Data structures

Linear data structures

- โครงสร้างข้อมูล (Data structure) เป็นวิธีการเก็บข้อมูลลงในหน่วยความจำของคอมพิวเตอร์ มันมีความสำคัญในการเลือกโครงสร้างข้อมูลที่เหมาะสมสำหรับปัญหา เพราะว่าแต่ละโครงสร้างข้อมูลนั้นมีข้อดีข้อเสียแตกต่างกันไป
- คำถามที่สำคัญคือ การดำเนินการใดมีประสิทธิภาพจากโครงสร้างข้อมูลที่เลือกมา
- ในเรื่องนี้เราจะแนะนำโครงสร้างข้อมูลที่สำคัญใน C++ standard library ก่อน มันเป็นความคิดที่ดีในการใช้ standard library เพราะว่าช่วยประหยัดเวลา จากนั้นในคราวต่อไปจะมีโครงสร้างข้อมูลที่ซับซ้อนขึ้นที่ไม่มีใน standard library

Static arrays

- เป็นที่ชัดเจนว่า array นั้นถูกใช้งานบ่อยในการแข่งขัน เมื่อไรที่มีกลุ่มของข้อมูลที่ต้องการเก็บและหลังจากนั้นต้องการเข้าถึงข้อมูลโดยใช้ index static array ก็จะถูกใช้งาน
- เนื่องจาก input size สูงสุดนั้นจะถูกระบุในโจทย์ array size จึงสามารถถูกประกาศเป็นค่ามากที่สุดของ input พร้อมทั้งมีส่วน extra นิดหน่อยเพื่อความปลอดภัย
- โดยทั่วไป array 1D 2D 3D ถูกใช้ในการแข่งขันละปัญหาแทบจะไม่ใช้ array ที่มิติสูงกว่านี้ ส่วนการดำเนินการของ array ได้แก่การ access ด้วย index การเรียงข้อมูล การค้นหาแบบ linear หรือ binary search

Dynamic arrays

- Dynamic array เป็น array ที่สามารถเปลี่ยนแปลงขนาดได้ระหว่างการประมวลผล Dynamic array ที่นิยมที่สุดใน C++ คือ vector ซึ่งสามารถใช้งานได้ใกล้เคียงกับ array ปกติ ควรใช้ vector แทน array ถ้าขนาดของลำดับสมาชิกไม่รู้ขณะ compile
- โดยทั่วไปเราจะกำหนดขนาดเริ่มต้นด้วยการประมาณ
- การดำเนินการของ vector ที่ใช้บ่อยได้แก่ push_back(), at(), การใช้ [ ], assign(), clear() erase() และ iterator สำหรับการท่องไปใน vectors
- นอกจากนี้ operation ที่ใช้บ่อยอีกสองอย่างได้แก่ sort และ search (lower_bound upper_bound binary_search)

- คราวก่อนที่เป็น array

```
auto k = lower_bound(array,array+n,x)-array;
```

- ถ้าเป็น vector

```
auto k = lower_bound(s.begin(),s.end(),x)-s.begin();
```

- binary_search

```
vector<int> s={2,3,3,3,5,5,6};
    if(binary_search (s.begin(), s.end(), 4)){
        cout<<"found";
    }else{
        cout<<"not found";
    }
```

- ตัวอย่าง code ต่อไปเป็นการสร้าง vector เปล่าและเพิ่มสมาชิกลงไป

```
vector<int> v;
v.push_back(3); // [3]
v.push_back(2); // [3,2]
v.push_back(5); // [3,2,5]
```

- หลังจากสร้างแล้ว สมาชิกสามารถถูกเข้าถึงได้เช่นเดียวกับ array

```
cout << v[0] << "\n"; // 3
cout << v[1] << "\n"; // 2
cout << v[2] << "\n"; // 5
```

- ฟังก์ชัน `size` จะคืนค่าจำนวนสมาชิกใน `vector` ตัวอย่างต่อไปเป็นการวน `print` ทุกสมาชิก

```
for (int i = 0; i < v.size(); i++) {
    cout << v[i] << "\n";
}
```

- มีวิธีในการวนรอบ vector ที่สั้นกว่าดังนี้

```
for (auto x : v) {
    cout << x << "\n";
}
```

auto เป็น default storage class

compiler จะเลือก type ให้

ซึ่งใช้ได้กับการเป็น local variable

auto มีตั้งแต่ C++11

- ฟังก์ชัน back จะคืนค่าสมาชิกตัวสุดท้ายใน vector และฟังก์ชัน pop_back จะเอาสมาชิกตัวสุดท้ายออก

```
vector<int> v;
v.push_back(5);
v.push_back(2);
cout << v.back() << "\n"; // 2
v.pop_back();
cout << v.back() << "\n"; // 5
```

- วิธีการสร้าง vector อีกวิธีคือกำหนดค่าตั้งแต่ประกาศ

```
vector<int> v = {2,4,2,5,1};
```

- นอกจากนี้ อีกวิธีในการประกาศ vector คือการระบุจำนวนและกำหนดค่าเริ่มต้นของแต่ละตัว

```
// size 10, initial value 0
vector<int> v(10);

// size 10, initial value 5
vector<int> v(10, 5);
```

- การ implement ภายในของ vector นั้นใช้ array ธรรมดา ถ้าขนาดของ vector เพิ่มขึ้นหรือเล็กลง array ใหม่จะถูก allocate และทุกสมาชิกจะถูกย้ายไปยัง array ใหม่ อย่างไรก็ตามไม่ค่อยเกิดเหตุการณ์นี้และโดยเฉลี่ยแล้ว time complexity ของ push_back เป็น O(1)
- http://www.cplusplus.com/reference/vector/vector/

string

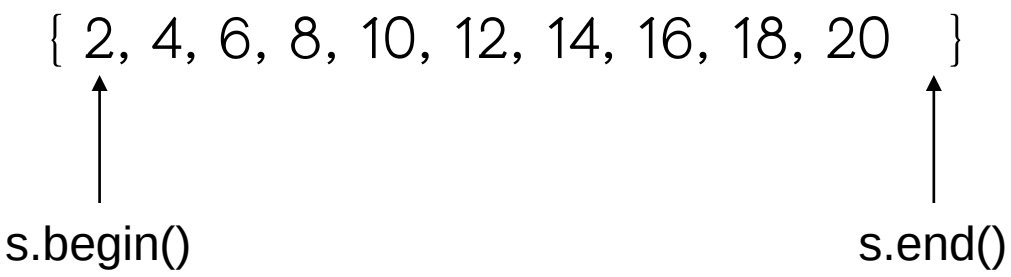
- โครงสร้าง string เป็น dynamic array เช่นกันคล้ายกับ vector เพิ่มเติมคือมี syntax พิเศษของ string ที่ไม่มีในโครงสร้างข้อมูลอื่น
- String สามารถรวมกันได้โดยใช้สัญลักษณ์ +
- ฟังก์ชัน substr(k,x) คืนค่า substring ที่เริ่มต้นตำแหน่งที่ k และมีความยาว x
- ฟังก์ชัน find(t) หาตำแหน่งของ substring t ที่พบตำแหน่งแรก

```
string a = "hatti";
string b = a+a;
cout << b << "\n"; // hattihatti
b[5] = 'v';
cout << b << "\n"; // hattivatti
string c = b.substr(3,4);
cout << c << "\n"; // tiva
```

http://www.cplusplus.com/reference/string/string/

Iterators and ranges

- หลายๆ ฟังก์ชันใน C++ standard library ที่ทำงานด้วย iterator
- Iterator คือตัวแปร ที่ชี้ไปยังสมาชิกในโครงสร้างข้อมูล
- Iterator ที่ถูกใช้บ่อยคือ begin และ end นิยามโดยช่วงที่เก็บสมาชิกทุกตัวในโครงสร้างข้อมูล iterator begin นั้นชี้ไปที่สมาชิกตัวแรกในโครงสร้างข้อมูล ส่วน end ชี้ไปยังตำแหน่งถัดจากตัวสุดท้าย หากมองจะได้แบบนี้



- สังเกตว่า iterator สองตัวนี้ต่างกัน s.begin() ชี้ไปยังสมาชิกแต่ s.end() ชี้ภายนอกโครงสร้างข้อมูล ดังนั้น range ถูกนิยามโดย iterator นั้นเป็นแบบ half-open

- Working with ranges
- iterators ถูกใช้ใน C++ standard library function ที่ถูกกำหนดช่วงของสมาชิกในโครงสร้างข้อมูล โดยทั่วไปเราต้องการที่จะดำเนินการกับสมาชิกทุกตัวในโครงสร้างข้อมูล ดังนั้น iterator begin และ end จะถูกกำหนดมาให้

- ตัวอย่าง code การเรียงข้อมูลใน vector โดยใช้ sort การเรียงข้อมูลจากมากไปน้อยโดยใช้ reverse และการสลับลำดับของสมาชิกโดยใช้ random_shuffle

```
sort(v.begin(), v.end());
reverse(v.begin(), v.end());
random_shuffle(v.begin(), v.end());
```

- ฟังก์ชันเหล่านี้สามารถใช้กับ array ธรรมดาได้ ในกรณีนี้เราก็จะส่ง pointer ให้ฟังก์ชันแทน iterator

```
sort(a, a+n);
reverse(a, a+n);
random_shuffle(a, a+n);
```

Other structures

- Bitset เป็น array ที่แต่ละค่าเป็นได้เพียง 0 หรือ 1 ตัวอย่างการสร้าง bitset ที่มีสมาชิก 10 ตัว

```
bitset<10> s;

s[1] = 1;

s[3] = 1;

s[4] = 1;

s[7] = 1;

cout << s[4] << "\n"; // 1

cout << s[5] << "\n"; // 0
```

- ประโยชน์ของการใช้งาน bitset คือว่าใช้ memory น้อยกว่า array ทั่วไป เพราะว่าแต่ละสมาชิกใน bitset นั้นใช้หน่วยความจำเพียงแค่ 1 บิต ตัวอย่างเช่น ถ้า n bit ถูกเก็บใน array ที่เป็น integer จะใช้หน่วยความจำ 32n bit แต่หากเป็น bitset ใช้เพียง n บิต
- เพิ่มเติมค่าของ bitset สามารถเปลี่ยนแปลงได้อย่างมีประสิทธิภาพโดยการใช้ bit operator ซึ่งทำให้เรา optimize algorithms ของเราได้
- ต่อไปเป็นการสร้าง bitset ก่อนหน้าอีกวิธี

```
bitset<10> s(string("0010011010")); // from right to left

cout << s[4] << "\n"; // 1

cout << s[5] << "\n"; // 0
```

- ฟังก์ชัน count คืนค่าจำนวนสมาชิกที่เป็น 1 ใน bitset

```
bitset<10> s(string("0010011010"));
cout << s.count() << "\n"; // 4
```

- ต่อไปเป็นตัวอย่างการดำเนินการทาง bit

```
bitset<10> a(string("0010110110"));
bitset<10> b(string("1011011000"));
cout << (a&b) << "\n"; // 0010010000
cout << (a|b) << "\n"; // 1011111110
cout << (a^b) << "\n"; // 1001101110
```

http://www.cplusplus.com/reference/bitset/bitset/

Linked list

- แม้ว่าเราจะพบ linked list ใน data structure และ algorithms แต่ linked list ไม่ค่อยแนะนำให้ใช้ในการแข่งกัน เพราะว่าไม่ค่อยมีประสิทธิภาพในการเข้าถึงสมาชิก(ต้อง search จากตัวแรกไปยังตัวสุดท้ายใน list) และการใช้งาน pointer ผิดพลาดง่าย ดังนั้นส่วนใหญ่ linked list เราจะใช้ vector แทน
- อย่างไรก็ตาม หากต้องใช้ list ใน c++ มี list ให้ใช้ ซึ่งใช้งานคล้ายกับ vector การดำเนินการที่ใช้บ่อยเช่น push_back() pop_back() insert() push_front() pop_front() เหมาะกับปัญหาที่มีการแทรกระหว่างสายข้อมูล
- ข้อระวัง list insert แล้ว iterator ตัวก่อนหน้าใช้ได้ปกติ แต่ vector iterator จะถูกต้องถ้าใช้ push-back

Deque

- Deque เป็น dynamic array ที่สามารถเปลี่ยนแปลงขนาดได้ที่จุดปลายทั้งสองข้างของ array เช่นเดียวกับ vector deque นั้นมีฟังก์ชัน push_back และ pop_back แต่มันยังมี push_front และ pop_front เพิ่มเข้ามาในขณะที่ vector ไม่มี

```
deque<int> d;

d.push_back(5); // [5]

d.push_back(2); // [5,2]

d.push_front(3); // [3,5,2]

d.pop_back(); // [3,5]

d.pop_front(); // [5]
```

- การ implement ภายในของ deque นั้นซับซ้อนกว่า vector ด้วยเหตุนี้ทำให้ deque ทำงานช้ากว่า vector
- แต่อย่างไรก็ตามทั้ง การเพิ่มข้อมูลและลบข้อมูลใช้เวลา O(1) โดยเฉลี่ย
- โครงสร้างนี้เหมาะกับการทำ sliding windows
- http://www.cplusplus.com/reference/deque/deque/

Stack

- Stack เป็นโครงสร้างข้อมูลที่มีสองการดำเนินการที่ใช้ O(1) นั่นคือการเพิ่มข้อมูลไว้บนสุด(push) และนำข้อมูลบนสุดออก (pop) นอกจากนี้ยังสามารถสอบถามข้อมูลตัวบนสุดได้

```
stack<int> s;

s.push(3);

s.push(2);

s.push(5);

cout << s.top(); // 5

s.pop();

cout << s.top(); // 2
```

http://www.cplusplus.com/reference/stack/stack/

- stack เป็นส่วนหนึ่งที่ถูกใช้ในหลายๆ algorithms เช่น จับคู่วงเล็บ, การคำนวณ postfix, แปลง infix เป็น postfix, หา strongly connected component เป็นต้น
- stack ทำงานแบบ Last in first out โดย insert(push) ใน O(1) และ delete(pop) ใน O(1) ทางปลายด้านหนึ่ง นอกจากนี้ยังมี top() สอบถามตัวบนสุด และ empty()

Queue

- Queue มี 2 O(1) operations นั่นคือเพิ่มข้อมูลไปต่อท้ายใน Queue และนำเอาข้อมูลด้านหน้าออกจาก Queue การเข้าถึงข้อมูลใน Queue เข้าถึงได้เพียงตัวหน้าสุด และหลังสุดของ Queue (ข้อควรระวัง มันใช้ push/pop เหมือน stack แต่ทำงานคนละทาง)

```
queue<int> q;
q.push(3);
q.push(2);
q.push(5);
cout << q.front(); // 3
q.pop();
cout << q.front(); // 2
```

http://www.cplusplus.com/reference/queue/queue/

- queue ถูกใช้ใน algorithm ที่คล้าย breadth first search(bfs) โดย queue insert ใน O(1) โดยการ enqueue เข้าไปด้านท้าย และ delete ใน O(1) โดย dequeue ทางด้านหน้าออก
- การทำงานเช่นนี้เราเรียกว่า First in first out