Name: Yan, Zi
Course: CIS 502
Assignment: HW4

Problem 1

Assume the sorted array is in decreasing order, if not use $n-i$ to access element i. Let $s=1, e=n, mid=\lfloor(s+e)/2\rfloor$, then compare $A[mid]$ with mid^2. If $A[mid] > mid^2$, let $s=mid+1, mid=\lfloor(s+e)/2\rfloor$; if $A[mid] < mid^2$, let $n=mid-1, mid=\lfloor(s+e)/2\rfloor$; then continue to compare $A[mid]$ and mid^2 recursively. If $A[mid]=mid^2$, we just find that wanted index, but until $s>e$, we still cannot find $A[mid]=mid^2$, we can output "does not exist".

Proof. Every time we look at the number $A[mid]$ in the middle of the array, there are three possibilities:

- If $A[mid] > mid^2$, and because the array is in decreasing order, we have $A[mid-1] > A[mid] > mid^2 > (mid-1)^2$, therefore each element before $A[mid]$ will always be greater than the square of its index. Consequently, we only need to look at the right part.
- If $A[mid] < mid^2$, we have $A[mid+1] < A[mid] < mid^2 < (mid+1)^2$, then all elements after $A[mid]$ will be less than their indices respectively. Consequently, we only need to look at the left part.
- If $A[mid] = mid^2$, we just find what we need.

In all these cases, we perform one probes of the array A and reduce the problem to one of at most half the size. Thus the runtime will be $T(n)=T(n/2)+c$, namely $T(n)=O(\log n)$. □

Problem 2

The algorithm is:

1) divide the array into two groups recursively.
2) find the largest-sum subinterval in each group.
3) while merging, first we should find the largest-sum subinterval in the merged group, starting from the middle mid of the group. From middle to the left of the group, we find a index a, where the sum of interval $[a, mid]$ is the largest one in the left half, only if not negative. Similarly, we can find a index b which makes $[mid, b]$ the interval with largest non-negative sum. Finally, we will get the largest-sum interval $[a, b]$ by concatenating $[a, mid]$ and $[mid, b]$.
4) then we choose the largest sum subinterval from those three, pushing it to the upper level of merging.

Proof. The step 1 will take constant time, and the step 2 will take $T(n/2)$ time, so the crucial part is the step 3, the merging step. Once we prove that in step 3, we can get the required subinterval, then we are done.

The algorithm maintains a property that every merging will provide the largest sum of subintervals in the merged group.

Base case: at the bottom of the recurrence, each group will have only one number, therefore the only number is the largest-sum subinterval.

Inductive step: assume at level i, the merging result of each merged group is the largest-sum subinterval in it. Then we step into level $i+1$. While merging two groups g_L and g_R, we know from level i that intervals $[a_L, b_L], [a_R, b_R]$ are the largest-sum subintervals in g_L and g_R respectively. Besides these two intervals, we should also consider the intervals which cross both groups, because at level i, we only have the largest-sum interval from each of g_L and g_R, not both.

By combining the largest-sum interval ending in the last element of g_L with the one starting at the first element of g_R, we can get the largest-sum interval across two group. Suppose not. Assume the combined interval is $[a, b]$, then there must be another interval $[a', b']$ which also cross two groups has larger sum than $[a, b]$. In accordance with the algorithm, assuming the last element of g_L is x, the first element of g_R is y, the sum of $[a, x]$ is greater than that of $[a', x]$, and the sum of $[y, b]$ is greater than that of $[y, b']$, so the sum of $[a, b]$ is greater than $[a', b']$. It contradicts with the assumption. So $[a, b]$ is the interval with largest sum. And the way of finding $[a, b]$ is simply picking out the largest sum of intervals from the middle element to left end and right end, which should take $O(n)$ time.

Consequently, after choosing the largest one from $[a_L, b_L], [a_R, b_R]$ and $[a, b]$, we can get the largest-sum interval at level $i + 1$. $\square$

The runtime is $T(n) = 2T(n/2) + O(n)$, namely $T(n) = O(n \log n)$

Problem 3

We can use convolution to solve this problem.

First, we build a vector for the coefficients.

Let $w = (\frac{c}{(n-1)^2}, \frac{c}{(n-2)^2}, \ldots, \frac{c}{2^2}, \frac{c}{1^2}, 0, -\frac{c}{1^2}, -\frac{c}{2^2}, \ldots, -\frac{c}{(n-2)^2}, -\frac{c}{(n-1)^2})$ and $b = (q_n, q_{n-1}, \ldots, q_2, q_1)$. Then we can use FFT to calculate the convolution $S_j = \sum_{(k,l):k+l=2n-1+j} w_k b_l$. And $F_j = S_j \times q_j$.

The FFT way of the convolution will take $O(n \log n)$, and the multiplication will take $O(n)$. So total runtime is $O(n \log n)$.

Problem 4

First, let $P(n)$ denote the parent of node n, $C_L(n)$ denote the left child of node n, and $C_R(n)$ denote the right child of node n.

We start from the root r. If $x_{C_L(r)} > x_r$ and $x_{C_R(r)} > x_r$, then r is a local minimum. If not, we choose a child that has its value less than x_r of r to continue. For the condition that both of r's children have the less value than r, we choose either one. Then recursively repeat the check procedure same as the root one, until we come to a leaf. During this process, there will be two situations, first there is a node n satisfying $x_{C_L(n)} > x_n$ and $x_{C_R(n)} > x_n$, then n is a local minimum, second we reach a leaf n', then n' is a local minimum.

Proof. We have already proved the root. For internal nodes, once we reach a node n, we know $x_{P(n)} > x_n$, because our algorithm will only choose a child with smaller value than its parent. Therefore, once we can provide $x_{C_L(n)} > x_n$ and $x_{C_R(n)} > x_n$, the node n is a local minimum. For leafs, because the implication of our algorithm makes it have smaller value than its parent, and its parent is the only node jointed by edges to it, the leaf is a local minimum. $\square$

Because we only choose one child, and we probe three nodes, itself, its left child, and its right child, at one time, the runtime $T(n) = T(n/2) + c$, namely $T(n) = O(\log n)$