搜尋演算法與資料結構

二分搜尋、二元搜尋樹

nevikw39

點石學園

September 30, 2021

Section 1

Abstract

Introduction to Searching

Definition (Search)

Find elements in an array such that some condition be satisfied.

Introduction to Searching

Definition (Search)

Find elements in an array such that some condition be satisfied.

- If there is an element less than, equal to or greater than x??
  How many elements less than, equal to or greater than x??

Introduction to Searching

Definition (Search)

Find elements in an array such that some condition be satisfied.

- If there is an element less than, equal to or greater than x??
  How many elements less than, equal to or greater than x??
- The problem we are concerned about is the relation rather than order.

Pseudocode

procedure LINEAR SEARCH($*begin, *end, target$)
    $result \leftarrow \infty$
    for $itr \in [begin, end)$ **do**
        if $*itr \geq target \wedge *itr < result$ **then**
            $result \leftarrow *itr$
        end if
    end for
    return $result$
end procedure

Efficiency

If there are plenty of queries, then Linear Search may not fit in time limit.

Efficiency

If there are plenty of queries, then Linear Search may not fit in time limit.
Therefore, we could do some preprocess such as sorting and utilize the monotonicity to improve the efficiency.

Section 2

Binary Search

Efficient Way to Search in Ordered Sequence

Divide and Conquer

Divide Split array into two subarrays ($b = 2, O(1)$)

Efficient Way to Search in Ordered Sequence

Divide and Conquer

Divide Split array into two subarrays ($b = 2, O(1)$)

Conquer Search the subarray in which the target is ($a = 1$)

Efficient Way to Search in Ordered Sequence

Divide and Conquer

Divide	Split array into two subarrays ($b = 2, O(1)$)
Conquer	Search the subarray in which the target is ($a = 1$)
Combine	Nothing to do

"Although the basic idea of binary search is comparatively straightforward, the details can be surprisingly tricky."

— Knuth, D. E.

Pseudocode: Recursion

procedure BINARY SEARCH($*begin, *end, target$)
    if $begin = end$ **then**
        return $begin$
    end if
    $mid \leftarrow \frac{begin+end}{2}$
    if $target < *mid$ **then**
        return BINARY SEARCH(begin, mid, target)
    else if $target = *mid$ **then**
        return mid
    else
        return BINARY SEARCH(mid+1, end, target)
    end if
end procedure

Tail Call & Tail Recursion

Definition

It's called **Tail Call** that the return statement of a function is to call some function.
If the called one is the calling one itself, it's called **Tail Recursion**.

Tail Call & Tail Recursion

Definition

It's called **Tail Call** that the return statement of a function is to call some function.
If the called one is the calling one itself, it's called **Tail Recursion**.

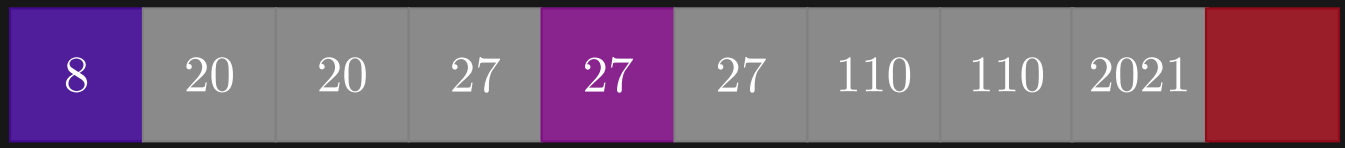
On top of the fact that recursion requires quite a few resources, we could use loop to attain better performance.

Pseudocode: Loop

procedure BINARY SEARCH($*begin, *end, target$)
    while $begin \neq end$ **do**
        $mid \leftarrow \frac{begin+end}{2}$
        if $target < *mid$ **then**
            $end \leftarrow mid$
        else if $target = *mid$ **then**
            return mid
        else
            $begin \leftarrow mid + 1$
        end if
    end while
    return $begin$
end procedure

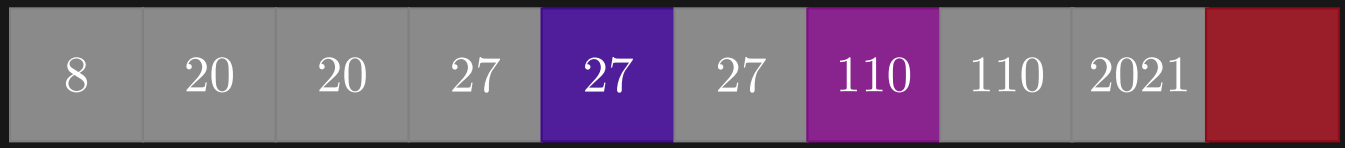
Example

$target = 110$



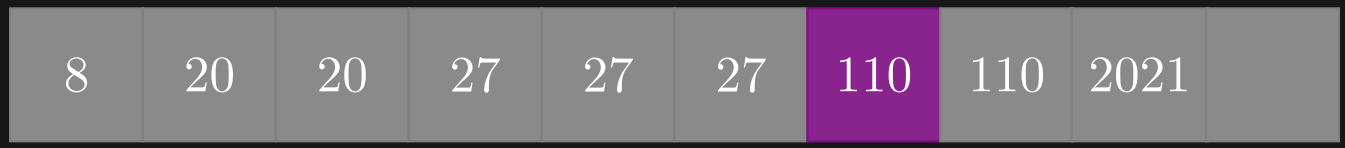
Example

$target = 110$



Example

$target = 110$

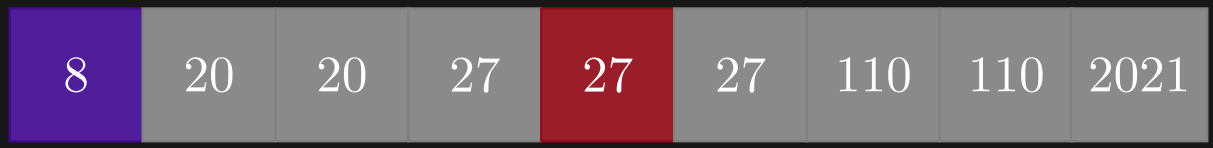


Pseudocode

procedure BINARY SEARCH($*begin, *end, target$)
    for $n \leftarrow end - begin, jump \leftarrow n/2; jump > 0; jump \leftarrow jump/2$ **do**
        while $begin + jump < end \wedge *(begin + jump) < target$ **do**
            $begin \leftarrow begin + jump$
        end while
    end for
    return $begin + 1$
end procedure

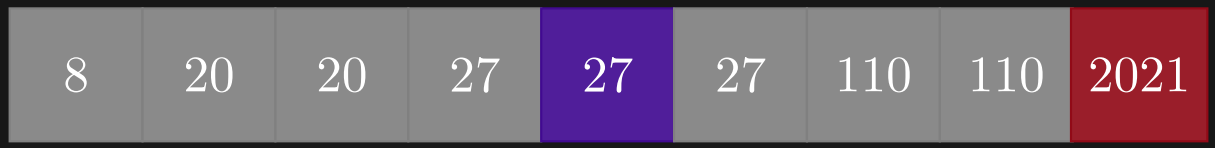
Example

$target = 110$



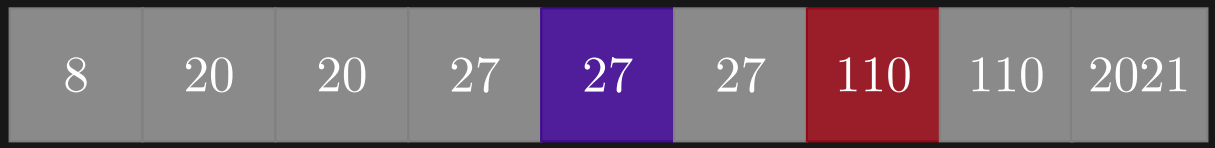
Example

$target = 110$



Example

$target = 110$

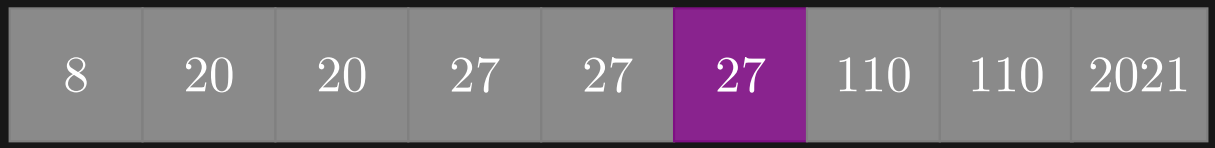


Example

$target = 110$

Example

$target = 110$



std::lower_bound()

Find the minimum element that is no less than (greater than or equal to) *target*

```
result = lower_bound(v.begin(), v.end(),
    target);
if (result != v.end())
     cout << "The minimum element that no less
         than (greater than or equal to) " <<
         target << " is " << *result << '\n';
else
     cout << "Each element is less than " <<
         target << '\n';
```

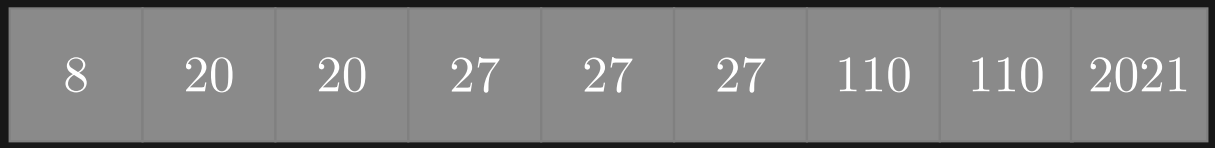
std::upper_bound()

Find the minimum element that is greater than *target*

```
result = upper_bound(v.begin(), v.end(),
    target);
if (result != v.end())
    cout << "The minimum element that greater
        than " << target << " is " << *result <<
        '\n';
else
    cout << "Each element is less than " <<
        target << '\n';
```

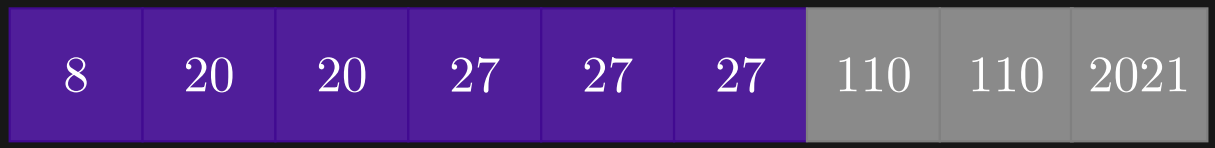
Examples

$target = 110$



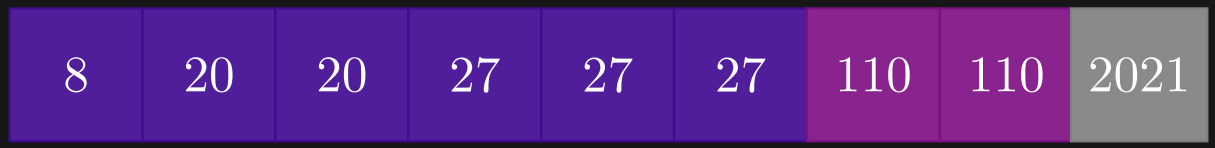
Examples

$target = 110$



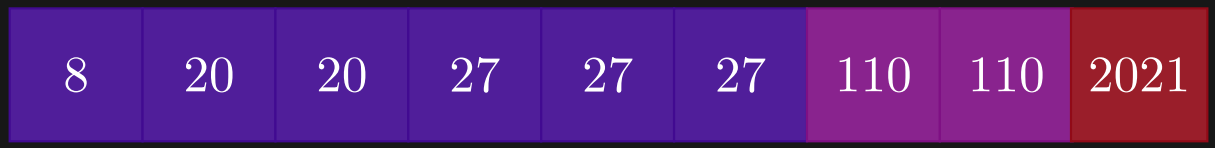
Examples

$target = 110$



Examples

$target = 110$



Examples

`std::lower_bound()`

If vector v is sorted:

Examples

`std::lower_bound()`

If vector v is sorted:

- Range of elements that less than x, $\{i \mid \forall i \in v \implies i < x\}$:

Examples

`std::lower_bound()`

If vector v is sorted:

- Range of elements that less than x, $\{i \mid \forall i \in v \implies i < x\}$: $[v.begin(), lower\_bound(v.begin(), v.end(), x))$

Examples

`std::lower_bound()`

If vector v is sorted:

- Range of elements that less than x, $\{i \mid \forall i \in v \implies i < x\}$: $[v.begin(), lower\_bound(v.begin(), v.end(), x))$
- Range of elements that no less than (greater than or equal to) x, $\{i \mid \forall i \in v \implies i \not< (\geq) x\}$:

Examples

`std::lower_bound()`

If vector v is sorted:

- Range of elements that less than x, $\{i \mid \forall i \in v \implies i < x\}$:
  $[v.begin(), lower\_bound(v.begin(), v.end(), x))$
- Range of elements that no less than (greater than or equal to) x,
  $\{i \mid \forall i \in v \implies i \not< (\geq) x\}$:
  $[lower\_bound(v.begin(), v.end(), x), v.end())$

Examples

`std::upper_bound()`

If vector v is sorted:

Examples

`std::upper_bound()`

If vector v is sorted:

- Range of elements that no greater than (less than or equal to) x, $\{i \mid \forall i \in v \implies i \not> (\leq) x\}$:

Examples

`std::upper_bound()`

If vector v is sorted:

- Range of elements that no greater than (less than or equal to) x, $\{i \mid \forall i \in v \implies i \not> (\leq) x\}$:
  $[v.begin(), upper\_bound(v.begin(), v.end(), x))$

Examples

`std::upper_bound()`

If vector v is sorted:

- Range of elements that no greater than (less than or equal to) x, $\{i \mid \forall i \in v \implies i \not> (\leq) x\}$: $[v.begin(), upper\_bound(v.begin(), v.end(), x))$
- Range of elements that greater than x, $\{i \mid \forall i \in v \implies i > x\}$:

Examples

`std::upper_bound()`

If vector v is sorted:

- Range of elements that no greater than (less than or equal to) x, $\{i \mid \forall i \in v \implies i \not> (\leq) x\}$:
  $[v.begin(), upper\_bound(v.begin(), v.end(), x))$
- Range of elements that greater than x, $\{i \mid \forall i \in v \implies i > x\}$:
  $[upper\_bound(v.begin(), v.end(), x), v.end())$

Examples

If vector v is sorted:

Examples

If vector v is sorted:

- Range of elements that equal to x, $\{i \mid \forall i \in v \implies i = x\}$:

Examples

If vector v is sorted:

- Range of elements that equal to x, $\{i \mid \forall i \in v \implies i = x\}$: $[lower\_bound(v.begin(), v.end(), x), upper\_bound(v.begin(), v.end(), x))$

Examples

If vector v is sorted:

- Range of elements that equal to x, $\{i \mid \forall i \in v \implies i = x\}$: $[lower\_bound(v.begin(), v.end(), x), upper\_bound(v.begin(), v.end(), x))$
- Range of elements that greater than x and less than y, $\{i \mid \forall i \in v \implies x < i < y\}$:

Examples

If vector v is sorted:

- Range of elements that equal to x, $\{i \mid \forall i \in v \implies i = x\}$:
  $[lower\_bound(v.begin(), v.end(), x), upper\_bound(v.begin(), v.end(), x))$
- Range of elements that greater than x and less than y,
  $\{i \mid \forall i \in v \implies x < i < y\}$:
  $[upper\_bound(v.begin(), v.end(), x), lower\_bound(v.begin(), v.end(), y))$

Examples

If vector v is sorted:

- Range of elements that equal to x, $\{i \mid \forall i \in v \implies i = x\}$: $[lower\_bound(v.begin(), v.end(), x), upper\_bound(v.begin(), v.end(), x))$
- Range of elements that greater than x and less than y, $\{i \mid \forall i \in v \implies x < i < y\}$: $[upper\_bound(v.begin(), v.end(), x), lower\_bound(v.begin(), v.end(), y))$
- Range of elements that greater than or equal to x and less than or equal to y, $\{i \mid \forall i \in v \implies x \leq i \leq y\}$:

Examples

If vector v is sorted:

- Range of elements that equal to x, $\{i \mid \forall i \in v \implies i = x\}$: $[lower\_bound(v.begin(), v.end(), x), upper\_bound(v.begin(), v.end(), x))$
- Range of elements that greater than x and less than y, $\{i \mid \forall i \in v \implies x < i < y\}$: $[upper\_bound(v.begin(), v.end(), x), lower\_bound(v.begin(), v.end(), y))$
- Range of elements that greater than or equal to x and less than or equal to y, $\{i \mid \forall i \in v \implies x \leq i \leq y\}$: $[lower\_bound(v.begin(), v.end(), x), upper\_bound(v.begin(), v.end(), y))$

Small Factor

UVa 11621

We could construct $C_{2,3}$ first. For every m inputted, the answer is $*lower\_bound(c.begin(), c.end(), m)$.

Exact Sum

UVa 11057

How could we determine whether there exist any exact value v in an ordered sequence s??

Exact Sum

UVa 11057

How could we determine whether there exist any exact value v in an ordered sequence s??

- The range of elements that equal to v is $[lower\_bound(s.begin(), s.end(), v), upper\_bound(s.begin(), s.end(), v))$

Exact Sum

UVa 11057

How could we determine whether there exist any exact value v in an ordered sequence s??

- The range of elements that equal to v is $[lower\_bound(s.begin(), s.end(), v), upper\_bound(s.begin(), s.end(), v))$
- If the length of range (i.e. $upper\_bound(s.begin(), s.end(), v) - lower\_bound(s.begin(), s.end(), v)$) is greater than 0, then it means v occurs at least one time.

Triangles

AtCoder Beginner Contest 143 p. D

Condition to Construct a Triangle

Let a, b, c be the three edges of the triangle, where $a \leq b \leq c$, then $b - a < c < a + b$

Triangles

AtCoder Beginner Contest 143 p. D

Condition to Construct a Triangle

Let a, b, c be the three edges of the triangle, where $a \leq b \leq c$, then $b - a < c < a + b$

Lest we may count duplicate ones, we could enumerate the shorter edges and search the range of the third edge, whose closed left end is upper bound of $b - a$ and opened right end is lower bound of $a + b$.

Triangles

AtCoder Beginner Contest 143 p. D

Condition to Construct a Triangle

Let a, b, c be the three edges of the triangle, where $a \leq b \leq c$, then $b - a < c < a + b$

Lest we may count duplicate ones, we could enumerate the shorter edges and search the range of the third edge, whose closed left end is upper bound of $b - a$ and opened right end is lower bound of $a + b$. Note that because c should be the longest edge, we mustn't search from $v.begin()$ but from the next iterator point to b.

Tsundoku

AtCoder Beginner Contest 172 p. C

Problem

Find the max $i+j$ such that $\sum_{k=0}^{i} A_k + \sum_{l=0}^{j} B_l \leq K$.

Tsundoku

AtCoder Beginner Contest 172 p. C

Problem

Find the max $i+j$ such that $\sum_{k=0}^{i} A_k + \sum_{l=0}^{j} B_l \leq K$.

Prefix sum

We could define $PSA_i \leftarrow \sum_{k=0}^{i} A_k$, $PSB_j \leftarrow \sum_{l=0}^{j} B_l$.

Tsundoku

AtCoder Beginner Contest 172 p. C

Problem

Find the max $i+j$ such that $\sum_{k=0}^{i} A_k + \sum_{l=0}^{j} B_l \leq K$.

Prefix sum

We could define $PSA_i \leftarrow \sum_{k=0}^{i} A_k$, $PSB_j \leftarrow \sum_{l=0}^{j} B_l$.
If $\forall i \in A, i \geq 0 \wedge \forall i \in B, i \geq 0$, then it's obviously that the prefix sum of A and B is increasing.

Tsundoku

AtCoder Beginner Contest 172 p. C

Problem

Find the max $i+j$ such that $\sum_{k=0}^{i} A_k + \sum_{l=0}^{j} B_l \leq K$.

Prefix sum

We could define $PSA_i \leftarrow \sum_{k=0}^{i} A_k$, $PSB_j \leftarrow \sum_{l=0}^{j} B_l$.
If $\forall i \in A, i \geq 0 \wedge \forall i \in B, i \geq 0$, then it's obviously that the prefix sum of A and B is increasing.

Thus, we could calculate the prefix sum of A and B first, and for all i in the prefix sum of A, we perform Binary Search to find the max j in the prefix sum of B such that $i+j \leq K$ i.e. $j \leq K-i$.

Long Sequence

AtCoder Beginner Contest 220 p. C

Problem

Consider B to be an infinity sequence where $B_i = A_{i \bmod N}$.
Find the min i such that $\sum_{k=0}^{i} B_k > X$.

Long Sequence

AtCoder Beginner Contest 220 p. C

Problem

Consider B to be an infinity sequence where $B_i = A_{i \bmod N}$.
Find the min i such that $\sum_{k=0}^{i} B_k > X$.

Since X may be enormous, we could calculate $\lfloor \frac{X}{\sum A} \rfloor$ and binary search $X \pmod{\sum A}$ in the prefix sum of A.

Coordinate Compression

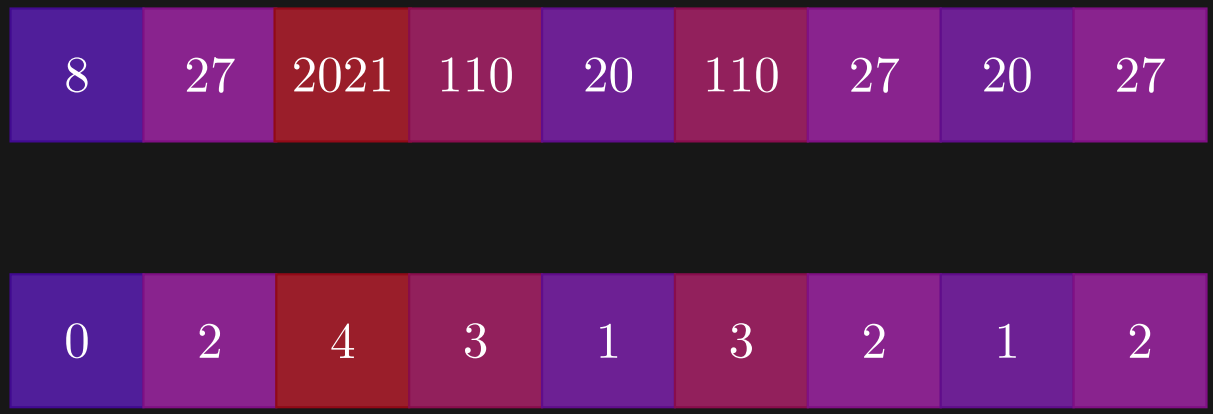
If the value range of input data is quite enormous and we care about not the exact values but their relative relation, we could apply a technique called **Coordinate Compression**, mapping the value in codomain into $[0, n)$.

Pseudocode

procedure COORDINATE COMPRESSION(v)
    $rk \leftarrow v, cc \leftarrow \{\}$
    SORT(rk.begin(), rk.end())
    UNIQUE(rk.begin(), rk.end()) ▷ Remove duplicate elements in rk
    for $i \in v$ **do**
        $cc.push_back($LOWER BOUND$(rk.begin(), rk.end(), i) - rk.begin())$
    end for
    return cc
end procedure

Examples

Examples



std::unique()

Move duplicate elements in an ordered sequence to end and return new valid end

```
sort(rank.begin(), rank.end());
auto itr = unique(rank.begin(), rank.end()); //
    move duplicate elements to end and return
    new valid end
rank.erase(itr, rank.end());                 //
    remove them
```

Reorder Cards

AtCoder Beginner Contest 213 p. C

"It can be proved that these positions are uniquely determined without depending on the order in which the operations are done."

Reorder Cards

AtCoder Beginner Contest 213 p. C

> *"It can be proved that these positions are uniquely determined without depending on the order in which the operations are done."*

Hence, rows and columns could be reordered separately.

Reorder Cards

AtCoder Beginner Contest 213 p. C

> *"It can be proved that these positions are uniquely determined without depending on the order in which the operations are done."*

Hence, rows and columns could be reordered separately.
It's obvious that any operation cannot affect the relative position of cards.

Reorder Cards

AtCoder Beginner Contest 213 p. C

> *"It can be proved that these positions are uniquely determined without depending on the order in which the operations are done."*

Hence, rows and columns could be reordered separately.
It's obvious that any operation cannot affect the relative position of cards.
At the end of process, the coordinate of cards must be in $[1, n]$.

Reorder Cards

AtCoder Beginner Contest 213 p. C

> *"It can be proved that these positions are uniquely determined without depending on the order in which the operations are done."*

Hence, rows and columns could be reordered separately.
It's obvious that any operation cannot affect the relative position of cards.
At the end of process, the coordinate of cards must be in $[1, n]$.
Voilà! All we have to do is to perform coordinate compression in both x and y axis.

It's not hard for us to refer to array as a sort of function.

It's not hard for us to refer to array as a sort of function.
Then if we could perform Binary Search in monotonic array,
couldn't we also perform Binary Search in monotonic function??

Buy an Integer

AtCoder Beginner Contest 146 p. C

Apparently, $\forall i \geq 0, d(i) \leq d(i+1)$.

Buy an Integer

AtCoder Beginner Contest 146 p. C

Apparently, $\forall i \geq 0, d(i) \leq d(i+1)$.
Therefore, it could be easily proven that
$\forall i \geq 0, price(i) = A \times i + B \times d(i) < A \times (i+1) + B \times d(i+1) = price(i+1)$.

Buy an Integer

AtCoder Beginner Contest 146 p. C

Apparently, $\forall i \geq 0, d(i) \leq d(i+1)$.
Therefore, it could be easily proven that
$\forall i \geq 0, price(i) = A \times i + B \times d(i) < A \times (i+1) + B \times d(i+1) = price(i+1)$.

Logs

AtCoder Beginner Contest 174 p. E

The minimum of shortest possible length x is 1, and the maximum is the maximum length of logs.

Logs

AtCoder Beginner Contest 174 p. E

The minimum of shortest possible length x is 1, and the maximum is the maximum length of logs.
Moreover, whether we could cut all logs to less than x in at most k times is monotonic.

Section 3

Binary Search Tree

Dynamic Problems

Some problems require us not only to answer some queries but also to modify the original data alternately.
If we sort the data each time we do modification, the time complexity would decline to $O(Q \times N \log N)$.
As a result, we need some data structure to avoid that.

Self-Balancing Binary Search Tree

Tree is a kind of abstract data type on which we could easily insert or traverse.

Self-Balancing Binary Search Tree

Tree is a kind of abstract data type on which we could easily insert or traverse.

Binary Tree is a kind of rooted tree that each node could have at most two child nodes.

Self-Balancing Binary Search Tree

Tree is a kind of abstract data type on which we could easily insert or traverse.

Binary Tree is a kind of rooted tree that each node could have at most two child nodes.

Binary Search Tree is a kind of binary tree where the value of internal node always greater than all values in its left subtree and less than all values in its right subtree.

Self-Balancing Binary Search Tree

Tree is a kind of abstract data type on which we could easily insert or traverse.

Binary Tree is a kind of rooted tree that each node could have at most two child nodes.

Binary Search Tree is a kind of binary tree where the value of internal node always greater than all values in its left subtree and less than all values in its right subtree.

Self-Balancing Binary Search Tree is a BST which could balance its height by change root and/or rotating so that the max hegiht won't exceed $O(\log N)$.

Set in Math & C++

In math:

- The ordering of a set doesn't matter; that is, $\{8, 27\} = \{27, 8\}$.

In C++:

- Because the order of set is relevant, we could choose a unique order for representation. In the set of C++, the elements are sorted by their relation.

Set in Math & C++

In math:

- The ordering of a set doesn't matter; that is, $\{8, 27\} = \{27, 8\}$.
- Each element in set should be unique.

In C++:

- Because the order of set is relevant, we could choose a unique order for representation. In the set of C++, the elements are sorted by their relation.
- Each element in set should be unique.

Usage

Like `std::vector<>`, `std::set<>` is a kind of template class container in C++ Standard Template Library.
The most difference is that we couldn't use `[]` operator to get the access of elements.
In addition, We could still use iterators to traverse all elements, but the iterators are bidirectional, which means they doesn't support `+ n`, `- n` operations.

Usage (cont.)

`s.insert()` Add a new element.

`s.erase()` Erase an element by its value or iterator.

`s.find()` If the element exists in the set then return its iterator, else return `s.end()`.

`s.lower_bound()` The specific version of `std::lower_bound()`.

`s.upper_bound()` The specific version of `std::upper_bound()`.

Problems to Practice

- AtCoder Beginner Contest 164 p. C - gacha
- AtCoder Beginner Contest 073 p. C - Write and Erase

Cutting Woods

AtCoder Beginner Contest 217 p. D

How could we know the length of the segment a mark is in??

Cutting Woods

AtCoder Beginner Contest 217 p. D

How could we know the length of the segment a mark is in??

- We could find the greatest cut mark on its left and the least one on its right.

Cutting Woods

AtCoder Beginner Contest 217 p. D

How could we know the length of the segment a mark is in??

- We could find the greatest cut mark on its left and the least one on its right.
- Be sure to insert 0 and L first so that we could avoid boundary conditions.

Map, dictionary or associative array

Sometimes, we need to store extra values related to the nodes in a binary search tree.
`std::map<,>` is a C++ STL container that allow us to store data in that way.
The data used to build binary search tree are called **keys**, and the extra data related to some keys are called **values**.
In fact, a `std::set<>` could be deemed `std::map<,>` without **values** the extra data.

Usage

Just like `std::set<>`, the type of **key** or **value** could be any type if it supports `<`, `==` operations.
We could use `[]` operator to access the **value** of a **key**.
Iterators also work with `std::map<,>`, and the key is `itr->first` while the value is `itr->second`.

Usage (cont.)

`m[]` Return the reference to the **value** of a **key**.

`m.erase()` Erase an element by its **key** or iterator.

`m.find()` If the key exists in the map then return its iterator, else return `s.end()`.

`m.lower_bound()` The specific version of `std::lower_bound()`.

`m.upper_bound()` The specific version of `std::upper_bound()`.

Problems to Practice

- UVa 10420 - List of Conquests
- UVa 10226 - Hardwood species
- UVa 12250 - Language Detection
- AtCoder Beginner Contest 155 p. C - Poll
- AtCoder Beginner Contest 137 p. C - Green Bin

std::multiset<>, std::multimap<,>

The containers that accept duplicate keys.
Note that we cannot use `[]` operator in `std::multimap<,>`.

Section 4

Hash Functions & Tables

Hash Functions

A Hash Function is a function that map some data into value in fixed *range*.
The *range* of inputted data may be large, even infinite, whereas the *range* of Hash Function is fixed.
As a consequence, when it come to searching, instead of searching the whole *range* of inputted data, we could search in the *range* of Hash Function.

Hash Functions (cont)

Note that on top of the fact that Hash Function may not hold monotonicity, we could only search whether the *key* exist.
In addition, since the *range* of Hash Function is finite, the *hash values* of different *key* may be the same, which is called Collision.

Hash Container

C++ STL provides **unordered_set** and **unordered_map** which is the hash version of **set** and **map** respectively.
The usage is mostly the same, the main difference is that unordered container cannot perform Binary Search.