C++ Array, Pointer และ STL สำหรับแก้โจทย์

เครื่องมือพื้นฐานสำหรับโค้ช IOI

ค่ายอบรมโค้ช IOI

บรรยาย 3 ชั่วโมง

เป้าหมายของการบรรยายนี้

สไลด์นี้ออกแบบสำหรับ **ครูที่จะทำหน้าที่เป็นโค้ช** ในการฝึกนักเรียนเข้าแข่งขัน IOI
ไม่ใช่บรรยายต่อหน้านักเรียนโดยตรง

สิ่งที่ครูควรได้หลังจบบทเรียน:

- เข้าใจ เหตุผล ว่าทำไมต้องใช้ STL container แต่ละแบบ ไม่ใช่แค่ syntax
- สามารถประเมินเวลาทำงานของโปรแกรมก่อนเขียน เพื่อสอนนักเรียนให้คิดในรูปแบบเดียวกัน
- รู้จัก โจทย์ตัวอย่าง ที่แก้แบบไร้เดียงสาได้ในเวลา $O(n^2)$ แต่ดีขึ้นเป็น $O(n \log n)$ เมื่อใช้ container ที่เหมาะสม
- รู้จุดที่นักเรียนมักพลาด เพื่อเตรียมตัวอย่างและคำเตือน

คำแนะนำการสอน

แต่ละหัวข้อในสไลด์จบด้วย “โจทย์ตัวอย่าง” หนึ่งข้อ
ขอแนะนำให้นำไปดัดแปลงให้นักเรียนลองทำเองในห้องเรียนก่อน แล้วเฉลยพร้อมโชว์เทคนิคของ container

เนื้อหา 3 ชั่วโมง

I. การวิเคราะห์เวลาทำงาน (เบื้องต้น) (≈ 15 นาที)
II. Array (≈ 15 นาที)
III. Pointer (แค่เกริ่นนำสู่เบื้องหลังของ STL) (≈ 10 นาที)
IV. `std::vector` (≈ 20 นาที)
V. `std::pair` (≈ 10 นาที)
VI. `std::stack` และ `std::queue` (≈ 20 นาที)
VII. `std::priority_queue` (≈ 20 นาที)
VIII. `std::set`, `std::multiset`, `std::map` (≈ 35 นาที)
IX. การจัดเรียงและการค้นหาแบบทวิภาค (≈ 25 นาที)

ทุกหัวข้อจบด้วย โจทย์ตัวอย่าง ที่ใช้ container เพื่อปรับปรุงประสิทธิภาพ

ตอนที่ 1: การวิเคราะห์เวลาทำงาน

บทนำสั้น ๆ

เวลาจำกัดในการแข่ง

สมมติฐาน. โปรแกรมในการแข่งขันจะมีเวลาจำกัด โดยทั่วไป **1 หรือ 2 วินาที**

กฎเบื้องต้นบนเครื่องตรวจสมัยใหม่:

$$\approx 10^8 \text{ “การดำเนินการพื้นฐาน” ต่อวินาที}$$

“การดำเนินการพื้นฐาน” หมายถึงการบวก/ลบ/คูณ การเปรียบเทียบ การเข้าถึงสมาชิกอาร์เรย์ ฯลฯ

สองคำถามที่ครูควรฝึกให้นักเรียนถามทุกครั้ง ก่อนเริ่มเขียนโปรแกรม:

1. ขนาดข้อมูลใหญ่แค่ไหน? (n, m, . . .)
2. วิธีการของเราใช้เวลาทำงาน เพิ่มขึ้นตาม n อย่างไร?

สัญลักษณ์ Big-O แบบเข้าใจง่าย

เรากล่าวว่าโปรแกรมมีความซับซ้อน $O(f(n))$ ถ้าจำนวนการดำเนินการสำหรับ input ขนาด n ที่ใหญ่พอ มีขอบเขตบนเป็นค่าคงที่คูณกับ $f(n)$

อัตราการเติบโตที่พบบ่อย เรียงจากเร็วไปช้า:

$$1 \ll \log n \ll \sqrt{n} \ll n \ll n \log n \ll n^2 \ll n^3 \ll 2^n \ll n!$$

ตารางแนะนำ: input ขนาดเท่าใดที่วิ่งทันภายใน 1 วินาที ($\sim 10^8$ ops)?

ขนาด n	ความซับซ้อนที่ทำได้
$n \leq 10$	$O(n!)$
$n \leq 25$	$O(2^n)$
$n \leq 5000$	$O(n^2)$
$n \leq 200000$	$O(n \log n)$
$n \leq 10^7$	$O(n)$
$n \leq 10^{18}$	$O(\log n)$ หรือ $O(1)$

ตัวอย่างเปรียบเทียบสองวิธี

ปัญหา. อาร์เรย์เรียงแล้วขนาด n และ query หนึ่งค่า x ต้องการตรวจว่า x อยู่ในอาร์เรย์หรือไม่

สแกนเชิงเส้น $O(n)$:

```
bool contains(const int A[], int n, int x) {
    for (int i = 0; i < n; ++i)
        if (A[i] == x) return true;
    return false;
}
```

Binary search $O(\log n)$:

```
bool contains(const int A[], int n, int x) {
    int lo = 0, hi = n;
    while (lo < hi) {
        int mid = lo + (hi - lo) / 2;
        if      (A[mid] < x) lo = mid + 1;
        else if (A[mid] > x) hi = mid;
        else                 return true;
    }
    return false;
}
```

ผลต่าง. ที่ $n = 10^6$ และ 10^6 queries: เชิงเส้น = 10^{12} ops (16 นาที), binary search = $< 2 \cdot 10^7$

สรุปตอนที่ 1

- ก่อนเขียนโค้ดทุกครั้ง ประเมินจำนวนการดำเนินการก่อน
- เทียบ constraint ของโจทย์กับตารางความซับซ้อนข้างต้น
- ถ้าประเมินแล้วเกิน 10^8 ไม่ต้องเขียน ต้องหาวิธีหรือ data structure ที่ดีกว่า

คำแนะนำการสอน

ฝึกนักเรียนให้พูดออกมา ก่อนเริ่มเขียน “ข้อนี้ n เท่ากับ . . . ดังนั้นเราต้องการความซับซ้อน O(. . .)” นิสัยนี้ป้องกัน Time-Limit Exceeded (TLE) ได้กว่า 80% ของกรณี

เนื้อหาที่เหลือของวันนี้ = วิธีที่ STL container ช่วยลด $O(n^2)$ ให้เหลือ $O(n \log n)$

ตอนที่ 2: Array

Array แบบสแตติก

```cpp
int A[100];                    // 100 ints, uninitialized
int B[5] = {1, 2, 3, 4, 5};    // explicit initialization
int C[1000] = {0};             // zero-initialize all 1000 elements
```

การเข้าถึง $O(1)$: `A[i]` อ่าน/เขียนตำแหน่งที่ i
ดัชนีเริ่มที่ 0: `A[0]` คือสมาชิกตัวแรก, `A[n-1]` คือตัวสุดท้าย

ข้อผิดพลาดที่นักเรียนมักพบ

C++ ไม่ตรวจขอบเขตให้ การเข้าถึง `A[100]` สำหรับอาร์เรย์ขนาด 100 เป็น undefined behaviour: อาจ crash, อาจทำให้หน่วยความจำเสียโดยไม่ error, หรือทำงานผ่านในเครื่องนักเรียนแล้วผิดที่เครื่องตรวจ
ครูควรเตือนเรื่องนี้บ่อย ๆ

อาร์เรย์อยู่ที่ไหน?

อาร์เรย์ในฟังก์ชัน (stack): ใช้ได้ดี แต่ขนาด เล็ก ทั่วไปเครื่องตรวจให้ stack 1–8 MB เท่านั้น

```
int main() {
    int A[100];          // OK
    int B[10'000'000];   // CRASH: stack overflow
}
```

อาร์เรย์ระดับโลก (global / BSS): ขนาดใหญ่ได้ และถูกเริ่มต้นเป็น 0 อัตโนมัติ

```
int A[10'000'000];       // OK, 40 MB   BSS,    0
int main() { ... }
```

กฎปฏิบัติ. สำหรับอาร์เรย์ขนาดใหญ่ที่ทราบขนาดล่วงหน้า ประกาศเป็น global

อาร์เรย์หลายมิติ

```cpp
int grid[1000][1000];               // 10^6 ints = 4 MB, global
char board[8][8];
int  cube[50][50][50] = {0};

//          cache!
for (int i = 0; i < N; ++i)
    for (int j = 0; j < M; ++j)
        grid[i][j] = i + j;          // row-major:
```

การจัดเก็บหน่วยความจำ. อาร์เรย์ใน C++ จัดเก็บแบบ row-major: `grid[i][j]` อยู่ที่ตำแหน่ง `i*M + j` จากจุดเริ่มต้น

คำแนะนำการสอน

แนะนำให้นักเรียนวน `i` ก่อน แล้ว `j` ทีหลัง (row-major) จะเร็วกว่า column-major หลายเท่าในกรณีอาร์เรย์ใหญ่ เพราะ CPU cache ดึงข้อมูลที่อยู่ติดกันมาทีเดียว

โจทย์ตัวอย่าง – Prefix Sum

ปัญหา. ให้อาร์เรย์ A ขนาด n และ query จำนวน q แต่ละ query ถามหา $A[l] + A[l+1] + \cdots + A[r]$

แบบไร้เดียงสา $O(nq)$:

```
long long sum = 0;
for (int i = l; i <= r; ++i) sum += A[i];
```

ถ้า $n = q = 10^5$ จะใช้ 10^{10} ops ช้าเกินไป

คำนวณ prefix sum ก่อน แล้วตอบ query ในเวลา $O(1)$:

```
long long P[N+1];
P[0] = 0;
for (int i = 0; i < n; ++i) P[i+1] = P[i] + A[i];
// query [l, r]:      = P[r+1] - P[l]
```

รวม $O(n + q)$ ใช้แค่อาร์เรย์อย่างเดียว

ตอนที่ 3: Pointer

แค่พอเข้าใจเบื้องหลังของ STL

Pointer คืออะไร? – ฉบับย่อ

Pointer คือตัวแปรที่เก็บ ที่อยู่ในหน่วยความจำ ของตัวแปรอื่น

```
int x = 42;
int* p = &x;        // p    address    x
cout << *p;         // 42       ( p)
*p = 100;           //     x    100    p
```

ในการแข่ง CP เราไม่ค่อยเขียน `int*` ตรง ๆ เพราะใช้ `vector`, `set`, `map`, ... ซึ่งจัดการ pointer ภายในให้หมด

คำแนะนำการสอน

ไม่ต้องสอนนักเรียนทำงานกับ pointer แบบลึก ๆ ให้สอนแค่แนวคิด “ตัวแปรชี้ไปที่ข้อมูล” เพื่อให้เข้าใจว่า

- ทำไม `vector` ขยายขนาดได้
- ทำไม `set` วน iterate ออกมาเรียง
- ทำไมการเปลี่ยน vector อาจทำให้ iterator เก่าเสีย

STL Container เบื้องหลังใช้ Pointer อย่างไร

Container	โครงสร้างภายในโดยย่อ
`vector`	pointer ไปยัง buffer ใน heap + size + capacity
`stack`, `queue`	ห่อ `deque` ไว้ภายใน (ซึ่งเป็นบล็อกของ buffer ต่อกันด้วย pointer)
`priority_queue`	เก็บข้อมูลใน `vector` จัดเป็นรูป binary heap
`set`, `multiset`, `map`	red-black tree: แต่ละ node มี pointer ไปยังลูกซ้าย ลูกขวา และพ่อ
`unordered_set`, `unordered_map`	hash table: array ของ pointer ไปยังลิงค์ลิสต์ของ key (chaining)

สิ่งที่ครูควรย้ำให้นักเรียน: ทุก container ทำงานบน pointer เบื้องหลัง เราไม่ต้องเขียน pointer แต่ต้องรู้ว่า “ที่ตั้งของข้อมูลอาจย้าย” เมื่อ container ขยายขนาด จึงไม่ควรเก็บที่อยู่ของสมาชิก vector ไว้ใช้นาน ๆ

ตอนที่ 4: `std::vector`

container แรกที่ทุกคนควรเรียน

std::vector – อาร์เรย์ฉลาด

```cpp
#include <vector>
using namespace std;

vector<int> v;                      // empty
vector<int> w(10);                  // 10 zeros
vector<int> u(10, 7);               // 10 sevens
vector<int> a = {3, 1, 4, 1, 5, 9}; // from initializer list

v.push_back(42);                    // amortized O(1)
v.pop_back();                       // O(1)
int x = v[0];                       // O(1)
int n = v.size();                   // O(1)
v.clear();                          // O(n)
```

เพราะอะไรถึง “ฉลาด”? เมื่อ `push_back` แล้วเต็ม vector จะขยายความจุเป็น สองเท่า อัตโนมัติ จากการวิเคราะห์แบบ amortized: ต้นทุนเฉลี่ยต่อ `push_back` เป็น $O(1)$

Vector แบบ 2 มิติ

```
int n = 5, m = 7;
vector<vector<int>> grid(n, vector<int>(m, 0));   // 5x7 of zeros

grid[2][3] = 42;
for (int i = 0; i < n; ++i)
    for (int j = 0; j < m; ++j)
        cout << grid[i][j] << ' ';
```

เหมาะกับเมื่อ. ขนาดมิติยังไม่ทราบตอน compile หรือต้องการเปลี่ยนขนาดได้

หมายเหตุประสิทธิภาพ. อาร์เรย์ 2 มิติแบบ static เร็วกว่า vector ซ้อน vector เล็กน้อย (ไม่มี indirection พิเศษ) ถ้าวงในของอัลกอริทึมต้องการความเร็วสุดขีดบนกริดขนาดทราบล่วงหน้า ให้ใช้ `int grid[N][M]` แบบ global จะดีที่สุด

การวน vector

```cpp
vector<int> v = {3, 1, 4, 1, 5};

// 1.     index         ( index)
for (int i = 0; i < (int)v.size(); ++i)
    cout << v[i] << ' ';

// 2. range-for        ( index)
for (int x : v) cout << x << ' ';

// 3. range-for    reference          ( vector)
for (int& x : v) x *= 2;

// 4. iterator            ( STL)
for (auto it = v.begin(); it != v.end(); ++it)
    cout << *it << ' ';
```

ข้อผิดพลาดที่นักเรียนมักพบ

`v.size()` มีชนิดเป็น `size_t` (unsigned) ถ้าเขียน `for (int i = 0; i < v.size(); ++i)` แล้วเปรียบเทียบ `int` กับ `size_t` อาจได้คำเตือน (และเข้า loop ไม่ตามที่คิดในกรณีพิเศษ) ครูควรสอนให้แคสต์ `(int)v.size()` หรือใช้ `ssize/std::ssize`

Vector vs Array – เลือกใช้แบบไหนเมื่อใด?

คุณสมบัติ	`int A[N]`	`vector<int> v`
ขนาดคงที่	ใช่ (ขนาดที่ compile time)	ไม่ใช่ ขยายได้
ส่งเข้าฟังก์ชัน	ขนาดหาย	พกขนาดไปด้วย + มี `.at()`
หน่วยความจำ	stack หรือ BSS	heap
ความเร็ว	เร็วกว่าเล็กน้อย	ใกล้เคียงในทางปฏิบัติ
ความสะดวก	ต่ำ	สูง

ค่าเริ่มต้นใน

CP สมัยใหม่: ใช้ `vector` ก่อนเสมอ ใช้ raw array เมื่อจำเป็นต่อความเร็วของ inner loop จริง ๆ เท่านั้น

ตอนที่ 5: `std::pair`

จับสองค่ามาคู่กัน

`std::pair` – พื้นฐานที่ใช้บ่อยที่สุด

`pair` คือกล่องเก็บค่า 2 ค่าที่อาจต่างชนิดกัน

```cpp
#include <utility>          //    <vector>, <map>
pair<int, string> p = {7, "hello"};

cout << p.first;            // 7
cout << p.second;           // hello

auto [a, b] = p;            // structured binding (C++17)
```

ใช้ทำอะไรบ้าง?

- เก็บพิกัด (x, y) บนกริด หรือ (row, col)
- เก็บ (น้ำหนัก, ปลายทาง) ในกราฟ
- คืนค่าสองค่าจากฟังก์ชันโดยไม่ต้องสร้าง struct
- ใช้เป็นสมาชิกของ `vector`, `set`, `queue`, `priority_queue`

การเปรียบเทียบของ pair – Lexicographic

pair เปรียบเทียบโดย first ก่อน ถ้าเท่าจึงดู second

```
pair<int,int> a = {2, 9};
pair<int,int> b = {3, 1};
pair<int,int> c = {2, 10};

a < b;      // true  (2 < 3)
a < c;      // true  (first    , 9 < 10)
```

ประโยชน์ที่ใหญ่ที่สุด. sort, set, map, priority_queue ทำงานกับ pair ได้ทันทีไม่ต้องเขียน comparator

ตัวอย่าง: เรียงนักเรียนตามคะแนนมากไปน้อย ถ้าเท่าให้เรียงตามชื่อ

```
vector<pair<int,string>> v;       //    {,  }
v.push_back({80, "Bee"});
v.push_back({90, "Ann"});
v.push_back({80, "Ace"});

sort(v.begin(), v.end(),
     [](auto& x, auto& y){
         if (x.first != y.first) return x.first > y.first; //
         return x.second < y.second;                       //
```

ตัวอย่างการประยุกต์ – คืนค่าสองค่าจากฟังก์ชัน

โจทย์. หา “ค่าน้อยที่สุดและค่ามากที่สุด” ของ vector ในการสแกนครั้งเดียว

```
pair<int,int> minmax(const vector<int>& v) {
    int lo = v[0], hi = v[0];
    for (int x : v) {
        if (x < lo) lo = x;
        if (x > hi) hi = x;
    }
    return {lo, hi};
}

auto [lo, hi] = minmax(arr);
cout << lo << ' ' << hi;
```

หมายเหตุ. STL มี `std::minmax_element` อยู่แล้ว ใช้แทนได้
ตัวอย่างนี้ใช้สอนแนวคิด “คืนค่าสองค่าโดยไม่ต้องเขียน struct”

ของ 3 ค่าขึ้นไป? ใช้ `std::tuple<int,int,int>` ทำงานคล้าย pair

ตอนที่ 6: Stack และ Queue

std::stack – เข้าหลัง ออกก่อน

```cpp
#include <stack>
stack<int> s;

s.push(3);         // O(1)
s.push(1);
s.push(4);
cout << s.top(); // 4  --      push
s.pop();         //   4
cout << s.size();
cout << s.empty();
```

ทุก operation เป็น $O(1)$

ภาพในใจ. กองจานในซุปเปอร์มาร์เก็ต วางใหม่บนสุด หยิบจากบนสุด

ข้อผิดพลาดที่นักเรียนมักพบ

`s.pop()` ไม่ส่งค่ากลับ ต้องเรียก `s.top()` ดูค่าก่อน แล้วค่อย `pop()`

โจทย์ Stack – วงเล็บสมดุล

ปัญหา. สตริงที่ประกอบด้วย `()[]{}` เป็นวงเล็บสมดุลหรือไม่? เช่น `"({[]}())"` สมดุล, `"({[]}()))"` ไม่สมดุล

แนวคิด. push วงเล็บเปิดทุกตัว เจอวงเล็บปิดต้องเข้าคู่กับ top ของ stack

```
bool balanced(const string& s) {
    map<char,char> match = {{')','('}, {']','['}, {'}','{'}};
    stack<char> st;
    for (char c : s) {
        if (match.count(c) == 0) {                    //
            st.push(c);
        } else {                                      //
            if (st.empty() || st.top() != match[c]) return false;
            st.pop();
        }
    }
    return st.empty();
}
```

O(n) เวลา และหน่วยความจำ ตาราง `match` ทำให้โค้ดสั้นและขยายเพิ่ม bracket ใหม่ง่าย

โจทย์ Stack – Next Greater Element

ปัญหา. อาร์เรย์ขนาด n สำหรับแต่ละตำแหน่งหา index ของ “ค่าถัดไปทางขวาที่มากกว่า” (หรือ -1 ถ้าไม่มี)

แบบไร้เดียงสา $O(n^2)$: สำหรับแต่ละ i สแกน $i+1, i+2, \ldots$
แบบใช้ stack $O(n)$: เก็บ index ที่ยังไม่มีคำตอบ ในรูปแบบที่ค่าลดหลั่นลง (monotonic stack)

```
vector<int> next_greater(const vector<int>& a) {
    int n = a.size();
    vector<int> ans(n, -1);
    stack<int> st;              // indices, decreasing values
    for (int i = 0; i < n; ++i) {
        while (!st.empty() && a[st.top()] < a[i]) {
            ans[st.top()] = i;
            st.pop();
        }
        st.push(i);
    }
    return ans;
}
```

ทุก index push และ pop อย่างละครั้ง $\Rightarrow O(n)$

std::queue – เข้าก่อน ออกก่อน

```cpp
#include <queue>
queue<int> q;

q.push(3);          // O(1)
q.push(1);
q.push(4);
cout << q.front();  // 3 --
q.pop();            //    3
```

ภาพในใจ. ลูกค้าต่อแถวที่เคาน์เตอร์ มาก่อน-ได้รับบริการก่อน

ข้อผิดพลาดที่นักเรียนมักพบ

`q.pop()` ก็ไม่ส่งค่ากลับเช่นกัน อ่านด้วย `q.front()` ก่อนเสมอ

โจทย์ Queue – BFS บนกริด

ปัญหา. หาเส้นทางที่สั้นที่สุดจาก $(0,0)$ ไปยัง $(n-1, m-1)$ บนกริด โดยบางช่องเดินไม่ได้

แนวคิด. BFS = ขยายระยะทางจากต้นทางเป็นชั้น ๆ ใช้ queue

```
int dr[] = {-1,1,0,0}, dc[] = {0,0,-1,1};
int bfs(vector<string>& g) {
    int n = g.size(), m = g[0].size();
    vector<vector<int>> dist(n, vector<int>(m, -1));
    queue<pair<int,int>> q;
    q.push({0,0}); dist[0][0] = 0;
    while (!q.empty()) {
        auto [r, c] = q.front(); q.pop();
        for (int k = 0; k < 4; ++k) {
            int nr = r + dr[k], nc = c + dc[k];
            if (nr<0||nr>=n||nc<0||nc>=m) continue;
            if (g[nr][nc]=='#' || dist[nr][nc]!=-1) continue;
            dist[nr][nc] = dist[r][c] + 1;
            q.push({nr, nc});
        }
    }
    return dist[n-1][m-1];
}
```

ตอนที่ 7: `std::priority_queue`

Heap ในร่างใหม่

`std::priority_queue` คือ heap แบบทวิภาค (binary heap) ใช้ดึงค่ามากที่สุดได้รวดเร็ว

```
#include <queue>
priority_queue<int> pq;         // max-heap      ()
pq.push(3);                     // O(log n)
pq.push(7);
pq.push(1);
cout << pq.top();               // 7    O(1)
pq.pop();                       // O(log n)
```

อยากได้ค่าน้อยที่สุดบ้าง?

```
priority_queue<int, vector<int>, greater<int>> mn;  // min-heap
mn.push(3); mn.push(7); mn.push(1);
cout << mn.top();               // 1
```

โจทย์ Priority Queue – K ค่าที่มากที่สุด

ปัญหา. stream ของตัวเลข n ตัว (เข้ามาทีละตัว) ต้องแสดง K ค่าที่มากที่สุด จนถึงตอนนี้ ทุกขั้น

แบบไร้เดียงสา $O(nK)$: เก็บ list เรียงแล้วขนาด K insert ในตำแหน่งที่เหมาะ
แบบใช้ heap $O(n \log K)$: เก็บ min-heap ขนาด K

```
priority_queue<int, vector<int>, greater<int>> heap;   // min-heap

for (int x : stream) {
    heap.push(x);
    if ((int)heap.size() > K) heap.pop();   //   heap     K
}
// K         heap
```

เหตุผล. ใส่ทุกตัวเข้า heap ถ้าเกิน K ก็โยน ตัวเล็กสุด ทิ้ง สุดท้ายเหลือ K ตัวใหญ่สุดเสมอ

การประยุกต์อื่น. Dijkstra's shortest path, Huffman coding, การจัดตารางงาน

โจทย์ Priority Queue – รวม K list ที่เรียงแล้ว

ปัญหา. รวม K ลิสต์ที่เรียงแล้ว ทั้งหมดมี n สมาชิก ให้ผลลัพธ์เป็นลิสต์เรียง

แบบไร้เดียงสา $O(nK)$: ทุกครั้งสแกนหัวลิสต์ K ตัวเพื่อหาตัวเล็กที่สุด
แบบใช้ heap $O(n \log K)$: min-heap ของ `pair(value, listIndex)`

```
using pii = pair<int,int>;        // {value, listIndex}
priority_queue<pii, vector<pii>, greater<pii>> pq;
vector<int> idx(K, 0);            //

for (int i = 0; i < K; ++i)
    if (!lists[i].empty()) pq.push({lists[i][0], i});

vector<int> merged;
while (!pq.empty()) {
    auto [v, i] = pq.top(); pq.pop();
    merged.push_back(v);
    if (++idx[i] < (int)lists[i].size())
        pq.push({lists[i][idx[i]], i});
}
```

สังเกต. `pair` เปรียบเทียบโดย first ก่อน (จาก section ที่แล้ว) จึงไม่ต้องเขียน comparator

ตอนที่ 8: `set`, `multiset` และ `map`

balanced BST ในกระเป๋าหลัง

std::set – เรียง, ไม่ซ้ำ

```cpp
#include <set>
set<int> s;
s.insert(3);        // O(log n)
s.insert(1);
s.insert(4);
s.insert(1);        //   --
// s = {1, 3, 4}

s.erase(3);         // O(log n)
auto it = s.find(1);// O(log n); ==s.end()
cout << s.count(4); // 1       ()

for (int x : s) cout << x << ' ';   //         : 1 4
```

ภายใน: red-black tree
ทุก operation: O(log n)
การวน iterate: ได้ออกมาตามลำดับการเรียง

Killer feature – `lower_bound` / `upper_bound`

หาตัวที่น้อยที่สุดในเซตที่ $\geq$ x (หรือ $>$ x)

```
set<int> s = {1, 4, 7, 9, 13};
auto it = s.lower_bound(5);   //    7
auto jt = s.upper_bound(7);   //    9
auto kt = s.lower_bound(20);  // == s.end()    ()
```

การประยุกต์.

- “ก่อนหน้า / ถัดไป” ของ x ในชุดข้อมูลที่เปลี่ยนแปลงได้
- ตรวจสอบการชนของตารางนัด/การจอง
- หาจุดที่ใกล้ที่สุดบนเส้นจำนวน

ข้อผิดพลาดที่นักเรียนมักพบ

ใช้ `s.lower_bound(x)` ซึ่งเป็น member function ของ `set`
ไม่ใช่ `std::lower_bound(s.begin(), s.end(), x)` — ตัวหลังเป็น O(n) บน set

`std::multiset` – เรียง, ซ้ำได้

เหมือน set แต่อนุญาตให้มีค่าซ้ำ ทุก operation ยังเป็น $O(\log n)$

```cpp
#include <set>
multiset<int> ms;
ms.insert(3);
ms.insert(5);
ms.insert(3);              //   , ms = {3, 3, 5}
cout << ms.count(3);       // 2
cout << ms.size();         // 3

for (int x : ms) cout << x << ' ';   // 3 3 5   ()
```

เหมาะกับเมื่อ.

- ต้องการ “ค่ามากสุด / น้อยสุด” ใน multiset ของค่าที่อาจซ้ำ
- ต้องการ predecessor / successor และมีค่าซ้ำได้
- ต้องการ “ลบทีละครั้ง” ของค่าที่ซ้ำ (ไม่ใช่ลบหมดทีเดียว)

กับดักสำคัญ: `erase` ใน multiset

ปัญหา. ใน `multiset` การเรียก `ms.erase(value)` จะลบ ทั้งหมด ที่มีค่านั้น ไม่ใช่ลบแค่ตัวเดียว

```
multiset<int> ms = {3, 3, 5};
ms.erase(3);              // ms      {5}  --    2  !
```

ลบแค่ตัวเดียวต้องใช้ iterator:

```
multiset<int> ms = {3, 3, 5};
auto it = ms.find(3);     // O(log n)   iterator
if (it != ms.end()) ms.erase(it);   // ms = {3, 5}
```

ข้อผิดพลาดที่นักเรียนมักพบ

ครูควรย้ำเรื่องนี้กับนักเรียน เพราะเป็นบักที่หาเองยากมาก โปรแกรมยังคอมไพล์ผ่าน รันได้ ผลลัพธ์ออกมาผิดในกรณีที่มีค่าซ้ำเท่านั้น

โจทย์ Multiset – ค่า Min / Max ใน Sliding Window

ปัญหา. อาร์เรย์ a ขนาด n และความยาวหน้าต่าง k
ในแต่ละ index i ที่ $k-1 \le i < n$ ให้รายงานค่าน้อยที่สุดและมากที่สุดใน $a[i-k+1 \dots i]$

แบบไร้เดียงสา $O(nk)$: สแกนหน้าต่างใหม่ทั้งหน้าต่างทุกครั้ง
แบบใช้ multiset $O(n \log k)$:

```
multiset<int> win;
for (int i = 0; i < k; ++i) win.insert(a[i]);
cout << *win.begin() << ' ' << *win.rbegin() << '\n';

for (int i = k; i < n; ++i) {
    win.erase(win.find(a[i - k]));   //      !
    win.insert(a[i]);
    cout << *win.begin() << ' ' << *win.rbegin() << '\n';
}
```

หมายเหตุ. สำหรับ min/max เพียงอย่างเดียว deque แบบ monotonic ทำได้ใน $O(n)$ แต่ multiset ยืดหยุ่นกว่า (เช่นถามค่ามัธยฐานก็เพิ่มได้ทันที)

โจทย์ Multiset – ตารางการจองห้องเรียน

ปัญหา. ห้องเรียนรับนักเรียนได้พร้อมกัน C คน รับ event ทีละตัว: `ENTER`(t) คนเข้าเวลา t, `LEAVE`(t) คนออกเวลา t
ทุกครั้งที่ `ENTER` ให้รายงานว่ามีคนในห้องกี่คน

```
multiset<int> in_room;   //

void enter(int t) {
    in_room.insert(t);
    cout << "    " << in_room.size() << "  \n";
}
void leave(int t) {
    auto it = in_room.find(t);  //           t
    if (it != in_room.end()) in_room.erase(it);
}
```

สังเกต. เพราะอาจมีคนเข้าเวลาเดียวกันหลายคน multiset จำเป็น ไม่ใช่ set

`std::map` – คีย์ → ค่า

```cpp
#include <map>
map<string,int> freq;
freq["apple"] = 3;
freq["banana"] = 5;
freq["apple"]++;              // 4

for (auto& [k, v] : freq)
    cout << k << " -> " << v << '\n';
```

ทุก operation: $O(\log n)$
ลำดับการวน: ตามคีย์ น้อยไปมาก

ข้อผิดพลาดที่นักเรียนมักพบ

`operator[]` สร้างคีย์ใหม่ ถ้ายังไม่มี โดยมีค่าเป็น 0
ถ้าต้องการแค่ ตรวจการมีอยู่ โดยไม่ให้สร้าง ใช้ `find` หรือ `count` แทน

สรุปทางเลือก – set / multiset / unordered

Container	ค่าซ้ำได้?	การจัดเรียงและความซับซ้อน
`set` / `map`	ไม่	เรียงตามคีย์, $O(\log n)$
`multiset` / `multimap`	ใช่	เรียงตามคีย์, $O(\log n)$
`unordered_set` / `unordered_map`	ไม่	hash, $O(1)$ เฉลี่ย, แย่ที่สุด $O(n)$

เลือกอย่างไร

- ต้องการลำดับ / `lower_bound`? → `set` / `map`
- อนุญาตให้มีค่าซ้ำและยังต้องเรียง? → `multiset` / `multimap`
- แค่ตรวจการมีอยู่ / นับความถี่ ไม่สนลำดับ? → `unordered_*` (เร็วกว่า constant)
- Input ที่อาจถูกจงใจให้ชน hash ใช้ custom hash หรือกลับไปใช้ `set` ที่กันได้แน่

โจทย์ Map – นับความถี่

ปัญหา. ลิสต์ของคำ ให้เอาคำที่แตกต่างมาเรียงตามอักษร พร้อมจำนวนครั้ง

แบบไร้เดียงสา $O(n^2)$: วนซ้ำคู่หาคำที่ตรงกัน
แบบใช้ map $O(n \log n)$:

```
map<string,int> cnt;
for (const string& w : words) cnt[w]++;

for (auto& [w, c] : cnt)
    cout << w << ' ' << c << '\n';
```

สามบรรทัด ผลลัพธ์เรียงอักษรอัตโนมัติ เป็น idiom ที่นักเรียน CP ควรท่องจำ

คำแนะนำการสอน

ให้นักเรียนเปรียบเทียบเวลารันจริง ๆ ระหว่างวิธีคู่ขนานกัน $O(n^2)$ กับ map $O(n \log n)$ ที่ $n = 10^5$ ความรู้สึก “map ช่วยจริง” จะติดตัวไปตลอด

โจทย์ Set – ค่าที่ใกล้ที่สุดในเซตเปลี่ยนแปลงได้

ปัญหา. รักษาเซต S ที่รองรับ

- `insert`(x): เพิ่ม x เข้า S
- `nearest`(q): ตอบสมาชิกใน S ที่ใกล้ q ที่สุด

แบบไร้เดียงสา: สแกน S ทั้งหมดทุกครั้งที่ query — O(n) ต่อ query
แบบใช้ set + `lower_bound`: O(log n) ต่อ operation

```
set<int> s;

int nearest(int q) {
    auto it = s.lower_bound(q);       //      >= q      ()
    int ans = 0, best = INT_MAX;

    if (it != s.end())  { ans = *it;        best = *it - q; }
    if (it != s.begin()) {
        int lo = *prev(it);
        if (q - lo < best) ans = lo;
    }
    return ans;
}
```

ตอนที่ 9: Sort และ Binary Search

std::sort

```cpp
#include <algorithm>
vector<int> v = {3, 1, 4, 1, 5, 9, 2, 6};
sort(v.begin(), v.end());                          //
sort(v.begin(), v.end(), greater<int>());          //

// custom comparator
sort(v.begin(), v.end(),
     [](int a, int b){ return abs(a) < abs(b); });
```

ความซับซ้อน. $O(n \log n)$ ใน libstdc++ คือ introsort: quicksort + heapsort สำรอง

แบบเสถียร. `stable_sort` เป็น $O(n \log^2 n)$ รักษาลำดับเดิมของค่าที่เท่ากัน

ข้อผิดพลาดที่นักเรียนมักพบ

comparator ต้องเป็น strict weak ordering: `cmp(a,b)` และ `cmp(b,a)` ห้ามเป็น `true` พร้อมกัน และต้องมีคุณสมบัติส่งผ่าน (transitive) ใช้ `<` เท่านั้น ห้ามใช้ `<=` เพราะจะพังคุณสมบัตินี้

Binary Search ใน Standard Library

บนช่วงที่เรียงแล้ว:

```
vector<int> v = {1, 3, 5, 7, 9};
bool present  = binary_search(v.begin(), v.end(), 5);  // true
auto lo = lower_bound(v.begin(), v.end(), 5);  //     >= 5
auto up = upper_bound(v.begin(), v.end(), 5);  //      > 5
int idx_lo = lo - v.begin();                   // 2
int count_5 = up - lo;                         //     5
```

ทั้งสามฟังก์ชันเป็น $O(\log n)$

`lower_bound` และ `upper_bound` มีประโยชน์มากกว่า `binary_search` เพราะบอกได้ทั้ง มีหรือไม่ และอยู่ตำแหน่งใด

โจทย์รวม – Two Sum

ปัญหา. อาร์เรย์ n ตัวเลข และเป้าหมาย T มีเลขสองตัวที่บวกกันได้ T หรือไม่?

แบบไร้เดียงสา $O(n^2)$: วน loop ซ้อน loop
Sort + two pointers $O(n \log n)$:

```
sort(v.begin(), v.end());
int l = 0, r = (int)v.size() - 1;
while (l < r) {
    int s = v[l] + v[r];
    if      (s == T) return true;
    else if (s <  T) ++l;
    else             --r;
}
```

Hash-based $O(n)$ เฉลี่ย:

```
unordered_set<int> seen;
for (int x : v) {
    if (seen.count(T - x)) return true;
    seen.insert(x);
}
```

โจทย์รวม – ตัดเชือก

ปัญหา. มีเชือก n เส้น เส้นที่ i ยาว ℓ_i หน่วย
ต้องการตัดเชือกออกเป็น**ชิ้นที่ยาวเท่ากันทั้งหมด** โดยตัดเส้นใดก็ได้เป็นกี่ชิ้นก็ได้ (เศษทิ้งได้)
ให้**ได้ชิ้นรวมกันอย่างน้อย** k **ชิ้น**
ถามว่าความยาวต่อชิ้น L ที่ มากที่สุด ที่ทำได้คือเท่าใด?

ตัวอย่าง. เชือก $= [10, 24, 15, 7]$, $k = 4$

L	$\lfloor 10/L \rfloor$	$\lfloor 24/L \rfloor$	$\lfloor 15/L \rfloor$	$\lfloor 7/L \rfloor$	รวม	ผล
9	1	2	1	0	4	ทำได้
10	1	2	1	0	4	ทำได้
11	0	2	1	0	3	ทำไม่ได้

คำตอบ: $L = 10$

ตัดเชือก – วิธีคิดและโค้ด

สังเกตคุณสมบัติเชิงเอกพันธ์. ถ้าตัดได้ $\geq$ k ชิ้นที่ L ก็ตัดได้ $\geq$ k ชิ้นที่ทุก $L' < L$
(เพราะชิ้นสั้นกว่าจำนวนชิ้นย่อมไม่น้อยลง)
$\Rightarrow$ ขอบเขตของ “L ที่ทำได้” เป็นช่วงต่อเนื่อง ใช้ binary search บนคำตอบ

```
bool feasible(const vector<int>& l, int L, int k) {
    long long pieces = 0;
    for (int x : l) pieces += x / L;
    return pieces >= k;
}

int largest_L(const vector<int>& l, int k) {
    int lo = 1, hi = *max_element(l.begin(), l.end());
    while (lo < hi) {
        int mid = lo + (hi - lo + 1) / 2;          // round up
        if (feasible(l, mid, k)) lo = mid;         //
        else                     hi = mid - 1;
    }
    return lo;
}
```

เทคนิค. “binary search บนคำตอบ” พบบ่อยใน CP ครูควรสอนให้นักเรียนแยก “ฟังก์ชันตรวจสอบ `feasible`” กับ “การวนหา L ที่มากที่สุด” ออกจากกัน

สรุป – เลือก container อย่างไร

สิ่งที่ต้องการ	Container	ต้นทุนทั่วไป
buffer ขนาดคงที่	`vector` / array	$O(1)$ access
ต่อท้าย ขยายเป็นระยะ	`vector`	$O(1)$ amortized push
จับ 2 ค่ามาคู่กัน	`pair`	เปรียบเทียบ lexicographic
เข้าหลัง-ออกก่อน (LIFO)	`stack`	$O(1)$
เข้าก่อน-ออกก่อน (FIFO)	`queue`	$O(1)$
ค่ามาก/น้อยสุดเร็ว	`priority_queue`	$O(\log n)$
เซตเรียง ไม่ซ้ำ + predecessor	`set`	$O(\log n)$
เซตเรียง ซ้ำได้	`multiset`	$O(\log n)$
key $\rightarrow$ value เรียง	`map`	$O(\log n)$
membership / count อย่างเดียว	`unordered_set/map`	$O(1)$ เฉลี่ย
จัดเรียง	`sort`	$O(n \log n)$
ค้นในช่วงเรียงแล้ว	`lower/upper_bound`	$O(\log n)$

ข้อสรุปสำคัญ – สำหรับครูฝึก

1. **ฝึกประเมินก่อนเขียน.** ถามว่า n เท่าใด และวิธีของเราเป็น $O(?)$
2. **Array + STL container ไม่กี่ตัว แก้โจทย์แข่งได้ส่วนใหญ่.** ไม่ต้องเขียน data structure เอง
3. **Pointer** – เข้าใจแค่แนวคิดว่าทุก container ใช้ pointer เบื้องหลังก็พอ ไม่ต้องเขียนเอง
4. **`vector` เป็น container เริ่มต้น** และ **`pair` เป็นตัวประกอบที่ใช้ได้ทุกที่**
5. **`multiset` ระวัง `erase(value)` ลบหมด!** ให้นักเรียนจำคู่กับการใช้ `erase(iterator)`
6. **เห็น $O(n^2)$ ในใจ ให้ถามทันที.** container ใดจะลดเป็น $O(n \log n)$ ได้?

คำแนะนำการสอน

ในการสอนจริง ให้แต่ละหัวข้อมีโจทย์ก่อนเริ่ม-หลังเรียน เพื่อเห็นพัฒนาการ
และนำโค้ดในสไลด์ไปเปิดให้นักเรียนพิมพ์รันจริงในห้อง การได้รันเอง 1 ครั้ง = อ่าน 10 ครั้ง

แหล่งเรียนรู้และโจทย์ฝึก

หนังสือและเอกสารอ้างอิง.

- Antti Laaksonen — Competitive Programmer's Handbook (PDF ฟรี)
- Steven Halim — Competitive Programming 4
- `cppreference.com` — เอกสารอ้างอิง STL อย่างเป็นทางการ
- `cp-algorithms.com` — สูตรอัลกอริทึมและบทพิสูจน์

เว็บฝึกโจทย์.

- Codeforces — ใช้ tag: `implementation`, `data structures`, `sortings`, `binary search`
- USACO Guide — ระดับ Bronze และ Silver ครอบคลุมเนื้อหาทั้งหมดในสไลด์นี้
- AtCoder Beginner Contest
- programming.in.th — โจทย์ภาษาไทย เหมาะกับนักเรียนเริ่มต้น

ขอบคุณครับ

คำถามและข้อเสนอแนะยินดีต้อนรับ