

Feel free to work in groups of at most 4 for these - if you have a group of more than 4, please run it by me first. If you do work in a group, please include the names of those that you worked with. **However: each student should submit a separate copy where the solutions have been written by yourself.**

1. (10 points) Recall the inversion problem we worked through in class. Given a list of n items $(e_1, \dots, e_n)$, we say (e_i, e_j) is a mega-inversion if $e_i > 3e_j$ and $i < j$. Design a $O(n \log n)$ -runtime algorithm that calculates the number of mega-inversions and convince me of its correctness.

Solution: Consider the following algorithm: we recursively divide and recurse on our input. As we merge back, we will first combine our sublists as in merge sort, and then count the mega-inversions after. The following is a pseudocode implementation:

```
def sortAndCount(list):
    n = len(list)
    if n==1:
        return 0
    else:
        (l1, c1) = sortAndCount(list[1:n/2])
        (l2, c2) = sortAndCount(list[n/2+1:n])
        (l3, c3) = mergeAndCount(l1, l2)
        return (l3, c1 + c2 + c3)

def mergeAndCount(l1, l2):
    l3 = merge(l1, l2) # This is just the merge from MergeSort
    inversions = 0
    index1 = index2 = 0
    while index1 < len(l1) and index2 < len(l2):
        if l1[index1] > 3 * l2[index2]:
            inversions += len(l1) - index1
            index2 += 1
        else:
            index1 += 1
    return (l3, inversions)
```

First, we analyze `mergeAndCount`: let n be the maximum of the lengths of `l1` and `l2`. Then we know from the analysis of `mergeSort` that the `merge` step takes $O(n)$ time. Further, the while loop advances either `index1` or `index2` by 1 every iteration, thus the number of iterations is at most $2n$. Since each iteration takes a constant time to run, the total runtime of `mergeAndCount` is $O(n)$. Letting $T(n)$ be the runtime of `sortAndCount`, we have that $T(n) = 2T\left(\frac{n}{2}\right) + O(n) = O(n \log n)$, as shown in class.

Lastly, to see why this works: we break the argument up into two parts. First, we claim that any inversions we add are indeed inversions: note that at the moment we identify `l1[index1] > 3 * l2[index2]`, we know that the remaining elements past

`index1` are all mega-inversions with `12[index2]`, since `11` has already been sorted. Next, we claim that when we advance either index, we are done with that index. When we increase `index1`, we have that `11[index1] ≤ 3 12[index2]`, and since all future indices of `12` are larger than `12[index2]`, there can be no more mega-inversions involving `11[index1]`. On the other hand, when we increase `index2`, we have `11[index1] > 3 12[index2]`, so we add in all inversions involving `11[index2]` at this moment. Hence all mega-inversions are counted and no extra are counted, showing that our algorithm is correct.

- (10 points) Imagine your buddy and you are playing a **very fun** version of 20 questions: your bud has two separate lists (say list A and list B), each of n values written down behind their back. No value is repeated (meaning if 3 is in A then 3 is not in B). You are allowed to specify a list and ask for the k th smallest item, to which they must truthfully respond. So for example, you could ask for the 3rd smallest item of list B , etc. Determine a strategy of questions that will determine the median (we'll say, the n th smallest item) of the two lists combined in $O(\log n)$ questions.

Solution: Let's say the sorted order of A is $A = (a_1, \dots, a_n)$ and the sorted order of B is $B = (b_1, \dots, b_n)$. Let $k = \lfloor \frac{n}{2} \rfloor$. We will ask what a_k and b_{n-k+1} are (the k th smallest item in A and the $n - k + 1$ th smallest in B respectively). Otherwise, we recurse on $A' = (a_1, \dots, a_k)$ and $B' = (b_{n-k+1}, \dots, b_n)$; if $a_k < b_k$, instead we recurse on $A' = (a_{n-k+1}, \dots, a_n)$ and $B' = (b_1, \dots, b_k)$. Note that $n - (n - k + 1) + 1 = k$, so the lists remain the same size in recursion.

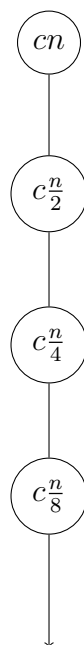
Note that if $a_k > b_{n-k+1}$, then for any $i > k$ we have that $a_i > a_k > b_k$, so $a_i > a_k, a_{k-1}, \dots, a_1$ and $a_i > b_{n-k+1}, b_{n-k}, \dots, b_1$. Thus there are $k + (n - k + 1) = n + 1$ items smaller than a_i , so a_i is bigger than the median. Similarly, for any $j < n - k + 1$ we have that $b_j < b_{n-k+1} < a_k$, so $b_j < b_{n-k+1}, b_{n-k+2}, \dots, b_n$ and $b_j < a_k, a_{k+1}, \dots, a_n$. Thus there are $(n - (n - k + 1) + 1) + (n - k + 1) = k + n - k + 1 = n + 1$ items larger than b_j , so b_j is less than the median. Note that due to the symmetry of cases, the same results apply in the case $a_k \leq b_{n-k+1}$. Thus at the moment of recursion, we remove k items less than and k items greater than the median of the combined lists. Hence, the median of the combined lists A and B is the median of the new combined lists of A' and B' .

Note that since we're cutting our input in half each round, letting $T(n)$ denote the maximum number of questions we'd ask on an input of length n we have that $T(n) = T(\frac{n}{2}) + 2$. As shown in class, this means that $T(n) = O(\log n)$.

Thus we ask $O(\log n)$ questions and correctly identify the median.

- (10 points) Suppose you are given $T(n) = T(\frac{n}{2}) + O(n)$. Using the tree based method described in class, show that $T(n) = O(n)$.

Solution: For this formula, we have $T(n) \leq T(\frac{n}{2}) + cn$ for some constant c . Our recursion tree is thus a path:



Hence, we have that the total work is at most $cn + c_{\frac{n}{2}}^n + c_{\frac{n}{4}}^n + c_{\frac{n}{8}}^n + \dots = cn(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots) = 2cn = O(n)$.

4. The input of this problem is a list of positive numbers $(v_1, \dots, v_n)$. The optimal solution is a collection of the numbers such that no two sequential numbers are taken (so if v_i is taken then v_{i-1} and v_{i+1} are off limits) but the summation of the collection is maximal.

- (a) (5 points) Consider the following ‘greedy’ algorithm: sequentially choose the greatest numbers not yet chosen that are not directly before or after any numbers already chosen. Show that this algorithm is incorrect.

Solution: Consider the instance $v_1 = v_3 = 2$ and $v_2 = 3$. Then the algorithm takes v_2 first, but cannot take any others, giving a total value of 3. The optimal solution, however, gives a total value of 4 by taking v_1 and v_3 . Thus the algorithm is incorrect.

- (b) (5 points) Consider the following algorithm: let C_1 be the collection of all even indexed numbers, and C_2 the collection of all odd indexed numbers, and select the collection which has greater total summation. Show that this algorithm is incorrect.

Solution: Consider the the instance $v_1 = v_4 = 100$ and $v_2 = v_3 = 1$. Then the algorithm either takes v_1 and v_3 or v_2 and v_4 with a total value of 101. The optimal solution, however, takes v_1 and v_4 giving a total value of 200. Thus the algorithm is incorrect.

- (c) (10 points) Design a dynamic programming solution and convince me that it correct and has polynomial runtime.

Solution: We will construct an array A such that $A(i)$ will store the maximal solution on the first i values $v_1, \dots, v_i$. Note that we have the following recursive formula:

$$A(i) = \max(A(i-1), A(i-2) + v_i),$$

where $A(1) = v_1$ and $A(0) = 0$ (since without any values the total value would be 0).

In order to compute this, initialize $A(i) = 0$ for all i and $A(1) = v_1$. For each index $i = 0$ to $n - 2$ (in that order), set $A(i + 1) = \max(A(i + 1), A(i))$ and $A(i + 2) = \max(A(i + 2), A(i) + v_i)$.

Since each index i only induces a constant amount of work, calculating A takes $O(n)$ time.

In order to determine the collection itself, we can use backtracking: starting from $i = n$ and until $i = 0$,

- i. if $A(i) = A(i - 1)$, do not include v_i and reduce i by 1
- ii. otherwise, include v_i , do not include v_{i-1} and reduce i by 2.

When $i = 0$, the set of the v_i included forms that maximal solution. Hence we can produce the collection itself in $O(n)$ time.