COMP3121: Assignment 1 – Q4

Gerald Huang

$z5209342$

Updated: June 15, 2020

Let A be a 2-dimensional array of size $4n \times 4n$. Our goal is to design an algorithm that returns the $n \times n$ grid that produces the largest number of apples in $O(n^2)$ time.

Begin by pre-processing A. During this stage, we calculate the sum of the first $n \times 1$ rectangles in each column and store the sum in some temporary array structure B. We can do this by first calculating the first $n \times 1$ rectangle by traversing through the first n elements in the first column which takes $O(n)$ time complexity. This creates a running sum which keeps track of the total sum of the elements that have been traversed through. In each subsequent rectangle after the first rectangle, we can do this in $O(1)$ time complexity by adding the next element in the column and subtracting the uppermost element still in the running sum. That is, we can define the following recursion

$$B[i][j] = B[i-1][j] + A[i+n-1][j] - A[i-1][j].$$

This can be illustrated by considering a 1-dimensional column array,

$$A_1 = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix}^{\mathsf{T}}.$$

We calculate the running sum by keeping track of the sum after having traversed each element. So in the first 2×1 rectangle, we have a running sum of $S_1 = A_1[1][1] + A_1[2][1] = 1 + 2 = 3$. In the next rectangle, we add the next element $A_1[3][1]$ and subtract the upper most element still in the running sum $A_1[1][1]$. This gives us a new running sum of $S_2 = S_1 + A_1[3][1] - A_1[1][1] = 3 + 3 - 1 = 5$. We continue this process until we hit the last possible 2×1 rectangle which is the $8 - (2 - 1) = 7$th rectangle.

More generally, we consider $4n - (n-1) = 4n - n + 1 = 3n + 1$ rectangles in a column. But each column runs in $O(n)$ time since every rectangle after the first rectangle's entry in B can be computed in $O(1)$ time, assuming that performing arithmetic operations take constant time. Since each column takes $O(n)$ time to compute and there are $4n$ columns to compute, then it follows that the construction of B can be computed in $O(n^2)$ time.

By our construction of B, this gives us all of the vertical $n \times 1$ strip combinations possible. Each $n \times n$ square can therefore be constructed by computing the sum of n consecutive elements of B along its **row**. We approach the calculations of the sum of each square similarly to our construction of B. In the first row, we compute the sum of the first n elements which takes $O(n)$ time. However, in each subsequent calculation, we can take $O(1)$ time by adding and subtracting certain elements. Again, we can keep a running sum that calculates the sum of all of the elements, and use this sum to add the next element on the row and subtract the leftmost element that is still in the running sum. That is, the sum of the $n \times n$ square with starting index (i, j) can be recursively defined as

$$\text{Sum} = \text{current running sum} + B[i][j+n-1] - B[i][j-1].$$

To keep track of the maximum sum, we introduce some variable `max_sum` and update it when a new sum exceeds the value stored in `max_sum`. When `max_sum`, we also update the position of the leftmost and uppermost index in the square for reference. Each comparison and update also take $O(1)$ time, by assumption. We repeat this process for the remaining $3n$ rows. Each row takes $O(n)$ time complexity and since there are $3n + 1$ rows, searching for the maximum sum will take $O(n^2)$ time.

The algorithm then returns the square that starts from index of the leftmost and uppermost square (i, j) which also corresponds to the (i, j) square in our original grid A and ends at index of the rightmost and lowermost square $(i + n - 1, j + n - 1)$.