Problem 1.
Solve the following recurrence; make your bounds as tight as possible. Use strong induction if necessary.

\[ T(n) = T(n/2) + \log n, \quad \text{where} \quad T(2) = 1. \]

Problem 2.
Remember Problem 11 of Problem Set 1? Do not use a divide and conquer approach and do NOT assume that \( n \) is a power of two. Solve this problem again but for a \( \lceil 3n/2 \rceil - 2 \) bound.

Problem 3.
How can you implement a stack or a queue if you only have access to a TimeFunction and a PriorityQueue. That is implement \( \text{Push}(S, x) \) and \( \text{Pop}(S) \) for inserting \( x \) into a stack, and \( \text{Enqueue}(Q, x) \) and \( \text{Dequeue}(Q) \) for inserting \( x \) into a queue by using a PriorityQueue \( P \) and operations \( \text{Insert}(P, x, \text{pr}(x)) \) and \( \text{ExtractHigh}(P, \text{op}) \) where the former inserts into \( P \) an element \( x \) with priority \( \text{pr}(x) \) and the latter extracts from \( P \) the element with the highest priority: if \( \text{op} \) is \( \text{min} \) it is the lowest \( \text{pr}(x) \) valued element, if \( \text{op} \) is \( \text{max} \) is the highest \( \text{pr}(x) \) valued element.

Problem 4.
How can you realize a data structure that supports \( \text{ExtractMax} \), \( \text{ExtractMin} \) and \( \text{Insert} \) all three of them in \( O(\log n) \) time using structures already defined and used in class? Use as little space as possible.

Problem 5.
The input is an array \( A \) of \( n \) keys in no particular order. We are interested in determining whether there exists a key \( x \) in \( A \) that appears more than \( n/3 \) times in \( A \). Give a worst-case linear-time algorithm that determines whether such a key exists or not. If such a key exists, you return the key otherwise you return "NO KEY". Give the algorithm in pseudocode, explain why it works, and analyze (correctly) its worst-case running time.

Problem 6.
Which algorithm among heap-sort, radix-sort, count-sort is asymptotically faster for each one of the following sorting problems. Justify your answer. In case of a tie, the tie-breaker is an in-place algorithm.
(a) Sorting of \( n \) keys, where each key takes values in the range \( 0 \ldots n^4 - 1 \).
(b) Sorting of \( n \) keys, where each key takes values in the range \( 0 \ldots n^{2\log n} - 1 \).
(c) Sorting of \( n \) keys, where each key takes values in the range \( 0 \ldots n^n - 1 \).

Problem 7.
How can one change quicksort in such a way that its worst-case running time is \( O(n \log n) \)? Explain and justify your answer.

Problem 8.
Given a black box worst-case linear time median subroutine, give a simple, linear-time algorithm that solves the selection problem for an arbitrary order statistic.

Problem 9.
In the tree above perform (a) inorder, (b) preorder, (c) postorder traversals. Show your results.

**Problem 10.**
For \( n \) distinct numbers \( x_1, x_2, \ldots, x_n \) with positive weights \( w_1, w_2, \ldots, w_n \) such that \( \sum_{i=1}^{n} w_i = 1 \), the **weighted (lower) median** is the element \( x_k \) satisfying

\[
\sum_{x_i < x_k} w_i \leq 1/2 \\
\sum_{x_i > x_k} w_i \leq 1/2
\]

i. Argue that the median of \( x_1, x_2, \ldots, x_n \) is the weighted median of the \( x_i \) with weights \( w_i = 1/n \) for \( i = 1, \ldots, n \).

ii. Show how to compute the weighted median of \( n \) elements in \( O(n \log n) \) worst-case time.

**Problem 11.**
Give an \( O(n) \) worst case running time algorithm that, given a set \( S \) of \( n \) distinct numbers and a positive integer \( k \), \( k \leq n \), determines the \( k \) numbers in \( S \) that are closest to the median of \( S \).

**Problem 12.**
We have 4 coins all of the same weight except one that is fake and may weigh LESS or MORE than the other coins (we don’t know what the case is). We also have a balance scale; any number of coins can be put on one or the other side of the scale at any one time and the scale will tell us whether the two sides weigh the same or which side weighs more (or less). Can you find the fake coin with only 2 weighings? Explain (and justify your answer).

**Problem 13.**
The following two methods are proposed to find \( M_k \), the \( k \)-th smallest of \( n \) keys stored in array \( A[1..n] \).

a) Sort \( A \) and then output the \( k \)-th element of the sorted \( A \). This is \( M_k \).

b) Build a MIN-heap out of the \( n \) elements of \( A \), and then perform \( k \) EXTRACT-MIN operations. The last (i.e. \( k \)-th) EXTRACT-MIN operation returns \( M_k \).

Which of the two algorithms is asymptotically faster? Explain, i.e. provide details of efficient implementation of the various steps, and analyze their worst-case running time using asymptotic notation. Use case analysis, if necessary.

**Problem 14.**
(a) Analyze the binary entropy function \( f(x) = -x \log x - (1-x) \log (1-x) \), with \( 0 < x < 1 \).

Find \( \lim_{x \to 0} f(x) \), \( \lim_{x \to 1} f(x) \), \( f(1/2) \), and extrema.

Does \( F(x) = x \log x + (n-x) \log (n-x) \), \( 0 < x < n \), have a max or a min? \( F(n/4) \geq n \log n - n \)?

**Problem 15.**
We can use the results of the previous Problem in this one. Let \( a, b \) be positive constants such that \( 0 < a, b < 1 \) so that \( a + b < 1 \). Show that

\[
T(n) = T(an) + T(bn) + n
\]

has a solution that is \( T(n) = \Theta(n) \).
Solution Outline

Problem 1.

\[ T(n) = T(n/2) + \log n \] We use the iteration method, noting that \( T(2) = 1 \).

\[
T(n) = T\left(\frac{n}{2}\right) + \log n
\]

\[
= \left( T\left(\frac{n}{2^2}\right) + \left( \log \frac{n}{2} \right) \right) + \log n
\]

\[
= T\left(\frac{n}{2^2}\right) + \log \frac{n}{2^1} + \log \frac{n}{2^n}
\]

\[
= \ldots
\]

\[
= T\left(\frac{n}{2^i}\right) + \log \frac{n}{2^{i-1}} + \ldots + \log \frac{n}{2^1} + \log \frac{n}{2^n}
\]

The base case is \( T(2) = 1 \). We set \( n/2^i = 2 \), i.e. \( n = 2^{i+1} \). If we solve this for \( i \), we get \( i = \log n - 1 \). Then \( T(n/2^i) = T(2) = 1 \). In addition \( 2^i = n/2 \). Therefore the formula for \( T(n) \) becomes.

\[
T(n) = T\left(\frac{n}{2^i}\right) + \log \frac{n}{2^{i-1}} + \ldots + \log \frac{n}{2^1} + \log \frac{n}{2^n}
\]

\[
= T\left(\frac{n}{2^i}\right) + \log \frac{n}{2^{i-1}} + \ldots + \log \frac{n}{2^1} + \log \frac{n}{2^n}
\]

\[
= T(2) + (\log n - (\log 2^{k-1})) + (\log n - (\log 2^{k-2})) + \ldots + (\log n - \log 2) + (\log n - \log 2^0)
\]

\[
= 1 + (\log n - (\log n - 2)) + (\log n - (\log n - 3)) + \ldots + (\log n - 1) + (\log n)
\]

\[
= \log n + (\log n + 1)/2
\]

Therefore, \( T(n) = \log n(\log n + 1)/2 \). It is easy to verify that \( T(2) \) is indeed 1.

Problem 2.

Distinguish two cases odd \( n \) and even \( n \) i.e. \( n = 2k \) and \( n = 2k + 1 \). I’ll do the easy one. I am skipping details: pair the first two keys, then the next two and so on. As \( n = 2k \) we have \( k \) pairs. With one comparison per pair we know the min and the max per pair. Group the \( k \) mins in \( A \) and the \( k \) max in \( B \). Find the minimum of \( A \) in \( k - 1 \) comparisons and return it and the max of \( B \) in \( k - 1 \) comparisons and return it. Done (the returned values are the global min and max respectively). Total number of comparisons is \( k + (k - 1) + (k - 1) = 3k - 2 \). Since \( n = 2k \), this is also \( 3k - 2 = 3n/2 - 2 = |3n/2| - 2 \) as \( 3n/2 \) is an integer! You do the odd case!

Problem 3.

A FIFO (aka queue) means first-in first-out i.e. if you timestamp insertions the lowest time-stamped key is to be extracted first. A LIFO (aka stack) means last-in first-out i.e. the highest time-stamped key is to be extracted first. That Push(S,x) and Enqueue(Q,x) are implemented as Insert(S,x,TimeFunction) and Insert(Q,x,TimeFunction) identically. The only difference is extraction! Pop(S) and Dequeue(Q) are implemented as ExtractHigh(S,max) and ExtractHigh(Q,min) respectively. So stacks and queues are instances of a priority queue!

Problem 4.

An outline by skipping details. Use a Binary MinHeap and MaxHeap with two auxiliary arrays ToMax and ToMin. For a MinHeap[i] element, ToMax[i] shows its position (index) into the MaxHeap. Likewise for MaxHeap[i] element.

When an element is inserted it is inserted into the MinHeap first; in DownHeap when two elements are swapped say MinHeap[i] and MinHeap[j] to maintain the heap property, then make sure that the corresponding ToMin entries get swapped i.e. ToMin[ToMax[i]] and ToMin[ToMax[j]]. Then it is inserted into the MaxHeap similarly.

When Extraction is to be performed, say ExtractMax, the key is HeapExtracted from the MaxHeap first. Any DownHeap operations should update ToMax as well. When the key is erased from the MaxHeap, its ToMin[0] entry of the MinHeap must be ‘deleted’ and ‘not extracted’. Naive ‘deletion’ would create a gap: replace the gap with the last element as in an Extract operation. From that point on it works either as a DownHeap if the replaced value is smaller than the value replacing it, or upwards as an Insert operation otherwise. See example below and try to “delete” the 10 or the 25. The ExtractMin operation is analogous.
Problem 5.

\[ \text{FindKey}(A,n,n/3) \]
1. \( a_2 = \text{Select}(A,n,\text{ceiling}(n/3)) \);
2. \( a_3 = \text{Select}(A,n,2\text{ceiling}(n/3)) \);
3. \( f_2 = \text{Find-Frequency}(a_2,A,n) \);
4. \( f_3 = \text{Find-Frequency}(a_3,A,n) \);
5. if (\( f_2 > n/3 \)) return(a_2);
6. if (\( f_3 > n/3 \)) return(a_3);
7. return(NO-KEY);

\[ \text{Find-Frequency}(\text{key},A,n) \]
1. int count=0;
2. for(i=1;i<=n;i++)
3. \( \text{if} (A[i] == \text{key}) \) count++;
4. return(count);

Example.

\[
\begin{array}{c|cccccccc}
\text{n=8} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\text{Indices} & x & x & x &  &  &  &  &  \\
\text{3keys} & x & x & x &  &  &  &  &  \\
\text{Case 1} & x & x & x &  &  &  &  &  \\
\text{Case 2} & x & x & x &  &  &  &  &  \\
\text{Case 3} & x & x & x &  &  &  &  &  \\
\text{Case 4} & x & x & x &  &  &  &  &  \\
\text{Case 5} & x & x & x &  &  &  &  &  \\
\text{Case 6} & x & x & x &  &  &  &  &  \\
\end{array}
\]

Although we are not going to sort, consider the keys in \( A \) in sorted order. If a key in \( A \) appears more than \( n/3 \) times, then in the sorted sequence of these keys a block of \( n/3 \) such consecutive values or more will span the \( [n/3] \) and \( 2[n/3] \) positions. Call the keys in these positions \( a_2 \) and \( a_3 \) respectively. Thus if a keys appears more than \( n/3 \) times it will be \( a_2 \) or \( a_3 \) or possibly both. Our first action is thus to determine the \( [n/3] \)-th and \( 2[n/3] \)-th smallest keys of the (unordered) \( A \) \( (a_2, a_3) \) and it is these keys that are computed in lines 1 and 2 of FindKey. We only need to find these two keys, not sort the whole of \( A \). So after computing these two keys, in lines 3 and 4 we determine how often these two keys appear in the unsorted \( A \) by scanning the input sequence left-to-right and determining the frequency of \( a_2 \) and \( a_3 \) respectively. If one of them appear more than \( n/3 \) times such a value of \( a_2 \) or \( a_3 \) is returned through lines 5 and 6. If neither \( a_2 \) nor \( a_3 \) appears more than \( n/3 \) times, this means there is no key that appears that often. Line 7 returns NO-KEY then. The running time of lines 1 and 2 is \( O(n) \) if we ran the worst-case-linear-time selection algorithm, the running time of lines 3 and 4 is obviously \( O(n) \) (observe the code of Find-Frequency) and lines 5,6,7 take \( O(1) \) time. Overall \( O(n) \) is the asymptotic worst-case running time of FindKey.

We give an example with \( n = 8 \) and thus \( n/8 = 2 \) (i.e. determining whether a keys exists more than twice, i.e. 3 times or more). In the sorted input if such a key exists then the position of the first three occurrences (of the third or more) of the key are going to follow the pattern of one of the six cases depicted. All cases touch the \( a_2 \) and \( a_3 \) i.e. the \( [n/3] \)-th and \( 2[n/3] \)-th smallest of the \( n \) keys.

Problem 6.

(a) We use 4-digit radix-n sorting of the keys in the range from 0 to \( n^4 - 1 \). We use 4 rounds of count sort (i.e. radix-sort) for a total time of \( O(4n) = O(n) \). Radix-Sort beats Count-Sort that would require \( O(n^4) \) or heap-sort.

(b) If we use \( 2\log n \)-digit radix-n sorting of the keys we need time \( \Theta(\log n) \) rounds of count-sort for sorting keys in the radix-n range i.e. \( O(n) \) time per round for the Radix-Sort algorithm. Total is \( \Theta(n\log n) \). (One-round) Count-Sort requires \( O(n^2\log n) \) i.e. too much. Heap-Sort on the other hand also takes \( O(n\log n) \).

There is a tie between HeapSort and Radix-Sort. The tie-breaker is the in-place property of Heap-Sort. Heap-Sort wins.

(c) If we use Radix-Sort for the keys in the range from 0 to \( n^n - 1 \), we need \( n \) rounds of Count-Sort i.e. \( \Theta(n^2) \) time since we will view the keys as \( n \)-digit radix-n integers. Count-Sort itself requires too much time for initialization alone \( \Theta(n^n) \). Heap-Sort however uses only \( O(n\log n) \). The winner is Heap-Sort by a margin.
Problem 7.
Pick the pivot key to be the median of the \( n \) keys or the to be partitioned keys. Finding the median requires linear time (though with a very large constant). However it guarantees a 50-50 split. Thus the recurrence for the running time of this modified quick-sort will be: \( T(n) = 2T(n/2) + Cn \), where \( C \) is a big constant. The asymptotic solution is \( \Theta(n \log n) \), which is acceptable and asymptotically faster than \( \Theta(n^2) \) in the worst-case. Pseudocode wise we can just change line 1 of Partition slightly.

1. \( x = \text{MEDIAN}(A, l, r); \)
2. \( \text{index-of-}x = \text{index-of-MEDIAN}(A, l, r); \) //returns the index of the element in \( A[l..r] \) holding the median value \( x \).
3. \( \text{swap}(A[\text{index-of-}x], A[l]); \)
4. //Copy the remaining lines of Partition

Problem 8.
Let's call the black box function \( B(X, n) \) that if given a set \( X \) of size \( n \) returns the median of \( X \). Let the arbitrary statistic we are interested in finding be \( k \). The idea behind the solution is simple. We are going to use a binary search motivated procedure to find the \( k \) statistic. Binary search searches the middle element of a sorted set (which happens to be the median). As \( X \) in our case is not sorted we call \( B(X, n) \) to find the median. If \( k = m \), where \( m = \lfloor (n + 1)/2 \rfloor \) then we are done as the \( k \) statistic is the median which is returned by \( B(X, n) \). Otherwise we split \( X \) into two sets \( X_1 \) and \( X_2 \) around the median of \( X \) and decide which one contains the \( k \) statistic (depending on whether \( k \) is less than or greater than \( m \)). We repeat recursively the same method on either \( X_1 \) or \( X_2 \). In case \( X_2 \) is searched, we are then interested in finding not the \( k \) statistic of \( X_2 \) but the \( k - m \) statistic which is the \( k \) statistic of \( X \).

```
1. FindStatWithBinSearch(X,k,n) { // find k statistic of X . X is of size n
2.    if k = floor((n+1)/2) then return B(X,n)
3.    else
4.        if k < floor((n+1)/2) then {
5.            temp=B(X,n)
6.            collect in X1 all keys of X less than temp
7.            let m=floor((n+1)/2)
8.            FindStatWithBinSearch(X1,k,m-1)
9.        }else{
10.            temp=B(X,n)
11.            collect in X2 all keys of X greater than temp
12.            let m=floor((n+1)/2)
13.            FindStatWithBinSearch(X2,k-m,n-m)
14.        }
15. }
```

Let \( T(n) \) be the time for finding the \( k \)-th statistic of \( n \) keys using the previously defined procedure. Sets \( X_1 \) and \( X_2 \) are of about the same size and each one is of size at most \( n/2 \). Step 2 takes \( O(n) \) worst case time (median finding, by assumption on the properties of the black box) and so do Steps 5 and 10. Step 6 or 11 takes time \( O(n) \) and 7 or 12 take \( O(1) \) time. Step 8 or 13 takes time \( T(n/2) \). Note that either step 1 is executed, or steps 5-8, or steps 10-13. Therefore the worst case running time \( T(n) \) is given by the following recurrence: \( T(n) = T(n/2) + O(n) \). The solution to this recurrence is \( T(n) = O(n) \) as required (solution by any of the three known methods).

Problem 9.
(a) An inorder traversal yields the sequence: 3 5 6 10 11 12 13.
(b) A preorder traversal yields the sequence: 10 5 3 6 12 11 13.
(c) A postorder traversal yields the sequence: 3 6 5 11 13 12 10.

Problem 10.
Let the median of the \( n \) keys be \( M \). Then \( M \) is the \( \lfloor (n+1)/2 \rfloor \) smallest of the \( n \) keys. If all keys have the same weight the \( \lfloor (n+1)/2 \rfloor - 1 \) keys smaller than the median have weight less than 1/2, as \( \lfloor (n+1)/2 \rfloor - 1 \leq (n+1)/2 - 1 = (n-1)/2 \) and \( (n-1)/2 \cdot 1/n < 1/2 \).

Similarly the keys greater than the median have total weight less than or equal to 1/2 as well.

(i) The algorithm for finding \( M \) is given below.

(One easy way to prove it is to consider two cases
Case 1. \( n = 2k + 1 \) i.e. \( n \) is odd. Then the number of keys greater than \( M \) is \( k \) for a total weight of \( k/(2k+1) < 1/2 \).
Case 2. \( n = 2k \) i.e. \( n \) is even. Then the number of keys greater than \( M \) is also \( k \) for a total weight of \( k/(2k) = 1/2 \).
(ii) The algorithm for finding \( M \) is given below.
WM( A[1..n]) // A[i].key is the key, A[i].weight is the weight of key A[i].key
1. Y = MergeSort(A,n);
2. Z = empty-set; // Z will contain keys less than weighted median M
3. sum = 0; // stores weight of Z
4. for (i=1;i<=n;i++) {
   5.  Z = Z U { Y[i] }
   6.  sum = sum + Y[i].weight;
   7.  if sum < 1/2 continue;
   8.  else {
  10.  report Y[i] as weighted median
  11.  exit;
  12. }
  13. }

The time for step 1 is \( O(n \log n) \), for 2-3 is \( O(1) \), and for the loop of lines 5-13 is \( O(n) \). Briefly, we sort the keys in increasing order and if for the first \( i - 1 \) keys the total weight is \( < \frac{1}{2} \) but the weight of the first \( i \) keys is greater than or equal to \( \frac{1}{2} \), this means that the \( i \)-th sorted key is \( M \) and it is reported in line 10.

Problem 11.

Closest(A[1..n],k)
1. m= Select(A[1..n],n,floor(n/2)) // Find the median m of A[1...n]
2. for(i=1;i<=n;i++) { // Find the distance of A[i] from m
   3.  B[i].value = A[i]; // (distance is absolute value of difference)
   4.  B[i].key = |A[i]-m|; // |x| is the absolute value of x.
  5. }
6. s=Select(B[1..n].key,n,k); // select the k-th statistic use .key values for the selection
7. for(i=1;i<=n;i++) {
   8.  if B[i].key <= s // i.e. the keys with distance at most s.
      9.   print(B[i].value); // The values of these k keys is the ANSWER.
  9. }

We first find the median of the \( n \) number stored originally in \( A \). This takes \( O(n) \) worst-case running time and is performed in line 1. So far we have found the median \( m \). Next we create a new array that stores the \( n \) numbers in its value field. Attached to each such value is a key field. The key field indicates the distance between \( A[i] \) and the median. It does not matter whether \( A[i] \) is less or greater than \( m \), and thus we take absolute values to determine the distance. This creation of a new array \( B \) takes place in lines 2-4. In line 3, the \( A[] \) values are copied and the distance to the median \( m \) is computed in line 4. Total running time for lines 2-5 is also \( O(n) \).

In line 6, we select the \( k \)-th smallest element of \( B \), i.e. find the \( k \)-th statistic of \( B \) by using as keys the distance/key values previously created NOT the original numbers. This has the effect of finding the \( k \)-th number closest to \( m \). We record its distance to \( m \) in \( s \). This takes \( O(n) \) time.

In lines 7-9 we use \( s \) to find the \( k \) numbers that are close to the median \( m \) by checking in turn which of the \( n \) numbers (value fields) are at distance \( s \) or closer to \( m \). This also takes \( O(n) \) time. The overall time complexity is also \( O(n) \).

Note. The algorithm above assumes that all keys and also distances are distinct. If they are not then we can modify line 8 to output only the first \( k \) matches. We just need to add a counter in line 8. As soon as \( k \) results are output the loop is terminated.
Problem 12.

This is a decision-tree based problem. Call the coins $A,B,C,D$. If we weigh $A$ and $B$ and show it as in $A:B$ we can determine whether the fake coin is one of $C,D$ or one of $A,B$. A second weighing is required by weighing for example $A:C$.

The figure below is a binary tree, with the root on the left and extending rightwards. Note that for the weighings, we only need the equal, not-equal outcome; that one side might weigh less or more is not used in the solution.

![Binary Tree Diagram]

Problem 13.

The two algorithms are

a. $\text{Find}_M(k)_a (A[1..n], n)$
   
   1. HeapSort($A$);
   
   2. return($A[k]$);

b. $\text{Find}_M(k)_b (A[1..n], n)$
   
   1. BuildMINheap($A$, $n$);
   
   2. for($i=1; i<k; i++$)
   
   3. $z=\text{EXTRACTMIN}(A)$;
   
   4. }
   
   5. return($z$);

The two pseudocodes are given above.

Algorithm $\text{Find}_M(k)_a$ requires $O(n \lg n)$ worst-case running time for step 1 (HeapSort can be replaced by MergeSort) and step 2 requires constant time. The total is $O(n \lg n)$.

Algorithm $\text{Find}_M(k)_b$ requires $O(n)$ worst-case running time for the build-heap operation of step 1, and $O(1)$ time for step 5. The loop involves $k$ EXTRACT-MIN operations. Each one takes $O(\lg n)$ time for a total of $O(k \lg n)$. Note that the $z$ returned in step 5, is the result of the $k$-th EXTRACT-MIN operation, the $k$-th smallest of the $n$ keys of the input. Total running time is $O(n + k \lg n)$.

Let us asymptotically compare the performance of the two algorithms for various values of $k$. To make thing less confusing consider algorithm (a) has performance $n \lg n$ and (b) $n + k \lg n$.

Test Case 1. Let us start with small values of $k$. Let $k = \Theta(1)$, i.e. $k$ be constant. The running time of (a) is $n \lg n$ and (b) is $n + 1 \lg n = O(n)$. The latter algorithm is asymptotically faster.

How much can we increase $k$ beyond a constant and still get the same performance? For what values of $k$ is the term $k \lg n$ comparable to $n$? This is for $k \lg n = O(n)$, i.e. $k = O(n/ \lg n)$. For such values (b) is $\Theta(n)$ and (a) is $n \lg n$ and (b) is asymptotically faster.

Test Case 2. Suppose that $k = \omega(n/ \lg n)$. Algorithm (b) has performance $n + k \lg n = n + \omega(n/ \lg n) \lg n$ which is $\omega(n + (n/ \lg n) \lg n)$ i.e. it is $\omega(n)$.

Test Case 3. However as long as $k = o(n)$, the term $n + k \lg n$ is still $n + o(n \lg n) = o(n \lg n)$, i.e. algorithm (b) is still asymptotically faster that algorithm (a).

Case A.

Cases 1, 2, and 3 can become one case: if $k = o(n)$ then (b) is asymptotically faster than (a).

Case B. If $k$ is not $k = o(n)$, then $k = \Theta(n)$, and algorithm (b) has performance $n + k \lg n = O(n \lg n)$ and both algorithms have the same asymptotic performance.
Problem 14.
This problem reviews topics related to finding the extrema (min or max) of functions.

Consider

\[ f(x) = -x \log x - (1 - x) \log (1 - x) \]

and \( g(x) = -f(x) \) i.e.

\[ g(x) = x \log x + (1 - x) \log (1 - x). \]

If \( g(x) \) has a minimum or maximum then its negative i.e. \( f(x) \) will have the opposite (i.e. a maximum or a minimum respectively).

Remark 1. The first observation is that \( f(x), g(x) \) are symmetric around \( x = 1/2 \). Thus \( f(0) = f(1) \) if they were defined and in general \( f(x) = f(1 - x) \).

Remark 2. What is the limit \( \lim_{x \to 0} g(x) \)?

\[ \lim_{x \to 0} g(x) = \lim_{x \to 0} (x \log x + (1 - x) \log (1 - x)) = \lim_{x \to 0} (x \log x + (1 - 0) \log (1)) = \lim_{x \to 0} x \log x. \]

At this point we change variables \( y = 1/x \) or \( x = 1/y \). For \( x \to 0 \), or \( 1/y \to 0 \), we have \( y \to \infty \). Therefore

\[ \lim_{x \to 0} g(x) = \lim_{x \to 0} x \log x = \lim_{y \to \infty} (1/y) \log (1/y) = \lim_{y \to \infty} \frac{-\log y}{y} = \text{L'Hospital} \lim_{y \to \infty} \frac{-(\log y)'}{(y)'} = \lim_{y \to \infty} \frac{-1/y}{1} = 0^{-} \]

Thus the limit \( \lim_{x \to 0} g(x) = 0 \) from the left side (from the negative) and we represent this by having a minus as a superscript to the 0. It also suggests that \( \lim_{x \to 1} g(x) = 0^{-} \) as well by Remark 1.

Therefore for \( f(x) = -g(x) \) we have \( \lim_{x \to 1} f(x) = 0^{+} = \lim_{x \to 0} f(x) \).

Remark 3. What is \( g'(x) \)? Simple calculations reveal that

\[ g'(x) = \log x - \log (1 - x) \]

The first derivative is 0 for \( \log x = \log (1 - x) \) i.e. for \( x = 1/2 \). Therefore at \( x = 1/2 \) we have a minimum or maximum (most likely). To determine so we examine the second derivative

\[ g''(x) = 1/x + 1/(1 - x), \quad \text{and thus } g''(1/2) = 4 > 0. \]

Therefore \( g(x) \) has a minimum at \( x = 1/2 \) and symmetrically \( f(x) \) has a maximum. It is easy to find that \( g(1/2) = -1 \), and thus \( f(1/2) = 1 \). Remark 4. For \( F(x) \) note that \( 0 < x < 1 \). Therefore, let \( x = yn \), where \( y = x/n \) is such that \( 0 < y < 1 \). Then

\[
\begin{align*}
F(x) &= x \log x + (n - x) \log (n - x) \\
F(yn) &= yn \log (yn) + (n - yn) \log (n - yn) \\
&= yn \log y + yn \log n + n(1 - y) \log (1 - y) + n(1 - y) \log n \\
&= yn \log y + n(1 - y) \log (1 - y) + n \log n \\
H(y) &= n (y \log y + (1 - y) \log (1 - y)) + n \log n \\
\end{align*}
\]

As a function of \( y \), the expression above has a constant term \( n \log n \) and expression \( y \log y + (1 - y) \log (1 - y), 0 < y < 1 \) which is \( g(y) \). Therefore it retains a minimum at \( y = 1/2 \) i.e. \( x = yn = n/2 \).

Remark 5. As far as \( F(n/4) \) is concerned, for \( x = y/n \), i.e. \( y = 1/4 \) we have that

\[ F(n/4) = n (g(1/4)) + n \log n \]

Given that \( g(x) \) has a minimum for \( g(1/2) \) we have that \( g(1/4) \geq g(1/2) \). Therefore

\[ F(n/4) = n (g(1/4)) + n \log n \geq n g(1/2) + n \log n \geq n \log n - n. \]
Problem 15.

The \( \Omega \) part is obvious by the last term of the recurrence. We show the big-Oh part, i.e. we show that there exist \( c, n_0 \) positive and constant such that \( T(n) \leq cn \) for \( \forall n \geq n_0 \). Proof by induction. Since \( an < n \) and \( bn < n \) the induction hypothesis for \( i = an \) and \( i = bn \) gives

\[
T(n) = T(an) + T(bn) + n \\
\leq c(an) + c(bn) + n \\
= cn(a + b) + n
\]

For the last expression to be at most \( cn \) we need to have

\[
\begin{align*}
    cn(a + b) + n & \leq cn \\
    n & \leq c(1 - a - b)n \\
    1 & \leq c(1 - a - b)
\end{align*}
\]

Since \( a + b < 1 \) we have \( 1 - a - b > 0 \). Therefore if we choose \( c \) such that \( c > 1/(1 - a - b) \) the last inequality is true and the inductive step is proven.

**Question 1.** What if \( a + b = 1 \)? i.e. \( b = 1 - a \)?

Let us guess \( T(n) \leq c_2n \lg n \).

\[
T(n) = T(an) + T(1 - a)n + n \\
\leq c_2(an) \lg (an) + c_2(1 - a)n \lg (1 - a)n + n \\
= c_2(an) \lg (an) + c_2(1 - a)n \lg (1 - a)n + n \\
= c_2n \lg n + c_2(a \lg a + (1 - a) \lg (1 - a)) + n \\
\leq c_2n \lg n - c_2nA + n \\
= c_2n \lg n + (1 - c_2A)n \\
\leq c_2n \lg n \quad \text{for } c_2 > 1/A.
\]

where we used that the entropy function i.e. the expression \( a \lg a + (1 - a) \lg (1 - a) \) is negative for \( 0 < a < 1 \). Let that negative value be called \( -A = a \lg a + (1 - a) \lg (1 - a) \), where \( A > 0 \). We obtain the final answer (i.e. inductive step) by making sure that the term \((1 - c_2A)n\) is negative which is so for \( 1 - c_2A < 0 \) i.e. \( c_2 > 1/A \). Note that \( A \) is constant because \( a \) is constant. If \( a \) were not a constant, then \( A \) would not have been a constant either, and \( c_2 \) would not have been bounded by a constant either.

**Question 2 : Subproblem:** What if \( a + b > 1 \)?

Solve \( T(n) = T(n/2) + T(2n/3) + n \).

**Solution.**

We first observe that \( T(n) \) is such that \( T(n) \geq n \).

**NOTE 1.** \( T(n) = O(n) \) does not work.

We note that the guess and check (substitution method) does not work for \( T(n) \leq cn \).

**NOTE 2.** \( T(n) = O(n^2) \) works but it’s not asymptotically tight

A second choice, \( T(n) \leq cn^2 \) would work, however, as by the inductive hypothesis \( T(n/2) \leq c(n/2)^2 \) and \( T(2n/3) \leq c(2n/3)^2 \) and

\[
T(n) \leq cn^2/4 + 4cn^2/9 + n = c(25/36)n^2 + n \leq cn^2
\]

for large enough \( n \) and \( c > 0 \). It is not, however, the best possible one.

**NOTE 3.** Does there exist something better?

**Approximation 1.**

This leads us to try a solution \( T(n) \leq cn^\alpha \) where \( \alpha \) is a constant such that \( 1 < \alpha < 2 \). Similarly as before,

\[
T(n) \leq cn^\alpha((1/2)^\alpha + (2/3)^\alpha) + n \leq cn^\alpha
\]

The latter inequality holds asymptotically for large \( n \) provided that \( (1/2)^\alpha + (2/3)^\alpha < 1 \). If this value of \( (1/2)^\alpha + (2/3)^\alpha \) is 0.99 then 0.99cn^\alpha + n is indeed smaller than \( cn^\alpha \) asymptotically for large \( n \).
Approximation 2.

A more accurate solution is to try \( T(n) \leq cn^\alpha - \beta n \) (for some \( \beta \) a constant). A similar lower bound can also be proved.

For the choice \( T(n) \leq cn^\alpha - 6n \), where \( \alpha \) is the solution of \( (1/2)^\alpha + (2/3)^\alpha = 1 \), the recurrence is satisfied as by the inductive hypothesis \( T(n/2) \leq c(n/2)^\alpha - 6(n/2) \) and \( T(2n/3) \leq c(2n/3)^\alpha - 6(2n/3) \) and by using the recurrence and the inductive hypothesis for \( T(n/2) \) and \( T(2n/3) \) we get that

\[
T(n) \leq cn^\alpha((1/2)^\alpha + (2/3)^\alpha) - 3n - 4n + n \leq cn^\alpha - 6n
\]

An \( \alpha \) that satisfies \( (1/2)^\alpha + (2/3)^\alpha = 1 \) can only be numerically computed; it is \( 1.293 < \alpha < 1.294 \) (\( \alpha \) can be found numerically, but the exact value is of no particular interest; what matters, is that it is a constant between 1 and 2).

After we proving a similar lower bound, we conclude that \( T(n) = \Theta(n^{O(1)}) \), where the exponent of \( n \) is some constant between 1 and 2 that can be found as shown above.

Note: If you wonder why how I guessed the \(-6n\) term and how things worked out so nicely afterwards the answer is simple. I did not guess right; I used \( \beta \) instead and through calculations I found the best choice for \( \beta \) was indeed 6!