

เอกสารประกอบการอบรม **Data Structures and Algorithms** **(STL)**

ค่ายคอมพิวเตอร์โอลิมปิก สอว. ค่าย 2 2/2566
ศูนย์โรงเรียนสามเสนวิทยาลัย - มหาวิทยาลัยธรรมศาสตร์
ระหว่างวันที่ 18 มีนาคม – 3 เมษายน 2567

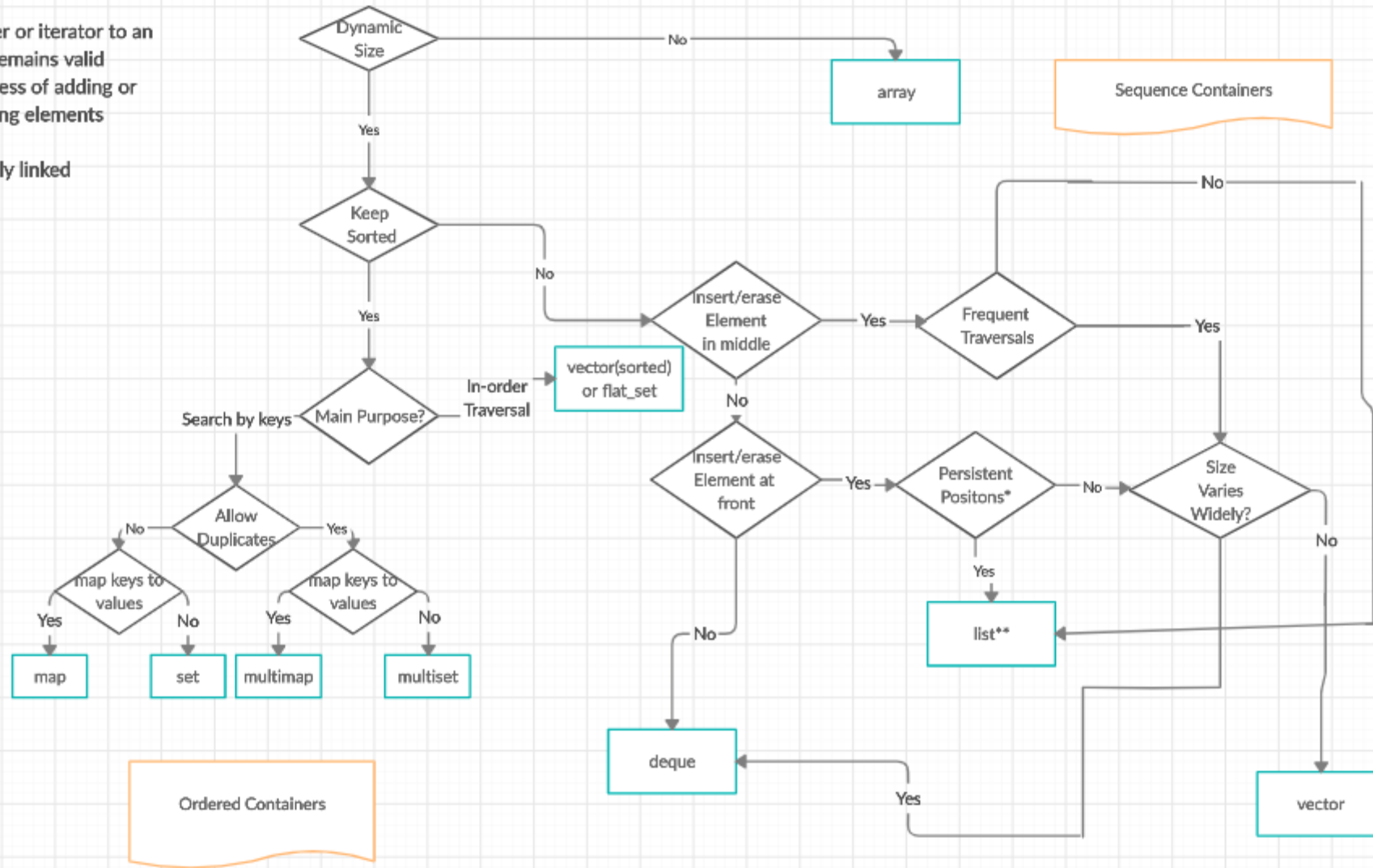
โดย
สาขาวิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์และเทคโนโลยี
มหาวิทยาลัยธรรมศาสตร์

Agenda

- C++ Standard Template Library (STL)
 - Containers
 - Sequence Containers
 - Adaptors
 - Associative Containers
 - Unordered Associative Containers
 - Algorithms
 - Iterators
 - Functions
 - Pairs
 - Others
- Recursive algorithms
 - Generating permutations
 - Generating subsets
 - Backtracking
- Binary search
 - Binary search
 - Search for optimal solution
- Minimizing sums

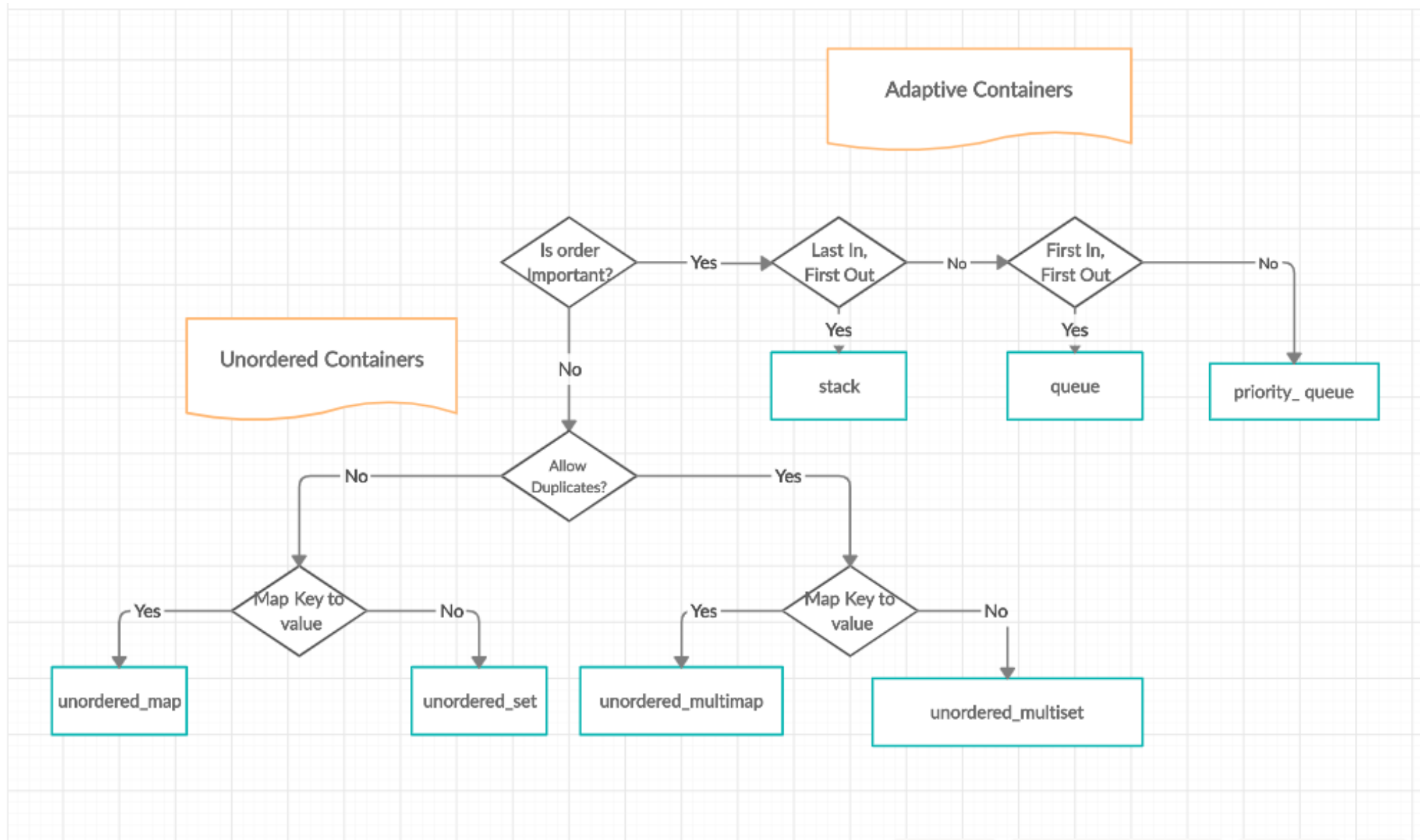
*pointer or iterator to an elem. remains valid regardless of adding or removing elements

**doubly linked



Ordered Containers

Sequence Containers



Iterators

- ใช้สำหรับเป็น Pointer ชี้ไปยัง Memory Addresses ของ STL Containers ต่าง ๆ
- Operations:
 - `begin()` คือฟังก์ชันที่จะคืนค่าตำแหน่งเริ่มต้นของ Container
 - `end()` คือฟังก์ชันที่จะคืนค่าตำแหน่งหลังสุดท้ายของ Container

Pairs

- เป็น Container ที่ใช้แทนคู่อันดับ (E1, E2) โดยที่ E1 และ E2 คือชนิดข้อมูล
- `make_pair()` คืนค่า pair ที่ถูก initialized ด้วยค่าที่ส่งเป็นพารามิเตอร์
- References:
 - `first` อ้างถึงสมาชิกตัวแรก (E1)
 - `Second` อ้างถึงสมาชิกตัวที่สอง (E2)

Data Structures

- โครงสร้างข้อมูล (Data Structures) เป็นขั้นตอนหนึ่งในการเก็บและจัดการกับข้อมูล
- แต่ละโครงสร้างข้อมูลมีจุดอ่อนจุดแข็งต่างกันออกไป เราควรต้องเข้าใจและรู้เกี่ยวกับ Time complexity ของ DS แต่ละอัน
- ดังนั้นการออกแบบ Algorithm ต้องเลือก DS ที่มีประสิทธิภาพในการ Insert, search, delete, query, update
- ทำไมไม่ควรใช้ Built-in Libraries ?

Linear DS

- Linear DS คือ DS ที่ Elements นั้นสร้างจาก Linear sequence กล่าวคือมันมีการเรียงจาก Left-right; Top-bottom
- เป็น DS ที่มีความสำคัญมาก
- ประกอบไปด้วย
 - Statis array (1D, 2D Arrays)
 - Dynamically-Resizable Array (Vector)

Dynamic Array

- A vector is a dynamic array that allows us to efficiently add and remove elements at the end of the structure.

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

Dynamic Array

Then, the elements can be accessed like in an ordinary array:

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

Another way to create a vector is to give a list of its elements:

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

We can also give the number of elements and their initial values:

```
vector<int> a(8); // size 8, initial value 0
vector<int> b(8,2); // size 8, initial value 2
```

Dynamic Array

The function `size` returns the number of elements in the vector. For example, the following code iterates through the vector and prints its elements:

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

A shorter way to iterate through a vector is as follows:

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

The function `back` returns the last element of a vector, and the function `pop_back` removes the last element:

```
vector<int> v = {2,4,2,5,1};  
cout << v.back() << "\n"; // 1  
v.pop_back();  
cout << v.back() << "\n"; // 5
```

Vectors are implemented so that the `push_back` and `pop_back` operations work in $O(1)$ time on average. In practice, using a vector is almost as fast as using an ordinary array.

Iterators and Ranges

Iterators

[5, 2, 3, 1, 2, 5, 7, 1]



v.begin()



v.end()

Ranges

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

```
vector<int> v = {2, 3, 3, 5, 7, 8, 8, 8};  
auto a = lower_bound(v.begin(), v.end(), 5);  
auto b = upper_bound(v.begin(), v.end(), 5);  
cout << *a << " " << *b << "\n"; // 5 7
```

deque

```
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]
```

The operations of a deque also work in $O(1)$ average time. However, deques have larger constant factors than vectors, so deques should be used only if there is a need to manipulate both ends of the array.

Stack

- Stack นั้นมีการ Insert (push) และ delete (pop) จาก Top เท่านั้น [Last In First Out]
- มีลักษณะการเพิ่มสมาชิกไว้ที่ท้ายคิวและเอาออกที่หัวคิว
- Operations:
 - push() เพิ่มสมาชิกเข้าไปในคิว
 - pop() ป้อนสมาชิกหัวแถวออก
 - top() คำนวณสมาชิกหัวแถว
 - empty() คำนวณจริงเมื่อคิวเป็นคว่าง คำนวณเท็จเมื่อคิวมีสมาชิกอย่างน้อย 1 ตัว
 - size() คำนวณจำนวนสมาชิกทั้งหมด

Queue (แถวคอย)

- เป็น Container ที่มีลักษณะการดำเนินการแบบ “เข้าก่อนออกก่อน” หรือ “First In First Out” หรือ FIFO
- มีลักษณะการเพิ่มสมาชิกไว้ที่ท้ายคิวและเอาออกที่หัวคิว
- Operations:
 - push() เพิ่มสมาชิกเข้าไปในคิว
 - pop() ป้อนสมาชิกหัวแถวออก
 - front() คำนวณค่าสมาชิกหัวแถว
 - empty() คำนวณค่าจริงเมื่อคิวเป็นคิวงว่าง คำนวณเท็จเมื่อคิวมีสมาชิกอย่างน้อย 1 ตัว
 - size() คำนวณค่าจำนวนสมาชิกที่อยู่ในแถวคอย

ตัวอย่าง

- ให้ดูในไฟล์ `queue_stack_pq.cpp`

Non-Linear DS with Built-in Libraries



Priority Queue

- Priority Queue เป็น Container ที่ถูกออกแบบมาให้หัวแถวจะต้องมีค่ามากที่สุดเสมอ (เทียบกับสมาชิกตัวอื่นในคิว)
- Operators:
 - push() เพิ่มสมาชิกเข้าไปในคิว
 - pop() ป้อนสมาชิกหัวแถวออก
 - top() **คืนค่าสมาชิกหัวแถว**
 - empty() คืนค่าจริงเมื่อคิวเป็นคิวง่าง คืนเท็จเมื่อคิวมีสมาชิกอย่างน้อย 1 ตัว
 - size() คืนค่าจำนวนสมาชิกที่อยู่ในแถวคอย
- เราสามารถ customize ตัวเปรียบเทียบ (Comparator) หรือกำหนด priority ของ queue เองได้
- Insertion and removal take **$O(\log n)$** time, and retrieval takes **$O(1)$** time.

Priority Queues

- A priority queue is a multiset that supports element insertion and, depending on the type of the queue, retrieval and removal of either the minimum or maximum element.
- Insertion and removal take $O(\log n)$ time, and retrieval takes $O(1)$ time.

Priority Queues

By default, the elements in a C++ priority queue are sorted in decreasing order, and it is possible to find and remove the largest element in the queue. The following code illustrates this:

```
priority_queue<int> q;  
q.push(3);  
q.push(5);  
q.push(7);  
q.push(2);  
cout << q.top() << "\n"; // 7  
q.pop();  
cout << q.top() << "\n"; // 5  
q.pop();  
q.push(6);  
cout << q.top() << "\n"; // 6  
q.pop();
```

If we want to create a priority queue that supports finding and removing the smallest element, we can do it as follows:

```
priority_queue<int, vector<int>, greater<int>> > q;
```

ตัวอย่าง

- ให้ดูในไฟล์ `queue_stack_pq.cpp`

Sets (เซต)

- เซตคือ Container ที่สมาชิกแต่ละตัวต้องมีความแตกต่างกัน (Unique)
- สมาชิกของ Set ไม่สามารถถูกแก้ไขตรง ๆ ได้ อาจแก้ไขได้ด้วยการลบสมาชิกตัวที่ต้องการแก้ไขออกและเพิ่มค่าที่แก้ไขแล้วลงไปใหม่
- Operation:
 - begin() คำนวณค่า Iterator ที่ไปยังสมาชิกตัวแรกของเซต
 - end() คำนวณค่า Iterator ที่ไปยังสมาชิกตัวหลังตัวสุดท้ายของเซต
 - size() คำนวณค่าจำนวนสมาชิกที่อยู่ในเซต
 - insert() เพิ่มสมาชิกลงไปใน Set
 - upper_bound() คำนวณค่า iterator ไปยังสมาชิกตัวแรกที่เท่ากับค่าพามิเตอร์ที่ใส่เข้าไป หรือ สมาชิกตัวที่จะไม่มาก่อนค่าดังกล่าว
 - lower_bound() คำนวณค่า iterator ไปยังสมาชิกตัวแรกที่เท่ากับค่าพามิเตอร์ที่ใส่เข้าไป หรือ สมาชิกตัวที่จะไม่มาทีหลังค่าดังกล่าว
- ใช้ Iterator ในการเข้าถึงสมาชิกภายในเซต

Set

- `set` is based on a balanced binary search tree and its operations work in $O(\log n)$ time.
- `unordered_set` is based on a hash table and its operations work, on average,¹ in $O(1)$ time.

¹The worst-case time complexity of the operations is $O(n)$, but this is very unlikely to occur.

```
set<int> s;  
s.insert(3);  
s.insert(2);  
s.insert(5);  
cout << s.count(3) << "\n"; // 1  
cout << s.count(4) << "\n"; // 0  
s.erase(3);  
s.insert(4);  
cout << s.count(3) << "\n"; // 0  
cout << s.count(4) << "\n"; // 1
```

```
set<int> s;  
s.insert(3);  
s.insert(3);  
s.insert(3);  
cout << s.count(3) << "\n"; // 1
```

Set

A set can be used mostly like a vector, but it is not possible to access the elements using the `[]` notation. The following code prints the number of elements in a set and then iterates through the elements:

```
cout << s.size() << "\n";  
for (auto x : s) {  
    cout << x << "\n";  
}
```

The function `find(x)` returns an iterator that points to an element whose value is `x`. However, if the set does not contain `x`, the iterator will be `end()`.

```
auto it = s.find(x);  
if (it == s.end()) {  
    // x is not found  
}
```


Set

- **Ordered Sets** The main difference between the two C++ set structures is that **set** is *ordered*, while **unordered_set** is not. Thus, if we want to maintain the order of the elements, we have to use the set structure.
 - For example, consider the problem of finding the smallest and largest value in a set. To do this efficiently, we need to use the set structure. Since the elements are sorted, we can find the smallest and largest value as follows

```
auto first = s.begin();  
auto last = s.end(); last--;  
cout << *first << " " << *last << "\n";
```

Set

- The set structure also provides the functions `lower_bound(x)` and `upper_bound(x)` that return an iterator to the smallest element in a set whose value is at least or larger than `x`, respectively.

```
cout << *s.lower_bound(x) << "\n";  
cout << *s.upper_bound(x) << "\n";
```

- In both the functions, if the requested element does not exist, the return value is `end()`.

ตัวอย่าง

- ให้ดูในไฟล์ sets.cpp

Map

- Map เป็น container ที่มีโครงสร้างแบบ Key แมพไปยัง Value โดย Keys จะต้องไม่ซ้ำกัน
- ใช้ Pair เพื่อแทนคู่อันดับการแมพ (Key, Value)
- Operations ที่สำคัญ:
 - pair_insert() เพิ่มสมาชิกใหม่เป็นคู่อันดับ (Key, Value)
 - erase() ลบสมาชิกโดยใช้ Key หรือ Iterator ในตำแหน่งที่ต้องการลบเป็นพารามิเตอร์
 - clear() ลบสมาชิกทุกตัวใน map
 - find() คืนค่า iterator ของคู่อันดับที่ค่า Key ตรงกับพารามิเตอร์
- สามารถเข้าถึง/เปลี่ยนแปลงค่า Value ด้วยการใช้อุปกรณ์ [] เช่น mymap[key]
- unordered_map จะคล้ายกับ map แต่การเก็บค่าไม่ได้เป็นแบบเรียงลำดับ เพราะฉะนั้นจึงไม่อาจจะทำ operations บางอย่างเช่น upper_bound หรือ lower_bound ได้

Maps

- A map is a set that consists of key-value pairs. A map can also be seen as a generalized array. While the keys in an ordinary array are always consecutive integers $0, 1, \dots, n-1$, where n is the size of the array, the keys in a map can be of any data type and they do not have to be consecutive values.
- The C++ standard library contains two map structures that correspond to the set structures: **map** is based on a balanced binary search tree and accessing elements takes $O(\log n)$ time, while **unordered_map** uses hashing and accessing elements takes $O(1)$ time on average.

Maps

The following code creates a map whose keys are strings and values are integers:

```
map<string,int> m;  
m["monkey"] = 4;  
m["banana"] = 3;  
m["harpsichord"] = 9;  
cout << m["banana"] << "\n"; // 3
```

If the value of a key is requested but the map does not contain it, the key is automatically added to the map with a default value. For example, in the following code, the key “aybabbu” with value 0 is added to the map.

```
map<string,int> m;  
cout << m["aybabbu"] << "\n"; // 0
```

The function `count` checks if a key exists in a map:

```
if (m.count("aybabbu")) {  
    // key exists  
}
```

Then, the following code prints all keys and values in a map:

```
for (auto x : m) {  
    cout << x.first << " " << x.second << "\n";  
}
```

ตัวอย่าง

- ให้ดูในไฟล์ maps.cpp

Algorithms

- `sort(first_iterator, last_iterator, [comparator])` รับพารามิเตอร์เข้ามาเป็น iterators 2 ตัว เป็นจุดเริ่มต้นและจุดสิ้นสุดของตำแหน่งที่ต้องการจะทำการเรียงลำดับ
- `reverse(first_iterator, last_iterator)` รับพารามิเตอร์เข้ามาเป็น iterators 2 ตัว เป็นจุดเริ่มต้นและจุดสิ้นสุดของตำแหน่งที่ต้องการจะทำการกลับด้านลิสต์
- `max_element(first_iterator, last_iterator)/min_element(first_iterator, last_iterator)` รับพารามิเตอร์เข้ามาเป็น iterators 2 ตัว เป็นจุดเริ่มต้นและจุดสิ้นสุดของตำแหน่งที่ต้องการหาค่าสูง/ต่ำสุด (ค่าที่คืนจะเป็น iterator ชี้ไปยังตำแหน่งผลลัพธ์)
- `binary_search(first_iterator, last_iterator, x)` รับพารามิเตอร์เข้ามาเป็น iterators 2 ตัว เป็นจุดเริ่มต้นและจุดสิ้นสุดของตำแหน่งที่ต้องการทดสอบว่า x นั้นปรากฏอยู่ในลิสต์หรือไม่
- `lower_bound(first_iterator, last_iterator, x)` รับพารามิเตอร์เข้ามาเป็น iterators 2 ตัว เป็นจุดเริ่มต้นและจุดสิ้นสุดของตำแหน่งที่ต้องการคืนค่า iterator ที่เป็น lower_bound
- `upper_bound(first_iterator, last_iterator, x)` รับพารามิเตอร์เข้ามาเป็น iterators 2 ตัว เป็นจุดเริ่มต้นและจุดสิ้นสุดของตำแหน่งที่ต้องการคืนค่า iterator ที่เป็น upper_bound

แบบฝึกหัด 3

- กำหนดให้ List ต่อไปนี้ 10, 12, 15, 19, 20, 1, -3, -7, 8, 14, 12, 30, 58, -100, 44
- ให้เขียนโปรแกรมเพื่อตอบคำถามต่อไปนี้
 - ให้พิมพ์ค่า List แบบเรียงลำดับจากน้อยไปหามาก
 - ให้พิมพ์ค่า List แบบเรียงลำดับจากมากไปหาน้อย
 - หาค่าสูงสุด/ต่ำสุด ของ List ดังกล่าว
 - ให้หาอย่างมีประสิทธิภาพว่าใน List มีค่า 7 อยู่หรือไม่
 - ให้หาอย่างมีประสิทธิภาพว่าใน List มีค่า -7 อยู่หรือไม่
 - ให้พิมพ์ค่าแบบเรียงลำดับจากน้อยไปหามากตั้งแต่ตำแหน่งที่มีค่ามากกว่า 0 (เรียกใช้ lower_bound)
 - ให้พิมพ์ค่าแบบเรียงลำดับจากน้อยไปหามากตั้งแต่ตำแหน่งที่มีค่าน้อยกว่า 40 (เรียกใช้ upper_bound)

DATA STRUCTURES WITH OUR OWN LIBRARIES

- Graph
- Union-find Disjoint Sets
- Segment Tree
- Binary Indexed (Fenwick) Tree

Algorithms



Sorting Algorithms

- $O(n^2)$ comparison-based sorting algorithms: Bubble/Selection/Insertion Sort
 - พยายามหลีกเลี่ยง
- $O(n \log n)$ comparison-based sorting algorithms: Merge/Heap/Quick Sort, etc.
 - เป็น Algorithm สำหรับจัดเรียงประเภท Comparison-based ที่เร็วในเกือบทุกกรณี
 - `sort()`, `partial_sort()`, `stable sort()` [ใน STL C++ algorithm]
- Special purpose sorting algorithms: $O(n)$ Counting/Radix/Bucket Sort, etc.

It is appropriate to discuss two operations commonly performed on Arrays: **Sorting** and **Searching**. These two operations are well supported in C++ and Java.

There are *many* sorting algorithms mentioned in CS books [7, 5, 54, 12, 40, 58], e.g.

1. $O(n^2)$ comparison-based sorting algorithms: Bubble/Selection/Insertion Sort, etc. These algorithms are (awfully) slow and usually avoided in programming contests, though understanding them might help you solve certain problems.
2. $O(n \log n)$ comparison-based sorting algorithms: Merge/Heap/Quick Sort, etc. These algorithms are the default choice in programming contests as an $O(n \log n)$ complexity is optimal for comparison-based sorting. Therefore, these sorting algorithms run in the ‘best possible’ time in most cases (see below for special purpose sorting algorithms). In addition, these algorithms are well-known and hence we do not need to ‘reinvent the wheel’⁵—we can simply use `sort`, `partial_sort`, or `stable_sort` in C++ STL `algorithm` (or `Collections.sort` in Java) for standard sorting tasks. We only need to specify the required comparison function and these library routines will handle the rest.
3. Special purpose sorting algorithms: $O(n)$ Counting/Radix/Bucket Sort, etc. Although rarely used, these special purpose algorithms are good to know as they can reduce the required sorting time if the data has certain special characteristics. For example, Counting Sort can be applied to integer data that lies in a small range (see Section 9.32).

Searching Algorithms

- $O(n)$ Linear Search: Consider every element from index 0 to index $n - 1$
 - ควรหลีกเลี่ยง
- **$O(\log n)$** Binary Search: Use lower bound, upper bound, or binary search in C++ STL algorithm.
 - ควรใช้
 - ถ้า Input ไม่ได้ถูกเรียงมาก่อนให้ Sort ด้วย $O(n \log n)$ Algorithm ก่อนแล้วจึงทำการ Search ด้วย Binary search
- $O(1)$ Hashing

Performance Comparison

Data Structures

| Data Structure | Time Complexity | | | | | | | | Space Complexity |
|--------------------|-----------------|--------------|--------------|--------------|--------------|--------------|--------------|--------------|------------------|
| | Average | | | | Worst | | | | Worst |
| | Indexing | Search | Insertion | Deletion | Indexing | Search | Insertion | Deletion | |
| Basic Array | $O(1)$ | $O(n)$ | - | - | $O(1)$ | $O(n)$ | - | - | $O(n)$ |
| Dynamic Array | $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| Singly-Linked List | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| Doubly-Linked List | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ | $O(n)$ | $O(1)$ | $O(1)$ | $O(n)$ |
| Skip List | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n \log(n))$ |
| Hash Table | - | $O(1)$ | $O(1)$ | $O(1)$ | - | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| Binary Search Tree | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| Cartesian Tree | - | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | - | $O(n)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| B-Tree | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(n)$ |
| Red-Black Tree | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(n)$ |
| Splay Tree | - | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | - | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(n)$ |
| AVL Tree | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(\log(n))$ | $O(n)$ |

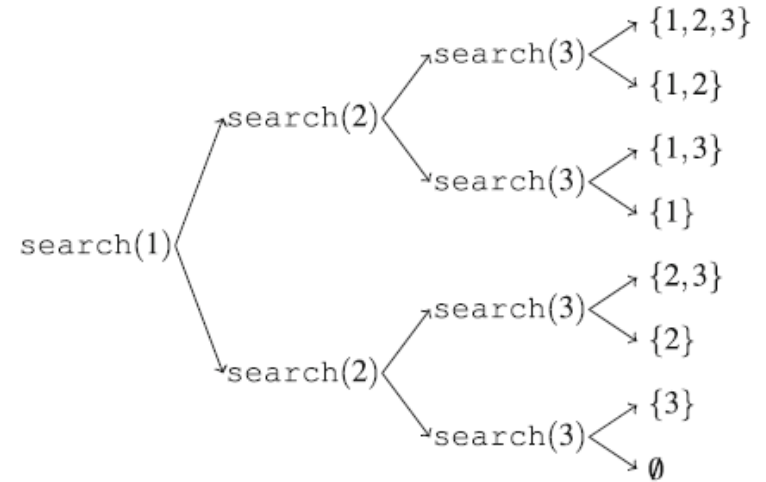


Recursive Algorithms



Generating Subsets

```
void search(int k) {  
    if (k == n+1) {  
        // process subset  
    } else {  
        // include k in the subset  
        subset.push_back(k);  
        search(k+1);  
        subset.pop_back();  
        // don't include k in the subset  
        search(k+1);  
    }  
}
```



Generating Permutations

```
vector<int> permutation;
```

```
bool chosen[n+1];
```

```
void search() {  
    if (permutation.size() == n) {  
        // process permutation  
    } else {  
        for (int i = 1; i <= n; i++) {  
            if (chosen[i]) continue;  
            chosen[i] = true;  
            permutation.push_back(i);  
            search();  
            chosen[i] = false;  
            permutation.pop_back();  
        }  
    }  
}
```

```
for (int i = 1; i <= n; i++) {  
    permutation.push_back(i);  
}  
do {  
    // process permutation  
} while (next_permutation(permutation.begin(),  
                           permutation.end()));
```

Backtracking

Fig. 2.4 Numbering of the arrays when counting the combinations on the 4×4 board

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |
| 0 | 1 | 2 | 3 |

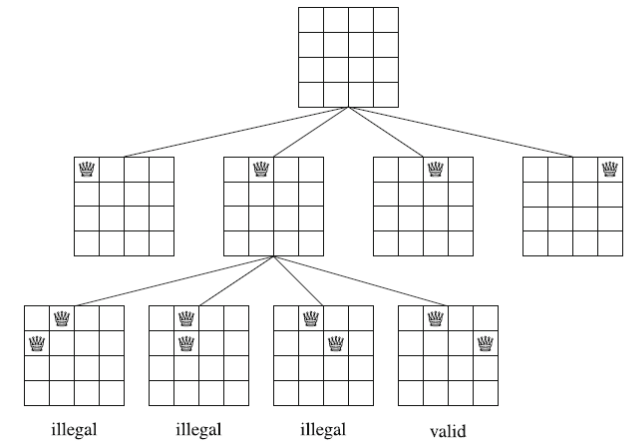
col

| | | | |
|---|---|---|---|
| 0 | 1 | 2 | 3 |
| 1 | 2 | 3 | 4 |
| 2 | 3 | 4 | 5 |
| 3 | 4 | 5 | 6 |

diag1

| | | | |
|---|---|---|---|
| 3 | 4 | 5 | 6 |
| 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 |
| 0 | 1 | 2 | 3 |

diag2



นับจำนวน Solution ของ N-Queens ใน Board

```
void search(int y) {  
    if (y == n) {  
        count++;  
        return;  
    }  
    for (int x = 0; x < n; x++) {  
        if (col[x] || diag1[x+y] || diag2[x-y+n-1]) continue;  
        col[x] = diag1[x+y] = diag2[x-y+n-1] = 1;  
        search(y+1);  
        col[x] = diag1[x+y] = diag2[x-y+n-1] = 0;  
    }  
}
```

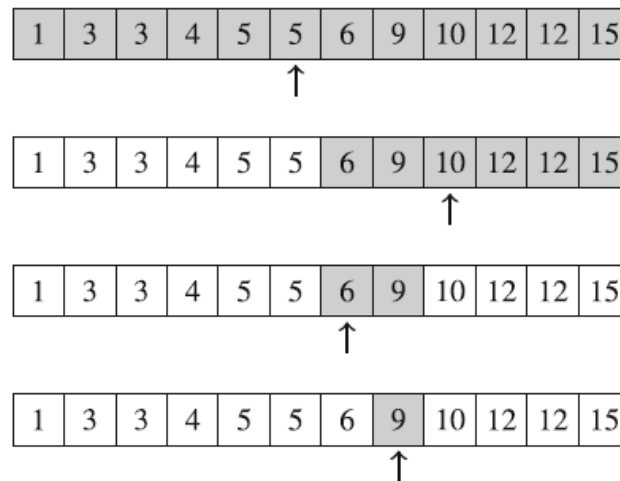
Binary Search



Binary Search

```
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) a = k+1;
    else b = k-1;
}
```

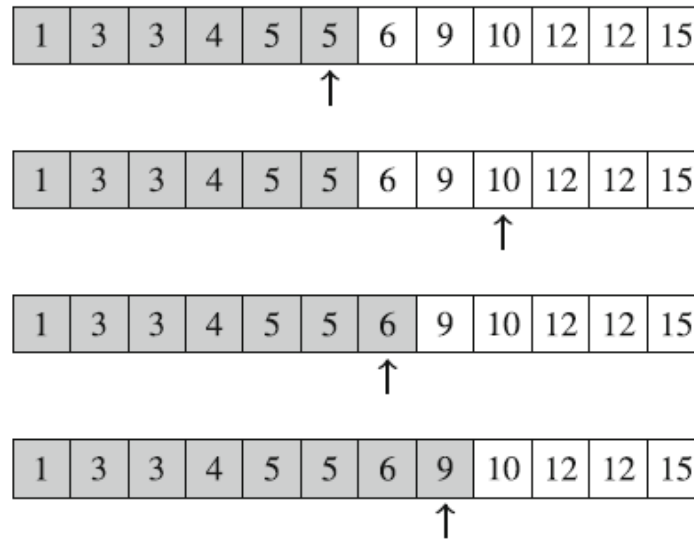
Fig.4.14 The traditional way to implement binary search. At each step we check the middle element of the active subarray and proceed to the left or right part



Binary Search-2

```
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
}
```

Fig.4.15 An alternative way to implement binary search. We scan the array from left to right jumping over elements



Finding Optimal Solutions

- สมมติว่าเรากำลังแก้ปัญหาที่มีฟังก์ชัน valid(x) ที่คืนค่า true ถ้า x คือ valid solution และถ้าไม่ใช่ Valid solution ให้คืน false.
- และเราจะรู้ว่าฟังก์ชัน valid(x) จะคืนค่า false เมื่อ $x < k$ และเมื่อ $x \geq k$
 - ** ในสถานการณ์แบบนี้เราจะใช้ Binary search หา Solution ได้อย่างมีประสิทธิภาพ
- แนวคิด หาค่าที่ใหญ่ที่สุดเพื่อให้ valid(x) มีค่าเป็น false และค่า $k = x+1$ จะเป็นค่าที่น้อยที่สุดที่จะทำให้ valid(x) มีค่าเป็น True

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

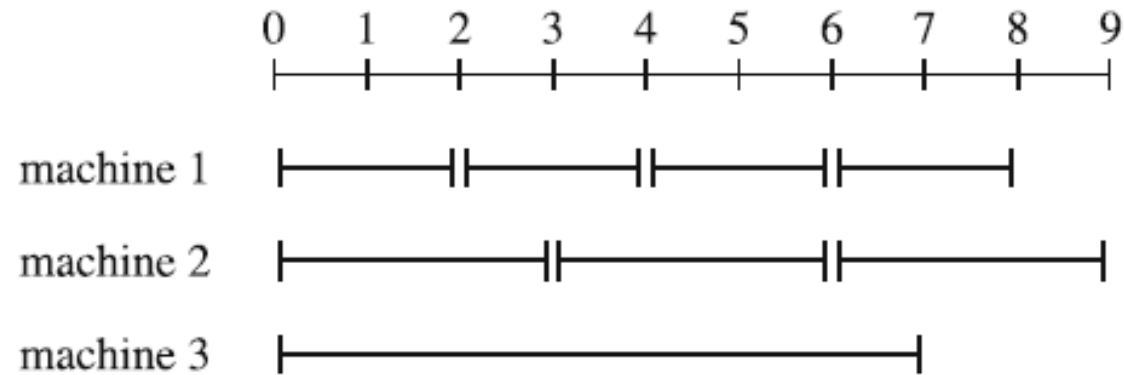

Finding Optimal Solutions

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

- Initial jump is z (Upper bound สำหรับคำตอบ – เป็นค่าที่ Valid(x) จะเป็น True แน่ ๆ)
- Algorithm จะเรียก valid(x) ทั้งหมด $O(\log z)$ ครั้ง
- ประสิทธิภาพรวมของโปรแกรมจะขึ้นอยู่กับ การคำนวณค่า valid(x)
- เช่นถ้า valid(x) ใช้เวลา n ประสิทธิภาพรวมคือ $O(n \log z)$

ตัวอย่าง

Fig. 4.16 An optimal processing schedule:
machine 1 processes four jobs, machine 2 processes three jobs, and machine 3 processes one job



Example Consider a problem where our task is to process k jobs using n machines. Each machine i is assigned an integer p_i : the time to process a single job. What is the minimum time to process all the jobs?

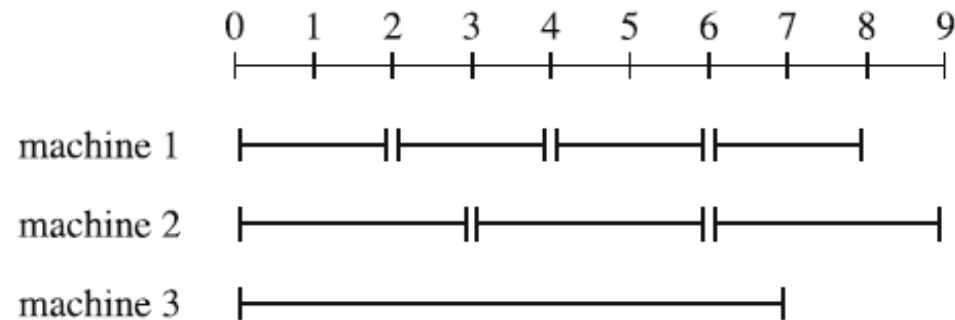
For example, suppose that $k = 8$, $n = 3$ and the processing times are $p_1 = 2$, $p_2 = 3$, and $p_3 = 7$.

คำตอบคือ 9 (ในรูปแบบ)

วิธีคิด

- ให้ $\text{valid}(x)$ เป็นฟังก์ชันที่ตรวจสอบว่างานทั้งหมดนั้นสามารถถูกทำได้ภายในเวลา x หรือไม่ ในกรณีด้านล่างจะพบว่า $x=9$ นั้น $\text{valid}(9) \Rightarrow \text{True}$ ในขณะที่ $x=8$ นั้น $\text{valid}(8) \Rightarrow \text{False}$

Fig.4.16 An optimal processing schedule:
machine 1 processes four jobs, machine 2 processes three jobs, and machine 3 processes one job



- ข้อสังเกต: การคำนวณค่าของฟังก์ชัน $\text{valid}(x)$ จะง่าย เพราะเราแค่ต้องคำนวณว่าเวลาที่ x แต่ละ Machine สามารถ Execute ได้กี่งาน (แล้วหาผลรวม)

Minimizing Sums



Minimizing Sums

- กำหนดตัวเลข n ตัวคือ a_1, \dots, a_n พิจารณาปัญหาการหาค่า x โดยที่

$$|a_1 - x| + |a_2 - x| + \dots + |a_n - x|.$$

มีค่าน้อยที่สุด

- ให้ตอบค่า Median ของ a_1, \dots, a_n (เป็น Optimal solution)

- กำหนดตัวเลข n ตัวคือ a_1, \dots, a_n พิจารณาปัญหาการหาค่า x โดยที่

$$(a_1 - x)^2 + (a_2 - x)^2 + \dots + (a_n - x)^2.$$

มีค่าน้อยที่สุด

- ให้ตอบค่าเฉลี่ยของ a_1, \dots, a_n (เป็น Optimal solution)

End.



Interesting Topics

- Range Queries...


| | | | |
|---|---|--|---|
| Optimization Common: <ul style="list-style-type: none">- Fake/dfs- DP/greedy/bf- Binary Search/TS- Branch & Bound- RMQ/LCA- Line sweep- AlgoX Minimization <ul style="list-style-type: none">- MCMF- Min cut / vertex- MST / Dijkstra- Chull / mec Maximization <ul style="list-style-type: none">- Max flow / MCMF- Max Independent Set- Kruskal Reverse- LIS/GCD | DP General <ul style="list-style-type: none">- State representation(s)- Diff sub-states calls?-- move to state- Cycles?-- Depth?-- Dijkstra / Bfs-- Dec(rement)-inc-dec Types <ul style="list-style-type: none">- Restricted / Range- Counting- Tree / Partitioning- Extending table Concerns <ul style="list-style-type: none">- Base case order- Search space?-- Constrained pars- Redundant pars States <ul style="list-style-type: none">- Canonical states?- Local Minima- Small substates cnt?- Large pars- Reduces fast? (e.g. /) Counting Problems <ul style="list-style-type: none">- DP- Combinations / Perms- Inclusion-exclusion- Graph Power | Data Structures <ul style="list-style-type: none">- Set/Heap /DisjointSets- BIT- Segmentation Tree- Treab, KDT- LCA/RMQ- Hashing- Interval Compression- Quad Tree Graph Algorithms <ul style="list-style-type: none">- MST: Kruskal / Prime- Dijkstra / Topological- Convex Hull / Floyd- Max Flow/Min Cut- Max Matching- Max Indep Set- Min path/vertex cover- Bellman / DConsts- Euler/Postman String Algorithms <ul style="list-style-type: none">- Trie- Permutation Cycles- LIS / LCS- Polynomial Hashing- KMP / Aho Corasick- Suffix tree/array | Mathematics <ul style="list-style-type: none">- GCD/LCM/Phi/Mob- NIM/Grundy/Chinese- Seive/Factorization- System of Linear Eqs- Determinant- Simplex/ Pick's Theo- Numerical Integration- Matrix Power- Closed Form- Pigeon Hole- Triangle inequality- Voronoi diagram Adhock Algorithms <ul style="list-style-type: none">- Greedy- Line Sweep- Sliding Window- Canonical Form- Grid Compression- Constructive algos- Test cases driven- Randomization- Time cut-off- Stress Test & Observe Decision Algorithms <ul style="list-style-type: none">- 2SAT- Difference constraints- Grundy- Bipartite? |
|---|---|--|---|

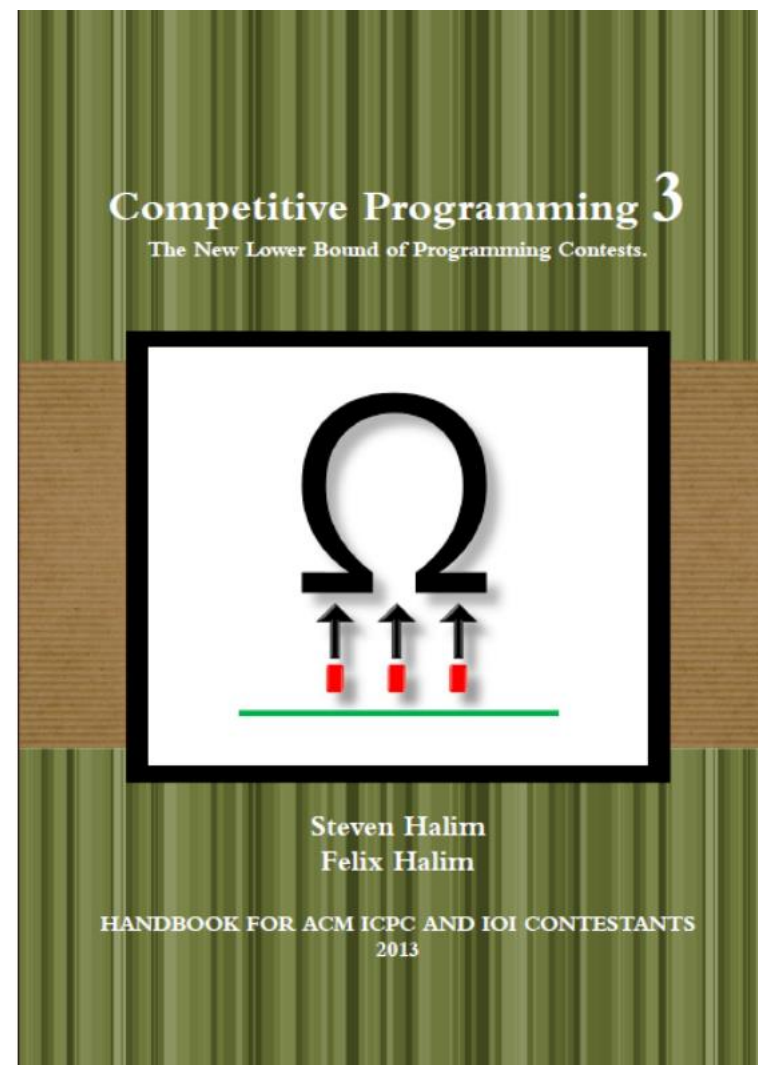
References

Antti Laaksonen

Guide to Competitive Programming

Learning and Improving Algorithms Through Contests

 Springer



Problem sets

- [CSES - CSES Problem Set - Tasks](#)