วิธีแก้ปัญหา Range Sum และ Maximum Sum of Contiguous Subsequence

ปัญหา *Range Sum* คือ การหาผลบวกในช่วงของอาเรย์ที่ติดกันเป็นจำนวนช่องที่แน่นอน ในขณะที่ปัญหา *Maximum Sum of Contiguous Subsequence* คือ การหาผลบวกในช่วงของอาเรย์ที่ติดกันแต่เป็นจำนวนช่องเท่าใดก็ได้ขอแค่ติดกันและให้ผลบวกที่มากที่สุดก็พอ ปัญหาทั้งสองนี้มีความเกี่ยวข้องกันอย่างใกล้ชิดและเราอาจจะใช้เทคนิค Range Sum มาช่วยแก้ปัญหา Maximum Sum of Contiguous Subsequence ก็ได้แม้ว่าประสิทธิภาพอาจจะไม่ดีเทียบเท่ากับการใช้วิธีที่เจาะจงกับ Maximum Sum ก็ตาม เราจะพิจารณาตัวอย่างต่อไปนี้ เพื่อให้เข้าใจปัญหา Range Sum ได้ดีขึ้น จากนั้นเราจะดูการใช้งานในรูปแบบต่าง ๆ ก่อนที่จะกล่าวถึงปัญหา Maximum Sum อย่างละเอียด

Range Sum กับอาเรย์หนึ่งมิติ

ตัวอย่าง หากเรามีอาเรย์ขนาดสิบช่อง ที่ประกอบไปด้วยเลขจำนวนเต็มดังนี้ 7 5 3 4 1 9 6 8 10 2 และเราต้องการหาผลบวกของอาเรย์ 5 ช่องที่ติดกันว่าแต่ละช่วงมีค่าเท่าใดบ้างผลบวกที่ได้ในแต่ละช่วงเป็นดังนี้

1. 7 + 5 + 3 + 4 + 1 = 20
2. 5 + 3 + 4 + 1 + 9 = 22
3. 3 + 4 + 1 + 9 + 6 = 23
4. 4 + 1 + 9 + 6 + 8 = 28
5. 1 + 9 + 6 + 3 + 10 = 34
6. 9 + 6 + 3 + 10 + 2 = 35

ลองสังเกตผลบวกชุดที่หนึ่งกับชุดที่สอง ดูจะพบว่ามีตัวเลขต่างกันแค่สองตัวคือเลข 7 ที่มีเฉพาะในชุดที่หนึ่ง และเลข 9 ที่มีเฉพาะในชุดที่สอง ข้อสังเกตที่สำคัญอีกอย่างหนึ่งก็คือว่าหากเรานำผลบวกชุดที่หนึ่งมาลบออกด้วย 7 และบวกเข้าด้วย 9 เราก็จะได้ผลบวกชุดที่สอง ในทำนองเดียวกันจากผลบวกชุดที่สอง ถ้าเรานำ 5 มาลบออกและบวกเข้าด้วย 6 เราก็จะได้ผลบวกของชุดที่สาม

การบวกเข้าและลบออกนี้สามารถเพิ่มความเร็วของโปรแกรมเราได้เพราะหากเราทำการหาผลบวกโดยตรงเราจะต้องทำการบวกเลขสี่ครั้งเพื่อหาผลลัพธ์ของเลขแต่ละชุด แต่การบวกเข้าและลบออกนี้ทำให้เราสามารถหาผลบวกเลขในชุดต่อ ๆ มาด้วยการบวกและลบอย่างละหนึ่งครั้งเท่านั้น ความแตกต่างด้านปริมาณการคำนวณจะสูงยิ่งขึ้นหากจำนวนข้อมูลในแต่ละชุดมีมากขึ้นและอาเรย์ยาวขึ้น เป็นต้นว่าหากอาเรย์ยาวหนึ่งล้านช่องและข้อมูลในแต่ละชุดมีทั้งหมด 1,001 ตัว การหาผลบวกในเลขแต่ละชุดจะต้องทำการบวก 1,000 ครั้ง ในขณะที่เทคนิคบวกเข้าและลบออกจะใช้การบวกและลบอย่างละหนึ่งครั้งเท่าเดิมโดยไม่สำคัญว่าเลขแต่ละชุดจะยาวเท่าใด

เพื่อความเข้าใจที่มากขึ้นเกี่ยวกับวิธีเขียนโปรแกรม ผมขอแนะนำให้ดูโค้ด range_sum_1d.cpp ประกอบ

โค้ดจากไฟล์ range_sum_1d.cpp

```
#include <cstdio>
int main() {
    int input[] = {7, 5, 3, 4, 1, 9, 6, 8, 10, 2};

    // Initialize first sum of 5 contiguous array elements.
    int sum = 0;
    for(int i = 0; i < 5; ++i)
        sum += input[i];
    printf("sum 1 = %d\n", sum);

    // We can quickly compute other range sum by one addition
    //   and one subtraction.
    // Notice that if we directly compute range sum,
    //   we need 4 additions.
    for(int t = 5; t < 10; ++t) {
        sum = sum - input[t-5] + input[t];
        printf("sum %d = %d\n", t - 3, sum);
    }

    return 0;
}
```

เทคนิคการบวกเข้าและลบออกยังสามารถทำได้อีกแบบหนึ่ง แม้ว่าวิธีที่จะเสนอต่อไปนี้จะต้องใช้อาเรย์อีกอันมาเก็บผลลัพธ์ชั่วคราว แต่มันเป็นวิธีการที่สามารถนำไปประยุกต์ใช้กับปัญหาสองมิติได้สะดวกขึ้น ซึ่งเราสามารถอธิบายการทำงานของวิธีที่สองจากตัวอย่างเดิมได้ดังนี้

1. เราเริ่มจากการสร้างอาเรย์ขนาดเท่ากับข้อมูลเข้า จากตัวอย่างนี้เราจะได้อาเรย์เปล่า ๆ ขนาด 10 ช่อง
2. กำหนดให้ช่องที่ k เก็บผลบวกจากช่องที่ 1 ถึงช่องที่ k (ผลบวกรวมช่องที่ 1 กับ k ด้วย ในที่นี้สมมติให้อาเรย์เริ่มที่ช่องหมายเลข 1) ดังนั้นขั้นตอนนี้จะได้ผลลัพธ์ในอาเรย์ที่สร้างใหม่เป็น 7, 12, 15, 19, 20, 29, 35, 43, 53, 55
3. อาเรย์ช่องที่ห้าคือคำตอบของผลบวกชุดที่หนึ่ง ส่วนชุดที่สองได้มาจากช่องที่หกลบด้วยช่องที่หนึ่ง ซึ่งก็คือ 29 – 7 ชุดที่สามได้มาจากช่องที่เจ็ดลบด้วยช่องที่สอง ซึ่งก็คือ 35 – 12 ชุดที่สี่คำนวณได้จากช่องที่แปดลบช่องที่สาม ซึ่งก็คือ 43 – 15 ส่วนชุดที่เหลือก็คำนวณได้ในลักษณะเดียวกัน

เราสามารถศึกษาวิธีการเขียนโปรแกรมได้จากโค้ดจากไฟล์ range_sum_1d_w_array.cpp

โค้ดส่วนของโปรแกรมจากไฟล์ range_sum_1d_w_array.cpp

```
int main() {
  int input[] = {7, 5, 3, 4, 1, 9, 6, 8, 10, 2};

  /// Create a temporary array and fill it with sum from
  ///   the beginning.
  int temp[10];
  temp[0] = input[0];
  for(int i = 1; i < 10; ++i)
    temp[i] = temp[i-1] + input[i]; /// Note how we sum up the input.

  /// We can quickly compute other range sum by subtracting an
  ///   irrelevant sum from the 'entire sum'.
  /// Note that if we directly compute range sum,
  ///   we need 4 additions.
  printf("sum 1 = %d\n", temp[4]);
  for(int t = 5; t < 10; ++t) {
    int sum = temp[t] - input[t-5];
    printf("sum %d = %d\n", t - 3, sum);
  }
  return 0;
}
```

เราสามารถเข้าใจเรื่องความถูกต้องของวิธีการนี้ได้ไม่ยากเพราะในตอนแรกอาเรย์เก็บผลลัพธ์ชั่วคราวนี้บวกเลขทุกอย่างที่ขวางหน้า เพื่อที่จะให้เราได้ผลลัพธ์เฉพาะในช่วงที่ต้องการ เราจึงทำการลบค่าของกลุ่มตัวเลขที่ไม่เกี่ยวข้องออกไป เรื่องที่น่าสนใจก็คือว่าจำนวนการบวกลบของวิธีที่สองเท่ากับวิธีแรก และภาระการคำนวณไม่ขึ้นกับจำนวนตัวเลขในแต่ละชุด

Range Sum กับอาเรย์สองมิติ

คราวนี้ลองมาดูวิธีการประยุกต์ใช้วิธีที่สองกับอาเรย์สองมิติบ้าง สมมติว่าเรามีอาเรย์ขนาด 5 แถว 6 คอลัมน์ ดังแสดงข้างล่างนี้

```
4 3 1 9 8 7
5 2 9 3 6 2
0 8 7 0 2 1
1 2 5 3 4 8
7 6 0 8 9 2
```

ต่อมาเราต้องการหาผลบวกในพื้นที่ติดกันขนาด 3 แถว 4 คอลัมน์ จากตัวอย่างนี้พื้นที่แรกจะประกอบด้วย

```
4  3  1  9
5  2  9  3
0  8  7  0
```

ในขณะที่พื้นที่ถัดมาในแถวเดียวกันจะมีตัวเลขเป็น

```
3  1  9  8
2  9  3  6
1  2  5  3
```

ความแตกต่างของพื้นที่ทั้งสองเป็นไปในลักษณะเดียวกันกับปัญหาของอาเรย์ 1 มิติ กล่าวคือพื้นที่แรกจะมีเลข 4, 5 และ 0 จากคอลัมน์ซ้ายสุด ในขณะที่พื้นที่ที่สองจะมีเลข 8, 6 และ 3 จากคอลัมน์ทางขวาสุดในพื้นที่ที่สอง ดังนั้นจากพื้นที่แรก ถ้าเราลบ 4, 5 และ 0 ออก แล้วบวก 8, 6 และ 3 เข้าไปก็จะได้ผลลัพธ์เป็นพื้นที่ที่สอง

สำหรับอีกทิศทางหนึ่ง หากพื้นที่ถัดมาอยู่ในคอลัมน์เดียวกันกับพื้นที่แรกแต่เลื่อนลงมาหนึ่งแถว เราก็จะได้พื้นที่ที่สามเป็น

```
5  2  9  3
0  8  7  0
1  2  5  3
```

ในลักษณะเดียวกัน จากพื้นที่แรก ถ้าเราลบแถวแรกที่ประกอบไปด้วยเลข 4, 3, 1 และ 9 ออก แล้วบวกเลข 5, 2, 9 และ 3 เข้าไป เราก็จะได้พื้นที่ที่สาม

เรื่องยุ่ง ๆ ของปัญหาสองมิติก็คือว่า ถ้าเรามุ่งไปตามแถวเดียวกัน เมื่อจบแถวแล้วเราก็ต้องมาคำนวณค่าเริ่มต้นกันใหม่และก็ต้องทำการบวกลบเลขบางอย่างซ้ำซาก ปัญหานี้จะหนักขึ้นถ้าหากพื้นที่ที่สนใจมีขนาดใหญ่ เพราะการเริ่มคำนวณใหม่แต่ละรอบจะใช้เวลามาก นอกจากนี้การบวกลบคอลัมน์ก็จะใช้เวลานานด้วย ในขณะที่เราอยากได้วิธีที่มีประสิทธิภาพในระดับที่ว่า ไม่ว่าขนาดพื้นที่ที่สนใจจะใหญ่เท่าใด ก็จะใช้เวลาในการคำนวณเท่า ๆ เดิม

เราสามารถบรรลุวัตถุประสงค์ดังกล่าวได้ด้วยการหาผลบวกจากจุดเริ่มต้นไปยังทุกตำแหน่งที่สนใจอย่างที่ทำมาก่อนหน้ากับอาเรย์หนึ่งมิติด้วยวิธีที่สอง นั่นคือเราเริ่มจากการสร้างอาเรย์สองมิติสำหรับเก็บผลลัพธ์ขึ้นมาก่อนโดยให้มันมีขนาดเท่ากับอาเรย์ของอินพุต จากนั้นเราจะทำให้ข้อมูล ณ แถวที่ r คอลัมน์ที่ c คือผลบวกจากแถวและคอลัมน์แรกมาจนถึงแถวที่ r และ c (ผลบวกนี้จะไม่มีตัวเลขจากคอลัมน์ที่เกินกว่า c และแถวที่เกินกว่า r ปนอยู่ด้วย) ซึ่งจะได้ผลการบวกในขั้นตอนนี้เป็น

```
  4   7   8  17  25  32
  9  14  24  36  50  59
  9  22  39  51  67  77
 10  25  47  62  82 100
 17  38  60  83 112 132
```

ซึ่งวิธีการบวกเช่นนี้อยู่ในไฟล์ range_sum_2d.cpp

โค้ดส่วนการบวกข้อมูลเข้าอาเรย์เก็บคำตอบชั่วคราวในไฟล์ range_sum_2d.cpp

```
int input[5][6] = {
  {4, 3, 1, 9, 8, 7},
  {5, 2, 9, 3, 6, 2},
  {0, 8, 7, 0, 2, 1},
  {1, 2, 5, 3, 4, 8},
  {7, 6, 0, 8, 9, 2},
};

int temp[5][6];  /// Array for intermediate results

/// Fill the first row
temp[0][0] = input[0][0];
for(int c = 1; c < 6; ++c)
  temp[0][c] = temp[0][c-1] + input[0][c];

/// Fill other rows
for(int r = 1; r < 5; ++r) {
  int row_sum = 0;

  /// We combine the sum of the current row and
  ///   the sum of all preceding rows.
  for(int c = 0; c < 6; ++c) {
    row_sum += input[r][c];
    temp[r][c] = row_sum + temp[r-1][c];
  }
}


/// The following part is for studying the behavior of the method.
/// Print out the intermediate outputs to see what's going on inside.
printf("temp:\n");
for(int r = 0; r < 5; ++r) {
  for(int c = 0; c < 6; ++c) {
    printf("%3d ", temp[r][c]);
  }
  printf("\n");
}
```

เมื่อเราได้อาเรย์เก็บคำตอบชั่วคราวมาแล้ว ก็ถึงเวลาที่เราจะกล่าวถึงวิธีใช้มันหาคำตอบที่ต้องการ กำหนดให้ A เป็นอาเรย์ชั่วคราวที่นับแถวและคอลัมน์จากหนึ่ง และสมมติว่าเราต้องการหาผลบวกจากแถวที่ 2 คอลัมน์ที่ 3 ไปจนถึงแถวที่ 4 คอลัมน์ที่ 6 เราจะเห็นได้ว่าข้อมูลในอาเรย์ที่เราคำนวณไว้ตรงตำแหน่งแถวที่ 4 คอลัมน์ที่ 6 ซึ่งก็คือ A[4][6] มีผลบวกส่วนเกินที่ไม่เกี่ยวข้องจากแถวที่ 1 ทั้งหมด (คอลัมน์ที่ 1 ถึง 6) ซึ่งก็คือ A[1][6] นอกจากนี้ยังมีผลบวกส่วนเกินจากคอลัมน์ที่ 1 และ 2 (แถวที่ 1 ถึงแถวที่ 4) ซึ่งก็คือ A[4][2] ดังนั้นถ้าเราลบผลบวกที่ไม่เกี่ยวข้องพวกนี้ออกไป เราก็จะได้คำตอบที่เราต้องการ

พิจารณาการลบส่วนเกินด้วยนิพจน์ A[4][6] – A[1][6] – A[4][2] เราจะพบว่ามีปัญหากับการทเช่นนี้เพราะเราลบบางส่วนซ้ำกันสองครั้ง ซึ่งส่วนดังกล่าวก็คือแถวแรกตรงคอลัมน์ที่หนึ่งและสอง เนื่องจากตำแหน่งสองอันนี้ต่างก็ถูกรวมอยู่ใน A[1][6] และ A[4][2] ปัญหาการลบเกินนี้ สามารถแก้ได้ด้วยการบวกส่วนที่ลบซ้ำกลับเข้าไปคืน ทำให้เราได้นิพจน์สำหรับการคำนวณค่าเป็น A[4][6] – A[1][6] – A[4][2] + A[1][2]

หากเราต้องการหาผลบวกของพื้นที่อื่น ๆ ในอาเรย์สองมิติ เราก็ทำได้ในลักษณะเดียวกัน ทั้งนี้การคำนวณของตำแหน่งอื่น ๆ จะมีการลบสองครั้งและการบวกหนึ่งครั้งเท่าเป็นอย่างมาก และหากเราเปลี่ยนข้อกำหนดของปัญหาให้จัดการกับพื้นที่ขนาดใหญ่ขึ้น เราก็จะพบว่าภาระในการคำนวณแทบจะไม่มีอะไรเปลี่ยนแปลงเลย แต่ก่อนที่จะลองทำเรื่องที่ซับซ้อนระดับนั้น ลองมาดูวิธีเขียนโค้ดสำหรับการใช้อาเรย์เก็บผลลัพธ์ชั่วคราวในการหาผลบวกในพื้นที่ก่อน

โค้ดส่วนการหาผลบวกในพื้นที่จากอาเรย์เก็บคำตอบชั่วคราวในไฟล์ range_sum_2d.cpp
หมายเหตุ โปรแกรมทำการหาผลบวกเฉพาะพื้นที่ที่มีจำนวนสามแถวและสองคอลัมน์ครบและพิมพ์ผลลัพธ์ออกมาด้วย

```
/// Compute range sum and print it out
printf("sum:\n");
for(int r = 2; r < 5; ++r) {
  for(int c = 3; c < 6; ++c) {
    int sum = temp[r][c];
    if(r > 2) sum -= temp[r-3][c];
    if(c > 3) sum -= temp[r][c-4];
    if(r > 2 && c > 3) sum += temp[r-3][c-4];
    printf("%d ", sum);
  }
  printf("\n");
}
```

ทดลองใช้อัลกอริทึม Range Sum ในอาเรย์สองมิติแก้ปัญหา

คราวนี้เราจะมาลองใช้อัลกอริทึมที่เสนอมาในการแก้ปัญหาที่สมจริงมากขึ้น ตัวอย่างของปัญหาที่จะเสนอในที่นี้คือโจทย์ข้อ 'หาทำเลที่ตั้ง (Location)' ซึ่งเป็นโจทย์ในการแข่งขันคอมพิวเตอร์โอลิมปิกวิชาการ สอวน. ที่มหาวิทยาลัยศิลปากรเป็นเจ้าภาพในปี 2555 แต่ในที่นี้ปรับแต่งเล็กน้อย [ดูเอกสารแนบ problem_location.pdf]

ในปัญหานี้ขนาดของพื้นที่ที่ต้องทำการบวกจะถูกกำหนดมาเป็นพารามิเตอร์ของปัญหา แต่เราก็จะเห็นได้ว่าภาระงานคำนวณไม่ได้ขึ้นอยู่กับขนาดของพื้นที่เลย โค้ดที่ใช้ก็มีลักษณะเกือบจะเหมือนเดิมทุกประการ ถ้าหากใครต้องการศึกษาที่จุดนี้ก็ดูได้จากไฟล์ location_toi.cpp แต่ถ้าหากอยากได้โจทย์ในรูปแบบ ACM ICPC ขอให้ดูตัวโจทย์จาก problem_location_acm_style.pdf และดูโค้ดจาก location_acm.cpp

อีกปัญหาที่มีการใช้งานจริง (แต่ไม่ขอนำมาเป็นตัวอย่างในที่นี้เพราะจัดเอกสารและตัวอย่างข้อมูลไม่ทัน) ก็คือการหาค่าเฉลี่ยในพื้นที่ของภาพซึ่งมีอยู่ในวิชา Image Processing ซึ่งถ้าใช้ตัวกรองภาพแบบพื้นฐานที่สุดจะใช้เวลา $O(M\,N\,K^2)$ เมื่อ M, N และ K คือความสูงและความกว้างของภาพ และ ความกว้างของตัวกรองตามลำดับ แต่หากใช้ตัวกรองแบบแยกสองส่วนได้ก็จะเร็วขึ้นเป็น $O(M\,N\,K)$ และสุดท้ายหากใช้วิธี Range Sum ก็จะเร็วที่สุดเป็น $O(M\,N)$ คือความเร็วที่ได้ไม่ขึ้นอยู่กับขนาดของตัวกรองเลย

Maximum Sum of Contiguous Subsequence กับอาเรย์หนึ่งมิติ

ปัญหานี้ต้องการหาผลบวกของช่วงอาเรย์ที่ติดโดยพยายามทำให้ผลบวกที่ติดกันมีค่ามากที่สุด ในตอนแรกเราอาจจะคิดว่าทำไมไม่บวกอาเรย์ทั้งหมดไปเลย คำตอบก็คือว่าถ้าหากอาเรย์มีเลขติดลบอยู่ด้วย การบวกอาเรย์ทั้งหมดอาจจะให้ค่าที่ผิดได้ เช่น จากอาเรย์ 3, -9, 1, 1, 2, -3, 3, -1, 2, -5, 3, 1, -2, 1 ถ้าเราบวกเลขทั้งหมดเข้าด้วยกันจะได้ค่าติดลบ ในขณะที่การบวกของช่วง 1 + 1 + 2 – 3 + 3 – 1 + 2 จะให้ค่าที่มากที่สุดคือ +5

อุปสรรคของการคิดแก้ปัญหานี้ก็คือว่า ไม่รู้ว่าควรจะเริ่มช่วงค่าที่อาเรย์ช่องใด และ ไม่รู้ว่าควรจะให้ช่วงอาเรย์ยาวติดต่อกันเท่าไหร่ถึงจะดีที่สุด ถ้าหากเราจะหาผลบวกที่เป็นไปได้แบบตรง ๆ ทุกรูปแบบ การคำนวณของเราก็จะมีลักษณะเป็นลูปสามชั้นดังนี้ [โค้ดดัดแปลงมาจากไฟล์ max_cont_subseq_naive.cpp]

```
int max_sum = INT_MIN;
for(int i = 0; i < n; ++i) {
  for(int j = i; j < n; ++j) {
    int sum = 0;
    for(int k = j; k <= i; ++k) {
      sum += input[k];
    }
    if(sum > max_sum)
      max_sum = sum;
    }
}
```

จากการคำนวณทางด้านบน ลูปชั้นนอกระบุตำแหน่งเริ่มต้นของช่วงอาเรย์ที่จะบวก ลูปชั้นที่สองระบุตำแหน่งสิ้นสุดของช่วงอาเรย์ ส่วนลูปชั้นที่สามทำการบวกจากตำแหน่งเริ่มต้นถึงตำแหน่งสิ้นสุด

วิธีการข้างบนจัดเป็นวิธีการที่ช้ามาก เราสามารถนำวิธี Range Sum มาประยุกต์ช่วยแก้ปัญหาได้ แม้วิธีนี้จะไม่เร็วที่สุดแต่เป็นวิธีการที่ควรเรียนรู้ไว้เช่นกัน การใช้ Range Sum ในที่นี้เราจะเริ่มจากการกำหนดให้ความยาวช่วงอาเรย์ที่ติดกันมีค่าเท่ากับหนึ่ง จากนั้นเราจะใช้ Range Sum หาผลบวกที่ความยาวนี้ เสร็จแล้วก็จะเปลี่ยนความยาวช่วงอาเรย์ให้เป็นสองแล้วทำแบบเดียวกัน และค่อย ๆ เพิ่มค่าความยาวช่วงไปเรื่อย ๆ จนความยาวช่วงเท่ากับความยาวอาเรย์ วิธีนี้โค้ดจะลดเหลือเป็นลูปสองชั้น ดังแสดงข้างล่างนี้ [โค้ดดัดแปลงมาจาก max_cont_subseq_not_so_naive.cpp]

```
int sum = 0;
for(int i = 0; i < n; ++i) {
  sum += input[i];
  sumTo[i] = sum;
}

int max_sum = INT_MIN;
for(int i = 1; i <= n; ++i) {        // range size
  for(int j = 0; j <= n-i; ++j) {    // start index
    int cont_sum = sumTo[j+i-1];
    if(j > 0) cont_sum -= sumTo[j-1];
    if(cont_sum > max_sum)
      max_sum = cont_sum;
    }
}
```

แต่วิธีที่เร็วกว่านี้มากก็มีอยู่ ซึ่งถูกนำเสนอโดย Jay Kadane ในปี 1984 วิธีการนี้ตั้งอยู่บนข้อสังเกตที่ว่าเราสามารถทำการบวกค่าอาเรย์จากจุดเริ่มต้นมาได้เรื่อย ๆ พร้อมกับเก็บผลบวกที่มากที่สุดเอาไว้ เมื่อใดก็ตามที่ผลบวกลดลงต่ำกว่าศูนย์ แสดงว่ามันดีกว่าที่จะลองเริ่มต้นใหม่จากจุดที่ติดลบนั้น (เป็นเหมือนการ reset การคำนวณทุกอย่างแต่เปลี่ยนจุดเริ่ม)

ตัวอย่าง กำหนดอาเรย์ 3 -2 4 -1 2 -7 4 -1 2 3 -5 3 1 -2 1 เราจะหาผลบวกในช่วงอาเรย์ที่ติดกันที่ทำให้ได้ค่าที่มากที่สุด

หากเราใช้วิธีของ Kadane เราจะเก็บผลบวกจากจุดเริ่มจนถึงจุดปัจจุบันไว้จนกว่าจะ 'หมดหวัง' จึงทำการเริ่มต้นการบวกใหม่แต่เปลี่ยนตำแหน่งเริ่ม ซึ่งอธิบายกับตัวอย่างได้ดังนี้

1. กำหนดจุดเริ่มคืออาเรย์ช่องที่หนึ่ง และ กำหนดตำแหน่งปัจจุบันคืออาเรย์ช่องที่หนึ่ง
2. ผลบวกจากจุดเริ่มถึงตำแหน่งปัจจุบันคือ 3
3. เลื่อนตำแหน่งปัจจุบันไปช่องที่สอง
4. ผลบวกจากจุดเริ่มถึงตำแหน่งปัจจุบันคือ 3 – 2 = 1
5. เลื่อนตำแหน่งปัจจุบันไปช่องที่สาม
6. ผลบวกจากจุดเริ่มถึงตำแหน่งปัจจุบันคือ 3 – 2 + 4 = 5 จากนั้นเลื่อนตำแหน่งปัจจุบันไปช่องที่สี่
7. ผลบวกจากจุดเริ่มถึงตำแหน่งปัจจุบันคือ 3 – 2 + 4 – 1 = 4 จากนั้นเลื่อนตำแหน่งปัจจุบันไปช่องที่ห้า
8. ผลบวกจากจุดเริ่มถึงตำแหน่งปัจจุบันคือ 3 – 2 + 4 – 1 + 2 = 6 จากนั้นเลื่อนตำแหน่งปัจจุบันไปช่องที่หก
9. ผลบวกจากจุดเริ่มถึงตำแหน่งปัจจุบันคือ 3 – 2 + 4 – 1 + 2 – 7 = -1

ณ ขั้นตอนที่ 9 นี้เอง เราพบว่าผลบวกมันติดลบแล้ว การจะเก็บค่านี้แล้วบวกต่อไปมีแต่จะทำให้เสียประโยชน์ สู้เริ่มต้นใหม่เสียยังจะดีกว่า ดังนั้นในขึ้นตอนต่อไปจะเป็นดังนี้

10. กำหนดจุดเริ่มคืออาเรย์ช่องที่เจ็ด และ กำหนดตำแหน่งปัจจุบันคืออาเรย์ช่องที่เจ็ด

11. ผลบวกจากจุดเริ่มถึงตำแหน่งปัจจุบันคือ 4 จากนั้นเลื่อนตำแหน่งปัจจุบันไปช่องที่แปด

ลองพิจารณาขั้นตอนที่ 11 ดูก็จะพบว่าถ้าหากเรายังเก็บผลบวกที่ติดลบไว้แล้วทำต่อ ขั้นตอนที่ 11 นี้จะได้ผลลัพธ์เป็น 3 ซึ่งน้อยกว่าการเริ่มต้นใหม่ จะเห็นได้ว่าถ้าผลบวกมันติดลบแล้วมัน 'หมดหวัง' แล้วจริง ๆ ในขณะที่การบวกในขั้นตอนที่ 4 ถึง 8 เราจะเห็นได้ว่าถ้าผลบวกยังไม่ติดลบการพยายามบวกเพิ่มเข้าไปย่อมทำประโยชน์ได้ และบางทีมันก็นำไปสู่การได้ค่าผลบวกที่ดีขึ้นด้วย ในตัวอย่างนี้หากเราทำซ้ำกระบวนการนี้ไปเรื่อย ๆ เราจะพบว่าผลบวกค่ามากที่สุดที่เราพบในระหว่างการบวกคือ 8 ซึ่งได้จาก 4 – 1 + 2 + 3

จุดสังเกตของ Kadane นี้นำไปสู่โค้ดที่เรียบง่ายและมีประสิทธิภาพสูงมากคือเหลือเพียงลูปชั้นเดียว ดังแสดงในโค้ดข้างล่าง [โค้ดดัดแปลงมาจากไฟล์ max_cont_subseq_kadane.cpp]

```
int max_sum = 0;
int max_end_here = 0;

for(int i = 0; i < n; ++i) {
  if(max_end_here + input[i] < 0)
    max_end_here = 0;
  else {
    max_end_here += input[i];
    if(max_end_here > max_sum)
      max_sum = max_end_here;
  }
}
```

Maximum Sum of Contiguous Subsequence กับอาเรย์สองมิติ

อัลกอริทึมของ Kadane สามารถนำไปประยุกต์ใช้กับอาเรย์สองมิติได้เช่นกัน แต่กระบวนการคิดมันจะซับซ้อนขึ้น เพื่อความสะดวกเราจะเรียนรู้จากปัญหา UVa 108 Maximum Sum ซึ่งตัวอย่างข้อมูลสำหรับทำการทดสอบคือ

```
 0 -2 -7  0
 9  2 -6  2
-4  1 -4  1
-1  8  0 -2
```

เราจะเริ่มจากการทดสอบว่า “ถ้าแถวเริ่มต้นในการหาผลบวกสองมิติคือแถวแรก (คือแถวแรกต้องมีส่วนร่วมด้วยเสมอ) ผลบวกในพื้นที่สองมิติที่มากที่สุดคือเท่าใด” จากนั้นก็จะเปลี่ยนไปทดสอบว่า “ถ้าแถวเริ่มต้นในการหาผลบวกสองมิติคือแถวที่สอง (คือแถวที่สองต้องมีส่วนร่วมด้วยเสมอและแถวแรกไม่เกี่ยวข้องในการคำนวณแล้ว) ผลบวกในพื้นที่สองมิติที่มากที่สุดคือเท่าใด” จากนั้นก็วนทำการทดสอบเช่นนี้ไปเรื่อย ๆ เวลาที่ทำแบบนี้ เราจะหยิบเอาวิธีของ Range Sum และข้อสังเกตของ Kadane มาใช้ในระดับคอลัมน์ เช่น ถ้าแถวเริ่มต้นคือแถวที่สองและแถวสิ้นสุดที่กำลังพิจารณาคือแถวที่แปด เราจะเริ่มด้วยการกำหนดคอลัมน์เริ่มต้นเป็นคอลัมน์ที่หนึ่ง จากนั้นดูว่าการบวกในระดับคอลัมน์จากแถวที่สองถึงแปดในคอลัมน์แรกติดลบหรือไม่ ถ้าไม่ติดลบแสดงว่า 'ยังมีความหวัง' เราจะบวกคอลัมน์ที่สองจากแถวในชุดเดียวกันเข้ามาเสริม (การหาผลบวกตรงนี้วิธีคล้าย ๆ กับ Range Sum) ถ้าหากติดลบเราก็จะเริ่มต้นใหม่เพราะถือว่าหมดหวังแล้ว แต่ถ้าหากไม่ติดลบก็จะพยายามบวกต่อไปเรื่อย ๆ วิธีนี้ใช้เวลาเป็น $O(N^3)$

เราควรทำข้อนี้ด้วยตัวเอง เพราะมันเป็นแบบฝึกหัดทดสอบความเข้าใจของแนวคิด Range Sum และ Maximum Sum of Contiguous Subsequence ในคราวเดียว แต่หากใครคิดไม่ออกจริง ๆ ต้องการเฉลยมาศึกษา ก็ลองดูได้จากไฟล์ uva_108_maximum_sum_kadane.cpp ซึ่งมีคอมเมนต์อธิบายวิธีคิดไว้ด้วย แต่โค้ดที่ใช้อธิบายนั้นจะ 'ยาวกว่าปรกติ' เพื่อให้คิดตามได้ง่าย และโค้ดยังมีการพิมพ์ค่า intermediate results ด้วย (สังเกตการใช้ #define DEBUG ในข้อนี้ด้วย เพราะมันเป็นวิธีที่ช่วยเราได้ทั้งในการเขียนโค้ดจริงและโค้ดแข่งขัน) ถ้าอยากรู้ว่าแบบโค้ดแบบสั้น ๆ มันเป็นยังไง ให้ดูที่ไฟล์ uva_108_maximum_sum_kadane_v2.cpp

เรื่องน่าสนใจของข้อ UVa 108 อีกอย่างก็คือว่า คนออกแบบโจทย์อนุญาตให้ใช้วิธี Range Sum เพื่อแก้ปัญหาได้โดยไม่ต้องใช้วิธีของ Kadane ดังนั้นถ้าใครอยากลองใช้ Range Sum กับปัญหาสองมิติอีกรอบเพื่อทดสอบความเข้าใจ นี่ก็เป็นโอกาสอันดีแล้ว **หมายเหตุ** วิธีนี้ทำให้ภาระการคำนวณเป็น $O(N^4)$

การประยุกต์อื่น ๆ ของ Maximum Sum of Contiguous Subsequence

บางทีปัญหาที่เราต้องแก้มันก็ไม่ได้ออกมาในรูปแบบที่โจ่งแจ้งเหมือน UVa 108 แต่มันซ่อนตัวมาให้เราค้นหาการประยุกต์ใช้ที่เหมาะสมและบังคับให้เราต้องดัดแปลงโค้ดให้อยู่ในรูปที่เหมาะสมด้วย เนื้อหาในส่วนนี้จะรวบรวมตัวอย่างประยุกต์ใช้งาน Kadane's Algorithm ที่น่าจดจำเอาไว้ [ตัวอย่างการประยุกต์คงเพิ่มขึ้นเรื่อย ๆ แต่ตอนนี้มีแค่อันเดียว]

1. UVa 10667 Largest Block เป็นการประยุกต์เพื่อหาพื้นที่สี่เหลี่ยมมุมฉากที่ปราศจากสิ่งกีดขวางและมีพื้นที่มากที่สุด

Maximum Sum of Multiple Contiguous Subsequences

[อันนี้ลำดับย่อยที่ติดกันแบ่งเป็นหลาย ๆ ชุดได้ แต่คงต้องเรียบเรียงเนื้อหาอีกนานกว่าจะเขียนเสร็จ]