ESO207A: Data Structures and Algorithms

Homework *1* **Due: August 23, 2017, end of class.**

Problem 1. Recurrence Relations. Assume $T(1) = 1$ as the base case for all the recurrences below.

1.1 Solve the following recurrence relations by unfolding the recursion tree. Assume n to be an appropriate power of 2.

a. $T(n) = 4T(n/2) + n \log n$.

Assume $\log n$ to be $\log_2 n$. Unrolling the recursion tree, we get,

$$\begin{aligned}
T(n) &= n \log n + 4T(n/2) \\
&= n \log n + 4(n/2)(\log(n/2)) + 4^2 T(n/2^2) \\
&= n \log n + 4(n/2) \log(n/2) + 4^2 (n/2^2) \log(n/2^2) + 4^3 T(n/2^3) \\
&\cdots \\
&= n \log n + 4(n/2) \log(n/2) + \ldots + 4^{\log n - 2}(n/2^{\log n - 2}) \log(n/2^{\log n - 2}) + 4^{\log n - 1} T(2)
\end{aligned}$$

Let $T(2) = c_2$ for some constant $c_2 > 0$. So the above sum is

$$T(n) = n \log n + 2n \log(n/2) + 2^2 n \log(n/2^2) + \ldots + 2^{\log n - 2} n \log(n/2^{\log n - 2}) + (n^2/4)c_2$$

Consider the sum

$$U_1 = n \log n + 2n \log(n/2) + 2^2 n \log(n/2^2) + \ldots + 2^{\log n - 2} n \log(n/2^{\log n - 2})$$

The ratio ρ_k of the kth term to the $k-1$th term in the series for U_1 is, for $k = 2, 3, \ldots, \log n - 1$

$$\rho_k = \frac{2^{k-1} n \log(n/2^{k-1})}{2^{k-2} n \log(n/2^{k-2})} = \frac{2 \log(n/2^{k-1})}{\log(n/2^{k-2})} = 2\left(1 - \frac{1}{\log(n/2^{k-2})}\right)$$

ρ_k decreases as a function of k. Setting $k = \log n - 1$, we have, $\log(n/2^{k-2}) = \log(n/2^{\log n - 3}) = 3$. Therefore, $\rho_k \geq \rho_{\log n - 1} = 4/3 = \rho$ (say). Writing the sum U_1 backwards, we have,

$$\begin{aligned}
U_1 &= 2^{\log n - 2} n \log(n/2^{\log n - 2}) + 2^{\log n - 3} n \log(n/2^{\log n - 3}) + \ldots + 2n \log(n/2) + n \log n \\
&\leq 2^{\log n - 2} n \log(n/2^{\log n - 2})(1 + \rho^{-1} + \rho^{-2} + \ldots + \rho^{-(\log n - 1)}) \\
&\leq 2^{\log n - 2} n \log(n/2^{\log n - 2}) \Theta(1) \\
&= (n/4) n (2) \Theta(1) \\
&= \Theta(n^2)
\end{aligned}$$

Further, $U_1 \geq 2^{\log n - 2} n \log(n/2^{\log n - 2}) = n^2/8 = \Omega(n^2)$. Hence, $U_1 = \Theta(n^2)$. Therefore,

$$T(n) = U_1 + (n^2/4)c_2 = \Theta(n^2) \ .$$

b. $T(n) = 4T(n/2) + n^2$.

$$\begin{aligned}
T(n) &= n^2 + 4T(n/2) \\
&= n^2 + 4(n/2)^2 + 4^2T(n/2^2) \\
&\cdots \\
&= n^2 + 4(n/2)^2 + 4^2(n/2^2)^2 + \ldots + 4^{\log n-1}(n/2^{\log n-1})^2 + 4^{\log n}T(1) \\
&= n^2 + n^2 + n^2 + \ldots + n^2 + n^2T(1) \\
&= n^2 \log n + n^2T(1) \\
&= \Theta(n^2 \log n)
\end{aligned}$$

since $T(1)$ is a constant.

c. $T(n) = 2T(n/4) + \sqrt{n} \log n$.

Assume n be a power of 4. Here, $\log n$ denotes $\log_2 n$.

$$\begin{aligned}
T(n) &= \sqrt{n} \log(n) + 2T(n/4) \\
&= \sqrt{n} \log n + 2\sqrt{n/4} \log(n/4) + 2^2T(n/4^2) \\
&= \sqrt{n} \log(n) + 2\sqrt{n/4} \log(n/4) + 2^2\sqrt{n/4^2} \log(n/4^2) + \\
&\qquad \ldots + 2^{\log_4 n-1}\sqrt{n/4^{\log_4 n-1}}(\log_4 n - (\log_4 n - 1)) + 2^{\log_4 n}T(1) \\
&= \sqrt{n} \log(n) + \sqrt{n}(\log n - 2) + \sqrt{n}(\log n - 4) + \ldots + \sqrt{n}(\log n - (\log_n -2)) + \sqrt{n}T(1) \\
&= \sqrt{n}(2 + 4 + 6 + \ldots + \log n - 2) + \sqrt{n}T(1) \\
&= 2\sqrt{n}(1 + 2 + \ldots + (\log n/2) - 1) + \sqrt{n}T(1) \\
&= 2\sqrt{n}(\log(n/2))((\log(n/2)) - 1)/2 + \sqrt{n}T(1) \\
&= \sqrt{n}(\log n - 1)(\log n - 3) + \sqrt{n}T(1) \\
&= \Theta(\sqrt{n} \log^2 n)
\end{aligned}$$

1.2 Solve using a combination of the recursion tree method and the substitution method. Assume that for the base case, $T(n) = O(1)$, for $n \leq$ some constant. (*Hint:* First use the recursion tree method to obtain a good guess and then use the substitution method to prove that the guess is correct, up to $\Theta(\cdot)$.)

a. $T(n) = 7T(\lceil n/2 \rceil + 2) + n^2$

Suppose we first assume the recursion tree method, ignore the $+2$ in the recurrence and assume n to be a power of 2. Then, we get the recurrence relation $T(n) = 7T(n/2) + n^2$ whose solution by the recurrence method is

$$\begin{aligned}
T(n) &= n^2 + 7(n/2)^2 + 7^2(n/2^2)^2 + \ldots + 7^{\log n-1}(n/2^{\log n-1})^2 + 7^{\log n}T(1) \\
&= n^2 + (7/4)n^2 + (7/4)^2n^2 + \ldots + (7/4)^{\log n-1}n^2 + 7^{\log n}T(1)
\end{aligned}$$

Now, $7^{\log n} = n^{\log 7}$. So, the above sum is

$$\begin{aligned}
T(n) &= n^2 + (7/4)n^2 + \ldots + (7/4)^{\log n} n^2 \\
&= (7/4)^{\log n} n^2 \left(1 + (4/7) + (4/7)^2 + \ldots + (4/7)^{\log n}\right) \\
&= (7/4)^{\log n} n^2 \Theta(1) \\
&= 7^{\log n} \Theta(1) \\
&= n^{\log 7} \Theta(1) \\
&= \Theta(n^{\log 7})
\end{aligned}$$

So, the guess we obtain from the recursion tree method is $T(n) = \Theta(n^{\log 7})$.
Let the guess for $T(n)$ be as follows (guess for upper bound):

$$T(n) \leq \sum_{k=0}^{2} a_k n^{\log 7 - k} + a_3 + dn^2, \quad \text{for } n \geq n_0$$

where, the constants, a_0, a_1, a_2 and d and n_0 are to be determined.
Assuming $T(n)$ is of the above form, we use the equation $T(n) \leq 7T(\lceil n/2 \rceil + 2) + n^2$ and substitute. Let

$$S(n) = \sum_{k=0}^{2} 7a_k(\lceil n/2 \rceil + 2)^{\log 7 - k} + a_3 + d(\lceil n/2 \rceil + 2)^2 + n^2$$

For $a_k \geq 0$, let

$$\begin{aligned}
t_k =& 7a_k(\lceil n/2 \rceil + 2)^{\log 7 - k} \\
\leq& 7a_k((n+1)/2 + 2)^{\log 7 - k} \\
=& 7a_k(n/2 + 5/2)^{\log 7 - k} \\
=& 7a_k(n/2)^{\log 7 - k}(1 + 5/(2n))^{\log 7 - k} \\
\leq& \frac{7a_k n^{\log 7 - k}}{2^{\log 7} 2^{-k}} \left(1 + \frac{b_k}{n}\right)
\end{aligned}$$

where, b_k is a positive constant arising out of degree 1 Taylor polynomial approximation of the function $(1+x)^{\log 7 - k}$ around $x = 0$, and for $|x| <$ some constant. Therefore,

$$t_k = a_k 2^k n^{\log_7 - k} + 2^k a_k b_k n^{\log 7 - k - 1}, \quad a_k \geq 0$$

For $a_k < 0$, $(\lceil n/2 \rceil + 2)^{\log 7 - k} \geq (n/2 + 2)^{\log 7 - k} = (n/2)^{\log 7 - k}(1 + 4/n)^{\log 7 - k} \geq (n/2)^{\log 7 - k} + b_k(n/2)^{\log 7 - k - 1}$. In this case too, t_k has the same form as above, except that b_k may be a different constant.
Performing the same exercise for the n^2 term, and assuming $d \geq 0$,

$$d(\lceil n/2 \rceil + 2)^2 + n^2 \leq d(n+5)/2)^2 + n^2 \leq d(2n/3)^2 + n^2 = n^2(4d/9 + 1)$$

assuming for large enough $n \geq \Omega(1)$, that $(n+5)/2 < (2n/3)$. In particular, $d(\lceil n/2 \rceil + 2)^2 + n^2 \leq dn^2$ if $n^2(4d/9 + 1) \leq dn^2$, which is satisfied if $d \geq 9/5$.

Substituting in the expression for $S(n)$, and assuming $d = 9/5$, we have,

$$\begin{aligned} S(n) &\leq \sum_{k=0}^{2} \left(a_k 2^k n^{\log 7-k} + 2^k a_k b_k n^{\log 7-k-1} \right) + a_3 + d(\lceil n/2 \rceil + 2)^2 \\ &= a_0 n^{\log 7} + n^{\log 7-1}(a_0 b_0 + 2a_1) + n^{\log 7-2}(2a_1 b_1 + 4a_2) \\ &\quad + a_3 + 4a_2 b_2 n^{\log 7-3} + a_3 + dn^2(4d/9 + 1) \end{aligned}$$

We have to show that $S(n) \leq \sum_{k=0}^{2} a_k n^{\log 7-k} + dn^2 + a_3$. We will try to choose the constants a_k's so that the $a_k \geq$ the coefficient of $n^{\log 7-k}$, for each $k = 0, 1, 2$ and the constant term. We have already seen that with $d \geq 9/5$, the coefficient of n^2 in $S(n)$ is less than dn^2. Comparing the coefficients of $n^{\log 7-k}$ in $S(n)$ and the guessed form of $T(n)$, we have,

$$\begin{aligned} a_0 &\leq a_0 \\ a_0 b_0 + 2a_1 &\leq a_1 \\ 2a_1 b_1 + 4a_2 &\leq a_2 \\ a_3 + 4a_2 b_2 n^{\log 7-3} &\leq a_3 \end{aligned}$$

The first constraint is obvious for any a_0. We choose $a_0 = 1$. The second constraint is satisfied if $a_1 \leq -a_0 b_0$. For a fixed a_1, the third constraint is satisfied if $a_2 \leq -2a_1 b_1/3$. Choose a_2 to be negative, (and since all the b_k's are positive), the last constraint is obviously satisfied. Thus, there exist constants a_0, a_1, a_2, a_3 and d such that $T(n) \leq \sum_{k=0}^{2} a_k n^{\log 7-k} + a_3 + dn^2$, for $n \geq n_0$. Hence, $T(n) \leq O(n^{\log 7})$.

For the lower bound the analysis is analogous. Let $T(n) \geq \sum_{k=0}^{2} a_k n^{\log 7-k} + a_3 + dn^2$, for $n \geq n_0$. Let $t_k = 7a_k(\lceil n/2 \rceil + 2)^{\log 7-k}$. For $a_k > 0$, and using $\lceil n/2 \rceil + 2 \geq n/2 + 2$, we have, $t_k \geq 7a_k(n/2)^{\log 7-k}(1 + b_k/n)$, for some constant $b_k > 0$ (and possibly different from the earlier b_k). So $t_k \geq 2^k a_k n^{\log 7-k} + 2^k a_k b_k n^{\log 7-k-1}$.

Now $d(\lceil n/2 \rceil + 2)^2 + n^2 \geq d(n/2)^2 + n^2 = n^2(d/4 + 1) \geq dn^2$, provided, $d \leq 4/3$. So let $d = 1$. We get the same expression for $S(n)$. The constraints obtained by comparing the coefficients of $n^{\log 7-k}$, for $k = 0, 1, 2$ are the same as before, with inequality reversed, (but the b_k's are different)

$$\begin{aligned} a_0 &\geq a_0 \\ a_0 b_0 + 2a_1 &\geq a_1 \\ 2a_1 b_1 + 4a_2 &\geq a_2 \\ a_3 + 4a_2 b_2 n^{\log 7-3} &\geq a_3 \end{aligned}$$

So let $a_0 = 1$, $a_1 = -a_0 b_0$ satisfying second constraint, $a_2 = -(2/3)a_1 b_1$, thus satisfying the third constraint. Now $a_2 > 0$ and since $b_2 > 0$, the fourth constraint is also satisfied. Thus we have show that there exist constants a_k, such that $T(n) \geq \sum_{k=0}^{2} a_k n^{\log 7-k} + a_3 + n^2$, for $n \geq n_0$. This gives $T(n) = \Omega(n^{\log 7})$.

Together, we have $T(n) = \Theta(n^{\log 7})$.

Problem 2. Hadamard Matrices The Hadamard matrices $H_0, H_1, H_2, \ldots$ are defined as follows.

1. H_0 is the 1×1 matrix [1].

2. For $k \geq 1$, H_k is the $2^k \times 2^k$ matrix

$$H_k = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix}$$

Show that if v is a column vector of length $n = 2^k$, then the matrix-vector product $H_k v$ can be calculated in $O(n \log n)$ operations. Assume that the numbers in v are small enough so that basic arithmetic operations like addition and multiplication take unit time.

(*P.S.* Another interesting property of the Hadamard matrices is that it is (real) orthonormal, that is, $H_k^T H_k = I$. As shown in the class, the DFT matrix is also unitary (complex orthonormal).)

Let $n = 2^k$ and v be an n-dimensional vector. Write $v = [v_1, v_2]^T$ Then,

$$H_k v = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{k-1}v_1 + H_{k-1}v_2 \\ H_{k-1}v_1 - H_{k-1}v_2 \end{bmatrix}$$

Suppose we compute $s_{k-1} = H_{k-1}v_1$ and $t_{k-1} = H_{k-1}v_2$. Then, in this notation, the re-use of these computed values is clearer:

$$H_k v = (1/\sqrt{2}) \begin{bmatrix} s_{k-1} + t_{k-1} \\ s_{k-1} - t_{k-1} \end{bmatrix}$$

The time taken to compute s_{k-1} and t_{k-1} is each $T(n/2)$ time. The time taken to compute $(1/\sqrt{2}(s_{k-1}+t_{k-1})$ or $(1/\sqrt{2})(s_{k-1}-t_{k-1})$ is $O(n)$ time (coordinate wise addition of $n/2$-dimensional vectors). Hence, we obtain the recurrence relation, $T(n) = 2T(n/2) + O(n)$, whose solution is $T(n) = n \log n$.

This gives the following algorithm, with complexity $T(n) = O(n \log n)$.

HadamardVectorProduct(k, Vector v)
// computes the product $H_k v$
// Assumes v is 2^k-dimensional vector.
1. **if** $k == 0$
2. **return** v
3. **else**
4. **let** $v = [v_1, v_2]^T$, // where, each v_1, v_2 are 2^{k-1} dimensional.
5. $s =$ **HadamardVectorProduct**$(k-1, v_1)$
6. $t =$ **HadamardVectorProduct**$(k-1, v_2)$
7. $u_1 = (1/\sqrt{2})(s+t)$ // Vector Addition of dim 2^{k-1}
8. $u_2 = (1/\sqrt{2})(s-t)$ // Vector Addition of dim 2^{k-1}
9. $u = [u_1, u_2]^T$
10. return u

Problem 3. Counting inversions–the Kendall Tau distance. A permutation is an array of n integers where each of the integers between 1 and n appears exactly once. The Kendall tau distance between the two permutations is the number of pairs that are in different order in the two permutations (inversions). For example, the Kendall tau distance between 0 3 1 6 5 2 4 and 1 0 3 6 4 2 5 is 5 because the pairs $(0,1), (3,1), (2,4), (2,5)$ and $(5,4)$ are in different relative order in the two rankings but all other pairs are in the same relative order.

1. Design an efficient $O(n \log n)$ time algorithm for computing the Kendall tau distance between a given permutation and the identity permutation (that is, 0 1 $\ldots n-1$). (***Hint:*** Augment Merge-sort with inversion counting calculations.)

2. Extend the above algorithm to give an $O(n \log n)$ time algorithm for computing the Kendall tau distance between two permutations.

We will revise the notation to let the identity permutation be 1 2 3 $\ldots$, n. The idea of the algorithm is as follows. Let n be a power of 2 for convenience. Let $A[1, \ldots, n]$ be the given permutation of $1, 2, \ldots, n$. A pair of indices (i, j) with $1 \leq i < j \leq n$ is said to be an inversion if $A[i] > A[j]$. Then the Kendall tau distance of a permutation from the identity permutation is the number of inversions.

Following a divide and conquer strategy, let τ_1 be the number of inversions in the first half of A, that is, $1 \leq i < j \leq n/2$ and $A[i] > A[j]$. Similarly, let τ_2 be the number of inversions (i, j) in the second half of A, that is, $n/2 + 1 \leq i < j \leq n$ and $A[i] > A[j]$. Suppose we recursively compute τ_1 and τ_2. What remains to be computed are the number of all cross inversion pairs, that is, inversion pairs of the form (i, j) where i is in the first half and j is in the second half and $A[i] > A[j]$. Note that if we permute the elements of the first half among themselves and permute the elements of the second half among itself, then, the number of cross inversion pairs do not change.

With this property, suppose we now sort the first half and sort the second half. More precisely, let us design a recursive procedure that overloads the Merge-Sort routine and returns the number of inversions. Dividing the array into two halves, the recursive calls return the number of inversions of the first half and the number of inversions in the second half respectively. We are left with the problem of counting the number of cross inversions, assuming that the first half is sorted and the second half is sorted. Let $a_1, \ldots, a_m$ and $b_1, \ldots, b_n$ be the two sorted halves respectively. As we do merge-sort + inversion counting, suppose after the ith iteration of the standard merge-loop, we have the suffix $a_k, a_{k+1}, \ldots, a_m$ of the first half and the suffix $a_{l+m}, a_{m+l+1}, \ldots, a_{m+n}$ of the second half (where, $a_1, \ldots, a_{k-1}, a_{m+1}, \ldots, a_{l-1}$ have been merged in sorted order). As we compare a_k with a_{l+m}, there are two possibilities.

1. $a_k < a_{l+m}$. We have no evidence of an inversion in which a_{l+m} has participated yet, and we place a_k in its correct position in the sorted order, and leave the inversion count unchanged.

2. $a_{l+m} < a_k$. This is evidence of an inversion where a_{l+m} is involved, that is, $(k, l+m)$ is an inversion. Further, the index $(k+1, m+l)$ is also an inversion since, $a_{m+l} < a_k < a_{k+1}$ (permutation: hence no repetitions). And so on, $(k+2, m+l), \ldots, (m, m+1)$ is also an inversion. Thus, the element a_{m+l} is involved in $m - k + 1$ inversions. We keep a count of this, along with the merge operation. The merge + count inversions operation will insert a_{m+l} into its rightful position in the sorted order and add to a running count, the number $m - k + 1$ of inversions in which a_{m+l} participated. After a_{m+l} has been inserted into the sorted order, a_{m+l} will not contribute towards any inversions, that is, all cross-inversions in which it participated in are accounted for.

The algorithm is given below.

MERGE_COUNT_CROSS_INVERSIONS(A, p, q, r)

//input: assumes $A[p, \ldots, q]$ is sorted and $A[q+1, \ldots, r]$ is sorted
// returns $A[p, \ldots, r]$ in sorted order and also returns the cross inversion count
1. $n_1 = q - p + 1$
2. $n_2 = r - q$
3. Let $L[1, \ldots, n_1 + 1]$ and $R[1, \ldots, n_2 + 1]$ be new arrays
4. Copy $A[p, \ldots, q]$ into $L[1, \ldots, n_1]$
5. $L[n_1 + 1] = \infty$
6. Copy $A[q+1, \ldots, r]$ into $R[1, \ldots, n_2]$
7. $R[n_2 + 1] = \infty$
8. $i = 1, j = 1$
9. $inv\_count = 0$
10. **for** $k = p$ **to** r {
11. **if** $L[i] < R[j]$
12. $A[k] = L[i]$
13. $i = i + 1$
14. **else**// $L[i] > R[j]$
15. $A[k] = R[j]$
16. $j = j + 1$
17. $inv\_count = inv\_count + (n_1 - i + 1)$
18. }
19. **return** *inv_count*

The recursive procedure is given below.

MergeSort_Count_Inversions(A, p, r)
// Sort the array $A[p, \ldots, r]$ and also count the number of inversions in the segment $A[p, \ldots, r]$.
// Assumes $p \leq r$.
1. **if** $p == r$ return 0
2. **else**
3. $q = \lfloor (p + r)/2 \rfloor$
4. $lcount$ = MergeSort_Count_Inversions(A, p, q)
5. $rcount$ = MergeSort_Count_Inversions $(A, q + 1, r)$
6. $cross\_count$ = Merge_Count_Cross_Inversions (A, p, q, r)
7. **return** *lcount* + *rcount* + *cross_count*

For the second part of the question, we will reduce the problem to the first part, that is, reduce the problem of calculating the Kendall tau (KT) distance between permutations to that of counting inversions of a permutation.

We are given two arrays $A[1, \ldots, n]$ and $B[1, \ldots, n]$, each being a permutation of $1, 2, \ldots, n$. Let $B'[j] = j'$, for $j = 1, 2, \ldots, n$, that is, we rename $B[1]$ as $1'$, $B[2]$ as $2'$, $\ldots$, $B[n]$ as n'. Also let $1' < 2' < 3' < \ldots < n'$, as expected. Rename $A[1, \ldots, n]$ consistently, that is, $A'[j] = k'$ iff $A[j] = B[k]$. We wish to argue that any renaming will not change the Kendall tau distance. Why? Consider any pair $1 \leq u < v \leq n$. Suppose $A[i_1] = u$ and $A[i_2] = v$. Similarly, let $B[j_1] = u$ and $B[j_2] = v$. With the renaming, u is renamed as j'_1 and v is renamed as j'_2. So, $A'[i_1] = j'_1$ and $A'[i_2] = j'_2$.

Suppose $u < v$. The pair (u, v) contributes 1 to the KT distance of A and B iff ($i_1 < i_2$ and $j_1 > j_2$) or ($i_1 > i_2$ and $j_1 < j_2$). If $j_1 > j_2$, then, $B'[j_1] = j'_1 > j'_2 = B'[j_2]$. Then, $A'[i_1] = (A[i_1])' = j'_1$ and similarly, $A'[i_2] = j'_2$. But $i_1 < i_2$ and $j_1 > j_2$. So in the arrays A' and B', the pair (j'_1, j'_2) corresponds to (u, v) and is an inversion iff (u, v) caused an inversion with respect to A and B.

So, we can now design an algorithm. First, map B to the identity permutation, and rename A accordingly. Now just count the number of inversions in the renamed A.

KT_Distance(A, B, n)
// Find the KT distance between two given permutations A and B, each permutations of $1, 2, \ldots, n$.
// First, create a mapping array C for B. This is the inverse permutation of B.
// This is used to efficiently compute the renaming for A
1. **for** $k = 1$ **to** n
2. $\quad C[B[k]] = k$
// Now compute the renaming of A to A' assuming that B is renamed to identity.
3. **for** $k = 1$ **to** n
4. $\quad A'[k] = C[A[k]]$
5. **return** MergeSort_Count_Inversions($A', 1, n$)

Problem 4. FFT Given two n-dimensional vectors $a = (a_0, a_1, \ldots, a_{n-1})$ and $b = (b_0, b_1, \ldots, b_{n-1})$, a wrap-around convolution is defined as an n-dimensional vector $c = (c_0, c_1, \ldots, c_{n-1})$ whose coordinates are defined as follows.

$$c_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_k b_0 + a_{k+1} b_{n-1} + a_{k+2} b_{n-2} + \ldots + a_{n-1} b_{k+1}$$

Show how to evaluate this transform in $O(n \log n)$ time by viewing it as a convolution. (*Hint:* Express $\sum_{j=0}^{k} a_j b_{k-j}$ and $\sum_{j=k+1}^{n-1} a_j b_{n+k-j}$ as convolutions and add them.)

We will write the expression of c_k as the sum of two convolutions. Let $c_k = c'_k + d_k$, where, $c_k = a_0 b_k + a_1 b_{k-1} + \ldots + a_k b_0$, and $d_k = a_{k+1} b_{n-1} + a_{k+2} b_{n-2} + \ldots + a_{n-1} b_{k+1}$. Clearly, c' is a convolution $c' = a \otimes b$. Consider, $d_k = a_{k+1} b_{n-1} + a_{k+2} b_{n-2} + \ldots + a_{n-1} b_{k+1}$. Let $b'_j = b_{n-1-j}$ and $a'_j = a_{n-1-j}$ and $d'_{n-2-k} = d_k$. Then,

$$d_{n-2-k} = a'_{n-k-2} b'_0 + a'_{n-k-3} b_1 + \ldots + a'_0 b'_{n-k-2}$$

In other words, letting $u = n - 2 - k$

$$d'_u = a'_u b'_0 + a'_{u-1} b'_1 + \ldots + a'_0 b'_u$$

Thus, $d' = a' \otimes b'$. From d', d is easily calculated. Since convolutions are computed in $O(n \log n)$ time using FFT, the vector d can be computed in time $O(n \log n)$.