รายงานวิชา 2110327 Algorithm Design

ภาคการศึกษา 2023/3

Array Mode

midterm66

พีรณัฐ กิตติวิทยากุล

6330374121

โจทย์

หาค่าฐานนิยมของ input array ที่เรียงแล้วด้วยวิธี divide and conquer

Input: vector ของจำนวนเต็มที่เรียงกันแล้ว

Output: ค่า mode (ฐานนิยม) ของจำนวนชุดที่ส่งไป

Version 1: แบ่งครึ่งตรงกลาง

Code ของโจทย์ข้อนี้ใช้วิธี divide and conquer คล้าย ๆ กับ maximum subarray sum โดยจะแบ่งเป็นปัญหาใหญ่เป็นปัญหาย่อยที่เล็กลงโดยวิธีแรกคือการแบ่งครึ่งตรงกลาง โดยส่วนครึ่งซ้ายและครึ่งขวาคิดเหมือนกับ maximum subarray sum คือแบ่งตรงกลาง m = (start + stop) / 2 ส่วนที่ต่างกันคือการคิด r3 ที่เป็นค่า mode ที่เกิดขึ้นหากจำนวนนั้นถูกแบ่งครึ่งที่ m ทำให้จำนวนในครึ่งซ้ายไม่พอเอาชนะจำนวนที่ทั้งหมดอยู่ฝั่งซ้าย และจำนวนในครึ่งขวาไม่พอเอาชนะจำนวนที่ทั้งหมดอยู่ฝั่งขวา ดังนั้นกรณีที่จำนวนตรงกลางมีโอกาสเอาชนะ r1, r2 จากสองฝั่งได้คือถูกหั่นตรงกลางพอดี

วิธีการคิดส่วน r3 ของตรงกลางคือนับจำนวนค่าใน array ที่ถัดจากตรงกลางไปทั้งทางด้านซ้ายและด้านขวาจนเจอตัวเลขอื่น จะได้ค่าความถี่ของจำนวนที่ถูกหั่นที่ตรงกลางเป็นค่าในส่วน r3 นำไปเทียบกับ r1 และ r2 เพื่อนำมา conquer รวมคำตอบ

Code version 1

```
5  pair<int,int> array_mode_middle(vector<int> &v, int start, int stop) {
6      if (start == stop) return make_pair(1, v[start]);
7      int m = (start+stop)/2;
8  
9      pair<int,int> r1 = array_mode_middle(v, start, m);
10     pair<int,int> r2 = array_mode_middle(v, m+1, stop);
11     int r1f = r1.first;
12     int r2f = r2.first;
13 
14     int i=m,j=m;
15     while (i > start && v[i-1] == v[m]) i--;
16     while (j < stop && v[j+1] == v[m]) j++;
17 
18     int r3f = j-i+1;
19 
20     if(r3f > r1f && r3f > r2f) {
21         return make_pair(r3f, v[m]);
22     }
23     else if(r1f > r2f) return r1;
24     else return r2;
25 }
```

ฟังก์ชั่น array_mode_middle รับ vector และ index start, stop โดยจะ return เป็น pair<int,int> โดยค่าแรกคือค่าความถี่ซึ่งเป็นจำนวนครั้งที่ปรากฏ ค่าหลังคือค่าฐานนิยม (Mode) หรือตัวเลขนั้น ๆ

line 6 – 10 เหมือนกับ maximum subarray sum ในการคิดฝั่งซ้ายและขวา

line 14 – 18 เป็นการนับจำนวนความถี่ของจำนวนที่ถูกแบ่งตรงกลางโดยนับไปทั้งซ้ายและขวา

line 20 – 24 จะเป็นการเทียบว่าค่าฐานนิยมจากก้อนไหนใน 3 ก้อนมีความถี่มากที่สุดจึงส่งออก

Function Time Complexity Analysis

- Base case: line 6 => O(1)
- Recursive step: line 7 – 10 => call subproblem recursively on each half
- Middle step: line 14 – 18 => ใน worst case อาจจะวิ่งทั้ง array => O(n) , n=stop-start+1
- Conquering step : line 20 – 24 => O(1)

เนื่องจากเป็นการเรียกฟังก์ชั่นแบบ recursive call โดยทุกการเรียกขนาดของ vector ที่จะไปทำงานต่อลดลงครึ่งหนึ่ง และมีงานที่ต้องทำจริง O(n) ทุกครั้งจึงเขียน T(n) ได้ T(n)=2T(n/2)+O(n) และจาก master method จะได้ O(nlogn)

Factors affected Time Complexity

เมื่อลองพิจารณา Time Complexity O(nlogn) แล้ว ส่วนที่เป็น divide subproblem ครึ่งหนึ่งนั้นจะทำงานเสมอในทุกกรณีทำให้ตรง 2T(n/2) ทำงานแน่นอน แต่ส่วนที่เป็นงานที่ทำจริงแต่ละการ call คือ Middle step: line 14 – 18 ที่ทำงาน O(n) ส่วนนี้ของโปรแกรมจะทำให้ปัจจัยความเร็วของโปรแกรมแตกต่างกันเพราะตรง O(n) นี้สามารถเป็นได้ทั้งการทำงานที่เร็วมาก ๆ คือวน while loop ไม่กี่ครั้งก็หมด หรืออาจจะวนเยอะมากได้ถึง n รอบ

ดังนั้น input กรณีที่ทำแล้วเร็วสำหรับกรณีนี้คือ input ที่ทำ line 14 – 18 น้อย ๆ จะทำให้ O(n) เข้าใกล้ time constant นั่นคือหั่นตรงกลางแล้วตัวตรงกลางที่โดนความถี่น้อย ๆ

เมื่อลองเทียบ vector {1 2 2 2 2 2 2 3} หั่นตรงกลางเจอ 2 แล้วต้องวนนับจำนวนเลข 2 ซ้ายขวาถึง 6 ตัวจาก 8 ตัว เมื่อแบ่งครึ่งปัญหา ในปัญหาย่อยชั้นถัดไปก็ยังหั่นกลางที่ 2 แล้วนับซ้ายขวา 3 ตัว

Input ที่วิธีนี้คำนวนได้เร็วคือ input ที่ไม่มีตัวซ้ำกันเลย เช่น {1 2 3 4 5 6} input แบบนี้จะทำให้ส่วน while loop แค่เช็คว่าค่าไม่เท่ากันแล้วออกเลย ไม่ต้องวนใน while loop กลายเป็น O(1)

ส่วน input ที่วิธีนี้คำนวนได้ช้าคือ input ซ้ำกันตรงกลางเยอะหรือซ้ำกันหมด เช่น {2 2 2 2 2 2} input แบบนี้จะทำให้แบ่งกี่รอบ ในปัญหาย่อยก็วน while loop จนสุดขอบ vector เป็น Theta(n)

Version 2: แบ่งครึ่งตรงกลางแล้วขยับจนกว่าจะเจอรอยต่อ

วิธีของ version นี้จะใช้ divide and conquer เหมือนกัน และจะเอา code จาก version ที่ 1 มาดัดแปลง วิธีนี้จะเริ่มจากค่า m ที่เป็นค่าตรงกลางก่อน m = (start + stop) / 2 แต่ที่ต่างจากเดิมคือเราจะไม่ใช้ m นี้จริง ๆ แต่เราจะขยับ m ให้ไปอยู่ตำแหน่งรอยต่อของเลข 2 เลขที่ต่างกัน ด้วยการแบ่งที่รอยต่อนี้ทำให้ค่า mode ที่เกิดจากการที่จำนวนที่มีความถี่เยอะ ๆ ถูกแบ่งที่ index m ทำให้ไม่ชนะทั้งฝั่งซ้ายและขวาไม่เกิดขึ้นเพราะเราจะแบ่งตรงรอยต่อพอดี

วิธีนี้จึงไม่มีการคิด r3 มีแค่ r1 และ r2 แต่จะมีการคิดค่า m ใหม่เพิ่มขึ้นมา

Code version 2

```cpp
pair<int,int> array_mode_crack(vector<int> &v, int start, int stop) {
    if (start == stop) return make_pair(1, v[start]);
    // if subarray all contains the same number
    if (v[start] == v[stop]) return make_pair(stop - start + 1, v[start]);

    int m = (start+stop)/2;
    if (v[m] == v[m+1])
    {
        int i=1,j=1;
        while (i < (m - start + 1) && v[m-i] == v[m]) i++;
        while (j < (stop - m + 1) && v[m+j] == v[m]) j++;
        i--; j--;
        if (i < j) m = m-i;
        else m = m+j;
    }

    pair<int,int> r1 = array_mode_crack(v, start, m);
    pair<int,int> r2 = array_mode_crack(v, m+1, stop);

    return (r1.first > r2.first)? r1 : r2;
}
```

ฟังก์ชั่น array_mode_crack รับพารามิเตอร์และ return pair เหมือน version 1 แต่จะมี 2 ส่วนที่เปลี่ยนจาก version 1

line 30 เป็น base case อีก case ซึ่งดักกรณีที่แบ่งจนได้ช่วงที่เป็นจำนวนเดียวกันทั้งช่วง ไม่สามารถขยับ m เพื่อหารอยแบ่งได้อีกแล้ว ให้ return เป็นค่าความถี่นั่นคือ stop – start – 1 กับจำนวนนั้น ซึ่งถ้าตามกลไกของอัลกอริทึ่ม base case line 28 ก็ไม่จำเป็น สามารถละได้

line 33 – 41 เป็นการขยับ m เพื่อให้ไปอยู่ที่รอยต่อของจำนวนที่ต่างกัน โดยถ้าตรง m ไม่ใช่รอยต่อโดยกรองได้ใน line 33

line 36 คือนับไปทางซ้าย แล้ว increment i เรื่อย ๆ จนเจอรอยต่อ

line 37 คือทำเหมือนกันแต่นับไปทางขวา แล้วใช้ j เป็นระยะทางแทน i

line 39 – 40 จะเทียบกันว่าฝั่งไหนใกล้กว่าให้ไปรอยต่อฝั่งนั้น ทำให้ได้ค่า m ที่รอยต่อจำนวน

line 43 – 46 เป็นการ divide เป็น 2 subproblems แล้วนำค่าเทียบและ return เหมือนเดิมแต่ไม่มี r3 แล้วในฟังก์ชั่นนี้

Function Time Complexity Analysis

- Base case, Subarray check : line 28 – 30 => O(1)
- Middle adjustment step: line 32 – 41 => ใน worst case อาจจะวิ่งทั้ง array ครบซ้ายขวา => O(n) , n=stop-start+1
- Recursive step : line 43 – 44 => divide problem into m, n – m

Actual work คือ O(n) และแบ่งปัญหาเป็นจำนวน m และ n – m จึงเขียน T(n) ได้ดังนี้ $T(n)=T(m)+T(n-m)+O(n)$ ซึ่งต่างจาก version แรกที่เป็น $T(n)=2T(n/2)+O(n)$

ถ้าเราต้องการคำนวน big O ของ T(n)=T(m)+T(n – m)+O(n) ก็จะต้องพิจารณาค่า m ซึ่งมีค่าได้ตั้งแต่ 0 ถึง n ขึ้นกับการกระจายของข้อมูล โดยการวิเคราะห์ big O notation จะทำแบบ quick sort ทำให้ได้ดังนี้

- Worst case T(n) = T(n-1) + T(1) + O(n) => O(n^2)
- Best case T(n) = 2T(n/2) + O(n) => O(nlogn) *excluding base case
- Average case T(n) = n(ln(n)) + n => Theta(nlogn)

Factors affected Time Complexity

เมื่อลองพิจารณา Time Complexity T(n)=T(m)+T(n – m)+O(n) แล้ว ส่วนที่ส่งผลต่อความเร็วในการทำงานมี 2 ส่วนคือ ส่วนการขยับ m (Middle adjustment step : line 32 – 41) ซึ่งเป็นงานที่ทำจริงในแต่ละ recursive call ยิ่งรอยต่อของจำนวนอยู่ไกล ยิ่งต้องวน loop บ่อย ยิ่งใช้เวลานาน อีกส่วนคือการแบ่ง subproblem ที่ไม่เท่ากันที่ T(m)+T(n – m) หากได้ best case คือ m = n/2 สอดคล้องกับไม่ค่อยขยับ n ในส่วน Middle adjustment step จะทำให้ big O เร็วที่สุดคือ O(nlogn) แต่ถ้าโชคร้ายขยับ m ไปสุดขอบ ยิ่ง m เข้าใกล้ 1 หรือ n – 1 มากเท่าไหร่ time complexity ก็จะเข้าใกล้ O(n^2) มากขึ้น

ดังนั้น input กรณีที่ทำแล้วเร็วสำหรับกรณีนี้มี 2 แบบ

1. input ที่ทำ Middle adjustment step : line 32 – 41น้อย ๆ จะทำให้เกิด O(nlogn) ที่เป็น best case นั่นคือข้อมูลที่ Evenly Distributed Midpoints ทำให้ m ที่ได้ไม่ทำให้การเรียก recusion เกิดความ unbalance เช่น {1, 1, 1, 1, 2, 2, 2, 2} หั่นตรงกลางก็เป็นรอยต่อระหว่าง 1 กับ 2 เลย ไม่ต้องขยับ m และแบ่งได้ที่ n/2 พอดี พอแบ่งแล้วก็เข้า base case ที่ค่าตัวแรกกับตัวสุดท้ายเหมือนกัน return โดยไม่ต้องทำต่อ
2. อีกประเภทคือ input ที่เข้ามาแล้วโดน base case line 30 ที่ค่าตัวแรกกับตัวสุดท้ายเหมือนกันกรองออกทันที นั่นคือ array ของเลขเดียวกันทั้งหมด เช่น {1, 1, 1, 1, 1, 1, 1, 1} จะทำให้กลายเป็น O(1) ทันที

Version 3: แบ่งรอยต่อแรกจากซ้ายสุด

วิธีของ version นี้จะเอา code จาก version ที่ 2 มาดัดแปลง แต่วิธีนี้จะเริ่มจากค่า m ที่เป็นค่าตรงกลางแล้ว แต่จะเริ่มจาก k = start เลย แล้วขยับ k ไปอยู่ตำแหน่งรอยต่อของเลข 2 เลขที่ต่างกันรอยแรกที่อยู่ซ้ายสุด วิธีนี้ไม่มีการคิด r3 มีแค่ r1 และ r2 เช่นกัน

Code version 3

```cpp
48 pair<int,int> array_mode_leftcrack(vector<int> &v, int start, int stop) {
49     // if subarray all contains the same number
50     if (v[start] == v[stop]) return make_pair(stop - start + 1, v[start]);
51 
52     // find left crack
53     int k = start;
54     while ((k < stop) && (v[k+1] == v[start])) {
55         k++;
56     }
57 
58     pair<int,int> r1 = array_mode_leftcrack(v, start, k);
59     pair<int,int> r2 = array_mode_leftcrack(v, k+1, stop);
60 
61     return (r1.first > r2.first)? r1 : r2;
62 }
```

ฟังก์ชั่น array_mode_leftcrack รับพารามิเตอร์และ return pair เหมือนเดิม ส่วนที่ต่างจาก version 2 ที่หารอยต่อตรงกลางคือ line 53 – 56 ที่เป็นการหารอยต่อเริ่มจากฝั่งซ้ายของ vector แทน

โดยถ้าค่าของตัวที่ k+1 เป็นค่าเดียวกับ v[start] ก็จะขยับ k ไปเรื่อย ๆ จนกว่าจะเจอรอยต่อ เมื่อเจอรอยต่อแล้ว จะแบ่งเป็น 2 subproblems ได้ r1 กับ r2 แล้วเอาผลลัพธ์มาเทียบก่อน return กลับ

Function Time Complexity Analysis

- Base case : line 50 => O(1)
- Find left crack : line 53 – 56 => worst case อาจจะวิ่งทั้ง array จนเกือบสุดด้านขวา => O(n)
- Recursive step : line 58 – 59 => divide problem into k, n – k

จะได้ $T(n)=T(k)+T(n-k)+O(n)$ เหมือนกับ version ที่ 2

- Worst case $T(n) = T(n-1) + T(1) + O(n) \Rightarrow O(n^2)$
- Best case $T(n) = 2T(n/2) + O(n) \Rightarrow O(n\log n)$ *excluding base case
- Average case $T(n) = n(\ln(n)) + n \Rightarrow \Theta(n\log n)$

Factors affected Time Complexity

สำหรับ version ที่ 3 best case เลย คือ input ที่เข้ามาแล้วโดน base case line 50 ที่ค่าตัวแรกกับตัวสุดท้ายเหมือนกันกรองออกทันที นั่นคือ array ของเลขเดียวกันทั้งหมด เช่น {1, 1, 1, 1, 1, 1, 1, 1} จะทำให้กลายเป็น O(1) ทันที

ถัดมา input แบบที่ version 3 ทำงานเร็วคือ vector ที่มีรอยต่อหรือการเปลี่ยนจำนวนน้อย ๆ แล้วรอยต่ออยู่ใกล้ด้านซ้าย เช่น vector ที่มีรอยต่อเดียว จะลดความลึกของ recursion ให้เหลือน้อยลง เช่น {1, 2, 2, 2, 2, 2, 2, 2} ตัวอย่างนี้จะแบ่งรอยต่อระหว่าง 1 กับ 2 ครั้งเดียวแล้ว return ออกมาเลยไม่มีการ recursive call ซ้ำอีก

ส่วน input ที่ทำให้เกิด worst case คือ vector มีเลขซ้ำน้อย ๆ แล้วมีการเปลี่ยนเลขหรือรอยต่อถี่มาก ๆ จะยิ่งทำให้ฟังก์ชั่นเข้าใกล้ O(n^2) เช่น {1, 2, 3, 4, 5} จะมีการแบ่งที่รอยต่อทุก ๆ ค่าที่มีการขยับ index จะทำให้ version นี้ทำงานช้า

Conclusion

สำหรับ version 1 ที่แบ่งครึ่งตรงกลาง case ที่ทำได้เร็ว ๆ คือ หั่นตรงกลางโดนตัวความถี่น้อย ๆ เช่น {1 2 3 4 5 6} หั่นยังไงก็เจอความถี่ 1 ไม่ต้องนับตัวกลางว่าจะโดนหั่นมั้ย

Case ที่ version 1 ทำงานได้ช้าคือหั่นตรงกลางแล้วตัวตรงกลางที่โดนหั่นความถี่สูง ต้องวนนับไกลจาก middle เช่น {1 2 2 2 2 2 2 2 3} หรือ {2 2 2 2 2 2 2}

Version ที่ 2 และ 3 เป็นการแบ่งตรงรอยต่อเหมือนกัน สุดท้ายออกมา time complexity รูปเดียวกันคือ T(n)=T(m)+T(n – m)+O(n) และ T(n)=T(k)+T(n – k)+O(n) แต่ความแตกต่างคือ case ที่ version 2 ทำงานได้เร็วคือ m ไม่ต้องขยับจากตรงกลางเยอะ ชอบให้ m เป็นรอยต่ออยู่แล้วซึ่งตรงข้ามกับ version 1 พวกข้อมูลที่ Evenly Distributed Midpoints เช่น {1, 1, 1, 1, 2, 2, 2, 2} ที่หั่นไม่กี่รอบ ไม่ต้องขยับ m เยอะก็เข้า base case

ถ้าหากรอยต่อที่ต้องขยับจาก m อยู่ไกลมาก ๆ เช่น เกือบสุดซ้าย version 2 จะเสียเวลาวน loop ขยับ m แต่จะดีสำหรับ version ที่ 3 เพราะ vector ที่มีรอยต่อหรือการเปลี่ยนจำนวนน้อย ๆ แล้วรอยต่ออยู่ใกล้ด้านซ้ายจะลดความลึกของ recursion ให้เหลือน้อยลง เช่น {1, 2, 2, 2, 2, 2, 2, 2} ซึ่งแบ่งรอยต่อระหว่าง 1 กับ 2 ครั้งเดียวแล้วเจอ base case เลย

Finalized code: https://github.com/Peeranut-Kit/algorithm-design-coding/blob/main/problem/array_mode/array_mode.cpp