

The Reverse Problem of Range Query

Tadao Takaoka

*Department of Computer Science
University of Canterbury
Christchurch, New Zealand
E-mail: tad@cosc.canterbury.ac.nz*

Abstract

We design an efficient algorithm for the reverse problem of the two-dimensional range query. In the range query problem, we specify a range of a rectangular shape in a given (n, n) array, and count the number of points in the range. If the points have weights, we compute the sum of the weights in the range. In the reverse problem, we give a value v and find a range whose sum equals the value. The time for our algorithms is $O(n^3 \log n)$. We also give an algorithm with $O(vn^2 \log^2 n)$ time. This is fast for a small v . We briefly compare our problem with the maximum subarray problem where we obtain a subarray that maximizes the sum.

1 Introduction

The maximum subarray problem is to find an array portion of rectangular shape that maximizes the sum of array elements in it. This problem occurs in data mining [6] and [1] and graphics. In data mining, for example, record items of sales amounts can be stretched over a two-dimensional array, where each row and column corresponds to an age and an income level. Since the array elements are all non-negative, the obvious solution is the whole array. If we subtract the mean of the array elements from each array element, and consider the modified maximum subarray problem, we can have more accurate estimation on the sales trends with respect to some age groups and some income levels. In graphics, if we subtract the mean values from the each pixel value in a grey-scale graphic image, we can identify the brightest portion in the image. Sub-cubic time algorithms for this problem are known in [11] and [10].

The reverse range query problem is similar to the maximum subarray problem. It obtains a subarray whose sum equals the given value v . In comparison to the above applications, it looks for an array portion whose sales sum is a given value, or looks for some portion in graphics which is not brightest, but has some grey value. This problem can be generalized into one where we

*This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs*

find an array portion whose sum is between v and w . The efficiency of our algorithms for those problems is summarized in the following. The preprocessing for sorting is $O(n^3 \log n)$ and the time for one query is $O(n^3)$ for an (n, n) array. This shows some contrast with the maximum subarray problem whose worst case complexity is slightly below cubic and average case is close to quadratic. To show comparison, we start from a review of the maximum subarray problem in Section 2 and analyze the algorithm in Section 3. The analyzing technique in this section can be used for both the maximum subarray problem and the reverse range query problem. Then we proceed to review the saddle point search in Section 4, which plays the central role for the reverse range query. In Section 5, we introduce the concept of presort, and use it to develop an efficient algorithm for the reverse range query problem. In Section 6, we generalize the problem into one where we obtain an array portion whose sum lies between given values v and w . In Section 7, we give an algorithm with $O(vn^2 \log^2 n)$ time. This is fast when v is small. Section 8 concludes the paper.

2 Review of the maximum subarray problem

We first review distance matrix multiplication. The distance matrix multiplication is to compute the following distance product $C = AB$ for two n -dimensional matrices $A = [a_{i,j}]$ and $B = [b_{i,j}]$ whose elements are real numbers.

$$c_{i,j} = \min_{k=1}^n \{a_{i,k} + b_{k,j}\}$$

The best known algorithm for this problem for worst case is by Takaoka [8], the computing time of which is $O(n^3(\log \log n / \log n)^{1/2})$. The best expected time, $O(n^2 \log n)$, for this problem with a wide class of random matrices is by Moffat and Takaoka [7]. The meaning of $c_{i,j}$ is the shortest distance from vertex i in the first layer to vertex j in the third layer in an acyclic directed graph consisting of three layers of vertices, which are labelled $1, \dots, n$ in each layer, and the distance from i in the first layer to j in the second layer is $a_{i,j}$ and that from i in the second layer to j in the third layer is $b_{i,j}$. If we replace the above \min operation by \max , we can define a similar product, where we have longest distances in the above three layered graph. The algorithm can be tailored to this version easily by symmetric considerations. We refer to the original multiplication and the algorithm as the *min* version, and those with *max* as the *max* version.

Now we proceed to the maximum subarray problem and the reverse range query problem. We are given a two-dimensional array $a[1..m, 1..n]$ as input data. The maximum subarray problem is to maximize the array portion $a[k..i, l..j]$, that is, to obtain such indices (k, l) and (i, j) . The reverse range query is to find an array portion whose sum is equal to the given value v . We suppose the upper-left corner has co-ordinate $(1,1)$.

| | | | | | | | |
|----|----|----|----|----|----|----|----|
| -1 | 2 | -3 | 5 | -4 | -8 | 3 | -3 |
| 2 | -4 | -6 | -8 | 2 | -5 | 4 | 1 |
| {3 | -2 | 9} | -9 | 3 | 6 | -5 | 2 |
| {1 | -3 | 5} | -7 | 8 | -2 | 2 | -6 |

Example 2.1 *Let a be given by*

Then the maximum subarray is given by the rectangle defined by the upper left corner $(3, 5)$ and the lower right corner $(4, 6)$, given by vertical bars. The array portion whose sum is 13 is given by braces.

We assume that $m \leq n$ without loss of generality. We also assume that m and n are powers of 2. If not, we can show the same asymptotic complexity by embedding the array into the next powers of 2 beyond m and n . Bentley's algorithm finds the maximum subarray in $O(m^2n)$ time, which is defined to be cubic in this paper.

The central algorithmic concept in this section is that of prefix sum. The prefix sums $sum[1..n]$ of a one-dimensional array $a[1..n]$ are computed by

```
sum[0] := 0;
for i := 1 to n do sum[i] := sum[i - 1] + a[i];
```

This algorithm can be extended to two dimensions with linear time $O(mn)$, the details of which are omitted.

We use distance matrix multiplications of both *min* and *max* versions in this section. We compute the partial sums $s[i, j]$ for array portions of $a[1..i, 1..j]$ for all i and j with boundary condition $s[i, 0] = s[0, j] = 0$. These sums are often called the two-dimensional prefix sums of a . The outer framework of the algorithm is given below. Note that the prefix sums once computed are used throughout recursion.

Algorithm 1: Maximum subarray

If the array becomes one element, return its value.

Otherwise, if $m > n$, rotate the array 90 degrees.

Thus we assume $m \leq n$.

Let A_{left} be the solution for the left half.

Let A_{right} be the solution for the right half.

Let A_{column} be the solution for the column-centered problem.

Let the solution be the maximum of those three.

Here the column-centered problem is to obtain an array portion that crosses over the central vertical line with maximum sum, and can be solved in the following way.

$$A_{column} = \max_{k=1, l=0, i=1, j=n/2+1}^{i-1, n/2-1, m, n} \{s[i, j] - s[i, l] - s[k, j] + s[k, l]\}.$$

In the above we first fix i and k , and maximize it over l and j . Then the above problem is equivalent to maximizing the following for $i = 1, \dots, m$ and $k = 1, \dots, i - 1$.

$$A_{column}[i, k] = \max_{l=0, j=n/2+1}^{n/2-1, n} \{-s[i, l] + s[k, l] + s[i, j] - s[k, j]\}$$

Let $s^*[i, j] = -s[j, i]$. Then the above problem can further be converted into

$$A_{column}[i, k] = -\min_{l=0}^{n/2-1} \{s[i, l] + s^*[l, k]\} + \max_{j=n/2+1}^n \{s[i, j] + s^*[j, k]\} \quad (1)$$

The first part in the above is distance matrix multiplication of the *min* version and the second part is of the *max* version. Let S_1 and S_2 be matrices whose (i, j) elements are $s[i, j - 1]$ and $s[i, j + n/2]$. For an arbitrary matrix T , let T^* be that obtained by negating and transposing T . Then the above can be computed by multiplying S_1 and S_1^* by the *min* version and taking the lower triangle, multiplying S_2 and S_2^* by the *max* version and taking the lower triangle, and finally subtracting the former from the latter and taking the maximum from the triangle.

3 Analysis

Let us assume that m and n are each a power of 2, and $m \leq n$. As we saw in Section 2, we can reduce to the case where $m = n$ by chopping the array into squares. Thus we analyze the time $T(n)$ for the (n, n) array. We observe the algorithm splits the array vertically and then horizontally. We can multiply $(n, n/2)$ and $(n/2, n)$ matrices by 4 multiplications of size $(n/2, n/2)$. We analyze the number of comparisons. The rest is proportional to this. Let $M(n)$ be the time for multiplying two $(n/2, n/2)$ matrices. Thus we have the following recurrence.

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 4T(n/2) + 12M(n). \end{aligned}$$

We can show that if $M(n) = O(n^3(\log \log n / \log n)^{1/2})$ for the worst case, $T(n) = O(M(n))$, and if $M(n) = O(n^2 \log n)$ for the average case, $T(n) = O(M(n) \log n)$.

4 Saddle point search

In this section, to prepare for the reverse problem of range query, we review the saddle point search. Let $A[1..n]$ and $B[1..n]$ be two arrays sorted in non-decreasing order and non-increasing order respectively. The following algorithm, called the saddle point search, is folklore. The algorithm finds $A[i]$ and $B[j]$ such that $A[i] + B[j] = v$ for a given value v . After setting i and j to 1, if $A[1] + B[1] < v$, we increase i until $A[i] + B[j] < v$ becomes false. If $A[i] + B[j] > v$, we increase j until $A[i] + B[j] > v$ becomes false. This

process continues alternately until it finds v or hits the boundary. We call the behaviour of indices i and j an alternate move.

Algorithm 2: Saddle point search

```

i := 1; j := 1;
loop
  while i ≤ n and A[i] + B[j] < v do i := i + 1;
  if i = n + 1 or A[i] + B[j] = v then exit;
  while j ≤ n and A[i] + B[j] > v do j := j + 1;
  if j = n + 1 or A[i] + B[j] = v then exit;
end loop
if i = n + 1 or j = n + 1 then write('no solution')
else write('found at', i, j);

```

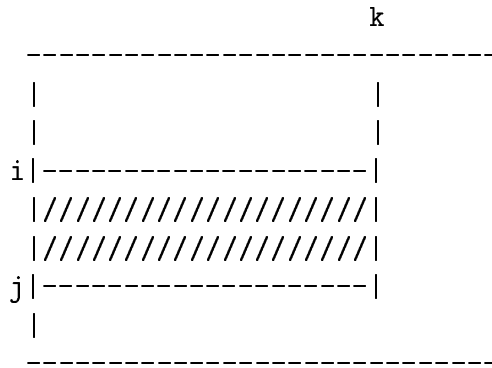
Since i and j move alternately, the time of this algorithm is $O(n)$. The correctness is seen from the following fact. Let i_1, i_2, \dots be the values of i when the first while loop is entered in this order, and j_1, j_2, \dots be those of j for the second while loop. We have the following three cases.

- (1) $A[1] + B[1] = v$. We find $A[i] + B[j]$ at the beginning, and finish.
- (2) $A[1] + B[1] < v$. In this case i changes first. For $k = 1, 2, \dots$, we have $A[i] + B[j] < v$ for $i < i_{k+1}$ and $j_k \leq j$, and $A[i] + B[j] > v$ for $i \geq i_{k+1}$ and $j < j_{k+1}$.
- (3) $A[1] + B[1] > v$. In this case j changes first. For $k = 1, 2, \dots$, we have $A[i] + B[j] > v$ for $i \geq i_k$ and $j < j_{k+1}$, and $A[i] + B[j] < v$ for $i < i_{k+1}$ and $j \geq j_{k+1}$.

Thus $A[i] + B[j] = v$ for some $(i, j) = (i_k, j_{k'})$, where k and k' differ by at most one, or there is no solution. This algorithm terminates at the first solution. We can convert this to one that finds all solutions in the same time complexity.

5 The reverse problem of range query

In this section, we consider the problem of obtaining a subarray whose sum is equal to a given value v . Let a two dimensional array a of size (m, n) and its two-dimensional prefix sum array s be given. We first sort the sums $S_{i,j}[k]$ of array portions $a[i..j, 1..k]$ for $k = 1, \dots, n$ and those $T_{k,l}[i]$ of $a[1..i, k..l]$ for $i = 1, \dots, m$. We call these computations the horizontal presort and the vertical presort. We denote the horizontally and vertically sorted arrays by $C_{i,j}$ and $D_{i,j}$ respectively. Array portion $a[i..j, k]$ is illustrated by the hatched part in the following. The sum in the hatched part is $S_{i,j}[k]$.

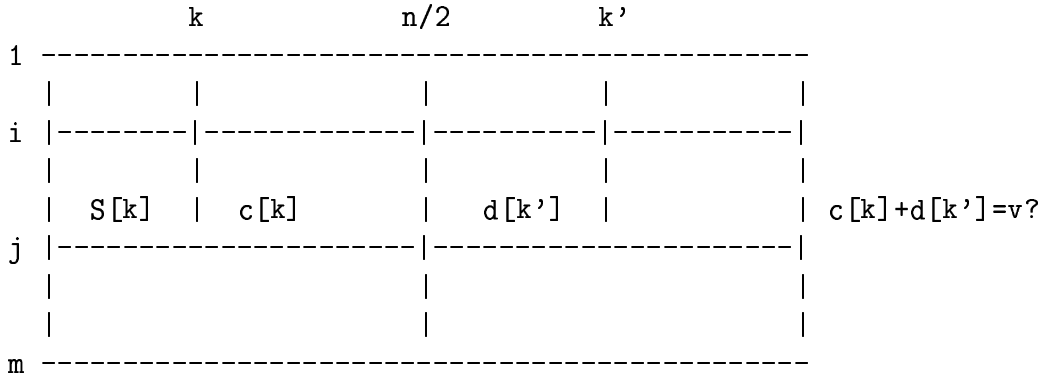


Algorithm 3: Horizontal presort
for $i := 1$ **to** m **do for** $j := i$ **to** m **do begin**
 for $k := 1$ **to** n **do** $b[k] := s[j, k] - s[i - 1, k]$;
 Let $S_{i,j} = b$;
 Sort array b in non-decreasing order;
 Let $C_{i,j} = b$
end.

Obviously this takes $O(m^2n \log n)$ time. The vertical presort can be done similarly, taking $O(mn^2 \log m)$ time. Now the algorithm for the reverse range query follows, where if there is no solution, “empty” is returned.

Algorithm 4: Reverse range query
 If the array becomes one element, check if its value is equal to v .
 Otherwise, if $m > n$, rotate the array 90 degrees.
 Thus we assume $m \leq n$.
 Let A_{left} be the solution for the left half.
 Let A_{right} be the solution for the right half.
 Let A_{column} be the solution for the column-centered problem.
 Return any non-empty solution of the above three, if any. Otherwise return empty

In the above, the column centered problem can be solved using the saddle point search in the following way. For each i and j , let $c_{i,j}[k]$ be the sums of array portions $a[i..j, k..n/2]$ for $k = 1, \dots, n/2$, sorted in non-increasing order. Let $d_{i,j}[k']$ be those of $a[i..j, n/2 + 1..n/2 + k']$ for $k' = 1, \dots, n/2$, sorted in non-decreasing order. Then by applying the saddle point search on $c_{i,j}$ and $d_{i,j}$, we can solve the column centered problem in $O(n)$ time for each i and j . In total, we spend $O(m^2n)$ time for the column centered problem. The following figure illustrates the column centered problem, in which indices i and j for arrays c , d , and S are omitted.



Now we need to explain how to obtain the above arrays $c_{i,j}$ and $d_{i,j}$. Let $H_{i,j}$ be the sum of array portion $a[i..j, 1..n/2]$. When we do the horizontal presort and the vertical presort using array elements as keys, we sort the positions of the array elements together. Then by scanning array $C_{i,j}$ made by the algorithm presort and using the position index, we can extract those elements into array $c_{i,j}$ and $d_{i,j}$, i.e., if the position index k belongs to the left half of a , it goes to $c_{i,j}$ with key $H_{i,j} - S_{i,j}[k]$ and position index k , and if the position k' goes to the right half, it goes to array $d_{i,j}$ with key $S_{i,j}[k'] - H_{i,j}$ and position index $k' - n/2$. This process of creating arrays $c_{i,j}$ and $d_{i,j}$ takes $O(n)$ time. The effort of the vertical presort is used when we rotate the array 90 degrees. For recursive calls, we can pass $c_{i,j}$ and $d_{i,j}$ as parameters, and regard them as $C_{i,j}$ or $D_{i,j}$ in the procedures called. Note that, we do not need to scan the whole original arrays in the recursive calls.

Now we analyze the time for the algorithm for $m = n$. We note that we have the same recurrence equation as in Theorem 1 except for the second term; instead of $12M(n)$ we have $O(n^3)$, since we do saddle point search for all i and j . Thus we have $T(n) = O(n^3)$. That is, after spending $O(n^2)$ time for prefix sum computation and $O(n^3 \log n)$ time for the presort, we can solve one reverse range query problem in $O(n^3)$ time.

We can easily generalize the analysis into an (m, n) array, and show that the time for presort is $O(m^2 n \log n)$, and that for one query is $O(m^2 n)$ for both the reverse range query and the generalized range query. If $m = 1$, the presort takes $O(n \log n)$ time, and we can solve the one-dimensional problem in $O(n)$ time for one query.

6 Generalization of reverse range query

In this section we consider the problem of obtaining a subarray whose sum is between v and w for $v \leq w$. We first modify the saddle point search slightly so that we can find the minimum $A[i] + B[j]$ such that $A[i] + B[j] \geq v$. If this minimum $A[i] + B[j]$ satisfies $A[i] + B[j] \leq w$, this is a solution. Otherwise we have no solution. Now the modified saddle point search follows.

Algorithm 5: Modified saddle point search

$i := 1; j := 1; min := \infty;$

repeat

while $i \leq n$ **and** $A[i] + B[j] < v$ **do** $i := i + 1;$

if $A[i] + B[j] < min$ **then** $min := A[i] + B[j];$

while $j \leq n$ **and** $A[i] + B[j] > v$ **do begin**

if $A[i] + B[j] < min$ **then** $min := A[i] + B[j];$

$j := j + 1;$

end

until $i = n + 1$ **or** $j = n + 1;$

At the end of the algorithm, min holds the minimum value of $A[i] + B[j]$ that is not less than v . If there is no such value, min is infinity.

As seen from this algorithm, the generalized reverse range query has the same time complexity as the reverse range query.

7 Faster algorithm for small v

In this section, we deal with integer values for the given array elements and the value v , and give an algorithm for reverse range query with $O(vn^2 \log^2 n)$ time. When v is a small integer, this algorithm is faster than that in Section 5.

For preparation, we begin with an enhanced saddle point search, called indexed saddle point search. Let array elements have two fields *key* and *index*. Arrays A and B are lexico-graphically sorted with *key* and *index*. That is, A is first sorted in non- decreasing order of *key*. Within the same *key* value, A is sorted in non-decreasing order of *index*. In the case of B , it is first sorted in non- increasing order of *key*. Within the same *key* value, B is sorted in non-decreasing order of *index*. The following algorithm finds i and j such that $A[i].key + B[j].key = v$ and $A[i].index = B[j].index$, if any.

Algorithm 6: Indexed saddle point search

$i := 1; j := 1;$

loop

while $i \leq n$ **and** $A[i].key + B[j].key < v$ **do** $i := i + 1;$

if $i = n + 1$ **then return** “no solution”;

while $i \leq n$ **and** $A[i].key + B[j].key = v$ **and** $A[i].index < B[j].index$

do $i := i + 1;$

if $i = n + 1$ **then return** “no solution”;

if $A[i].key + B[j].key = v$ **and** $A[i].index = B[j].index$ **then return** $(i, j);$

while $j \leq n$ **and** $A[i].key + B[j].key > v$ **do** $j := j + 1;$

if $j = n + 1$ **then return** “no solution”;

while $j \leq n$ **and** $A[i].key + B[j].key = v$ **and** $A[i].index > B[j].index$

```

do  $j := j + 1$ ;
if  $j = n + 1$  then return “no solution”;
if  $A[i].key + B[j].key = v$  and  $A[i].index = B[j].index$  then return  $(i, j)$ ;
end loop;

```

This algorithm is a generalization of the saddle point search in Section 4. After we find $A[i].key + B[j].key = v$, we further seek the condition $A[i].index = B[j].index$. Since the indices are sorted in non-decreasing order within the same *key* value, we can find the condition $A[i].index = B[j].index$ without overlooking. This can be viewed as a 2-way lexico-graphic saddle point search, if we convert the condition $A[i].index = B[j].index$ into $A[i].index - B[j].index = 0$.

Now we apply the indexed saddle point search to our problem of reverse range query. We divide the given rectangular array into four equal-sized arrays. The upper-left, upper-right, lower-left, and lower-right portions are symbolized by NW, NE, SW, and SE, abbreviation borrowed from geography. In the following the row-centered problem is to obtain a region that crosses over the horizontal center line. The center problem is to obtain a region that crosses over the center point.

Algorithm 7: Reverse range query for small v

If the array is one-dimensional, solve it by Algorithm 4 in one dimension.

Otherwise, if $m > n$, rotate the array 90 degrees.

Thus we assume $m \leq n$.

Let A_{NW} be the solution for the upper-left part.

Let A_{NE} be the solution for the upper-right part.

Let A_{SW} be the solution for the lower-left part.

Let A_{SE} be the solution for the lower-right part.

Let $A_{left-row}$ be the solution for the row-centered problem in the left half.

Let $A_{right-row}$ be the solution for the row-centered problem in the right half.

Let $A_{upper-column}$ be the solution for the column-centered problem in the upper half.

Let $A_{lower-column}$ be the solution for the column-centered problem in the lower half.

Let A_{center} be the solution for the center problem.

Return any non-empty solution of the above nine, if any. Otherwise return empty

Row-centered and column-centered problems are solved in the following. We only show the row-centered problem. The column-centered problem is similar.

Algorithm: Row-centered problem

If the array is one-dimensional, solve it by Algorithm 4 in one dimension.

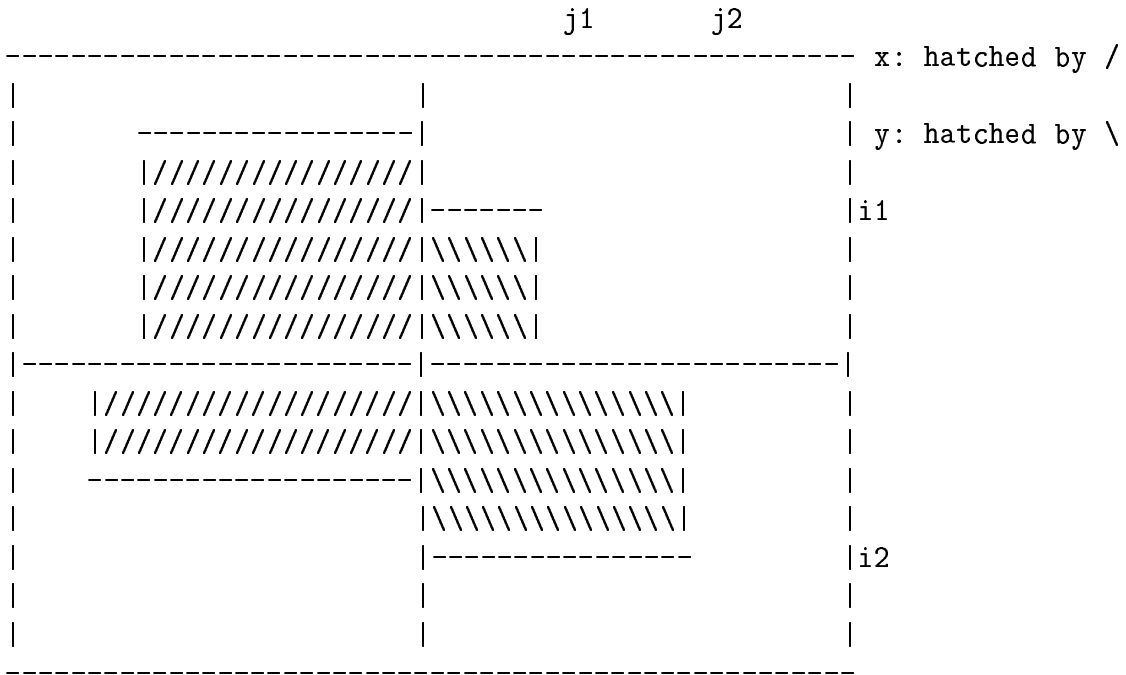
Let $A_{left-row}$ be the solution for the row-centered problem in the left half.
Let $A_{right-row}$ be the solution for the row-centered problem in the right half.
Let A_{center} be the solution for the center problem.
Return any non-empty solution of the above three, if any. Otherwise return empty.

Now we show how to solve the center problem. We define $NW[i, j]$, $NE[i, j]$, $SW[i, j]$, and $SE[i, j]$ by the partial sums of the region defined by (i, j) and the center. That is,

$$\begin{aligned} NW[i, j] &= s[m/2, n/2] - s[i, n/2] - s[m/2, j] + s[i, j] \\ NE[i, j] &= s[m/2, j] - s[i, j] - s[m/2, n/2] + s[i, n/2] \\ SW[i, j] &= s[i, n/2] - s[i, j] - s[m/2, n/2] + s[m/2, j] \\ SE[i, j] &= s[i, j] - s[i, n/2] - s[m/2, j] + s[m/2, n/2] \end{aligned}$$

We sort in non-decreasing order the partial sums $NW[i, j]$ for $i = 1, \dots, m/2; j = 1, \dots, n/2$ together with indices (i, j) . We sort them in lexicographic order of (key, j, i) in $O(mn \log mn)$ time, where key is the value of $NW[i, j]$ itself. We name the sorted list as $list[i], i = 1, \dots, mn/4$. We rename the above (key, j, i) by $(list[i].key, list[i].index1, list[i].index2)$. We can have the sorted lists for NE, SW, and SE in a similar way; NE in non-decreasing order, SW and SE in non-increasing order.

To solve the center problem for v , we divide the value as $v = x + y$. Then we find regions touching the center vertical line; the left whose sum is x and the right whose sum is y . To solve the right problem with y , we can use the indexed saddle point search. The array A is the sorted list for NE, and array B is that for SE. If we can find $A[I]$ and $B[J]$ such that $A[I].key + B[J].key = y$ and $A[I].index1 = B[J].index1$, the coordinates $(A[I].index2, A[I].index1)$ and $(B[J].index2, B[J].index1)$ define the right solution for y . We can similarly solve the left problem. To solve the center problem, we list up solutions $(A[I].index2, B[J].index2)$ obtained from the indexed saddle point search using key and $index1$. We sort those solutions for the left half and the right half, and sort those two lists in lexicographic order. By scanning those sorted lists by alternate move, we can find the solution in $O(mn)$ time, if it exists. Note that we check the equality of indices through the alternate move. The following figure illustrates the solution for y , where $j_1 = A[I].index1$ and $j_2 = B[J].index1$, and $i_1 = A[I].index2$ and $i_2 = B[J].index2$. We first seek solutions with j -coordinates matched in each of the left and right regions, and then with i -coordinates matched over the two regions.



After sorting, the time for each x and y such that $x + y = v$ is $O(mn)$. Thus for all x and y such that $x + y = v$, it takes $O(mn(v + \log mn))$ time. Let us set up the recurrence for the times. Let $T(m, n)$ be the time for the whole problem. Let $S(m, n)$ and $S(n, m)$ be the times for the row-centered and column-centered problems. The sorting effort for the four regions used in the first center problem can be used for the second center problem using a technique used in Section 5. The recurrence follows.

$$\begin{aligned}
 T(m, 1) &= O(m), T(1, n) = O(n) \\
 T(m, n) &= 4T(m/2, n/2) + 2S(m/2, n) + 2S(n/2, m) + O(mn(v + \log mn)) \\
 S(1, n) &= O(n) \\
 S(m, n) &= 2S(m/2, n) + O(vmn)
 \end{aligned}$$

By eliminating S , we have

$$T(m, n) = 4T(m/2, n/2) + O(vmn \log(mn))$$

From this, we have $T(m, n) = O(vmn \log^2(mn))$. The time for sorting $s[i, 1..n]$ and $s[1..m, j]$ for $i = 1, \dots, m; j = 1, \dots, n$ for use in the one-dimensional cases is absorbed in this complexity. For a square array, we have $T(n, n) = O(vn^2 \log^2 n)$.

8 Concluding remarks

We showed that the reverse range query can be solved within the same framework as the maximum subarray problem, but with time complexity slightly greater. It is open whether the reverse query can be solved in sub-cubic time. The time is dominated by that of presort. The time for one query is slightly

smaller than the time for presort. It is also open whether one query can be solved in subcubic time, when we spend $O(n^3 \log n)$ time for the presort. Note that if we spend $O(n^4 \log n)$ time to sort the n^4 possible subarrays with those sums as keys, one reverse range query can be solved in $O(\log n)$ time by binary search. We showed $O(vn^2 \log^2 n)$ time for a small v . When v approaches n , this complexity worsens.

There seems to be no room for improvement for the average case of the reverse range query problem, which is a sharp contrast to the maximum subarray problem.

References

- [1] Agrawal, R, T. Imielinski, and A. Swami, Mining Association Rules between Sets of Items in Large Databases, Proc. SIGMOD Conf. on Management of Data (1993) 207-216
- [2] Alon, N, Z. Galil, and O. Margalit, On the exponent of the all pairs shortest path problem, Proc. 32nd FOCS (1991) 569-575
- [3] Bentley, J., Programming Pearls - Algorithm Design Techniques, Comm. ACM, 27-9 (1984) 865-871
- [4] Bentley, J., Programming Pearls - Perspective on Performance, Comm. ACM, 27 (1984) 1087-1092
- [5] Feller, W, An Introduction to Probability Theory and its Applications, Volume 1, John Wiley and Sons (1950)
- [6] Fukuda, T., Y. Morimoto, S. Morishita, and T. Tokuyama, Data Mining Using Two-Dimensional Optimized Association rules: Scheme, Algorithms and Visualization, Proc. SIGMOD Conf. on Management of Data (1996) 13-23
- [7] Moffat, A. and T. Takaoka, An all pairs shortest path algorithm with $O(n^2 \log n)$ expected time, SIAM Jour. Computing, 16 (1987) 1023-1031
- [8] Takaoka, T., A New Upper Bound on the complexity of the all pairs shortest path problem, Info. Proc. Lett., 43 (1992) 195-199
- [9] Takaoka, T, Subcubic cost algorithms for the all pairs shortest path problem, Algorithmica, 20 (1998) 309-318
- [10] Takaoka, T, Efficient Algorithms for the Maximum Subarray Problem by Distance Matrix Multiplication, Proceedings of the Australasian Theory Conference, CATS02, (2002) 189-198
- [11] Tamaki, H. and T. Tokuyama, Algorithms for the Maximum Subarray Problem Based on Matrix Multiplication, Proceedings of the 9th SODA (Symposium on Discrete Algorithms) (1998) 446-452