Sorting

Sorting in C++

- ถ้าสมมติว่าเราต้องการเรียงเลขระหว่างการแข่งขัน มันคงไม่เป็นความคิดที่ดีในการเขียน sorting เองขึ้นมาตอนนั้น เพราะว่ามีการ implement sorting ที่ดีในภาษาโปรแกรมมาให้อยู่แล้ว
- ตัวอย่างเช่นใน C++ standard library มีฟังก์ชัน sort ที่เรียกใช้งานง่ายอยู่สำหรับการ sort array หรือโครงสร้างข้อมูลอื่น
- มีข้อดีหลายอย่างในการใช้ library function
  - อย่างแรก ช่วยประหยัดเวลาเพราะว่าไม่จำเป็นที่จำต้อง implement
  - อย่างที่สอง library นั้น implement มาถูกต้องแน่นอนและมีประสิทธิภาพ

- เราจะมาดูวิธีการใช้งาน C++ sort function
- code ต่อไปจะเรียงของใน vector จากน้อยไปมาก

```
vector<int> v = {4,2,5,3,5,8,3};
sort(v.begin(),v.end());
```

หลังจากการเรียง ข้อมูลใน vector จะเป็น [2,3,3,4,5,5,8]

ค่า default ของลำดับการเรียงคือ เรียงจากน้อยไปมาก แต่หากต้องลำดับจากมากไปน้อยให้ใช้

```
sort(v.rbegin(),v.rend());
```

- Array ทั่วไปสามารถถูกเรียงได้ดังนี้

```
int n = 7; // array size
int a[] = {4,2,5,3,5,8,3};
sort(a,a+n);
```

- string ก็สามารถถูกเรียงได้ดังนี้

```
string s = "monkey";
sort(s.begin(), s.end());
```

- การเรียง string หมายความว่าเป็นการ sort ตามอักขระ จากตัวอย่าง ผลลัพธ์ของ "monkey" จะได้ "ekmnoy"

Comparison operators

- ฟังก์ชัน sort นั้นต้องการ comparison operator ซึ่ง comparison operator เป็นตัวที่นิยามว่าชนิดข้อมูลนั้นสมาชิกจะถูกเรียงอย่างไร เมื่อมีการเรียง operation นี้จะถูกใช้เพื่อหาลำดับก่อนหลังของสมาชิกสองตัว
- ชนิดข้อมูลใน C++ ส่วนใหญ่มี built-in comparison operator และสมาชิกของชนิดข้อมูลเหล่านั้นสามารถถูกเรียงได้โดยอัตโนมัติอยู่แล้ว
- ตัวอย่างเช่น ตัวเลขถูก sort ตามค่าของมัน string ถูก sort ตามลำดับอักขระ

- คู่อันดับ (pair) ถูก sort ตามสมาชิกตัวแรก (first) อย่างไรก็ตามถ้าสมาชิกตัวแรกของ 2 คู่อันดับมีค่าเท่ากัน เราจะเรียงมันตามสมาชิกตัวที่สอง (second)

```
vector<pair<int,int>> v;
v.push_back({1,5});
v.push_back({2,3});
v.push_back({1,2});
sort(v.begin(), v.end());
for ( auto& i : v ) {
        cout << get<0>(i) << get<1>(i)  << endl;
        cout << i.first << i.second  << endl;
}
```

ลำดับของคู่อันดับที่ได้คือ (1,2), (1,5) และ (2,3)

- ในลักษณะเดียวกัน tuples ถูกเรียงตามสมาชิกตัวแรก ตามด้วยตัวที่สอง ตามด้วยตัวต่อๆไป

```
vector<tuple<int,int,int>> v;
v.push_back(make_tuple(2,1,4));
v.push_back(make_tuple(1,5,3));
v.push_back(make_tuple(2,1,3));
sort(v.begin(), v.end());
for ( auto& i : v ) {
    cout << get<0>(i) << get<1>(i) << get<2>(i) << endl;
}
```

- ซึ่งหลังจากนี้ tuple คือ (1,5,3),(2,1,3),(2,1,4)

User-defined structs

- Struct ที่เราเขียนขึ้นมาเองนั้น ไม่มี comparison operator มาให้โดยอัตโนมัติ
- operator ควรถูกนิยามภายใน struct ด้วยฟังก์ชัน operator< ซึ่ง parameter คือสมาชิกอีกตัวที่จะเทียบโดย operator จะคืนค่าจริงถ้าสมาชิกตัวที่เทียบน้อยกว่า parameter และเท็จในกรณีกลับกัน (ต้องมี const ด้วย)
- ตัวอย่าง
- struct P ต่อไปนี้เก็บ coordinate x และ y ของจุด comparison operator นิยามเพื่อที่ว่าจุดถูกเรียงตามแนวแกน x จากนั้นตามแนวแกน y

```
struct P {
        int x, y;
        bool operator<(const P& p)const {
                if (x != p.x) return x < p.x;
                else return y < p.y;
        }
};
```

```
int main()
{   struct P point[2];
    point[0].x=1;
    point[0].y=5;
    point[1].x=0;
    point[1].y=7;
    sort(point,point+2);
    for(auto p:point){
        cout<<p.x<<p.y<<endl;
    }
    return 0;
}
```

Comparison function

- นอกจากนี้เรายังสามารถ เขียน comparison function ไว้ภายนอกเพื่อเป็น callback function ให้กับ ฟังก์ชัน sort ได้ด้วย
- แต่ต้องรับ parameter 2 ตัว เพื่อเอาไว้เทียบกัน
- เช่นตัวอย่าง point ก่อนหน้า หากไม่เขียน **operator<** เราก็เขียนฟังก์ชันเองดังนี้

```
bool comp(struct P lhs, struct P rhs) { return lhs.x < rhs.x; }
```

เวลาจะ sort ให้เพิ่มฟังก์ชันเป็น parameter อีกตัว

```
sort(point,point+2,comp);
```

Comparison function

- อีกตัวอย่าง ฟังก์ชัน comp เป็น comparison function ในการเรียง string จากความยาวก่อน จากนั้นเป็นตามลำดับตัวอักษร

```
bool comp(string a, string b) {
        if (a.size() != b.size()) return a.size() < b.size();
        return a < b;
}
```

- จากนั้น vector ของ string สามารถถูกเรียงโดยการเรียก

```
sort(v.begin(), v.end(), comp);
```

Binary search

- วิธีทั่วไปในการค้นหาสมาชิกใน array คือการใช้ for loop ที่วนรอบผ่านสมาชิกแต่ละตัวใน array ตัวอย่างเน การค้นหา x ใน array

```
for (int i = 0; i < n; i++) {
        if (array[i] == x) {
                // x found at index i
        }
}
```

- ซึ่งเวลาในการทำงานเป็น O(n) เพราะว่ากรณีแย่สุดเราต้องตรวจสอบสมาชิกทุกตัวใน array ซึ่งถ้าลำดับของสมาชิกเป็นลำดับใดๆ การค้นหาตรงๆ นี้ก็เป็ฯวิธีการดีสุดที่ทำได้ เพราะว่าเราไม่มีข้อมูลในการค้นหา

- อย่างไรก็ตามถ้าข้อมูลใน array เรียงกัน สถานการณ์จะเปลี่ยนไป เราสามารถค้นหาเร็วขึ้นได้ เพราะว่ามีลำดับของสมาชิกใน array นั้น guide เรา
- ต่อไปเป็นการค้นหาแบบ binary search ซึ่งมีประสิทธิภาพสำหรับ array ที่มีการเรียงทำงานในเวลา O(logn)

Method 1

- วิธีการปกติในการ implement binary search คล้ายกับการค้นหาคำในพจนานุกรม การค้นหาจะเก็บช่วงที่ active ใน array ซึ่งเริ่มต้นจะเก็บสมาชิกทุกตัว หลังจากนั้นในแต่ละรอบเราจะแบ่งครึ่งช่วงนั้น
- ในแต่ละรอบ การค้นหาจะตรวจสอบสมาชิกตรงกลางของช่วงที่ active ถ้าสมาชิกตรงกลางเป็นตัวที่ต้องการค้นหา การค้นหาก็เสร็จ ถ้าไม่ใช่ การค้นหาจะทำงานต่ออย่าง recursive ไปยังส่วนทางซ้ายหรือทางขวาขึ้นอยู่กับค่าของสมาชิกตัวกลาง

```
int a = 0, b = n-1;

while (a <= b) {

        int k = (a+b)/2;

        if (array[k] == x) {

                // x found at index k

        }

        if (array[k] > x) b = k-1;

        else a = k+1;

}
```

- การ implement แบบนี้ ส่วนที่ active คือ a...b และเริ่มต้นเป็น 0...n–1 อัลกอริทึมนี้จะแบ่งครึ่งส่วนในแต่ละรอบ ดังนั้นเวลาในการทำงานเป็น O(logn)

Method 2

- อีกวิธีหนึ่งในการ implement binary search อยู่บนพื้นฐานของวิธีที่มีประสิทธิภาพในการวนผ่านสมาชิกใน array แนวคิดคือกระโดดแล้วลดความเร็วเมื่อใกล้กับตัวที่ต้องการ
- การค้นหาเริ่มต้นจากซ้ายไปขวา เริ่มต้นกระโดดไปยัง n/2 แต่ละขั้นความยาวของการกระโดดจะถูกแบ่งครึ่ง เริ่มจาก n/4 แล้วเป็น n/8 ไปเรื่อยๆ จนในที่สุดความยาวเป็น 1 หลังจากกระโดดก็จะพบเป้าหมายหรือไม่ก็ไม่มีเป้าหมายใน array

```
int k = 0;

for (int b = n/2; b >= 1; b /= 2) {

        while (k+b < n && array[k+b] <= x) k += b;

}

if (array[k] == x) {

        // x found at index k

}
```

- ระหว่างการค้นหา ตัวแปร b จะเก็บความยาวปัจจุบันของการกระโดด
- Time complexity ของ algorithm เป็น O(logn) เพราะว่า code ใน while loop ทำงานไม่เกินสองครั้งในแต่ละการกระโดด

C++ function

- C++ standard library นั้นมีฟังก์ชันที่มีหลักการของ binary search และทำงานใน logarithmic time
  - lower_bound จะคืนค่า pointer ของสมาชิกใน array ตัวแรกที่มีค่าอย่างน้อย x
  - upper_bound จะคืนค่า pointer ของสมาชิกใน array ตัวแรก ที่มีค่ามากกว่า x

ฟังก์ชันจะสมมติว่า array นั้นถูก**เรียงลำดับแล้ว(ถ้ายังไม่เรียงก็เรียงก่อน!!)** ถ้าไม่มีสมาชิกที่เราต้องการหา pointer จะชี้ไปยังสมาชิกตัวถัดจากสมาชิกตัวสุดท้ายใน array

ตัวอย่าง code ต่อไปเป็นการหาว่า array นั้นมีสมาชิกที่มีค่า x หรือไม่

```
auto k = lower_bound(array,array+n,x)-array;
if (k < n && array[k] == x) {
        // x found at index k
}
```

- ดังนั้นหากต้องการนับว่ามี x กี่ตัว ต้องทำอย่างไร

- ทั้งนี้สามารถใช้ equal_range ทำให้ code สั้นลงได้

```
auto r = equal_range(array, array+n, x);

cout << r.second-r.first << "\n";
```

ถ้าชนิดข้อมูลเป็น double ใช้

```
sort(begin(array),end(array));

auto r = equal_range(begin(array),end(array), x);
```

การหาคำตอบที่มีค่าน้อยที่สุด

- การใช้งานที่สำคัญของ binary search คือการหาตำแหน่งที่ทำให้ค่าของ function เปลี่ยน สมมติว่าเราต้องการหาค่า k ที่น้อยที่สุดที่ทำให้ฟังก์ชันเป็นจริง กำหนดให้เรามี function ok(x) ที่คืนค่า true ถ้า x เป็นคำตอบที่ถูกต้อง และ false ในกรณีอื่น เพิ่มเติม เรารู้ว่า ok(x) เป็น false เมื่อ x<k และเป็น true เมื่อ x>=k
- สามารถมองได้

x	0	1	...	k-1	k	k+1	...
ok(x)	false	false	...	false	true	true	...

- ค่า k สามารถถูกหาได้โดยใช้ binary search

```
int x = -1;
for (int b = z; b >= 1; b /= 2) {
        while (!ok(x+b)) x += b;
}
int k = x+1;
```

- การค้นหานั้นจะหาค่า x ที่มากที่สุดที่ ok(x) เป็น false ดังนั้นค่าต่อไป k=x+1 ก็คือค่าที่น้อยที่สุดที่ทำให้ ok(x) เป็นจริงนั่นเอง
- ความยาวของการกระโดดเริ่มต้น x จะต้องยาวพอ ตัวอย่างเช่นบางค่าที่เรารู้ว่า ok(x) เป็นจริง
- algorithm จะเรียก ฟังก์ชัน ok จำนวน O(logz) ครั้ง ดังนั้น total time complexity ขึ้นอยู่กับว่าฟังก์ชัน ok ใช้เท่าไร ตัวอย่างเช่นหากฟังก์ชัน ok ทำงานใน O(n) แล้ว total time complexity จะเป็น O(nlogz)

การหาค่าที่มากที่สุด

- binary search สามารถถูกใช้กับการหาค่าที่มากที่สุดสำหรับฟังก์ชันที่เริ่มแรกมีค่าเพิ่มขึ้นแล้วจากนั้นมีค่าลดลงได้
- งานของเราจะหาตำแหน่งที่ k ที่
  - f(x)<f(x+1) เมื่อ x<k และ
  - f(x)>f(x+1) เมื่อ x>=k
- แนวคิดคือใช้ binary search ในการหาค่ามากที่สุดของ x ที่ f(x) < f(x+1) นั่นคือ k=x+1 เพราะว่า f(x+1)>f(x+2) ตัวอย่าง code เป็นดังนี้

```
int x = -1;

for (int b = z; b >= 1; b /= 2) {

        while (f(x+b) < f(x+b+1)) x += b;

}

int k = x+1;
```

สังเกตว่า ไม่เหมือนกับการ binary search ทั่วไป ตัวอย่างนี้ไม่อนุญาตให้ค่าที่ติดกันของฟังก์ชันมีค่าเท่ากัน เพราะว่าในกรณีนี้มันจะไม่สามารถรู้ได้ว่าจะค้นหาต่ออย่างไร