Algorithms: CSE 202 — Homework 1 Solutions

Problem 1: Diameter of a tree (CLRS)

The *diameter* of a tree $T = (V, E)$ is given by

$$\max_{u,v \in V} \delta(u, v)$$

where $\delta(u, v)$ is the shortest path length between the vertices u and v. That is, the diameter of the tree is the length of the longest shortest-path between any two nodes in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

We are expecting a linear (in the number of vertices of the tree) time algorithm for this problem.

Solution: Diameter of a tree (CLRS)

First recall that a tree has a unique path between any two vertices. Therefore, the longest of all shortest paths between any two nodes must either run through the root of the tree or be entirely contained in a subtree rooted at one of the root's children. This leads to an intuitive recursive formulation. At each level of the recursion, we keep track of the longest path contained in that subtree and the longest path from the root of the subtree to any other node in the subtree. When we merge our solutions, we compare the maximum of the subtrees contained in each child's subtree with the maximum path that can be formed by combining one or two children's paths with the edges to the root node. This will output the longest path due to our observations.

Complexity: Because of the recursive formulation of this algorithm, we have only a constant amount of operation per node. Hence, this is a linear time algorithm.

In more detail, if it is a full binary tree then we have the recurrence relation $T(n) = 2T(n/2) + O(1)$, where n is the number of nodes, which give a runtime of $O(|V|)$.

In the more general case we can see that our recurrence tree is exactly the tree we are given. We visit each node once and do a constant amount of work at the node giving $O(|V|)$ total work.

Proof of Correctness: Consider the case when the tree consists of only one node. In this case, we have $l = 0$ given by our algorithm which is correct. When we have more than one nodes, we root it arbitrarily and call the *MaxShortestPaths(.)* function. Assume that this function works for a node at height i. We will show that this function will also work for a node at height $i + 1$.
Consider a node V_1 at height $i + 1$. By the assumption in the induction step, we know correctly the longest path contained in each subtree and calculate the longest leaf-to-root path in each subtree. Since by comparing the following three quantities - lengths of the longest path contained in each subtree and sum of the length of the longest leaf-to-root path in each subtree gives us the diameter of the tree rooted at V_1, we are done.
Inductively, we can deduce that this algorithm will work for any tree T.

Algorithm 1 Tree Diameter

```
Input: $T = (V, E)$
Output: Diameter of $T$
1: $(\ell_1, \ell_2) = $ MaxShortestPaths$(T)$
2: return $\ell_1$
```

Algorithm 2 MaxShortestPaths

```
Input: $T = (V, E)$
Output: Length of longest path in $T$, Length of longest path to root of $T$
if $|V| = 1$ then
    return $(0, 0)$
else
    $u \leftarrow Root(T)$
    $maxsubpath \leftarrow 0$
    $maxrootpath \leftarrow 0$
    $nextrootpath \leftarrow 0$
    for $v \in u.children$ do
        $(\ell_1, \ell_2) =$ MaxShortestPaths$(subtree(v))$
        if $\ell_1 > maxsubpath$ then
            $maxsubpath \leftarrow \ell_1$
        end if
        if $\ell_2 + w((u, v)) > maxrootpath$ then
            $nextrootpath \leftarrow maxrootpath$
            $maxrootpath \leftarrow \ell_2 + w((u, v))$
        else if $\ell_2 + w((u, v)) > nextrootpath$ then
            $nextrootpath \leftarrow \ell_2 + w((u, v))$
        end if
    end for
end if
```

Problem 2: Sorted matrix search

Given an $m \times n$ matrix in which each row and column is sorted in ascending order, design an algorithm to find an element.

If the element occurs multiple locations in the matrix, your algorithm is allowed to return the element from any location.

Solution: Sorted Matrix Search

Algorithm description:

Let $A_{i,j}$ be a 2-dimensional matrix where $0 \leq i \leq m - 1$ and $0 \leq j \leq n - 1$ such that every row and every column of A is sorted in increasing order. We search for an element x in A by following a walk in the matrix as follows:

- Let c be the element in the top right corner of the matrix.
- If $x = c$, we declare that x is found.
- If $x < c$, then we move one column to the left while remaining at the same row and search in the submatrix obtained by deleting the last column.
- If $x > c$, then we move one row down while remaining at the same column and search in the submatrix obtained by removing the first row of the matrix.
- This process is repeated until either the element is found or the matrix is an empty matrix without any rows and columns.

Proof of Correctness: If x is not in A it is clear that the algorithm is correct. If x is in A, then the correctness of the algorithm follows from this claim.

Claim 0.1. Let c be the element in the top right corner of a matrix whose rows and columns are sorted in increasing order and x be an element in the matrix. If $x < c$, then x is in the submatrix obtained by deleting the last column. If $x > c$, then s is in the submatrix obtained by removing the first row of the matrix.

We argue this claim by considering the two cases seperately. If $x < c$, then x is strictly less than every element in the last column of the matrix since the column is sorted in increasing order and c is its first element. Hence, it must be in the submatrix obtained by deleting the last column of the matrix.

Now suppose instead that $x > c$. Then x is greater than every element in the first row of the matrix since the row is sorted in increasing order and c is its last element. Hence, it must be in the submatrix obtained by deleting the first row of the matrix.

Pseudocode:

```
$A$ = given matrix
$x$ = element to be searched for
$i \leftarrow 0$
$j \leftarrow n - 1$
while ( $i \leq m - 1$ and $j \geq 0$)
    if($A_{i,j} == x$)
        return true
    else if($A_{i,j} > x$)
        $j = j - 1$
    else
        $i = i + 1$
return false
```

Complexity: In this algorithm, at every step we eliminate either a row or a column or terminate if the target is reached. So, we look at a maximum of $m + n$ elements. So the time complexity is $O(m + n)$. Since, we do not use any extra space, the space complexity is $O(1)$.

Problem 3: 132 pattern

Given a sequence of n distinct positive integers $a_1, \ldots, a_n$, a 132-pattern is a subsequence a_i, a_j, a_k such that $i < j < k$ and $a_i < a_k < a_j$. For example: the sequence $31, 24, 15, 22, 33, 4, 18, 5, 3, 26$ has several 132-patterns including 15, 33, 18 among others. Design an algorithm that takes as input a list of n numbers and checks whether there is a 132-pattern in the list.

Solution: 132 Pattern

This solution is based on the submission of Yuwei Wang.

High level description

Let $a_1, \ldots, a_n$ be a sequence of distinct integers. We will let $a_0 = \infty$ for the rest of the discussion as it simplifies our definitions without affecting the correctness.

Note: Students are advised to supply all the missing proofs in the following.

For $1 \leq i \leq n$, let $\pi(i) = \max\{j | 0 \leq j < i \text{ and } a_j > a_i\}$. In other words, $\pi(i)$ is the index of the nearest greater element to the left. For all $1 \leq i \leq n$, $\pi(i)$ is defined since $\{j | 0 \leq j < i \text{ and } a_j > a_i\}$ is not empty. Furthermore for $1 \leq i \leq n$, $0 \leq \pi(i) < i$.

For $1 \leq i \leq n$, let $\sigma(i) = j$ where j is such that $a_j = \min_{0 \leq k < i} a_k$. In other words, $\sigma(i)$ is the index of the smallest element to the left of position i. We have $0 \leq \sigma(i) < i$ for all $1 \leq i \leq n$. Note that $\sigma(i) \geq 1$ for all $i \geq 2$ since there is at least one element to the left of position i.

For $i \geq 1$, let $T(i) = (a_{\sigma(\pi(i))}, a_{\pi(i)}, a_i)$.

Fact 0.2. For $i \geq 3$ such that $\pi(i) \geq 2$, we have $1 \leq \sigma(\pi(i)) < \pi(i) < i$.

Fact 0.3. For $i \geq 3$ such that $\pi(i) \geq 2$, if $a_{\sigma(\pi(i))} < a_i$, then $T(i)$ is a 132 pattern.

For $1 \leq k < j < i$, let the triple $S = (k, j, i)$ be such that (a_k, a_j, a_i) is a 132 pattern, that is, $a_k < a_i < a_j$. We say S is canonical if

- $i \geq 3$ is the smallest index among all such triples,
- $j \geq 2$ is the largest index among all such triples with the smallest i, and
- $k = \sigma(j)$.

Fact 0.4. If the sequence has a 132 pattern, then there is a canonical triple (k, j, i) of indexes such that (a_k, a_j, a_i) is a 132 pattern.

For every position $1 \leq i \leq n$, our algorithm checks if the triple $(a_{\sigma(\pi(i))}, a_{\pi(i)}, a_i)$ forms a 132 pattern. In particular we test if $a_{\sigma(\pi(i))} < a_i$. We output true if the test succeeds for some $3 \leq i \leq n$, false otherwise.

For the computation of $\pi(i)$, the previous greater element in linear time, we refer to the solution of the "Next Greater Element" homework question. We will use it here as a black box.

Pseudocode

Algorithm 3 132Pattern

```
Input: Numbers $a_1, \ldots, a_n$
Output: true if and only if there is a 132 pattern in the input
 1: $\pi \leftarrow$ findPGE($a_1, \ldots, a_n$)
 2: $\sigma(1) \leftarrow 0$
 3: for $i = 2$ to $n$ do
 4:     $\sigma(i) \leftarrow \sigma(i-1)$ if $a_{\sigma(i-1)} < a_{i-1}$, else $i - 1$
 5: end for
 6: for $i = 3$ to $n$ do
 7:     $j \leftarrow \pi(i)$
 8:     if $j \geq 1$ and $a_{\sigma(j)} < a_i$ then
 9:         return True
10:     end if
11: end for
12: return False
```

Correctness proof

We show that the algorithm returns true if and only if there is a 132 pattern.

We first show that if the algorithm returns true, then there is a 132 pattern. Assume that the algorithm returns true during iteration i for $3 \leq i \leq n$. Consider the triple $(\sigma(\pi(i)), \pi(i), i)$. Let $j = \pi(i)$. We know $j < i$. Since the algorithm returned true during iteration i, we have $j \geq 1$ and $a_{\sigma(j)} < a_i$, which imply $j \geq 2$. Let $k = \sigma(j)$. Since $j \geq 2$, we have $1 \leq k < j$. Combining the inequalities we get $1 \leq k < j < i \leq n$. Since $a_j > a_i$ and $a_k < a_i$ (and thereby $a_k < a_j$), we conclude (k, j, i) forms a 132 pattern.

By Fact 0.4, we know that there is a canonical triple which forms a 132 a pattern if there is a 132 pattern. We show that if $T = (k, j, i)$ is a canonical triple that forms a 132 pattern (that is, $1 \leq k < j < i$ and $a_k < a_j < a_i$), the algorithm return true during iteration i. Since $a_j > a_i$ and j is the largest such index we conclude $j = \pi(i)$. We also have $k = \sigma(j) \geq 1$ since T is canonical. Therefore algorithm returns true during iteration i and would not return true in any earlier iteration since i is the smallest index for which a 132 pattern exists.

Runtime analysis

The algorithm scans the array twice with constant time per iteration. It also calls findPGE once. Since there is a $O(n)$ algorithm for findPGE we conclude that our algorithm runs in time $O(n)$.

Solution: 132 pattern

Algorithm description: Let $a_0, \ldots, a_{n-1}$ be a sequence of distinct positive integers. To find the existence of 132 pattern in the sequence, we employ the following two step algorithm.

- For $0 \leq i \leq n-1$ we compute the smallest number to the left of position i and assign it to $Min_A(i)$. We define $Min_A(0)$ to be ∞. We compute $Min_A(i)$ in linear time by scanning the array from left to right.
- For each location $1 \leq i \leq n-1$ we check if a_i is the middle element of a 132 pattern by scanning the list from right to left while maintaining a balanced binary search tree of all the elements seen so far. For each i we check if a_i is greater than the smallest number to its left, that is, if $a_i > Min_A(i)$. If so, we check if there is an element in between a_i and $Min_A(i)$ in the binary search tree. We do this as follows. We conduct the search for both elements in parallel. If at some point the searches lead to different branches of the tree, we conclude that there is an element between the two and terminate the process. Otherwise, we insert a_i in the binary search tree and continue the scan. If we reach a_0 in the process, we terminate and return False.

Pseudocode :

```
Input: List A of n positive distinct integers
Output: True if 132 pattern exists in the sequence, False otherwise
1:  procedure PatternExists(A)
2:      $Min_A \leftarrow$ empty list of size $n$
3:      $minElement \leftarrow \mathbf{A}(0)$
4:      $Min_A(0) \leftarrow \infty$
5:      for $i = 1$ to $n - 1$ do
6:          $Min_A(i) = minElement$
7:          if $A(i) < minElement$ then
8:              $minElement = A(i)$
9:          end if
10:     end for
11:     $BST \leftarrow$ empty $BST$
12:     for $j = n - 1$ to 1 do
13:         if $Min_A(j) > A(j)$ then
14:             continue
15:         end if
16:         if Search($A(j)$, $Min_A(j)$, BST.root) == True then
17:             return True
18:         else
19:             Insert $A(j)$ in BST
20:         end if
21:     end for
22:     return False
23: end procedure
24: procedure Search($a_j$, $a_i$, root)
25:     if root == NULL then
26:         return False
27:     end if
28:     if root.value $> a_i$ and root.value $< a_j$ then
29:         return True
30:     end if
31:     if root.value $> a_j$ then
32:         return Search($a_j$, $a_i$, root.left)
33:     end if
34:     if root.value $< a_j$ then
35:         return Search($a_j$, $a_i$, root.right)
36:     end if
37: end procedure
```

Proof of correctness:

Claim 0.5. For $1 \leq i \leq n-1$, at the beginning of the i-th iteration (of the loop in steps 6 through 10) *minElement* is the minimum of the elements in the subarray $A(0), \ldots, A(i-1)$.

Proof: The proof is by induction on the number of iterations, that is, on the value of i. For the base case, consider the state of the execution of the algorithm at the beginning of the first iteration of the loop. At this point, since $i = 1$, the prefix array under consideration has exactly one element, namely, the first element $A(0)$. Also, at this point we have *minElement* $= A(0)$. The value of *minElement* is indeed the minimum of the subarray $A(0)$. We have thus proved the claim for the case when $i = 1$.

Let $1 \leq x \leq n-1$ be an arbitrary integer. For the inductive step we assume that the claim is true for $1 \leq i \leq x$. Consider the state at beginning of iteration $x+1$. We are considering the subarray $A(0), \ldots, A(x)$ at this point.

We first show that *minElement* is the minimum of the subarray $A(0), \ldots, A(x)$. By the induction hypothesis, at the beginning of the iteration x, *minElement* is the minimum of the subarray $A(0), \ldots, A(x-1)$. During iteration x we compared it with $A(x)$ and updated it appropriately so we conclude that *minElement* is the minimum of the subarray $A(0), \ldots, A(x)$. Thus, at the beginning iteration $x+1$, the value of *minElement* is equal to the minimum element in the subarray $A(0), \ldots, A(x)$.

For $i < j < k$, we say that (a_i, a_j, a_k) is a *standard* 132 pattern if

- $a_i < a_k < a_j$,
- j is the largest index among all such triples,
- k is such that $\pi(k) = j$, and
- $i = \sigma(j)$.

where π and σ are defined in the previous solution.

Fact 0.6. If the array has a 132 pattern, then it has a standard 132 pattern.

Fact 0.7. Let T be a binary search tree and $\rho_1, \ldots, \rho_l$ be the sorted sequence of its keys. If two queries $q < q'$ (not in the search tree) take the same path in T, then one of the following holds:

- $\exists 1 \leq i \leq l-1$ such that $\rho_i < q < q' < \rho_{i+1}$
- $q < q' < \rho_1$
- $\rho_l < q < q'$

Claim 0.8. The algorithm returns true if there is at least one 132 pattern in the array.

Proof: Let (a_i, a_j, a_k) be a standard 132 pattern where $i < j < k$, $j = \pi(k)$ and $i = \sigma(j)$. We know that $Min_A(j) = a_i$ and $a_i < a_j$. Consider the iteration when the algorithm is processing the element a_j and assume that the algorithm did not return true prior to this point. If it did, the claim is true. Otherwise, at this point, it will search the binary search tree with the queries a_i and a_j since $Min_A(j) = a_i < a_j$. Since the binary search tree contains at least one element (that is, a_k) which is in between a_i and a_j, the two queries must necessarily disagree at some node in the binary search tree. Let $a_{k'}$ be the key at the node. Since the queries disagreed at the node, the algorithm return true.

Claim 0.9. If the algorithm returns true, there is at least one 132 pattern in the array.

Proof: If the algorithm returned true, there is some iteration during which it returned true. Let a_j be the element processed during the iteration and $i = \sigma(j)$. Since the algorithm returned during the iteration, the following statements hold:

- $Min_A(j) = a_i < a_j$, and
- during the iteration the algorithm searched the binary search tree with queries a_i and a_j and succeeded in finding a_k where $k \geq j$ and $a_i < a_k < a_j$.

This implies that there are elements a_i, a_j, and a_k in the array such that $i < j < k$ and $a_i < a_k < a_j$, which is a 132 pattern.

Complexity Analysis:

- The first step involves a single scan from left to right with constant time per element. The complexity of this step is $O(n)$.

- The second step involves a single scan from right to left with at most $O(n)$ find and insertion operations on the binary search tree. The complexity of this step is $O(n \log n)$.

Hence, the overall time complexity of the algorithm is $O(n \log n)$.

Problem 4: Next greater element

Given an array, print the Next Greater Element (NGE) for every element. The Next Greater Element of an item x is the first greater element to its right in the array.

Solution: Next greater element

Algorithm description: Let $a_1, a_2, \ldots, a_n$ be a list of n numbers. Let us assume without loss of generality that a_i is a positive integer for $1 \leq i \leq n$. If a_i does not have any greater element to its right, we define its next greater element to be -1. We compute the next greater element (NGE) for every element a_i by scanning the list from left to right and storing the elements for which we have not yet found the next greater element on the stack.

Here are the details of the algorithm. For each element a_i starting with $i = 1$ and an empty stack, we process a_i as follows until it is pushed onto the stack.

We push a_i onto the stack if the stack is empty or $a_i \leq t$ where t is the top of the stack. Otherwise, if $a_i > t$, we pop t from the stack and mark a_i as its next greater element and repeat the process of pushing a_i onto the stack.

After a_n is processed, we pop all the elements of the stack and mark -1 as their next greater element.

Pseudocode :

Algorithm 4 Next Greater Element

```
input $A[1...n]$
$S \leftarrow$ Empty Stack
for $i = 1$ to $n$ do
    while S is not empty AND $A[S.top] < A[i]$ do
        $t = S.pop()$
        $NGE(t) = A[i]$
    end while
    $S.push(i)$
end for
while S is not empty do
    $t = S.pop()$
    $NGE(t) = -1$
end while
output $NGE[1...n]$
```

Proof of Correctness: We prove the correctness of the algorithm by induction on the number of iterations of the algorithm. Each iteration exactly involves processing one element of the sequence. More precisely,

Claim 0.10. For $0 \leq i \leq n$, at the end of processing a_i, the following holds:

1. If the stack is not empty, indexes of the elements on the stack belong to the set $\{1, 2, \ldots, i\}$, are increasing and the values are decreasing as we go from the bottom of the stack to the top of the stack.

2. The elements on the stack do not have a next greater element among $a_1, \dots, a_i$.
3. If an element e is popped out of the stack during iteration i, then a_i is the next greater element for e.

We regard the beginning of iteration j as the end of iteration $j-1$ for $1 \le j \le n$. Although we only push indexes onto the stack, with a slight abuse of the notation, we use the phrase top value on the stack to refer to the value at the index given by the top element of the stack.

Proof.
Base case ($i = 0$): At the end of iteration 0 the stack is empty, so properties $1, 2$, and 3 are satisfied vacuously.
Inductive Step: Assume that the properties hold at the end of iteration j for all $0 \le j \le i < n$.

We will prove that the properties hold at the end of iteration $i+1$. Consider the state of the program at the beginning of the iteration $i+1$. There are several possibilities: the stack is empty; if not, a_{i+1} is less than the top value on the stack; or a_{i+1} is greater than the top value on the stack. In all scenarios, we prove that all the three properties will hold.

- The stack is empty: In this case, we simply push a_{i+1} onto the stack during iteration $i+1$. Property 1 holds trivially since there is only one element on the stack. Property 2 holds since the element a_{i+1} cannot have its next greater element in the list $a_1, \dots, a_{i+1}$. Property 3 holds vacuously since no element has been popped during iteration $i+1$ since the stack is empty at the beginning of the iteration.

- a_{i+1} is less than the top value on the stack: We push a_{i+1} onto the stack in this case. Since $i+1$ is the largest index of all the indices considered and a_{i+1} is smaller than the top value on the stack, it follows from induction hypothesis that Property 1 is satisfied. Again by induction hypothesis and since a_{i+1} is smaller than the top value we conclude that none of the elements on the stack have a next greater element in $a_1, \dots, a_{i+1}$. Since no elements have been popped from the stack, Property 3 holds vacuously.

- a_{i+1} is greater than the top value on the stack: Let $x_1, \dots, x_k$ be the indices of the elements on the stack with x_1 at the bottom and x_k at the top of the stack. Clearly $i+1 > x_k$ since $i+1$ is largest index we have accessed so far. We pop elements from the stack till the stack is empty or a_{i+1} is less than the top value on the stack. Hence, property 1 still holds at the end of iteration $i+1$.

  Let $a_{x_j}, \dots, a_{x_k}$ be the elements that were popped during the iteration. This implies that $a_{i+1} > a_{x_j} > \dots > a_{x_k}$. Additionally, we claim that for every element a_{x_m} where $m \in [j, k]$, there is no other element with index y such that $i+1 > y > m$ and $a_y > a_m$, since from property 2 at the end of iteration i we know that the elements on the stack do not contain their next greater element in the array $a_1, \dots, a_i$. Therefore all the popped elements have found their next greater element. Since a_{i+1} is smaller than the remaining elements on the stack, we conclude that values on the stack (including a_{i+1}) do not have their next greater element in $a_1, \dots, a_{i+1}$.

□

Terminating Step: At the end of the loop, if the stack contains any elements (with indices $x_1, x_2, \dots, x_k$) then $x_1 < x_2 < \dots < x_k$ and $a_{x_1} > a_{x_2} > \dots > a_{x_k}$. Since the properties in the claim hold at the end of iteration n, NGE does not exist for the elements on the stack in $a_1, \dots, a_n$ so we set their NGE values to -1.

Complexity Analysis: Every element in the array is pushed exactly once onto the stack and popped exactly once from the stack. Overall, there are only constant number of operations per element. Therefore, the algorithm runs in $\Theta(n)$ time.

Problem 5: Base conversion

Give an algorithm that inputs an array of n base b_1 digits representing a positive integer in base b_1 in little endian format (that is, the least significant digit is at the lowest array index) and outputs an array of base

b_2 digits representing the same integer in base b_2 (again in little endian format). Get as close as possible to linear time. Assume b_1, b_2 are fixed constants.

If we do the conversion digit-by-digit, the number of operations is really $O(n^2)$, since, for example, once we convert each digit we need to add up all the resulting $O(n)$ bit numbers. We can do better using a divide-and-conquer algorithm using a FFT integer multiplication algorithm as a sub-routine. Assume n is a power of 2; otherwise, pad with 0's in the most significant digits.

Solution: Base conversion

Convert[X[0,..n-1]: array of b_1 digits]: array of b_2 digits.

1. IF $n = 1$ output $X[0]$ in base b_2.
2. $Z[0, ..n/2] \leftarrow (b_1)^{n/2}$ in base b_1 (1 followed by n/2 0's).
3. $A \leftarrow Convert[X[0, ..n/2 - 1]]$;
4. $B \leftarrow Convert[X[n/2, ..n - 1]]$;
5. $C \leftarrow Convert[Z[0, ..n/2]]$;
6. Return $Add(Mult(A, C), B)$ (where Mult is the $O(nlogn)$ FFT multiplication algorithm from class and Add is the $O(n)$ grade-school addition algorithm).

If X has n b_1 digits, $X < (b_1)^n$, and X has at most $\lceil \log_{b_2} X \rceil < n \log_{b_2} b_1 + 1 \in O(n)$ b_1-digits. Therefore, the Mult and Add algorithms in the last line are called on $O(n)$ digit arrays, and their total time is $O(n \log n)$. Thus, the recurrence from the above is $T(n) = 3T(n/2) + O(n \log n)$. Since $n \log n \in O(n^{1+\epsilon})$ for any $\epsilon > 0$ and particularly for $\epsilon = 1/2$, we can apply the Master theorem with $a = 3, b = 2, k = 1.5$ and see that this is a bottom-heavy recursion with $T(n) = O(n^{\log_2 3})$. This is an improvement, but we can do much better.

Note that the above algorithm repeatedly computes b_1, b_1^2, b_1^4,...$b_1^{2^i}$,.. and $b_1^{n/2}$ in base b_2. Instead of computing these recursively, let's compute them once and save them for reuse. Then we can do the recursion with only two calls. To compute the powers of b_1, and store them in a two dimensional array $P[i, j], 1 \leq i \leq \log n, 1 \leq j \leq O(n)$ (where the constant is $\log_{b_2} b_1$.) $P[1, j]$ is b_1 in binary, and $P[i+1, 1..cn] = Mult(P[i, 1..n], P[i, 1..n])$. This uses $O(n \log n)$ memory, but really we could compress it down to $O(n)$ memory because the lengths of the powers are decreasing exponentially from $i = logn$ to $i = 1$. It takes at most $O(n \log n)$ time to perform each multiplication, so we get an $O(n \log^2 n)$ upper bound on the total time for to compute P. (Actually, since the lengths are increasing exponentially,t he total time is just $O(n \log n)$, but the rest of the algorithm is $O(n \log^2 n)$, so we won't analyze this precisely.)

Then once we have P as a global variable, the recursion becomes:

Convert[X[0,..n-1]: array of b_1 digits]: array of b_2 digits.

1. IF $n = 1$ output $X[0]$ in base b_2.
2. $Z[0, ..n/2] \leftarrow (b_1)^{n/2}$ in base b_1 (1 followed by n/2 0's).
3. $A \leftarrow Convert[X[0, ..n/2 - 1]]$;
4. $B \leftarrow Convert[X[n/2, ..n - 1]]$;
5. $C[0, ..n/2] \leftarrow P[\log n - 1, 0, ..n/2]$;
6. Return $Add(Mult(A, C), B)$ (where Mult is the $O(nlogn)$ FFT multiplication algorithm from class and Add is the $O(n)$ grade-school addition algorithm).

This main recursive procedure is then time: $T(n) = 2T(n/2) + O(n \log n) = 4T(n/4) + O(2n/2 \log n/2) + O(n \log n) = O(n \log n + n \log(n/2) + n \log n/4 + ...n \log 1) \leq O(n \log^2 n)$. The general master's theorem shows that this is tight. So the total time is $O(n \log^2 n)$.