CS381 Lecture Notes on Largest Square Block of Ones and on Maximum-Sum Contiguous Subarray

1 Finding a largest square block of 1s in a Boolean matrix

Let A be an $n \times n$ matrix of 0's and 1's. We seek an $O(n^2)$ time algorithm for finding the largest square block of A that contains 1's only. For example, the matrix shown below has a largest subblock of size 3 (the 3×3 square whose top-left corner is at the second row and third column).


```
0  1  1  0  1  0
0  0  1  1  1  0
1  1  1  1  1  1
1  1  1  1  1  0
1  0  0  1  1  1
1  0  0  0  1  1
```


1.1 Solution

Let M denote a matrix where $M[i, j]$ is the length of the side of the largest square block of 1's in A whose top-left corner is at position $[i, j]$ in A. Observe that $M[i, j] = 0$ if $A[i, j] = 0$, otherwise we have

$$M[i, j] = 1 + \min\{M[i + 1, j + 1], M[i + 1, j], M[i, j + 1]\}$$

(with the convention that out-of-bounds indices indicate a zero value). This implies that we can compute $M[i, j]$ in $O(1)$ time if we already know $M[i+1, j+1], M[i+1, j], M[i, j+1]$. Proceeding row by row, from the last row to the first and in right-to-left order within each row, we can therefore compute all of the $M[i, j]$s in $O(n^2)$ time. During this computation of the $M[i, j]$s we keep track of the largest $M[i, j]$ encountered so far (this will be the answer, after we finish computing the first row of M).

Note that the additional space for the above process (that is, the space used in addition to A itself) can be decreased from n^2 down to $O(n)$ because, when computing row i of M, we only need row $i + 1$ of M (so there is no need to store the other rows of M).

2 Finding a contiguous sub-array that maximizes sum of entries

Let A be an array of n numbers (positive or negative). We seek an $O(n)$ time algorithm that finds a contiguous interval in A such that the sum of the numbers in that interval is as large as possible. In other words, if for every pair of indices i, j where $i \leq j$ we use $S_{i,j}$ to denote $A[i] + \cdots + A[j]$, then the algorithm should find a pair of indices i, j for which $S_{i,j}$ is largest (possibly $i = j$). Note that we cannot afford to compute all $S_{i,j}$s, as that would take $O(n^2)$ time.

2.1 Non-recursive solution

This is a solution that goes through the array once, in left to right order (does not use recursion).

2.1.1 Preliminary solution

We first give an $O(n)$ solution that is convenient for conveying the main idea of the algorithm, but that contains multiple loops and uses linear additional space. Later we modify it into a solution that goes through A only once and uses a constant number of extra variables.

We first define a few quantities. Let $C[i]$, $Low[i]$, and $\Delta[i]$ be defined as follows.

- $C[i] = A[0] + \cdots + A[i]$
- $Low[i] = \min\{C[0], \ldots, C[i-1]\}$
- $\Delta[i] = C[i] - Low[i]$

If we view $C[i]$ as the closing price of a stock at the end of day i, and $A[i]$ as the price change on day i compared to the previous day, then $Low[i]$ is the lowest price ever reached by the stock prior to day i. Therefore $\Delta[i]$ is the largest "price increase" starting before day i and ending on day i. That is, $\Delta[i]$ is the largest value possible for $A[k] + \cdots + A[i]$, among all $k \leq i$. This implies that the largest of all of the $\Delta[i]$'s is equal to the largest of all the $S_{i,j}$'s. That is:

$$\max_{i,j} S_{i,j} = \max_{i} \Delta[i]$$

This suggests the following algorithm:

1. Compute the $C[i]$'s, $Low[i]$'s, and $\Delta[i]$'s in linear time. This can be done with a single left to right scan of the input array A, and it takes linear time because computing these values for an index $i > 0$ can be done in constant time by using

   - $C[i] = C[i-1] + A[i]$
   - $Low[i] = \min\{Low[i-1], C[i-1]\}$
   - $\Delta[i] = C[i] - Low[i]$

   If $i = 0$ then we simply set $C[0] = A[0]$, $Low[0] = 0$, $\Delta[0] = A[0]$.

2. Choose an index i that has largest $\Delta[i]$, call that index j. Time: Linear.

3. If $j > 0$, find an index k such that $k < j$ and $C[k] = Low[j]$ (such an index k must exist by the very definition of $Low[j]$). Time: Linear.

4. If $j > 0$ then output $k + 1, j$ as the answer index pair, and $\Delta[j]$ as the value that corresponds to that pair. If $j = 0$ then output $0, 0$ as the answer index pair, and $\Delta[0]$ (which is $A[0]$) as the value that corresponds to that pair.

2.1.2 A better implementation of the idea of the preliminary solution

This is the same idea as before but it uses a single pass through A and uses only a constant number of extra variables. Going through A in left to right order we maintain, in constant time for each position i in A, the variables *Cumulative_Sum*, *Low*, and *Best* whose significance and updating are as follows:

- When we reach position i in A, *Cumulative_Sum* is updated to contain $A[0]+\cdots+A[i]$ (in other words, at position i the *Cumulative_Sum* is the stock price at the end of day i). Note that *Cumulative_Sum* for position i can be updated from its value at the previous position $i-1$ by simply adding $A[i]$.

- When we reach position i in A, and after updating *Cumulative_Sum* as described above, *Low* is updated to contain (i) the smallest value ever achieved by *Cumulative_Sum* as we went from the start of A to the previous position $i-1$ in it; and (ii) the postition in A where that minimum occurred (in other words *Low* contains the all-time-low for that stock *before* day i, and the day when that low occurred). Note that *Low* can be updated in constant time by comparing its old value to $Cumulative\_Sum - A[i]$ (which is what *Cumulative_Sum* was at $i-1$): If the former is smaller then there is no need for an update to *Low*, but if the latter is smaller then *Low* needs to be set to $Cumulative\_Sum - A[i]$, together with the information that the new low occurred at index $i-1$.

- When we reach position i in A, and after updating *Cumulative_Sum* and *Low* as described above, *Best* is updated to contain the largest $Cumulative\_Sum - Low$ value ever encountered as we went from the start of A to the current position i. Note that *Best* can be updated in constant time by comparing its old value to $Cumulative\_Sum - Low$: If the former is larger then there is no need for an update to *Best*, but if the latter is larger then *Best* needs to be set to $Cumulative\_Sum - Low$, together with the information the pair of indices that achieve it are the index for *Low* and the current position i.

When the left-to-right sweep of A ends (at position $n-1$ in A) the answer is available in *Best*.

2.2 Recursive solution

We describe a recursive procedure $MaxSum(i, j)$ that operates on the part of the array A whose indices are in the interval $[i, j]$ and returns the following values:

- $M[i,j] = \max_{i \le a \le b \le j}(A[a] + \cdots + A[b])$, and the pair of indices a, b that achieve the max.
- $L[i,j] = \max_{i \le l \le j}(A[i] + \cdots + A[l])$ and the index l that achieves the max.
- $R[i,j] = \max_{i \le r \le j}(A[r] + \cdots + A[j])$ and the index r that achieves the max.
- $W[i,j] = A[i] + \cdots + A[j]$.

The steps of the recursive algorithm are given below; to simplify the presentation we have assumed that $j - i + 1$ (the problem size) is a power of 2. Also to simplify the notation, in the description of $MaxSum(i,j)$ that follows we dropped the explicit mention of i and j (they are understood) so the values to be returned are denoted simply by M, L, R, W and the returned indices that correspond to (respectively) M, L, R are denoted by (a,b), l, r.

1. If $i = j$ then return with all of M, L, R, W set equal to $A[i]$ and their corresponding indices set equal to i. Otherwise continue with the next steps.
2. Let ℓ be the index half-way between i and j: $\ell = (j + i - 1)/2$.
3. Recursively call $MaxSum(i, \ell)$. Let M', L', R', W' be the values returned by that recursive call, and let their corresponding returned indices be (a', b'), l', r'.
4. Recursively call $MaxSum(\ell + 1, j)$. Let M'', L'', R'', W'' be the values returned by that recursive call, and let their corresponding returned indices be (a'', b''), l'', r''.
5. Use the answers returned by the above two recursive calls in the following way.
   (a) Compute $M = \max\{M', M'', R' + L''\}$. If the maximum turns out to be M' then set $a = a'$ and $b = b'$, if it turns out to be M'' then set $a = a''$ and $b = b''$, and if turns out to be $R' + L''$ then set $a = r'$ and $b = l''$.
   (b) Compute $L = \max\{L', W' + L''\}$. If the maximum turns out to be L' then set $l = l'$, if it turns out to be $W' + L''$ then set $l = l''$.
   (c) Compute $R = \max\{R'', W'' + R'\}$. If the maximum turns out to be R'' then set $r = r''$, if it turns out to be $W'' + R'$ then set $r = r'$.
   (d) Compute $W = W' + W''$.
6. Return M, L, R, W and their corresponding $(a,b), l, r$.

The initial call (at the root of the recursion tree) is of course $MaxSum(0, n-1)$. The recurrence for its running time is: $T(1) = c_1$ and, for $n > 1$, $T(n) = 2T(n/2) + c_2$. The binary recursion tree has a linear number of nodes (n leaves and $n - 1$ internal nodes) at each of which constant work is done, so the total time is $O(n)$.