รายงานวิชา 2110327 Algorithm Design

ภาคการศึกษา 2023/3

Eating Futomaki

a66_q2a_eating_futomaki

พีรณัฐ กิตติวิทยากุล

6330374121

แนวทางการแก้ปัญหา

Version 1

Points	20.0/100
Comment	PPP-P---------------

int futomaki_score(vector<int> A, int n) version แรกจะเป็น function ที่รับข้อมูล input ที่เป็น vector ความอร่อย A และความยาว n ไปคำนวนแล้ว return ความความอร่อยที่จะได้กลับมา แนวคิดของ version นี้จะเป็นการใช้ brute force while loop ไล่ index left right แบบ naïve ซึ่งให้คำตอบที่ถูกต้องแค่ 4 จาก 20 testcases จึงเป็นอัลกอริทึ่มที่ไม่ถูกต้อง

<u>Summary on main()</u>

```cpp
int main(int argc, char const *argv[])
{
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    int n;
    cin >> n;
    vector<int> A(n);
    for (size_t i = 0; i < n; i++)
    {
        cin >> A[i];
    }
    cout << futomaki_score(A,n) << endl;
    return 0;
}

```

ตัวฟังก์ชั่น main ไม่ได้มีอะไรพิเศษนอกจากรับค่า n และ A ก่อนส่งไป futomaki_score()

แนวคิดของ futomaki_score_v1()

```cpp
int futomaki_score(vector<int> A, int n) {
    int score = 0;
    int left = 0, right = n - 1;
    while (left < right) {
        if (right - left > 1) {
            int bigger_left = max(A[left], A[left+1]);
            int bigger_right = max(A[right], A[right-1]);
            int smaller_left = min(A[left], A[left+1]);
            int smaller_right = min(A[right], A[right-1]);
            if ((smaller_left == smaller_right) && (bigger_left == bigger_right)) {
                score += A[left];
                left++;
                right--;
            }
            if (smaller_left > bigger_right) {
                score += A[left];
                left++;
                right--;
            }
            else if (smaller_right > bigger_left) {
                score += A[right];
                left++;
                right--;
            }
            else if (bigger_left > bigger_right) {
                if (A[left] < A[left+1]) {
                    score += A[left+1];
                    left += 2;
                }
                else {
                    if (A[left+1] < A[right])
                    {
                        score += A[left];
                        left += 2;
                    }
                    else {
                        score += A[left];
                        left++;
                        right--;
                    }
                }
            }
            else {
                if (A[right-1] > A[right]) {
                    score += A[right-1];
                    right -= 2;
                }
                else {
                    if (A[right-1] < A[left])
                    {
                        score += A[right];
                        right -= 2;
                    }
                    else {
                        score += A[right];
                        left++;
                        right--;
                    }
                }
            }
        }
        else {
            score += max(A[left++], A[right--]);
        }
    }
    return score;
}
```

แนวคิดคือจะวน while loop ตรวจสอบชิ้นซ้ายสุดและขวาสุดที่ยังเหลืออยู่จนกว่าจะหมดแท่ง โดยต่อไปนี้จะขอเรียกนายแด้ว่านาย ก และนายชินว่านาย ข ต้องการให้นาย ก ได้คะแนนเยอะที่สุด เราจึงมีแนวคิดว่าเราอยากให้ชิ้นที่คะแนนความอร่อยมากที่สุดในทุกการตัดที่เป็นไปได้ให้นาย ก กิน โดยการตัดสามารถตัดได้ทั้งหมด 3 แบบจึงมีทั้งหมด 4 ชิ้นที่สามารถกินได้ในแต่ละครั้ง คือ ซ้ายสุด ถัดจากซ้ายสุด ขวาสุด ถัดจากขวาสุด วิธีการทำคือหาว่าใน 4 ชิ้นนี้ ชิ้นไหนความอร่อยมากสุด ถ้าชิ้นนั้นไม่ได้อยู่ขอบ เราจะตัดได้แค่แบบเดียวคือถ้าอยู่ถัดจากซ้ายเข้ามา จะตัดสองชิ้นซ้ายแล้วกินชิ้นถัดจากซ้ายแล้วให้ชิ้นซ้ายสุดแก่นาย ข หรือตัดสองชิ้นขวาแล้วกินชิ้นถัดจากขวาแล้วให้ชิ้นขวาสุดแก่นาย ข แต่ถ้าชิ้นที่ความอร่อยมากที่สุดอยู่ขอบข้างใดข้างหนึ่ง เราจะเทียบระหว่างชิ้นข้างเดียวกันที่ถัดเข้ามากับชิ้นขอบอีกข้างและเราจะตัดแบบที่ให้อีกชิ้นที่ให้นาย ข ได้ความอร่อยน้อยกว่า

แต่จะมี 3 กรณีนอกเหนือจากที่กล่าวมาที่ต้องกรองก่อนคือกรณีทั้ง 4 ชิ้นมีค่าความอร่อยเท่ากัน จะตัดแบบซ้ายขวาแล้วกินชิ้นใดชิ้นหนึ่ง (line 14) และถ้าพบว่าชิ้นที่อร่อยน้อยกว่าของฝั่งซ้ายยังมากกว่าชิ้นเยอะกว่าของฝั่งขวา จะตัดแบบซ้ายขวาแล้วกินฝั่งซ้าย (line 19) โดยถ้าชิ้นที่อร่อยน้อยกว่าของฝั่งขวายังมากกว่าชิ้นเยอะกว่าของฝั่งซ้ายก็จะทำแบบเดียวกับแต่กินฝั่งขวา (line 24)

เมื่อตัดแล้วก็ขยับ left, right ซึ่งเป็นเหมือน pointer ขยับตามชิ้นที่เหลือ เมื่อทำจนครบก็ return ค่าความอร่อยรวมกลับมา

futomaki_score_v1() Time Complexity

time complexity ของ version นี้คือ O(n) จากการวน while loop แต่อัลกอริทึมนี้ไม่สามารถให้คำตอบที่ถูกต้องได้ในทุก instances โดยจะวิเคราะห์ว่าที่ได้ไม่เต็มเพราะอะไร และยกตัวอย่าง input ที่ทำให้ไม่ถูกต้องหัวข้อถัดไป จึงไม่มีความจำเป็นในการเอาความเร็ว time complexity มาพิจารณา

<u>ปัญหาของ futomaki_score_v1()</u>

หากพิจารณา testcase 4 cases ที่ถูกต้อง ปรากฏว่าเป็น instance ที่ input n มีขนาดน้อยมาก (4,4,6) 3 cases และมี input n ปานกลาง (28) อีก 1 case แปลว่าอัลกอริทึ่มนี้คำนวนข้าวปั้นสั้น ๆ อาจจะบังเอิญถูก แต่ถ้าพอมีความยาว มีค่าความอร่อยซับซ้อน อัลกอริทึ่มนี้ไม่ใช่วิธีที่ทำให้ได้ความอร่อยมากที่สุดเพราะอาจจะมีการตัดแบบที่ดีกว่าซึ่งไม่จำเป็นต้องตัดชิ้นที่มากที่สุดใน 4 ชิ้นขอบกิน แต่สามารถทำให้ผลรวมของค่าความอร่อยที่กินได้รวมแล้วมากกว่าได้ ดังนั้น ปัญหาของ v1 คือความถูกต้อง

Version 2

Points	40.0/100
Comment	PPPPPPPPTTTTTTTTTTTT

int futomaki_score(vector<int> &A, int n, int start, int stop) จะใช้หลักการ divide and conquer เข้ามาช่วยแก้ปัญหาโดยเราจะมองปัญหาให้เหมือนกันโจทย์ Maximum Subarray Sum โดยแบ่งปัญหาใหญ่เป็นการหั่นได้ 3 แบบได้แก่การหั่นสองชิ้นด้านซ้าย, หั่นสองชิ้นด้านขวา และหั่นด้านละชิ้นสองฝั่ง แล้วกินชิ้นที่มีค่าความอร่อยมากกว่า นั่นคือการหาค่า max ของสองชิ้นที่หั่นมา ดังนั้นเพื่อไล่ทำงานตาม index เหมือน Maximum Subarray Sum จึงต้องมีพารามิเตอร์ start และ stop ไปด้วยโดยเริ่มต้นที่ 0 กับ n-1 ตามลำดับ

<u>Summary on main()</u>

```cpp
int main(int argc, char const *argv[])
{
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    int n;
    cin >> n;
    vector<int> A(n);
    for (size_t i = 0; i < n; i++)
    {
        cin >> A[i];
    }
    cout << futomaki_score(A, n, 0, n - 1) << endl;
    return 0;
}
```

ใน main() ไม่มีอะไรเพิ่มเติมจาก v1 นอกจากมีการส่งค่า start, stop เพิ่มตามพารามิเตอร์ที่เพิ่มมาด้วย

แนวคิดของ futomaki_score_v2()

```
5   int futomaki_score(vector<int> &A, int n, int start, int stop)
6   {
7       if (start > stop) return 0;
8
9       int r1 = futomaki_score(A, n, start + 2, stop) + max(A[start], A[start + 1]);
10      int r2 = futomaki_score(A, n, start + 1, stop - 1) + max(A[start], A[stop]);
11      int r3 = futomaki_score(A, n, start, stop - 2) + max(A[stop], A[stop - 1]);
12
13      return max(max(r1, r2), r3);
14  }
```

ใช้หลักการ divide and conquer โดยเราจะมองปัญหาให้เหมือนกันโจทย์ Maximum Subarray Sum โดย base case คือถ้าทำจน start > stop หรือไม่มีข้าวปั้นให้ตัดแล้ว ให้ return 0 เป็นค่าความอร่อยเริ่มต้นซึ่งจะเอามา + กับค่า max ก้อนหลังของการหั่นทั้ง 3 แบบ โดย r1 จะเป็นค่าที่ได้จากการหั่นด้านซ้าย 2 ชิ้น โดยจะเอาคะแนนของชิ้นที่มากกว่าไปรวมกับของเก่าที่ max ออกมาแล้วฝั่งขวา ส่วน r2 คือค่าความอร่อยรวมที่ได้จากการหั่นด้านละชิ้น และสุดท้าย r3 คือค่าความอร่อยรวมที่ได้จากการหั่นด้านขวา 2 ชิ้น เมื่อได้ทั้ง 3 ค่าจึงนำไปหาว่าค่าใดจะได้ผลลัพธ์ค่าความอร่อยที่มากที่สุดใน line 13 แล้ว return ออกไปให้ node ที่เรียกมา

futomaki_score_v2() Time Complexity

สามารถแบ่งได้เป็น 2 ส่วนหลัก ๆ คือ

1. ส่วนที่เป็นการเรียก recursive subproblem call โดยใน 1 invocation ฟังก์ชั่นจะเรียก 3 recursive calls โดยแต่ละตัวจะลดขนาดปัญหาไป 2 หน่วย แตกต่างที่ index ที่เปลี่ยนแปลง แล้วทุกการเรียกก็จะเรียก 3 recursive calls ซ้ำไปเรื่อย ๆ จนเจอ terminating case
2. ส่วนที่เป็นงานที่ทำจริง ๆ ในทุกการเรียกซ้ำมี 2 ส่วนคือ
    a. line 7 เช็ค base case เป็น if clause => O(1)
    b. line 13 เรียก std::max() 2 ครั้ง => O(1)

จะได้ T(n) = 3T(n-2) +O(1) เมื่อคำนวน big O notation จากวิธี recurrence relation จะได้ O(3^n) ซึ่งยิ่งความยาว n เยอะเท่าไหร่ ก็จะใช้เวลาเยอะขึ้น exponentially

ปัญหาของ futomaki_score_v2()

จากผลลัพธ์ที่ออกมา ถูก 8 จาก 20 testcases ส่วนที่เหลือใช้เวลาเกิน จึงสันนิษฐานว่าอัลกอริทึมน่าจะให้คำตอบที่ถูกต้องแต่ยังใช้เวลามากเกินไปในบางกรณี และเมื่อลองดู testcase ที่ 1 ถึง 8 ที่ผ่าน พบว่า input n <= 30 ทั้งหมด และตั้งแต่ testcase ที่ 9 เป็นต้นไป n มีขนาดใหญ่ระดับมากกว่า 4000 ทั้งหมด จึงเป็นสาเหตุว่าทำไมอัลกอริทึมที่ใช้แค่ divide and conquer อย่างเดียวใช้เวลานานเกินไปใน case ที่ n มีขนาดใหญ่เพราะมี O(3^n) จึงต้องแก้ปัญหาโดยหาวิธีที่ทำให้การทำงานของ code เร็วขึ้น

Version 3

Points	100.0/100
Comment	PPPPPPPPPPPPPPPPPPPP

int futomaki_score(vector<vector<int>> &dp, vector<int> &A, int n, int start, int stop) จะใช้หลักการ dynamic programming เข้ามาช่วยแก้ปัญหาเพิ่มจากของเดิมที่เป็น divide and conquer ธรรมดา โดยจะจะมีพารามิเตอร์เพิ่มคือ vector<vector<int>> &dp เป็น vector 2D ที่เป็นเหมือนตารางสองมิติขนาด n x n ไว้เก็บค่าที่คำนวนไว้แล้วเป็น memorization ของ dynamic programming แบบ Top-down และ start กับ stop เป็น index ที่เราจะคอยไล่ขยับซ้ายขวาตาม index ของข้าวปั้น

<u>Summary on main()</u>

```cpp
int main(int argc, char const *argv[])
{
    ios_base::sync_with_stdio(false);
    cin.tie(0);
    int n;
    cin >> n;
    vector<int> A(n);
    for (size_t i = 0; i < n; i++)
    {
        cin >> A[i];
    }
    vector<vector<int>> dp(n);
    for (size_t i = 0; i < n; i++)
    {
        vector<int> tmp(n);
        dp[i] = tmp;
    }
    cout << futomaki_score(dp, A, n, 0, n - 1) << endl;
    return 0;
}
```

สิ่งที่เพิ่มมาจาก v2 คือการสร้าง dp เป็น vector ซ้อน vector ขนาด n x n และ initialized เป็นค่า default integer นั่นคือ 0 ไว้ทุก entries

แนวคิดของ futomaki_score_v3()

```
5   int futomaki_score(vector<vector<int>> &dp, vector<int> &A, int n, int start, int stop)
6   {
7       if (start > stop) return 0;
8       if (dp[start][stop] != 0) return dp[start][stop];
9   
10      int r1 = futomaki_score(dp, A, n, start + 2, stop) + max(A[start], A[start + 1]);
11      int r2 = futomaki_score(dp, A, n, start + 1, stop - 1) + max(A[start], A[stop]);
12      int r3 = futomaki_score(dp, A, n, start, stop - 2) + max(A[stop], A[stop - 1]);
13  
14      dp[start][stop] = max(max(r1, r2), r3);
15      return dp[start][stop];
16  }
```

สิ่งที่เพิ่มมาจาก v2 คือการ memorization เพื่อแก้ปัญหาความช้าของ divide and conquer ธรรมดา เนื่องจากการเรียก divide and conquer ตามแบบ v2 มีการเรียก futomaki_score(A, n, x, y) ที่ (x, y) ซ้ำเดิมบ่อยครั้งมาก ๆ ทำให้เกิด redundancy จึงเอา 2D vector dp มาเก็บค่าผลลัพธ์ของ subproblem โดยที่ dp[start][stop] จะเก็บค่า maximum score ที่สามารถคำนวนได้ใน subarray โดยนับจาก index start จนถึง index stop

ใน line 8 ถ้า dp[start][stop] ของค่าปัจจุบันมีการคำนวนไว้แล้ว นั่นคือไม่เท่ากับ 0 ซึ่งเป็นค่า default ที่ถูก initialized ฟังก์ชั่นจะ return ค่าที่เคยคำนวนได้เลยโดยไม่ต้องลงไปทำการเรียก 3 recursive subproblems เพื่อลดความ redundancy

ถ้ายังไม่เคยคำนวนใน index (start,stop) นี้ก็จะทำงานตามปกติแล้ว เอาค่า maximum ของทั้ง 3 ค่าไปเก็บไว้ใน dp[start][stop] เผื่อจะมีการเรียกซ้ำในอนาคตแล้ว return ค่านั้นกลับ

futomaki_score_v3() Time Complexity

หากเป็น divide and conquer ปกติใน v2 จะได้ O(3^n) แต่ใน v3 นี้เป็นอัลกอริทึ่มแบบ top-down dynamic programming approach with memorization เราจะพิจารณาจากจำนวน subproblem ที่สามารถเรียกได้โดย recursive function สามารถเรียกได้ตั้งแต่ค่า start และ stop ใน range 0 ถึง n-1 โดยที่ start < stop นั่นแปลว่าตาราง 2D vector dp แทนจำนวน subproblem ที่เกิดขึ้นได้ และจะไม่เกิดขึ้นมากกว่านี้เพราะถ้ามีการเรียกซ้ำก็จะ lookup ค่าจาก table นี้ให้ใน O(1) ได้เลย จึงได้ว่ามี

ทั้งหมด (n^2) / 2 subproblem จากขนาดของตาราง dp หารครึ่งเพราะ start < stop จึงเติมตารางแค่ครึ่งเดียว

ส่วนที่เป็นงานที่ทำจริง ๆ ในทุกการเรียกซ้ำก็ยังคงเป็น O(1) เช่นเดิม

เมื่อนำมารวมกันจะได้ overall time complexity O(n^2) ซึ่งลดลงจากเดิมที่เป็น O(3^n) ของ divide and conquer ปกติค่อนข้างมาก

ปัญหาของ futomaki_score_v3()

จริง ๆ แล้ววิธี top-down dynamic programming approach with memorization ก็ให้ผลลัพธ์และความเร็วที่น่าพึงพอใจ ซึ่งไม่ได้มีปัญหาอะไร แต่มีจุดที่สามารถพัฒนาให้ดีขึ้นได้ซึ่งแบ่งได้ 2 ประเด็น

1. ในโจทย์ข้อนี้ที่รู้รูปแบบ input แล้วว่าค่าความอร่อยในแต่ละท่อนมีค่าเป็นจำนวนบวก การประกาศตาราง 2D lookup table เก็บค่าก่อนหน้าจึงสามารถประกาศโดยใช้ default constructor ได้ เพราะไม่มี part ไหนที่ผ่านการคำนวนออกมาแล้วได้ 0 แต่ถ้ามีการกำหนดว่าบางท่อนสามารถเป็น 0 ได้ จะต้องมีการเปลี่ยนแปลงตอน initialization ใหม่โดยจะต้อง assign value เริ่มต้นเป็นค่า negative ให้หมดทุก entries เช่น -1 แล้วปรับ code line 8 ในฟังก์ชั่นเป็น if (dp[start][stop] != -1) return dp[start][stop]; แทน
2. ในแนวคิด dynamic programming มีการ approach ปัญหาได้อีกแบบนอกจาก Top-down คือ Botton-up approach ซึ่งจะใช้เวลาและ memory น้อยกว่ามาก แต่คิดยากกว่าซึ่งหากดู submission ที่เป็น Best Runtime ของ code C++ แล้วใช้วิธี Bottom-up และได้ผลลัพธ์โดยใช้ runtime และ memory น้อยกว่า รวมถึง code ก็สั้นกว่า

Version Model Solution (Submission: 563177)

Runtime	0.012 s
Memory	1016 kb

โดยที่ Submission ของ futomaki_score_v3() ได้ Runtime และ Memory ดังนี้(คะแนน100เท่ากัน)

Runtime	0.287 s
Memory	98668 kb

แนวคิดของ Model Solution

```
1  #include"stdio.h"
2  int n, A[5000], DP[5001];
3  inline int max(int a, int b) { return a > b? a: b; }
4  int main() {
5    scanf("%d", &n);
6    for (int i = 0; i < n; ++i) scanf("%d", A+i);
7    for (int i = 1; i < n; i+= 2)
8      for (int l = 0, r; (r = l+i) < n; ++l)
9        DP[l] = max(DP[l] + max(A[r-1], A[r]), max(DP[l+1] + max(A[l], A[r]), DP[l+2] + max(A[l], A[l+1]))
10   printf("%d", *DP);
11 }
```

Model Solution ใช้ Bottom-up approach โดยใช้ DP[5001] เป็น array 1 มิติเก็บการคำนวน อย่างเดียวทำให้ใช้ memory น้อยมาก

Line 5 และ 6 เป็นการรับข้อมูลตาม input

Line 7 ถึง 9 เป็นการวน loop เพื่อเติม DP array แบบ Bottom-up โดยจะวน loop นอก n ครั้ง และ loop ในอีกไม่เกิน n ครั้ง โดย loop นอกจะกำหนดขนาดของตัว futomaki โดยเริ่มจากขนาด 2 ก่อนนั่นคือตัวแปร index l กับ r มีค่าต่างกัน 1 แล้ว loop ในจะวนเติม DP[l] จน r >= n โดยที่ r = l+1 เสมอ เมื่อทำครบก็เพิ่มขนาด futomaki จาก 2 เป็น 4 นั่นคือ i เพิ่มขนาดจาก 1 ขึ้นเป็น 3 หรือ +2 ทุก iteration นั่นเอง

array DP จะถูก initialized ด้วย 0 แล้ว DP[l] จะเก็บค่า maximum score สำหรับ subarray ที่เริ่มจาก index l

เมื่อวนเติมวนครบจะได้ว่า DP[0] จะเก็บค่า maximum ของค่าความอร่อยที่เก็บได้จาก array ตั้งแต่ index ที่ 0 และ print ออกมาโดยใช้ตัว pointer *DP ใน line 10

Model Solution Time Complexity

จากการวิเคราะห์ code การทำงานหลัก ๆ จะอยู่ที่การวน loop เติม array DP ซึ่งเป็นการวน loop ซ้อน loop เพราะนอกจาก loop นี้ที่เหลือก็เป็นการทำงานแบบ O(1) ทั้งหมด

Outer loop: วิ่งตั้งแต่ i = 1 ถึง i < n และเพิ่มทีละ 2 จึงสรุปได้ว่าวนประมาณ n/2 ครั้ง

Inner loop: สำหรับทุกค่า i วิ่งตั้งแต่ l = 0 ถึง l + i < n และเพิ่มทีละ 1 จึงได้ว่าประมาณ n - i ครั้ง

การทำงานในชั้นในสุดของ Inner loop: update DP[i] ด้วย max() => O(1)

ดังนั้นจำนวนการทำงานทั้งหมดจะสามารถคำนวนได้ดังนี้

$$\sum_{i=1, i \text{ odd}}^{n-1} (n-i) \approx \sum_{k=0}^{n/2-1} (n-2k-1) \approx \sum_{k=0}^{n/2-1} n - \sum_{k=0}^{n/2-1} (2k+1)$$

$$\approx \frac{n}{2} \times n - \frac{n^2}{2} \approx \frac{n^2}{2} - \frac{n^2}{4} \approx \frac{n^2}{4}$$

จะได้ว่า time complexity โดยภาพรวมจะกลายเป็น O(2^n) ซึ่งเทียบกับ Top-down ก็เป็น O(2^n) เหมือนกัน แต่ตัว model solution ทำงานได้เร็วกว่าด้วยสาเหตุจากหลายปัจจัย การใช้ memory น้อย, การทำงานกับแค่ update array ที่เป็น time-constant อย่างเดียวไปเรื่อย ๆ, การที่มีการทำงาน O(n^2) แต่ไม่ได้มีการเรียก subproblem ย่อย ๆ หลายรอบ เช็ค lookup table หลายครั้ง รวมถึงการเขียน code ที่สั้นและเขียนให้ทำงานกับข้อมูลระดับ low level เช่น การใช้ pointer และ array ธรรมดาแทน vector ที่ต้อง include ทุก ๆ สาเหตุล้วนเป็นปัจจัยที่ทำให้ solution นี้ทำงานได้เร็วขึ้นจากวิธีเดิม

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