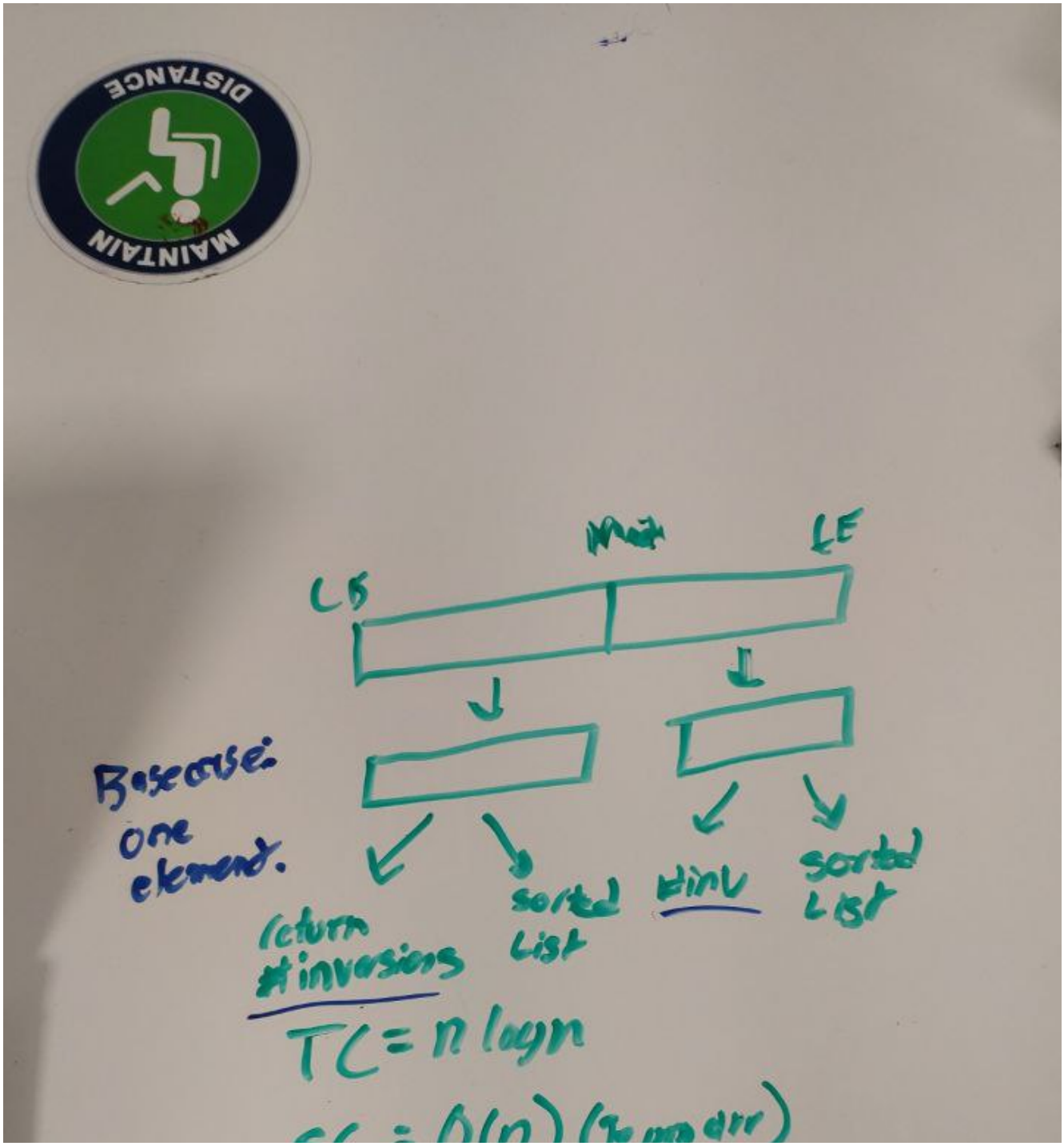
You are given an unsorted list L that has k ≥ 0 pairs of indices i < j such that L[i] > L[j]. These are called inverted pairs. Develop an O(n log n) algorithm that counts the number of inverted pairs (i.e. compute the value k).

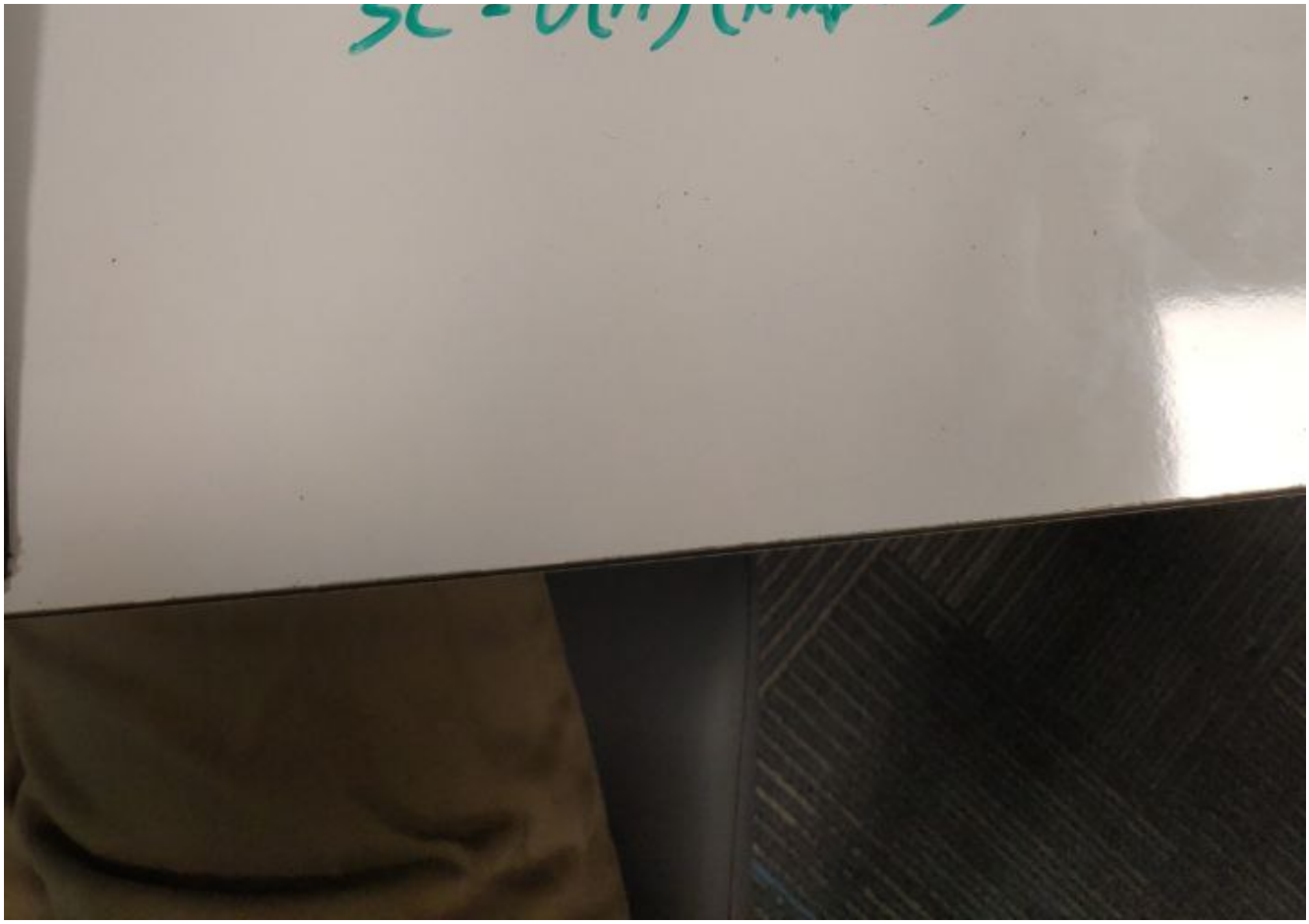
Solutions

A naieve solution using a standard merge sort would first go down the list, then back up as it rebuilds, meaning it would be of time complexity n^2, we need an algorithim that counts the inversions in the merge step, then sum the merges in the left, right, and merge() functions
Algorithim:

- Divide the array into two equal halves (within 1)
- Count the number of inversions when the two halves of the array are merged
    - Create two indices i and j
        - i is the index for the first half

        - j is the index for the second half
    - if a[i] > a[j]
        - there are mid - i inversions, since the respective sublists are sorted
        - all the remaining elements from a[i+1],a[i+2]...a[mid] will be greater than a[j]
    - Divide thse arrays into halves reccursively till theres only one element per given half
        - Sum the number of inversions in the first half, second half, and merging function inversions
- Return **Sum**
  Time Complexity:
- The Alg uses divide and conquor, in each level, one full array traversal is needed (n)
- the number of new levels is cut in half, meaning there are (log n) levels
- The Total Time complexity is $nlog(n)$





Problem 3: Best Subset Problem (40%)

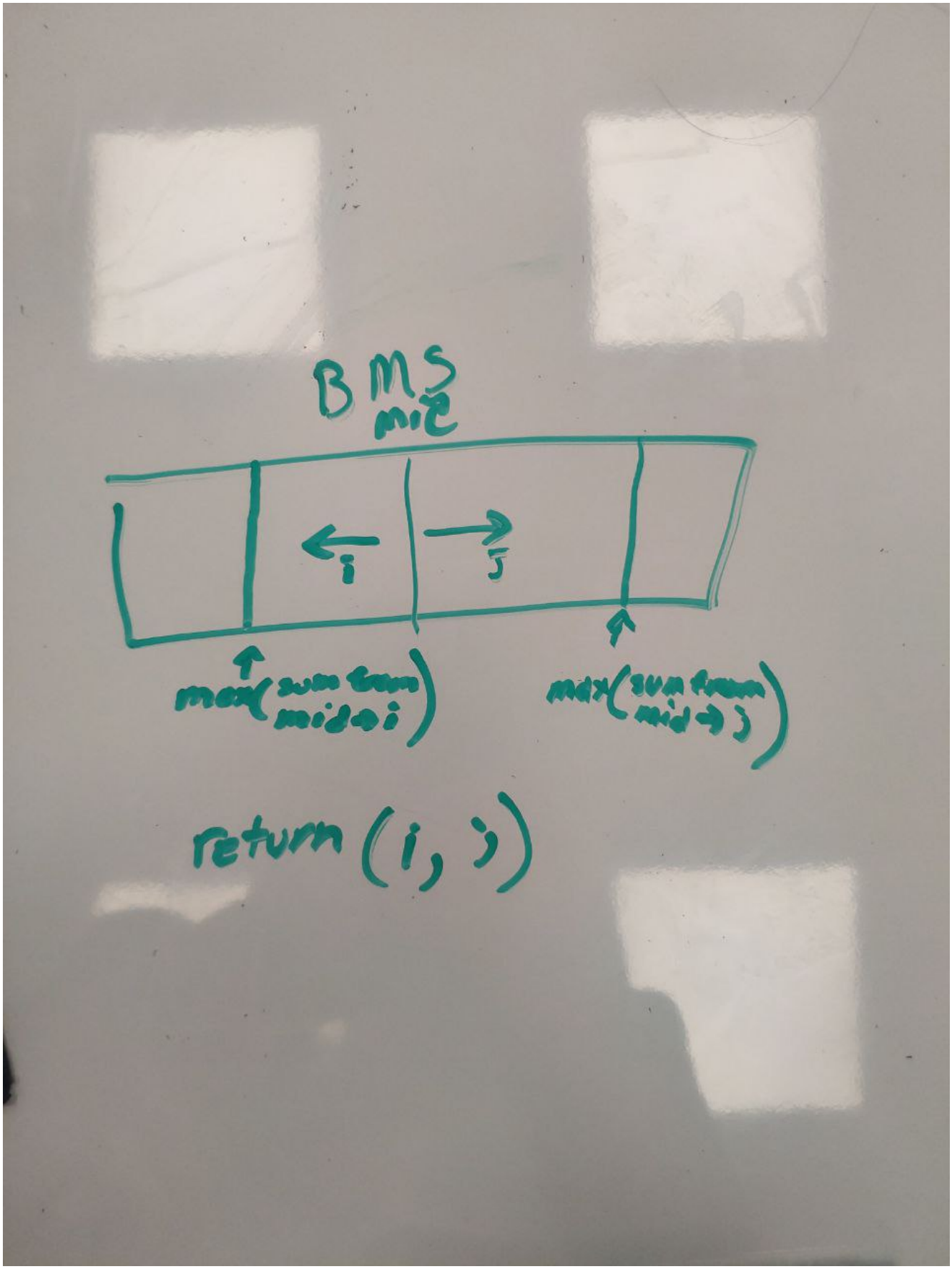
The best subset problem is defined as, given a list (x1, x2, . . . , xn) of integers (which can be positive, negative, or zero), find (i, j) such that xi + xi+1 + · · · + xj
is maximum for any 1 ≤ i ≤ j ≤ n. For example, if n = 10 and the input is (4, −8, −5, 8, −4, 3, 6, −3, 2, −11) then the output is x4 + x5 + x6 + x7 = 8 − 4 + 3 + 6 = 13.

1. Develop an O(n) algorithm for the related problem, best subset middle or BSM. The input to BSM is a list (x1, x2, . . . , xn) of integers (which can be positive, negative, or zero) and the output is the maximum value of xi + xi+1 + · · · + xj such that [i,j] spans n
2. in other words, for all possibilities for i and j such that 1 ≤ i ≤ n, 2 ≤ j ≤ n.
3. Design a recursive algorithm for the best subset problem with runtime O(n log n) that uses the BSM function.
4. Argue that your algorithm is indeed correct and prove the runtime is O(n log n).
5. (Extra credit: 5pts) Design an algorithm for the best subset problem that has O(n) runtime. Argue why your algorithm is correct and has O(n) runtime.

Solutions

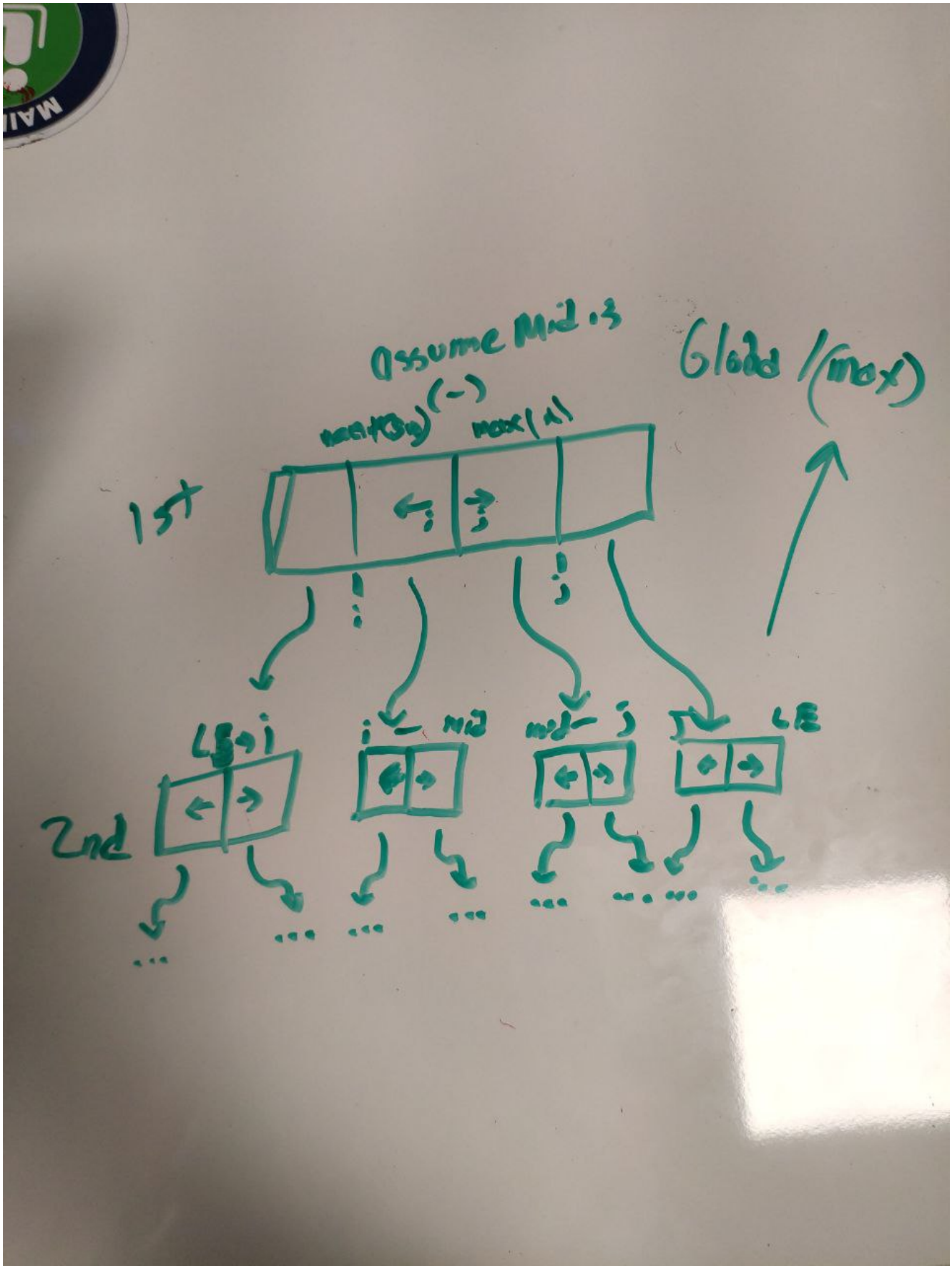
1. An algorithm for the best subset middle would be to first cut the list into two lists, one from list start (LS) to the middle (MID), and one from the Middle to list end (LE). your i variable could increment from MID to LS, and your j variable could increment from MID to LE. As these loops occur, they

should be summing the integers they come accross, and return the index at which this sum was the greatest (maxsubsequence (mid->j/i)). This means that if you take the cordinate returned by i and returned by j, you'll now have the BSM.



The worstcase runtime for this algorithm would be O(n), which occurs if the BSM is equal to the entirety of the list.

2. To design a reccursive algorith for the best sublist problem which finds the best subset using the BSM function, you would take the function and run it reccursively on the sublists formed from (LS,i), (i,MID),(MID,j), and (j,LE) splitting them in half and searching for the best sublist within them. The sublists they return will be they sublist which had the greatest size on this itteration. As these sublists return the greatest sublists they've seen thus far, we compare this to a global variable whcih holds the current best sublist, updating it if the returned sublist is greater than the current sublist. This is most easy to think about by assuming the MID of the original list is a negative number, the sublists found by the second iteration of the function (assiming the rest are positive integers) will be greater than that of the original list, allowing us to find the best subset.



This solution would be O(nlog(n)) because we first traverse potentially the entirety of the list, and then we look at problems which are at most a quarter of the size of the original problem reccursively, giving us an nlog(n) runtime.

Problem 4: Longest Common Subsequence (Extra Credit: 10pts)

Recall the Vinder problem from the first problem set. This is also known as the longest common subsequence problem between two strings. Design an improved algorithm that uses only O(n log n) memory, while still using time O(n^2). Your algorithm must compute the common subsequence, not just its length. You may assume what you showed in the Vinder problem, namely, that the dynamic programming algorithm correctly fills out a table whose (i, j) entry is the longest common subsequence between the i − prefix of the first string and the j − prefix of the second.

Solutions

To solve this problem, we need to initialize the first row to 0, and then recurse through c[i-1,j-1]+1 if i and j are greater than 0. we then take the max of the sublists returned by these two algorithims, and that result should take the same time complexity, but less space complexity due to the divide and conquor.

```
Initialize first row and column of C to 0
    Calculate c[1,j] for i<=j<=n;
    Calculate c[2,j] for 1<=j<=n
    Return c[m,n]
    Complexity O(mn)

    if i = 0 or j= 0:
        return 0;
    if (i,j)>0 and x_i = y_j:
        return c[i-1,j-1]+1;
    if (i,j)>0 and x_i<>y_j:
        return max(c[i,j-1],c[i-1,j]);
```

Then you should be able to find the LCS with recursion and divide and conquor.