Time complexity

- ประสิทธิภาพของอัลกอริทึมเป็นสิ่งสำคัญมากใน Competitive programming โดยทั่วไปแล้วมันไม่ยากมากในการออกแบบอัลกอริทึมที่แก้ปัญหาได้โดยไม่คำนึงถึงเวลา ซึ่งอาจจะแก้ปัญหาช้า แต่ในการแข่งขันจริงหากอัลกอริทึมช้า เราอาจจะได้คะแนนเพียงบางส่วนหรืออาจจะไม่ได้คะแนนเลยก็ได้
- **Time complexity** ในการทำงานของอัลกอริทึมนั้นเป็นการประมาณว่าอัลกอริทึมใช้เวลาในการทำงานเท่าไรสำหรับบาง input
- แนวคิดคือการแทนประสิทธิภาพการทำงานด้วยฟังก์ชันที่ parameter เป็นขนาดของข้อมูลเข้า จากการคำนวณ time complexity ทำให้เราทราบได้ว่าอัลกอริทึมของเรานั้นเร็วพอหรือไม่ โดยไม่ต้อง implement

กฎในการคำนวณ

- Time complexity ของอัลกอริทึมนั้นแทนด้วย O(...) โดยที่ ... แทนฟังก์ชันบางฟังก์ชัน โดยทั่วไป ตัวแปร n แทนขนาดของ input ตัวอย่างเช่น
  - ถ้า input เป็น array ของจำนวน n จะแทนขนาดของ array
  - ถ้า input เป็น string n ก็จะเป็นความยาวของ string

Loop

- เหตุผลอย่างหนึ่งที่ทำให้อัลกอริทึมของเราช้านั่นคือมันมี loop หลายอัน
- ยิ่งมี loop ซ้อนกันในอัลกอริทึมจำนวนมาก ยิ่งช้า ถ้ามี loop ปกติซ้อนกัน k ชั้น เวลาในการทำงานอาจจะเป็น $O(n^k)$

ตัวอย่างเช่น เวลาในการทำงานของ code ต่อไปนี้เป็น O(n)

```
for (int i = 1; i <= n; i++) {
// code
}
```

และตัวอย่างเวลาในการทำงานของ code ต่อไปนี้เป็น $O(n^2)$

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
    // code
    }
}
```

Order of magnitude

- Time complexity นั้นไม่ได้บอกเราถึงเวลาที่เป็นตัวเลขที่แน่นอนที่ code ภายใน loop ถูกประมวลผล แต่มันเพียงแสดงอันดับของขนาด
- ตัวอย่างต่อไป code ภายใน loop จะถูกประมวลผลในเวลา $3n$, $n + 5$ และ $\lceil n/2 \rceil$ แต่ time complexity ของ code แต่ละอันเป็น $O(n)$

```
for (int i = 1; i <= 3*n; i++) {
    // code
}

for (int i = 1; i <= n+5; i++) {
    // code
}
```

```
for (int i = 1; i <= n; i += 2) {
    // code
}
```

อีกตัวอย่างหนึ่งที่ Time complexity ของ code เป็น $O(n^2)$

```
for (int i = 1; i <= n; i++) {
    for (int j = i+1; j <= n; j++) {
        // code
    }
}
```

Phases

- ถ้า algorithm ของเราประกอบด้วยหลายเฟสต่อเนื่องกัน complexity รวมคือ complexity ที่มีค่ามากที่สุดของเฟสหนึ่งๆ เหตุผลเพราะเฟสที่ช้าที่สุดโดยทั่วไปแล้วคือคอขวดของ code

- ตัวอย่างต่อไปประกอบด้วย 3 เฟสที่มี time complexity เป็น $O(n)$ ตามด้วย $O(n^2)$ ตามด้วย $O(n)$ ทั้งนี้ complexity รวมเป็น $O(n^2)$

```
for (int i = 1; i <= n; i++) {
    // code
}
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= n; j++) {
        // code
    }
}
for (int i = 1; i <= n; i++) {
    // code
}
```

Several variables

- บางครั้ง Time complexity ขึ้นกับหลายปัจจัย ในกรณีนี้ Time complexity ขึ้นกับหลายตัวแปร
- ตัวอย่างเช่น Time complexity ของ code ต่อไปนี้ เป็น $O(mn)$

```
for (int i = 1; i <= n; i++) {
    for (int j = 1; j <= m; j++) {
        // code
    }
}
```

Recursion

```
#include <bits/stdc++.h>

using namespace std;
long fac(int n){
    if(n<=1)
        return 1;
    else
        return n*fac(n-1);
}

int main()
{
    int n;
    cin>>n;
    cout<<fac(n);
    return 0;
}
```

Base case

Recursive case

- Algorithm ที่มีการเรียก recursive call เวลาในการทำงานอธิบายได้ด้วย recurrence relation

- $$T(n) = \begin{cases} t_A & \text{กรณี } base\ case \\ t_B + t_C & \text{กรณีอื่นๆ} \end{cases}$$

- t_A = เวลาในการทำงานส่วนของ base case

- t_B = เวลาในการทำงานส่วนการเรียกตัวเอง (recursive) หรือเวลาของการแก้ปัญหาย่อย

- t_C = เวลาในการทำงานอื่นๆ ที่ไม่ใช่เวลาของการแก้ปัญหาย่อย หรือไม่ใช่ recursive

```
long fac(int n){
    if(n<=1)
        return 1;        ← Base case
    else
        return n*fac(n-1);    ← Recursive case
}
```

t_A = O(1)

t_B = T(n-1)

t_C = O(1)

BinarySearchRecursive(A[left..right])

```
if (left > right) return(-1)
m=(left + right)/2
if x == A[m] return m;
If x < A[m]
    return(BinarySearchRecursive (A[left..m-1]))
else
    return(BinarySearchRecursive (A[m...right]))
```

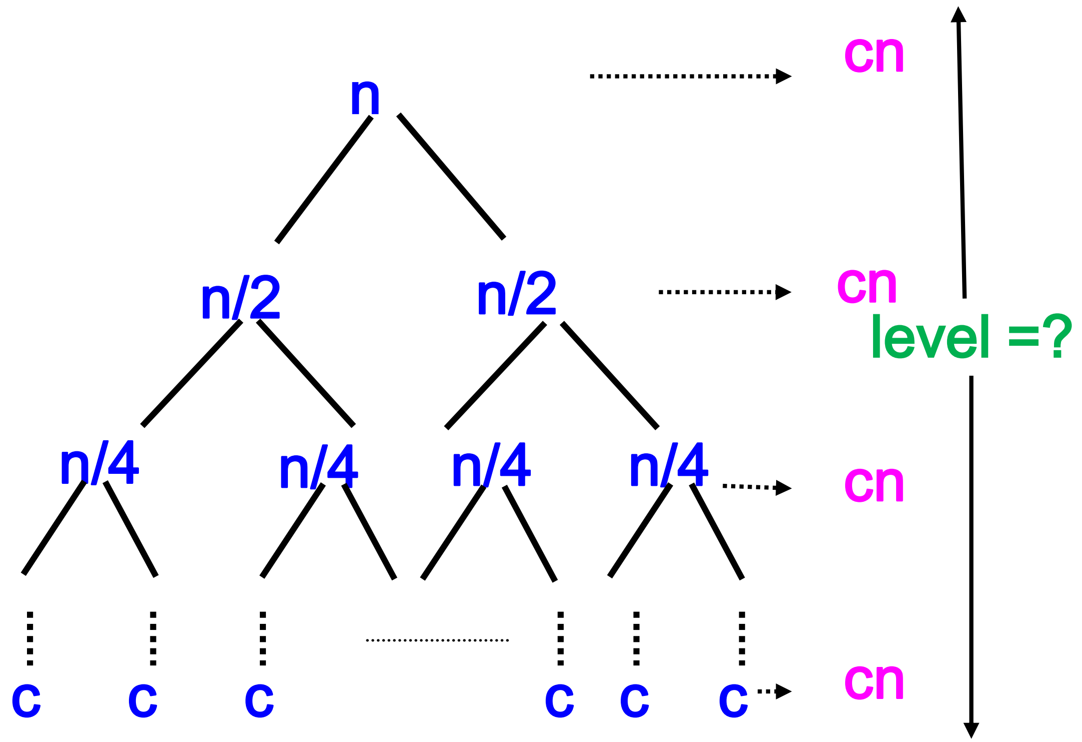
- Recurrence Relation เป็นอย่างไร

```
Merge_Sort(A,p,r)
1.If p < r then
2.    q= ⌊(p+r)/2⌋
3.    Merge_Sort(A,p,q)
4.    Merge_Sort(A,q+1,r)
5.    Merge(A,p,q,r)
```

```
Merge(A,p,q,r)
 1. i = p, j = q + 1, n = r - p + 1
 2. for k = 1 to n
 3.    if ((A[i]< A[j]) or (j > r)) and (i≤q)
 4.                B[k] = A[i]
 5.            i = i +1
 6.       else
 7.            B[k] = A[j]
 8.            j = j +1
 9. for k = 0 to n – 1
10.      A[p + k] = B[k]
```

Recurrence Relation เป็นอย่างไร

- $T(n) = 2T(n/2)+cn$



Total T(n) = ?

- $T(n) = T(n/2) + O(1)$
- $T(n) = T(n-1) + O(1)$
- $T(n) = 2\,T(n/2) + O(1)$
- $T(n) = T(n-1) + O(n)$
- $T(n) = 2\,T(n/2) + O(n)$

Complexity class

- O(1) เวลาการทำงานของอัลกอริทึมที่เป็นค่าคงที่ไม่ขึ้นกับขนาดของข้อมูลเข้า โดยทั่วไปเป็นอัลกอริทึมที่คำนวณสูตรโดยตรง
- O(logn) logarithmic algorithm ส่วนใหญ่จะแบ่งข้อมูลเข้าออกเป็นสองส่วนในแต่ละขั้น
- O($\sqrt{n}$) ทำงานช้ากว่า O(logn) แต่เร็วกว่า O(n) คุณสมบัติพิเศษของ square root คือว่า $\sqrt{n} = \mathrm{n}/\sqrt{n}$
- O(n) ทำงานผ่านข้อมูลเข้าด้วยจำนวนคงที่ algorithm นี้ส่วนใหญ่เป็น best possible time complexity เพราะว่ามันจำเป็นที่จะต้องเข้าไปยังข้อมูลเข้าทีละตัวอย่างน้อยหนึ่งครั้งก่อนที่จะให้คำตอบ

- O(nlogn) เวลาลักษณะนี้ส่วนใหญ่จะมีการเรียงข้อมูล (sort) เพราะว่า time complexity ของ sorting algorithms ที่มีประสิทธิภาพคือ O(nlogn) อีกอย่างที่เป็นไปได้คือ เป็น algorithm ที่ใช้โครงสร้างข้อมูลที่แต่ละการดำเนินการใช้เวลา O(logn)
- $O(n^2)$ quadratic algorithm ส่วนใหญ่จะมี loop ซ้อนกันสองชั้น อาจจะเป็นการตรวจดูทุกคู่ที่เป็นไปได้ของข้อมูลเข้าซึ่งใช้เวลา $O(n^2)$
- $O(n^3)$ cubic algorithm ส่วนใหญ่จะมี loop ซ้อนกันสามชั้น อาจจะเป็นการตรวจดูข้อมูลสามตัวทุกแบบที่เป็นไปได้

- $O(2^n)$ time complexity นี้ส่วนใหญ่เป็นการระบุว่า algorithm วนทุก subset ของข้อมูลเข้า ตัวอย่างเช่น subset {1,2,3} คือ {},{1}, {2}, {3}, {1,2}, {1,3}, {2,3}, {1,2,3}
- $O(n!)$ time complexity นี้ส่วนใหญ่เป็นการระบุว่า algrithm วนรอบทุกการเรียงสับเปลี่ยน (permutation) ตัวอย่างการเรียงสับเปลี่ยนของ {1,2,3} ได้แก่ (1,2,3), (1,3,2),(2,1,3),(2,3,1),(3,1,2),(3,2,1)
- อัลกอริทึมจะเป็น polynomial time ถ้า time complexity ไม่เกิน $O(n^k)$ เมื่อ k เป็นค่าคงที่ (และเราจะบอกว่ามีประสิทธิภาพด้วย)
- Time complexity ข้างต้นที่กล่าวมาแล้วยกเว้น $O(2^n)$ และ $O(n!)$ เป็น polynomial ทั้งนี้ในทางปฏิบัติค่าคงที่ k ควรจะน้อยๆ

- อัลกอริทึมส่วนใหญ่ที่แข่งจะเป็น polynomial time ทั้งนี้ยังมีหลายปัญหาที่สำคัญที่ยังไม่มีใครรู้ polynomial time algorithm นั่นคือ ยังไม่มีใครรู้วิธีแก้ที่มีประสิทธิภาพ

- NP-Hard problem คือเซตของปัญหาที่สำคัญที่ยังไม่มีใครรู้ polynomial time algorithm

Estimation efficiency

- จากการคำนวณ time complexity ของ algorithm มันเป็นไปได้ที่จะตรวจก่อนที่จะ implement algorithm ว่ามันมีประสิทธิภาพเพียงพอไหมในการแก้ปัญหา
- จุดเริ่มต้นของการประมาณคือ ความจริงที่ว่า เครื่องคอมพิวเตอร์สมัยใหม่ทำงานได้ หลายร้อยล้านคำสั่งต่อวินาที
- ตัวอย่างเช่นสมมตbว่า time limit ของปัญหาเป็น 1 วินาที และขนาดของข้อมูลเข้าเป็น $n = 10^5$ ถ้า time complexity เป็น $O(n^2)$ algorithm จะทำงานประมาณ $(10^5)^2 = 10^{10}$ คำสั่ง นั่นทำให้ต้องใช้เวลาอย่างน้อยหลายสิบวินาที ทำให้ algorithm นั้นดูช้าในการแก้ปัญหา

- อีกทางหนึ่ง เมื่อกำหนด input size มาให้ เราสามารถ เดา time complexity ที่ควรใช้ของ algorithm ในการแก้ปัญหาได้
- หาก time limit เป็น 1 วินาที ก็จะได้ประมาณนี้

Input size	**Required time complexity**
n <= 10	O(n!)
n <= 20	$O(2^n)$
n <= 500	$O(n^3)$
n <= 5000	$O(n^2)$
$n <= 10^6$	O(nlogn) หรือ O(n)
n is large	O(1) หรือ O(logn)

- ตัวอย่างเช่น ถ้า input size เป็น $n = 10^5$ เราอาจจะคาดว่า time complexity ของ algorithm คือ O(n) หรือ O(nlogn)
- ข้อมูลนี้ทำให้ง่ายขึ้นในการออกแบบ algorithm เพราะว่าเราไม่ควรสร้าง algorithm ที่มี time complexity แย่กว่านี้
- สิ่งสำคัญอีกอย่างคือ time complexity เป็นเพียงการประมาณประสิทธิภาพเพราะว่ามันซ่อน constant factor
- ตัวอย่างเช่น อัลกอริทึมที่ทำงานใน O(n) เวลาอาจจะเป็น n/2 หรือ 5n ก็ได้

Maximum subarray sum

- ปกติแล้วปัญหาหนึ่ง มีวิธีการแก้ได้หลายทาง ซึ่งแต่ละทางก็มี time complexity เหมือนหรือแตกต่างกันได้ ในตัวอย่างนี้จะมาดู algorithm ที่หากแก้ตรงๆ ใช้เวลา $O(n^3)$ ซึ่งหากออกแบบดีหน่อยจะกลายเป็น $O(n^2)$ และ $O(n)$

- กำหนด array ของ n จำนวนมาให้ จงคำนวณหา maximum subarray sum นั่นคือ ผลรวมที่มากที่สุดที่เป็นไปได้ของลำดับที่ติดกันใน array ปัญหานี้จะน่าสนใจขึ้นถ้าหากมีเลขลบใน array ตัวอย่างเช่น

-1	2	4	-3	5	2	-5	2

Algorithm1

- วิธีแก้แบบตรงๆ คือ ลองทุก subarray ทุกแบบที่เป็นไปได้ จากนั้นคำนวณค่า sum ของแต่ละ subarray และเก็บค่า maximum ไว้

```
int best = 0;
for (int a = 0; a < n; a++) {
        for (int b = a; b < n; b++) {
                int sum = 0;
                for (int k = a; k <= b; k++) {
                        sum += array[k];
                }
                best = max(best,sum);
        }
}
cout << best << "\n";
```

Algorithm1

- ตัวแปร a และ b เป็นตัวกำหนดจุดเริ่มต้นและจุดสิ้นสุดของ subarray
- ค่าผลรวมของแต่ละค่าคำนวณเก็บไว้ที่ sum
- ส่วนตัวแปร best เก็บค่าที่มากที่สุดที่พบระหว่างการ search ไว้

- Time complexity ของ algorithm เป็น $O(n^3)$ เพราะว่ามี 3 loop ซ้อนกันอยู่เมื่อรันผ่าน input

Algorithm2

การทำให้ algorithm1 มีประสิทธิภาพ โดยการลบ loop ออกอันหนึ่ง

ทำได้โดยการคำนวณค่า sum เวลาเดียวกับที่จุดปลายทางขวาขยับ

```
int best = 0;
for (int a = 0; a < n; a++) {
        int sum = 0;
        for (int b = a; b < n; b++) {
                sum += array[b];
                best = max(best,sum);
        }
}
cout << best << "\n";
```

Time complexity เป็น $O(n^2)$

Algorithm3

การแก้ให้ได้ O(n) หมายถึงลดให้เหลือ loop เดียว แนวคิดในการแก้ปัญหาคือ คำนวณค่าผลรวมที่มากสุดของ subarray ที่สิ้นสุดในแต่ละตำแหน่ง array หลังจากนั้นนำคำตอบที่มากที่สุดมาตอบ

พิจารณาปัญหาย่อยในการหาค่ามากที่สุดของ subarray ที่สิ้นสุดที่ตำแหน่งที่ k มี 2 อย่างที่เกิดขึ้นได้

1. subarray เก็บค่าที่ตำแหน่งที่ k แค่ตัวเดียว
2. Subarray ประกอบด้วย subarray ที่ลงท้ายด้วย k-1 แล้วรวม k เข้าไปด้วย

เริ่มต้นใหม่หรือนับต่อนั่นเอง

แบบฝึกหัด

Efficiency comparison

- ประสิทธิภาพของ algorithm เมื่อรันจริง
- มีการทดสอบแต่ละครั้งด้วยการสุ่มเลข ได้ดังนี้

Array size n	algorithm1	algorthim2	Algorithm3
10^2	0.0s	0.0s	0.0s
10^3	0.1s	0.0s	0.0s
10^4	>10.0s	0.1s	0.0s
10^5	>10.0s	5.3s	0.0s
10^6	>10.0s	>10.0s	0.0s
10^7	>10.0s	>10.0s	0.0s

- พบว่าทุก algorithm มีประสิทธิภาพเมื่อข้อมูลเข้าน้อยๆ แต่เมื่อข้อมูลเข้าใหญ่ขึ้นความแตกต่างเรื่องเวลาการทำงานก็มากขึ้น

Extra: รูปแบบการรับ/แสดงข้อมูล

- โดยทั่วไปในการแข่ง ความถูกต้องของ code ของเราถูกตัดสินจากการรัน code เรากับหลายๆ test cases แทนที่จะใช้หลาย test case บางครั้งก็จะใช้ test case เดียวแต่มีหลาย test case ย่อยในนั้น
- ในหัวข้อนี้จะยกตัวอย่างให้หลายๆ แบบ สมมติว่าโจทย์ให้รับเลขจำนวนเต็ม 2 จำนวนในบรรทัดเดียว จากนั้นแสดงผลรวม
- รูปแบบในการรับ/แสดงผลข้อมูล 3 แบบหลักๆ ได้แก่
  - จำนวน test case บอกในบรรทัดแรก
  - Test cases มีหลายอัน จบด้วย 0
  - Test case มีหลายอัน จบไฟล์ด้วย EOF (end-of-file)

Source code	Sample input	Sample output

`int TC, a, b;`
`scanf("%d", &TC);`
`while (TC--) {`
`    scanf("%d %d", &a, &b);`
`    printf("%d\n", a + b);`
`}`
	3
	1 2
	5 7
	6 3
		3
		12
		9

`int a, b;`
`// stop when both integers are 0`
`while (scanf("%d %d", &a, &b), (a || b))`
`    printf("%d\n", a + b);`
	1 2
	5 7
	6 3
	0 0
		3
		12
		9

`int a, b;`
`// scanf returns the number of items read`
`while (scanf("%d %d", &a, &b) == 2)`
`// or you can check for EOF, i.e.`
`// while (scanf("%d %d", &a, &b) != EOF)`
`    printf("%d\n", a + b);`
	1 2
	5 7
	6 3
		3
		12
		9

- ในบางปัญหาที่มีหลาย test case จะมีการระบุให้มีการพิมพ์ข้อความก่อนว่าเป็น test case ไหน เรียงกันไป บางข้อมีการให้มีบรรทัดว่างหลังจากแต่ละ test case
- ต่อไปเป็นตัวอย่าง "Case [NUMBER]: [ANSWER] จากนั้นตามด้วยบรรทัดว่าง 1 บรรทัด

<table>
<tr><th>Source code</th><th>Sample input</th><th>Sample output</th></tr>
<tr><td><pre>int a, b, c = 1;
while (scanf("%d %d", &a, &b) != EOF)
// notice the two '\n'
 printf("Case %d: %d\n\n",c++, a + b);</pre></td><td>1 2
5 7
6 3</td><td>Case 1: 3

Case 2: 12

Case 3: 9</td></tr>
</table>

- ข้อควรระวัง บางครั้งหากมี test case เดียว แล้วเรามีขึ้นบรรทัดใหม่เกิน ถ้าส่งกับเว็บ Uva ก็จะเกิดเหตุการณ์ "Presentation Error" ได้ เพราะว่าแสดงผลผิด แต่คำตอบถูก

<table>
<tr><th>Source code</th><th>Sample input</th><th>Sample output</th></tr>
<tr><td><pre>int a, b, c = 1;
while (scanf("%d %d", &a, &b) != EOF) {
    if (c > 1) printf("\n");
    printf("Case %d: %d\n", c++, a + b);
}</pre></td><td>1 2
5 7
6 3</td><td>Case 1: 3

Case 2: 12

Case 3: 9</td></tr>
</table>

- ปรับโจทย์ใหม่ถ้าแต่ละ test case (แต่ละบรรทัด) รับจำนวนเต็ม k (k>=1) ตามด้วยเลขจำนวนเต็ม k ตัว และให้แสดงผลเป็นผลรวมของ k ตัวนั้น สมมติว่าข้อมูลเข้าหยุดด้วย EOF แลแสดงผลไม่ต้องมี Case

<table>
<tr><th>Source code</th><th>Sample input</th><th>Sample output</th></tr>
<tr><td><pre>int k, ans, v;
while (scanf("%d", &k) != EOF) {
  ans = 0;
  while (k--) {
    scanf("%d", &v);
    ans += v;
  }
  printf("%d\n", ans);
}</pre></td><td>1 1
2 3 4
3 8 1 1
4 7 2 9 3
5 1 1 1 1 1</td><td>1
7
10
21
5</td></tr>
</table>

- ให้ลองไปสมัครเว็บ UVa
- https://uva.onlinejudge.org/