二分搜尋

日月卦長

排序好的陣列：兩個重要「邊界」

$$x = 7$$

$< x$			$= x$			$> x$			
2	3	3	7	7	7	9	9	12	12

排序好的陣列：四個重要「index」

$$x = 7$$

>= x 的最小值

> x 的最小值

0	1	2	3	4	5	6	7	8	9
2	3	3	7	7	7	9	9	12	12

< x 的最大值

<= x 的最大值

Check 函數: True 和 False 的交界

$$x = 7$$

$check(idx)$: $return\ arr[idx] <= x$

$>x$ 的最小值

	0	1	2	3	4	5	6	7	8	9
arr	2	3	3	7	7	7	9	9	12	12
$check(idx)$	True	True	True	True	True	True	False	False	False	False

$<= x$ 的最大值

Check 函數: True 和 False 的交界

$$x = 7$$

$check(idx)\text{: } return\ \ arr[idx] < x$

$>= x$ 的最小值

	0	1	2	3	4	5	6	7	8	9
arr	2	3	3	7	7	7	9	9	12	12
$check(idx)$	True	True	True	False	False	False	False	False	False	False

$< x$ 的最大值

想法1: 暴力法

#include <functional>

```cpp
#include <bits/stdc++.h>
using namespace std;

pair<int, int> search(int L, int R, function<bool(int)> check) {
  for (int i = L; i <= R; ++i)
    if (check(i) == false)
      return {i - 1, i};
  return {R, R + 1};
}

int main() {
  int arr[] = {2, 3, 3, 7, 7, 7, 9, 9, 12, 12};
  auto check = [&](int idx) { return arr[idx] < 7; };
  auto [a, b] = search(0, 9, check);
  cout << a << ' ' << b << '\n';
  return 0;
}
```

想法1: 暴力法

```cpp
#include <bits/stdc++.h>
using namespace std;

template <class Ty, class FuncTy>
pair<Ty, Ty> search(Ty L, Ty R, FuncTy check) {
  for (Ty i = L; i <= R; ++i)
    if (check(i) == false)
      return {i - 1, i};
  return {R, R + 1};
}

int main() {
  int arr[] = {2, 3, 3, 7, 7, 7, 9, 9, 12, 12};
  auto check = [&](int idx) { return arr[idx] < 7; };
  auto [a, b] = search(0, 9, check);
  cout << a << ' ' << b << '\n';
  return 0;
}
```

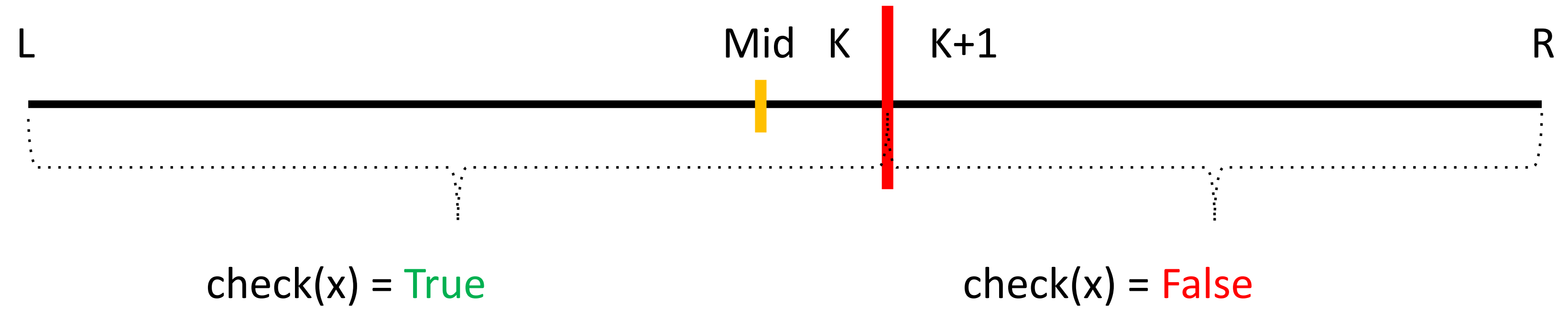
好好利用性質

-

- 設$Mid = L + (R - L)/2$
- 只會有兩種可能性
  - $check(Mid) = True$
  - $check(Mid) = False$

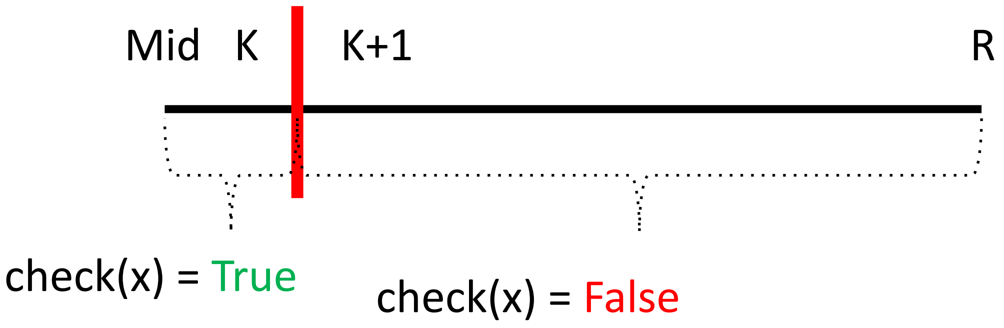
$$check(Mid) = True$$

- $Mid \leq K$



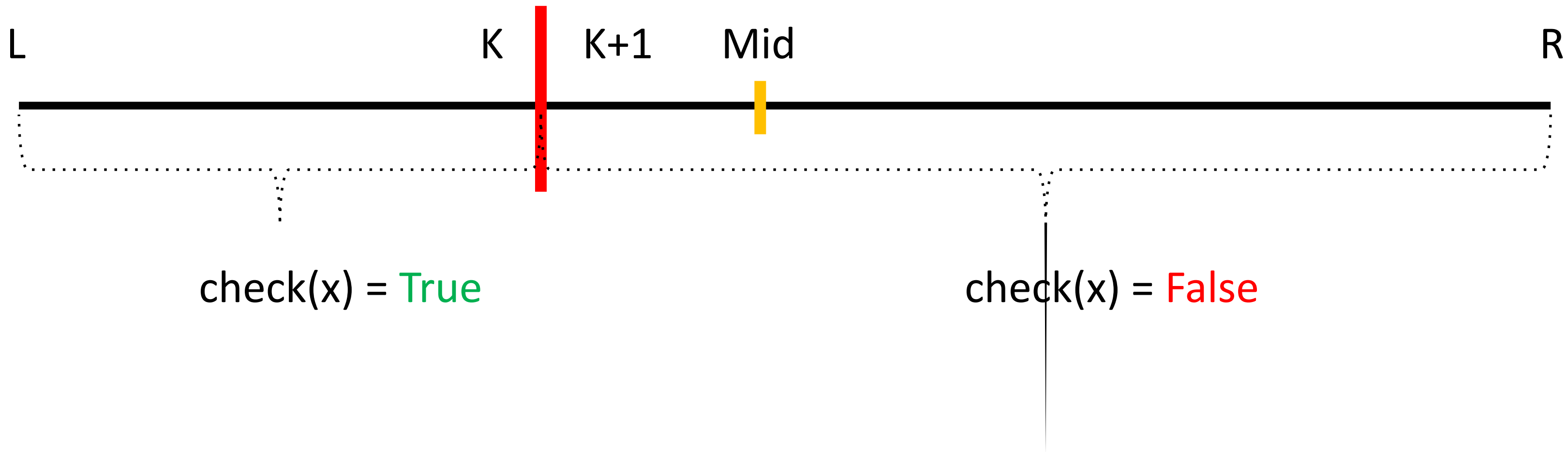
$$check(Mid) = True$$

- $Mid \leq K$
- $search(L, R, check) = search(Mid, R, check)$



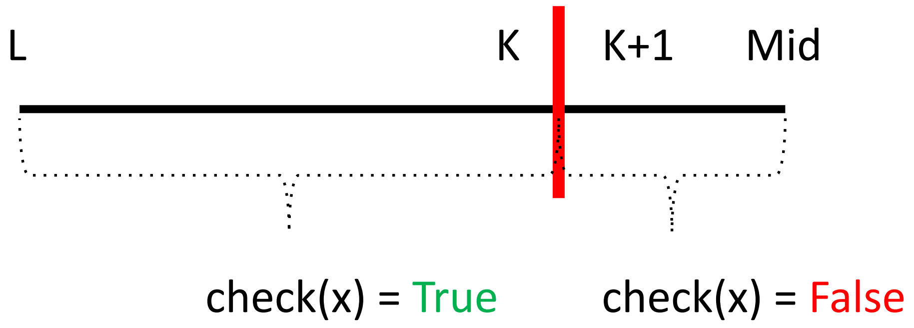
$$check(Mid) = False$$

- $Mid \geq K + 1$



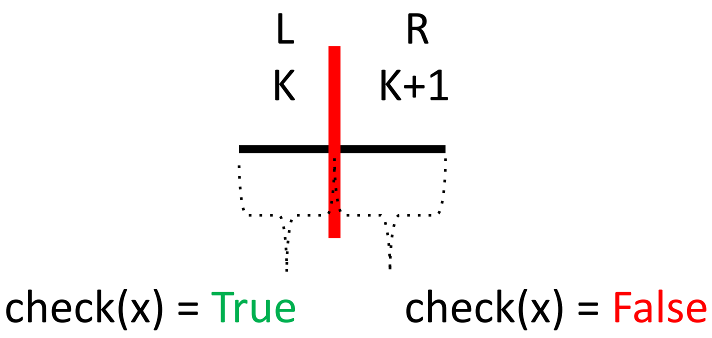
$check(Mid) = False$

- $Mid \geq K + 1$
- $search(L, R, check) = search(L, Mid, check)$



終止條件

- $L + 1 == R$
- 此時
  - $L = K$
  - $R = K + 1$



寫成遞迴函數

```cpp
template <class Ty, class FuncTy>
pair<Ty, Ty> search(Ty L, Ty R, FuncTy check) {
  if (L + 1 == R)
    return {L, R};
  Ty Mid = L + (R - L) / 2;
  if (check(Mid))
    return search(Mid, R, check);
  return search(L, Mid, check);
}
```

怎麼算中間值?

$$\boldsymbol{Mid = L + (R - L)/2}$$

- $L = 1$
- $R = 2$
- $L + \left\lfloor \frac{R-L}{2} \right\rfloor =$

$$1 + \left\lfloor \frac{1}{2} \right\rfloor = 1$$

$$\boldsymbol{Mid = (L + R)/2}$$

- $L = 1$
- $R = 2$
- $\left\lfloor \frac{L+R}{2} \right\rfloor =$

$$\left\lfloor \frac{3}{2} \right\rfloor = 1$$

怎麼算中間值?

$$\boldsymbol{Mid = L + (R - L)/2}$$

- $L = -2$
- $R = -1$
- $L + \left\lfloor \frac{R-L}{2} \right\rfloor =$

$$-2 + \left\lfloor \frac{1}{2} \right\rfloor = -2$$

不能整除結果會靠近 L

$$\boldsymbol{Mid = (L + R)/2}$$

- $L = -2$
- $R = -1$
- $\left\lfloor \frac{L+R}{2} \right\rfloor =$

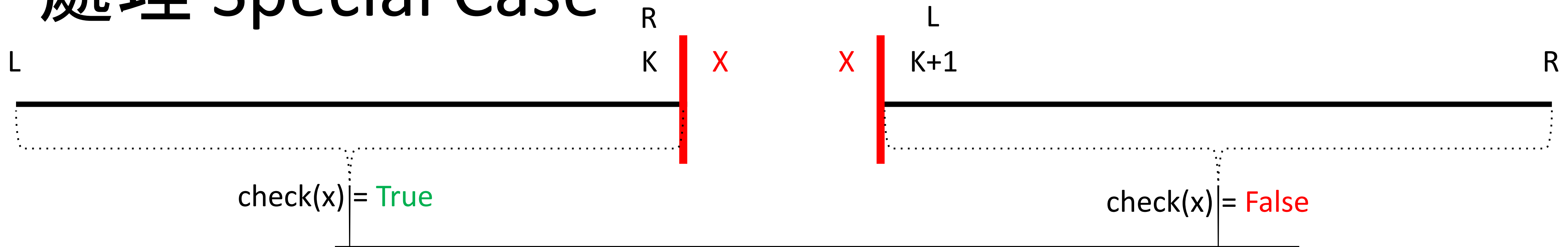
$$\left\lfloor \frac{-3}{2} \right\rfloor = -1$$

不能整除結果會靠近 R

遞迴常數太大了

```cpp
template <class Ty, class FuncTy>
pair<Ty, Ty> binarySearch(Ty L, Ty R, FuncTy check) {
  while (L + 1 < R) {
    Ty Mid = L + (R - L) / 2;
    if (check(Mid)) L = Mid;
    else R = Mid;
  }
  return {L, R};
}
```

處理 Special Case




```cpp
template <class Ty, class FuncTy>
pair<Ty, Ty> binarySearch(Ty L, Ty R, FuncTy check) {
  if (check(R) == true) return {R, R + 1};
  if (check(L) == false) return {L - 1, L};
  while (L + 1 < R) {
    Ty Mid = L + (R - L) / 2;
    if (check(Mid)) L = Mid;
    else R = Mid;
  }
  return {L, R};
}
```

程式碼中，你有看到任何「陣列」嗎?

$O(\log n)$

經典題變型

- https://leetcode.com/problems/search-in-rotated-sorted-array/
- 輸入一個排序好但rotate過的陣列，以及一個數字target 問你target在陣列的Index，不存在就輸出-1
- Example:
- Input = [4,5,6,7,0,1,2], target = 0
  - ans = 4
- Input = [4,5,6,7,0,1,2], target = 3
  - ans = -1

一次搜不出來就搜兩次

realR

target = 0

nums = [4, 5, 6, 7, 0, 1, 2]

check(x) = True　　check(x) = False

L=0　　R=nums.size()-1

```
auto check_1 = [&](int x) {
  return nums[x] >= nums[0];
};
```

第一次搜

nums[x] > nums[x+1]的分界線

一次搜不出來就搜兩次

realR=3

nums = [4, 5, 6, 7, 0, 1, 2]

Id	4	5	6	0	1	2	3
nums	0	1	2	4	5	6	7

check(x) = True

check(x) = False

L=0

R=nums.size()-1

```
auto getIdx = [&](int x) {
  return (x + realR + 1) % nums.size();
};
auto check_2 = [&](int x) {
  return nums[getIdx(x)] <= target;
};
```

第二次正常搜

用二分搜尋

```cpp
int search(vector<int> &nums, int target) {
  auto check_1 = [&](int x) {
    return nums[x] >= nums[0];
  };
  int realR = binarySearch(0, (int)nums.size() - 1, check_1).first;
  auto getIdx = [&](int x) {
    return (x + realR + 1) % nums.size();
  };
  auto check_2 = [&](int x) {
    return nums[getIdx(x)] <= target;
  };
  auto [L, R] = binarySearch(0, (int)nums.size() - 1, check_2);
  if (L == -1 || nums[getIdx(L)] != target)
    return -1;
  return getIdx(L);
}
```

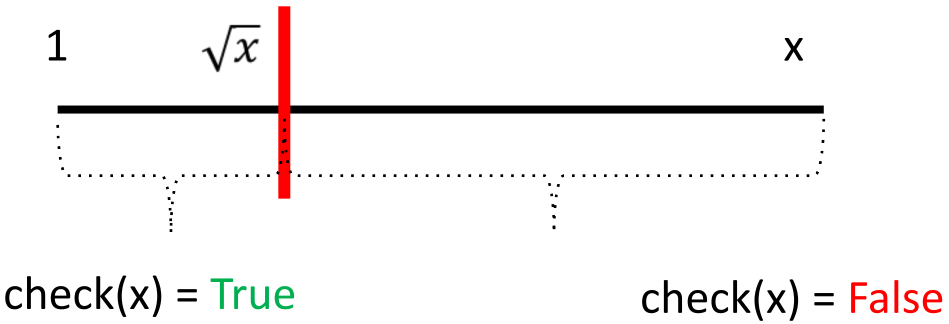
unsigned long long 開根號

- $\left\lfloor \sqrt{18014398241046527} \right\rfloor = ?$

```cpp
int main() {
  long long Base = 18014398241046527LL;
  double X = sqrt(Base);
  long long XL = X;
  if (XL * XL > Base)
    cout << "Error\n";
  return 0;
}
```

解法一：用二分搜尋

```
unsigned long long sqrt_ull(unsigned long long n) {
  auto [L, R] = binarySearch(0ull, min(n, 1ull * UINT_MAX),
                             [&](auto x) { return x * x <= n; });
  return L;
}
```

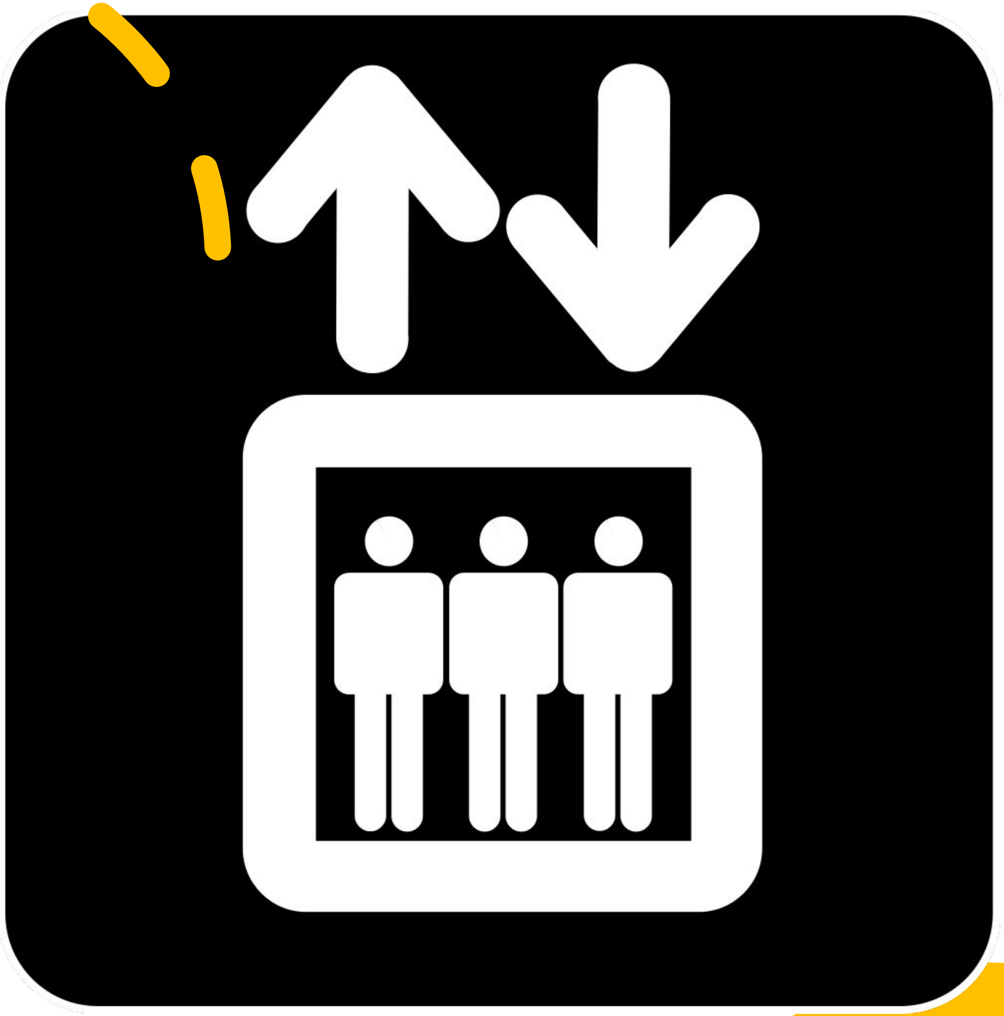


解法二：用 long double

```
int main() {
  long long Base = 18014398241046527LL;
  double X = sqrt(Base);
  long long XL = X;
  if (XL * XL > Base)
    cout << "Error\n";
  cout << XL << ' ' << (long long)sqrtl(Base) << endl;
  return 0;
}
```

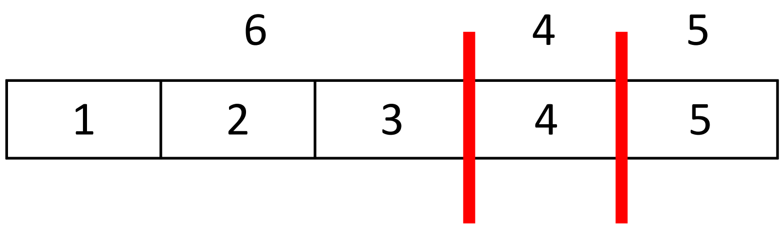
電梯向上

- 有 N 個人，每個人有各自的體重 $w_1 \sim w_n$ (保證都是正整數)
- 他們按邊號排隊從一樓搭電梯到頂樓
- 不可以插隊
- 問這台電梯的限重最少是多少才能在 K 次內將所有人送到頂樓



範例：$N = 5, K = 3, w = [1,2,3,4,5]$

- 限重最少要是 6
- 才能在 3 趟內將所有人送到頂樓



子問題

- $N = 5, w = [1,2,3,4,5]$
- 已知限重是 10
- 最少需要幾趟才能把人都送上去?

1	2	3	4	5

子問題

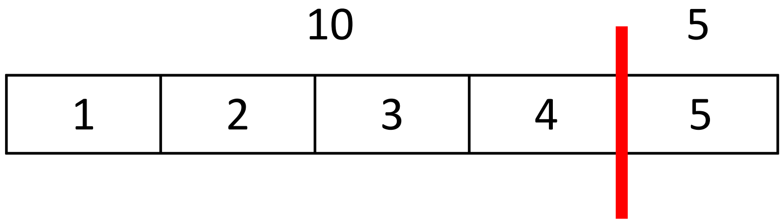
- $N = 5, w = [1,2,3,4,5]$
- 已知限重是 10
- 最少需要幾趟才能把人都送上去?
- 貪心法: 2 趟

		10		5
1	2	3	4	5

貪心法

```cpp
int greedy(const vector<int> &w, long long limit) {
  int times = 0;
  long long current = 0;
  for (int x : w) {
    current += x;
    if (current > limit) {
      current = x;
      ++times;
    }
  }
  if (current) ++times;
  return times;
}
```

不要忘記



觀察輸出結果

```
cout << greedy({1,2,3,4,5}, 5) << endl;
cout << greedy({1,2,3,4,5}, 6) << endl;
cout << greedy({1,2,3,4,5}, 7) << endl;
cout << greedy({1,2,3,4,5}, 8) << endl;
cout << greedy({1,2,3,4,5}, 9) << endl;
cout << greedy({1,2,3,4,5}, 10) << endl;
```

限重	最少需要的趟數
5	4
6	3
7	3
8	3
9	2
10	2

遞減

可以用二分搜尋

```
long long solve(const vector<int> &w, int k) {
  long long L = *max_element(w.begin(), w.end());
  long long R = 0;
  for (int x : w) R += x;
  return binarySearch(L, R, [&](long long mid) { return greedy(w, mid) > k; })
      .second;
}
```

常見名詞

- 單調性 (monotone)
  - 單調遞增
  - 單調遞減

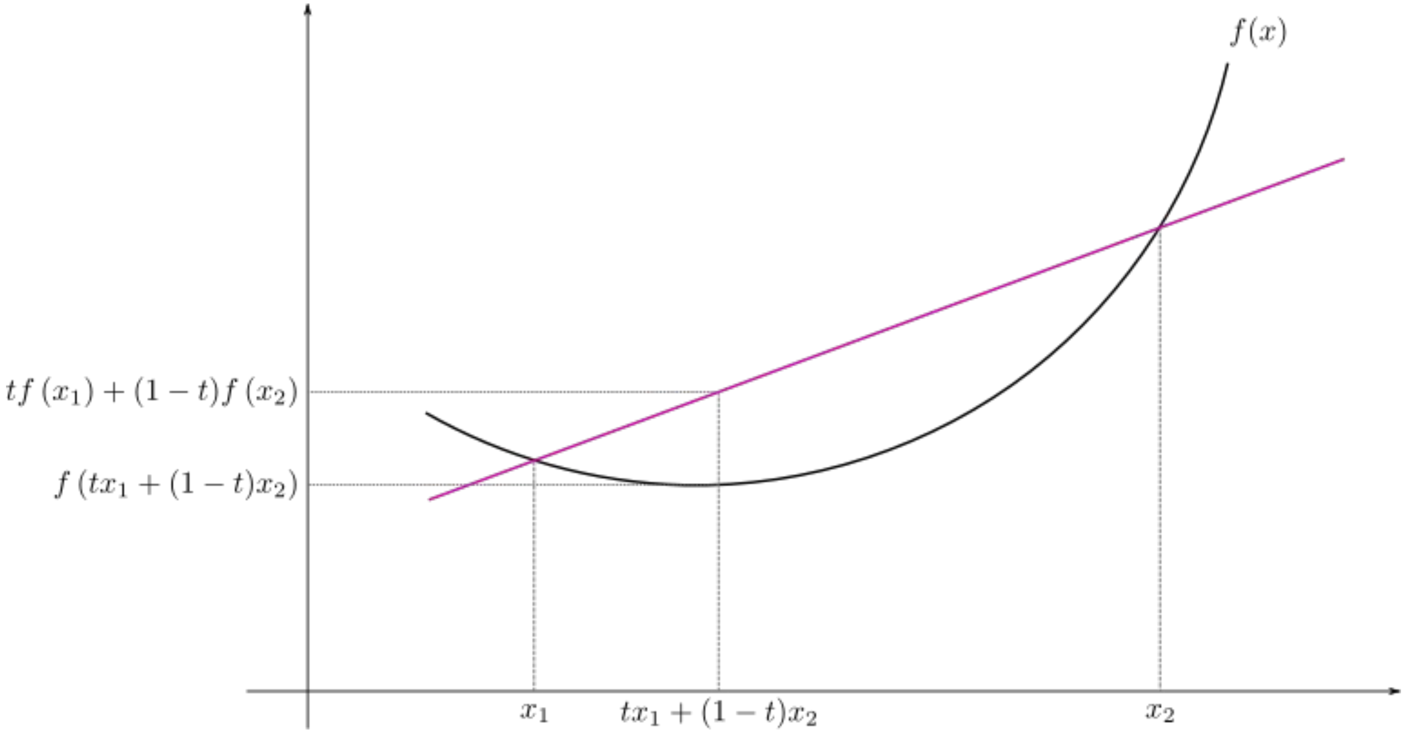
- 非嚴格遞減 (non-increasing) □ 由大到小排
- 非嚴格遞增(non-decreasing) □ 由小到大排

凸函數與三分搜

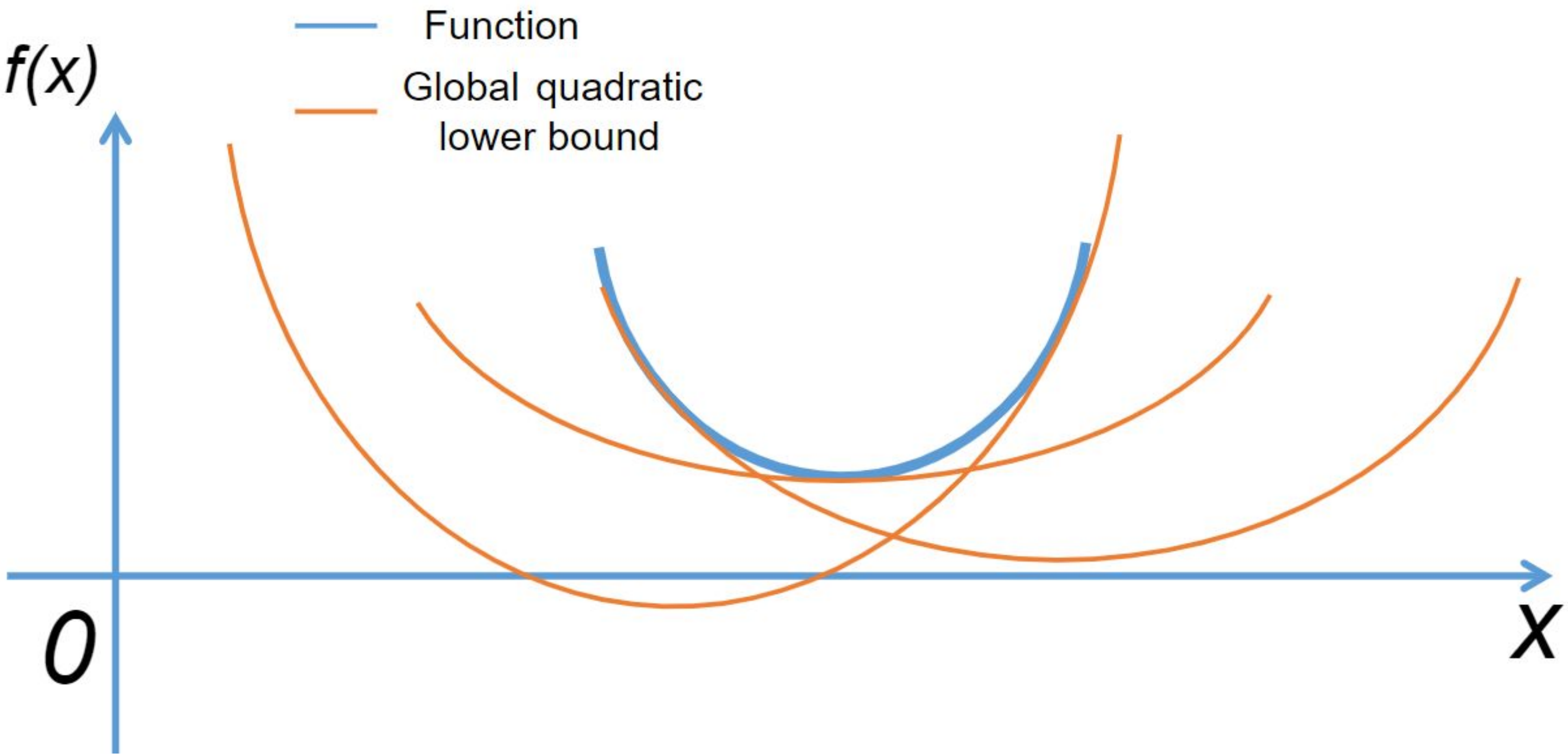
ternary search

凸函數

- 最簡單的說法
- 選函數上任意兩點連線都位於函數圖形的上方

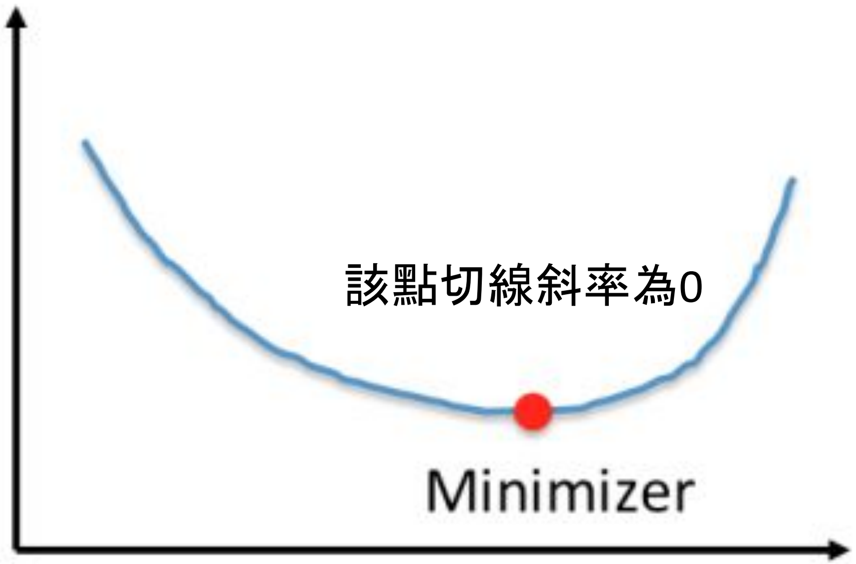


凸函數性質 – 交集也是凸函數

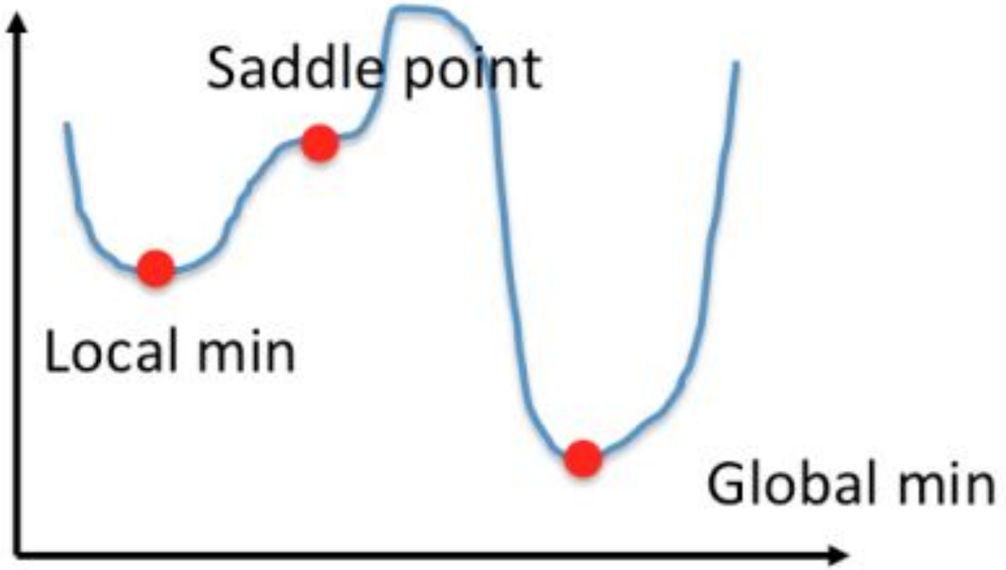


凸函數性質 – 只存在全域最小值

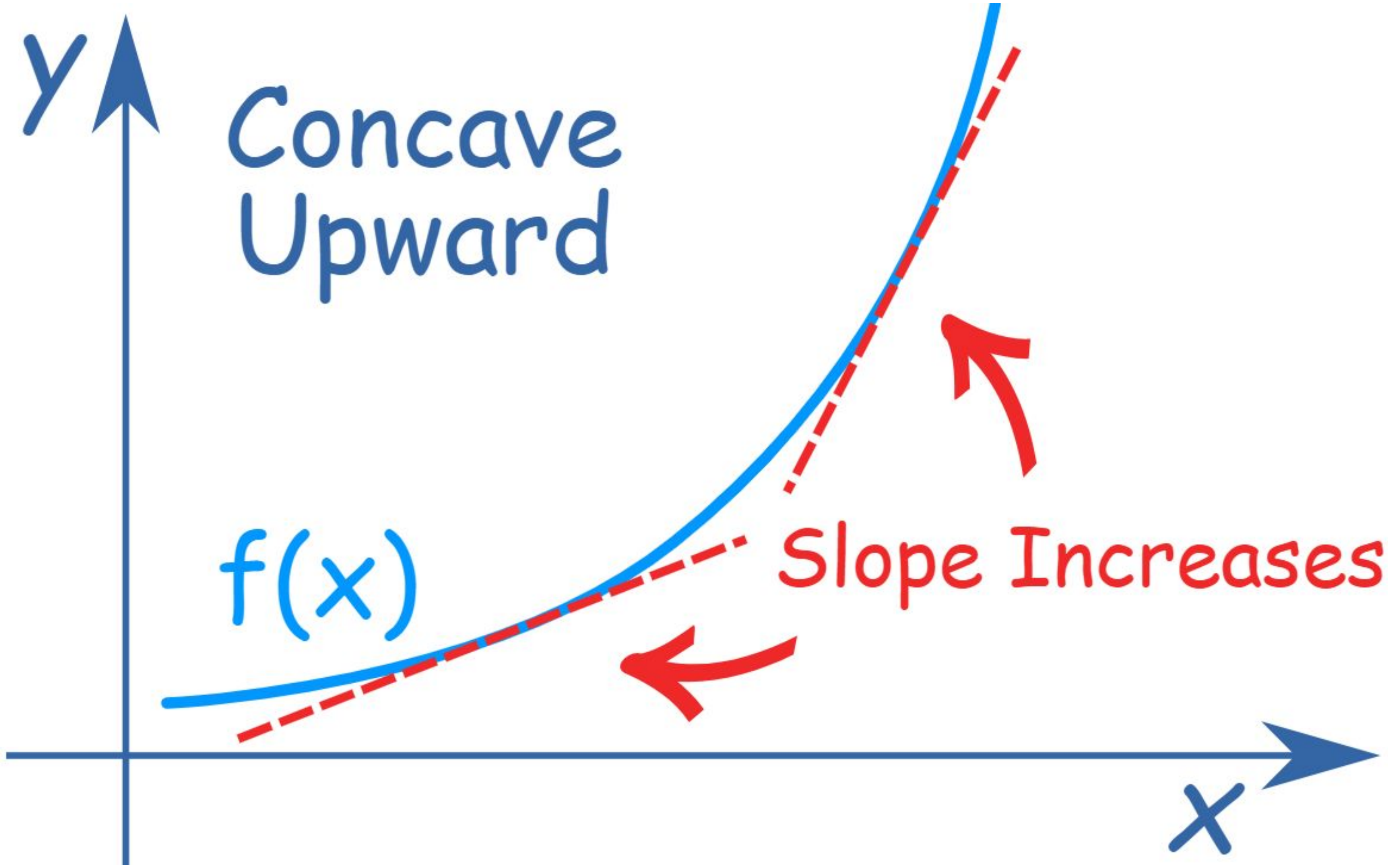
Convex



Non-Convex

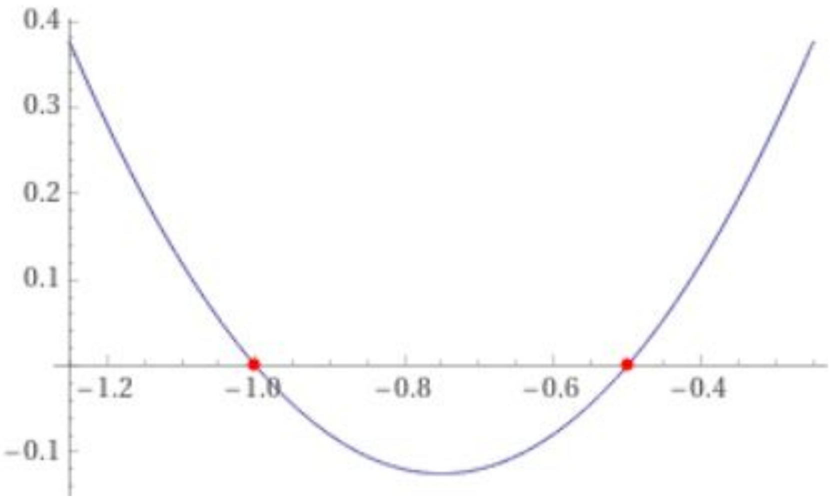


凸函數性質 – 斜率由左到右遞增



不透過微分算斜率 – 極近的兩點法

```
double f(double x) { return 2 * x * x + 3 * x + 1; }
double getSlope(double x) {
  static double eps = 1e-8;
  return (f(x + eps) - f(x)) / eps;
}
```



$$f(x) = 2x^2 + 3x + 1$$
$$f'(x) = 4x + 3$$

```
cout << getSlope(-8) << endl; // -29
cout << getSlope(-4) << endl; // -13
cout << getSlope(-0) << endl; // 3
cout << getSlope(4) << endl; // 19
cout << getSlope(8) << endl; // 35
```

浮點數二分搜 – 自定義精確度

```cpp
template <class Ty, class FuncTy>
pair<Ty, Ty> binarySearch(Ty L, Ty R, FuncTy check, Ty eps = 1) {
  if (check(R) == true) return {R, R + 1};
  if (check(L) == false) return {L - 1, L};
  while (L + eps < R) {
    Ty Mid = L + (R - L) / 2;
    if (check(Mid))
      L = Mid;
    else
      R = Mid;
  }
  return {L, R};
}
```

輕鬆找出最低點

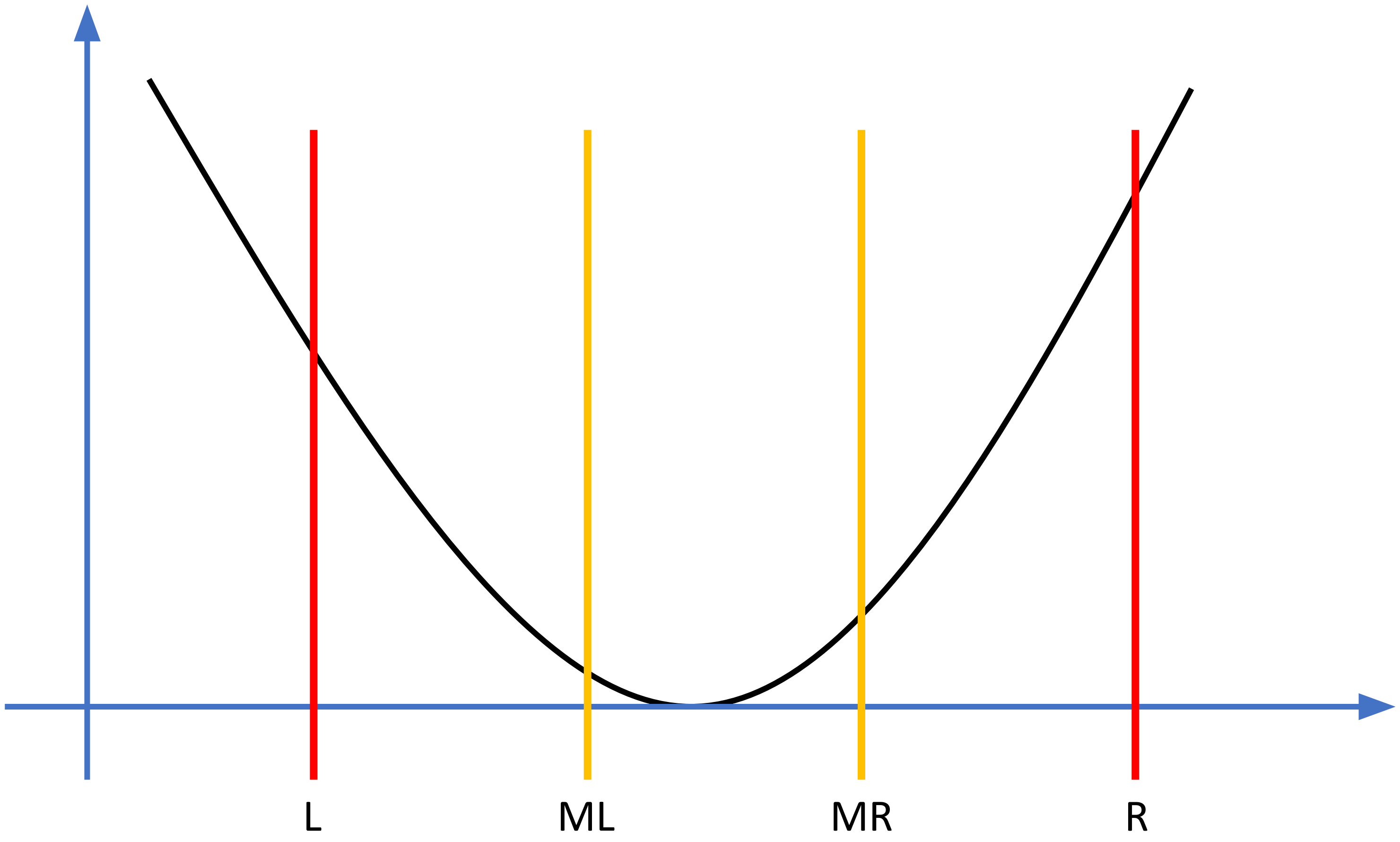
```cpp
double L = -10, R = 10;
tie(L, R) = binarySearch(
    L, R, [&](double mid) { return getSlope(mid) < 0; }, 1e-6);
cout << L << ' ' << R << endl; // -0.75 -0.75
```

注意 L 和 R 的 eps 不能比 getSlope 中的 eps 還要小

時間複雜度：$O\left(\log\frac{(R-L)}{eps}\right)$

有兩個誤差不精準？試試三分搜

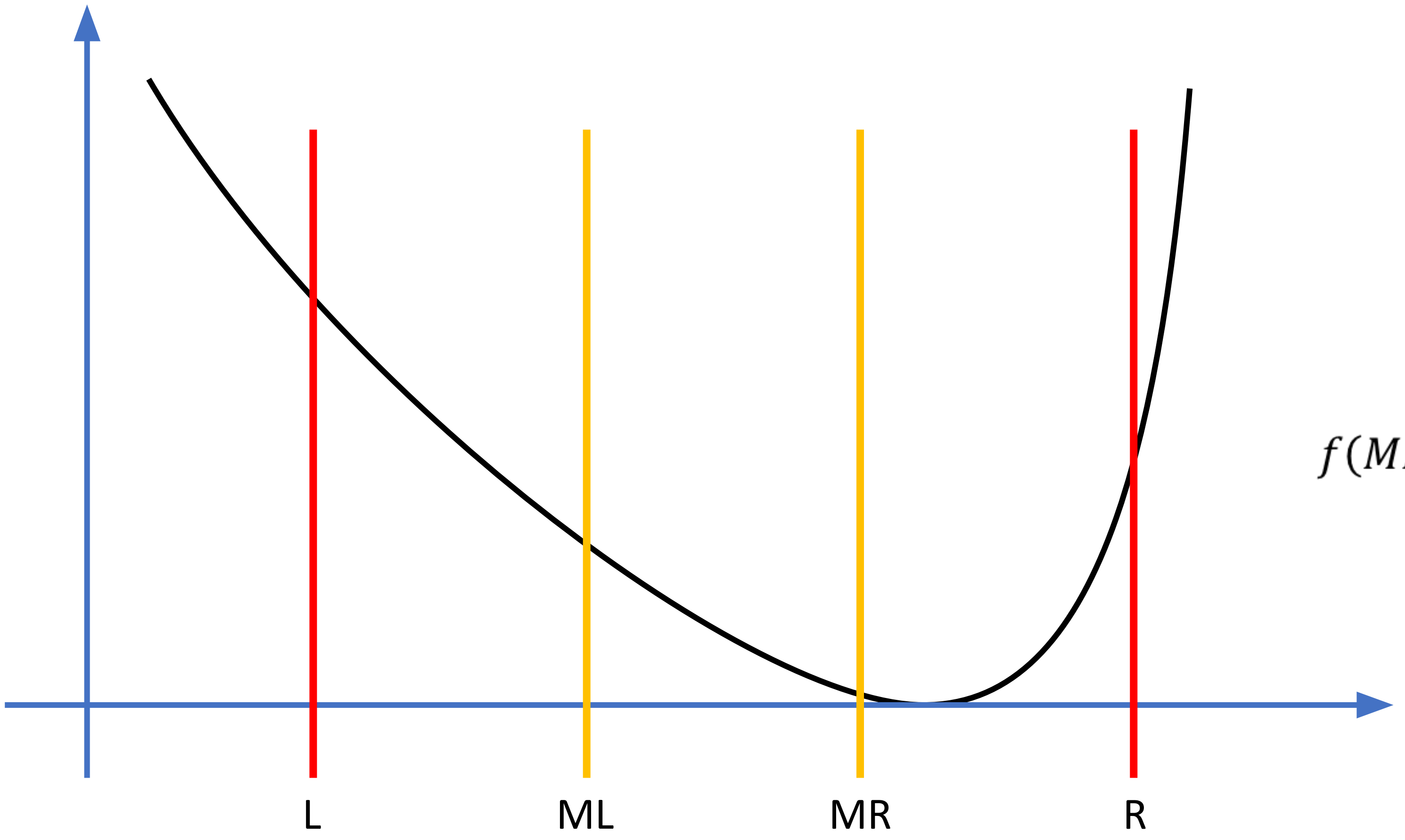
將 L R 區間分成三份



ML = L + (R-L)/3

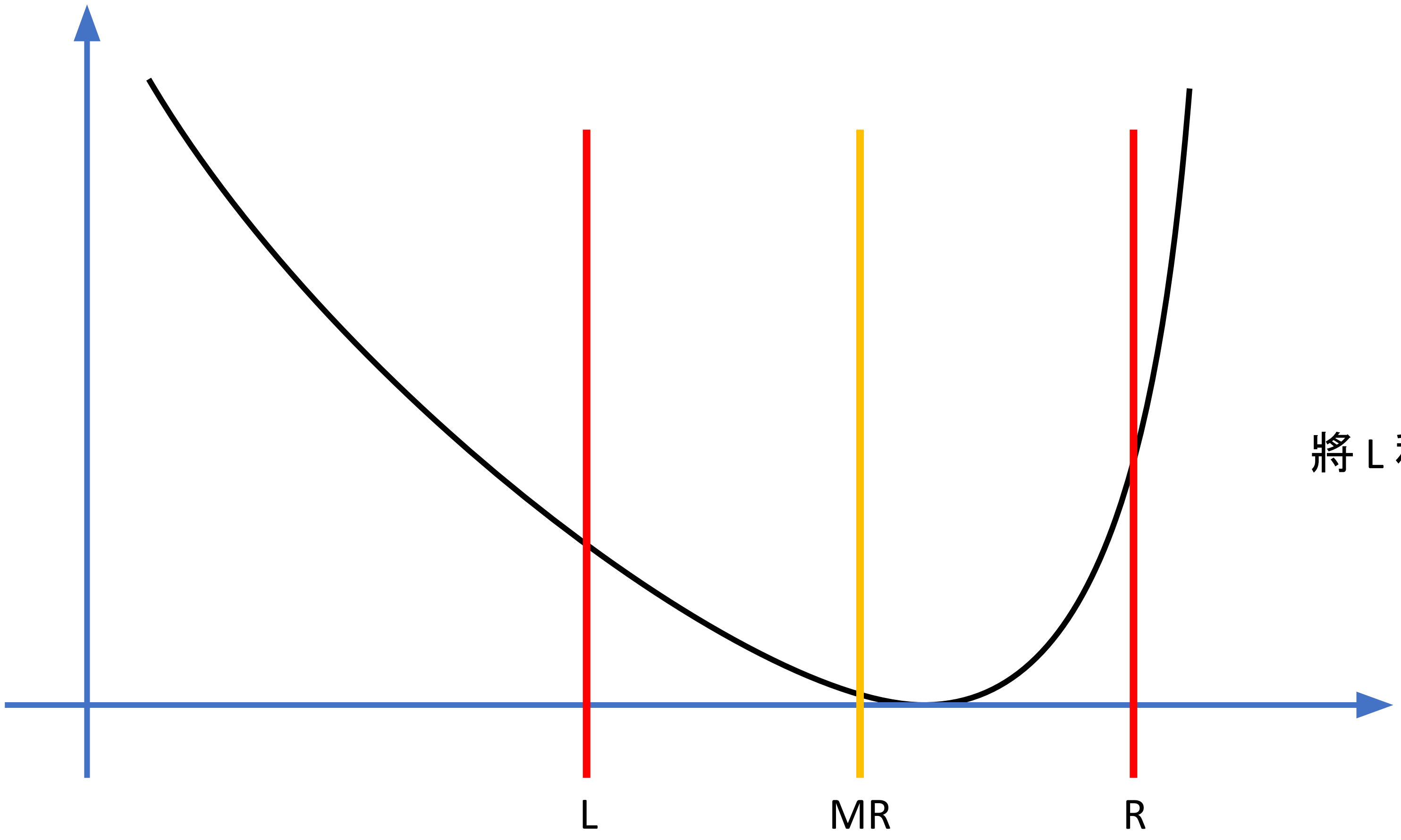
MR = R - (R-L)/3

Case 1: 最小值位於 MR 和 R 之間



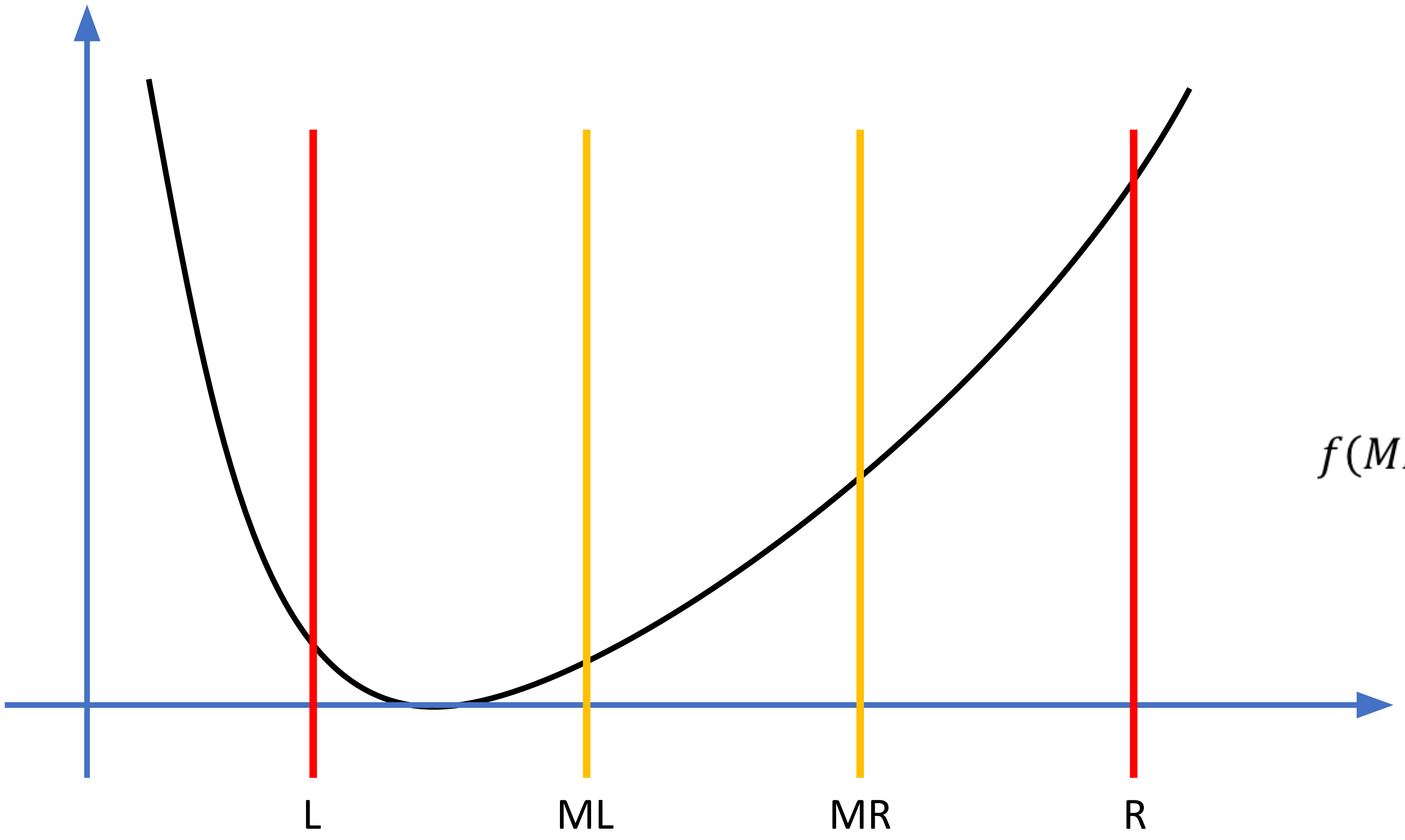
$f(ML) > f(MR)$

Case 1: 最小值位於 MR 和 R 之間



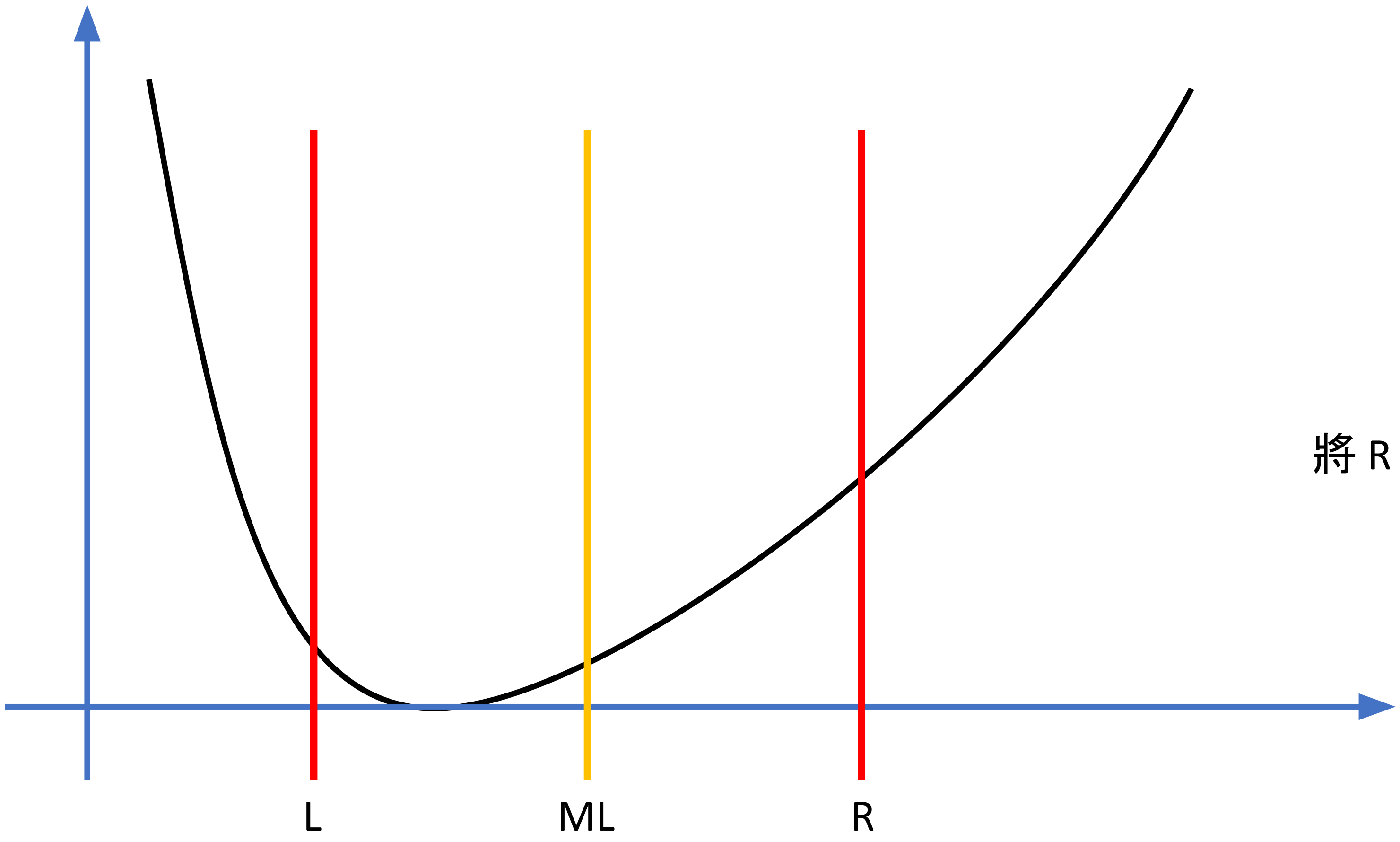
將 L 移動到 ML 不會影響答案

Case 2: 最小值位於 L 和 ML 之間



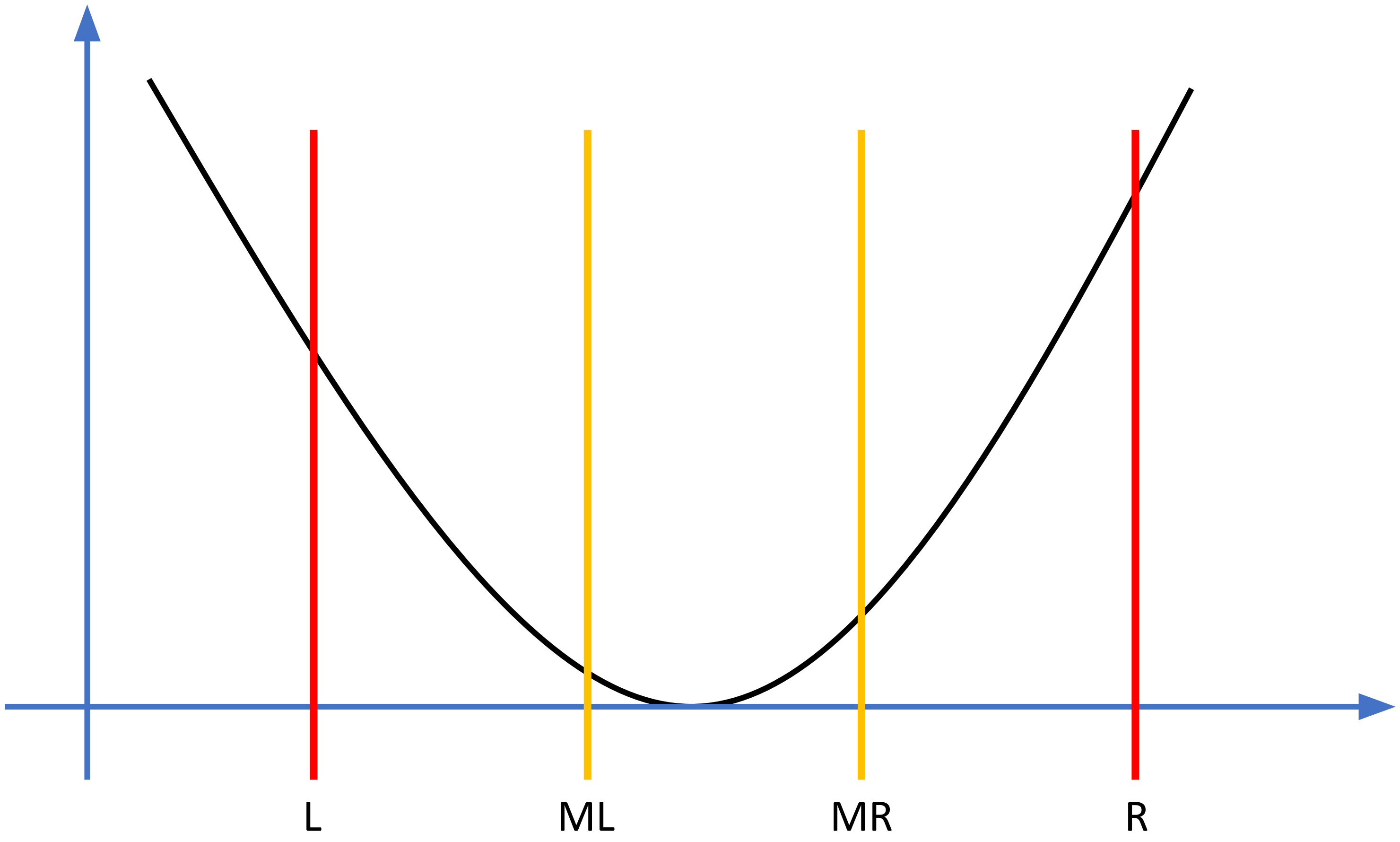
$f(ML) < f(MR)$

Case 2: 最小值位於 L 和 ML 之間



將 R 移動到 MR 不會影響答案

Case 3: 最小值位於 ML 和 MR 之間



將 L 移動至 ML
或 R 移動至 MR
都不會影響正確性

三分搜尋實作 $O\left(\log\frac{(R-L)}{eps}\right)$

```cpp
template <typename FuncTy>
pair<double, double> ternarySearch(double L, double R, FuncTy func,
                                   double eps = 1e-6) {
  while (L + eps < R) {
    double mL = L + (R - L) / 3;
    double mR = R - (R - L) / 3;
    if (func(mL) > func(mR))
      L = mL;
    else
      R = mR;
  }
  return {L, R};
}
```

```cpp
double f(double x) { return 2 * x * x + 3 * x + 1; }

int main() {
  double L = -10, R = 10;
  tie(L, R) = ternarySearch(L, R, f);
  cout << L << ' ' << R << endl; // -0.75 -0.75
  return 0;
}
```

STL 2 – 搜索相關的 STL

#include <algorithm>

- std::lower_bound
- std::upper_bound
- std::nth_element

Lower, Upper bound

$$x = 7$$

Lower bound
>= x 的最小值

Upper bound
> x 的最小值

0	1	2	3	4	5	6	7	8	9
2	3	3	7	7	7	9	9	12	12

Lower, Upper bound

$$x = 8$$

Upper bound
$>x$ 的最小值

0	1	2	3	4	5	6	7	8	9
2	3	3	7	7	7	9	9	12	12

Lower bound
$>= x$ 的最小值

std::lower_bound $O(\log n)$

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  const int n = 8;
  int A[n] = {1, 2, 2, 2, 3, 3, 4, 5};
  auto Ptr = lower_bound(A, A + n, 2);
  cout << Ptr - A << endl; // 1

  vector<int> V(A, A + n);
  auto Iter = lower_bound(V.begin(), V.end(), 3);
  cout << Iter - V.begin() << endl; // 4
  return 0;
}
```

std::upper_bound $O(\log n)$

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  const int n = 8;
  int A[n] = {1, 2, 2, 2, 3, 3, 4, 5};
  auto Ptr = upper_bound(A, A + n, 2);
  cout << Ptr - A << endl; // 4

  vector<int> V(A, A + n);
  auto Iter = upper_bound(V.begin(), V.end(), 3);
  cout << Iter - V.begin() << endl; // 6
  return 0;
}
```

綜合應用：某個數字出現幾次

```
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> V{1, 2, 2, 2, 3, 3, 4, 5};
  int x = 2;
  auto LowerIter = lower_bound(V.begin(), V.end(), x);
  auto UpperIter = upper_bound(V.begin(), V.end(), x);
  cout << "number of x = " << UpperIter - LowerIter << endl;
  // number of x = 3
  return 0;
}
```

兩者都可以使用比較函數

```cpp
#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;
int main() {
  vector<int> V{1, -2, 2, -2, 3, -3, 4, -5};
  auto cmp = [](int a, int b) { return abs(a) < abs(b); };
  int x = 2;
  auto LowerIter = lower_bound(V.begin(), V.end(), x, cmp);
  auto UpperIter = upper_bound(V.begin(), V.end(), x, cmp);
  cout << "number of x = " << UpperIter - LowerIter << endl;
  // number of x = 3
  return 0;
}
```

std::nth_element $O(n)$

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

void printVec(const std::vector<int> &vec) {
  cout << "v = {";
  for (int i : vec) cout << i << ", ";
  cout << "}\n";
}

int main() {
  vector<int> v{5, 10, 6, 4, 3, 2, 6, 7, 9, 3};
  printVec(v);
  auto m = v.begin() + v.size() / 2;
  nth_element(v.begin(), m, v.end());
  cout << "\nThe median is " << v[v.size() / 2] << '\n';
  printVec(v);
}
```

v = {5, 10, 6, 4, 3, 2, 6, 7, 9, 3, }

The median is 6

v = {3, 2, 3, 4, 5, 6, 10, 7, 9, 6, }

≤ 6 $\quad$ $= 6$ $\quad$ ≥ 6

當然也可以自定義比較函數

```cpp
int main() {
  vector<int> v{5, 10, 6, 4, 3, 2, 6, 7, 9, 3};
  printVec(v);
  auto m = v.begin() + v.size() / 2;
  nth_element(v.begin(), m, v.end(), greater<int>());
  cout << "\nThe median is " << v[v.size() / 2] << '\n';
  printVec(v);
}
```

v = {5, 10, 6, 4, 3, 2, 6, 7, 9, 3, }

The median is 5
v = {7, 10, 9, 6, 6, 5, 4, 3, 2, 3, }

≥ 5 ≤ 5

$= 5$

關聯容器

set map
multiset
multimap

關聯容器元素需要是可以比較(<)的

multi系列可以儲存重複的資料

關聯容器

- set / multiset

```
#include <set>
```

  - 可以儲存/刪除/查詢元素 $O(\log n)$

- map / multimap

```
#include <map>
```

  - 跟set差不多，但是可以儲存元素的對應關係。
  - map [a] = b;

新增元素insert / emplace $O(\log n)$

- 呼叫map[d]時就會在map建立d的引索，注意不要誤建多餘的資料

```
int main() {
  set<int> S;
  map<string, int> M;
  S.insert(1);
  S.emplace(1);
  M["ACD"] = 1; // Slow
  M.insert(make_pair("BGC", 2));
  M.emplace("GDS", 3);
  return 0;
}
```

查詢元素

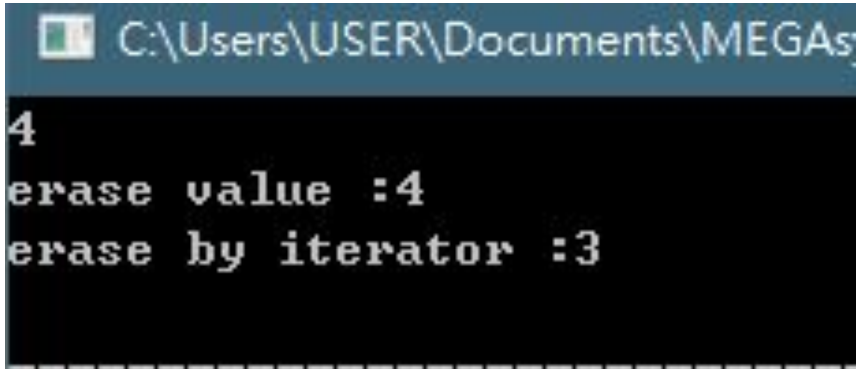
- count(d)的複雜度是$O(\log 容器大小 + 相等數值的數量)$
- 在multi系列有可能退化為$O(n)$，要注意

```cpp
int main() {
  multiset<int> S{1, 2, 6, 6, 6, 6, 6, 8, 5};
  if (S.find(2) != S.end()) // O(log n)
    cout << "Found 2\n";
  cout << "Num 6: " << S.count(6) << '\n'; // not O(log n)
  return 0;
}
```

移除元素Erase $O(\log n + \text{被移除的數量})$

```cpp
int main() {
  set<int> A{1, 2, 3, 4, 5};
  A.erase(1);
  cout << A.size() << '\n';

  multiset<int> B{1, 1, 1, 2, 2, 2, 2};
  B.erase(1);
  cout << "erase value :" << B.size() << '\n';
  B.erase(B.find(2));
  cout << "erase by iterator :" << B.size() << '\n';
  return 0;
}
```



lower_bound upper_bound $O(\log n)$

```cpp
int main() {
  multiset<int> S{7, 1, 2, 2, 2, 5, 3};
  auto Iter = S.lower_bound(2); // 大於或等於 2 的第一個 Iterator
  auto End = S.upper_bound(2);  // 大於 2 的第一個 Iterator
  int Cnt = 0;
  for (; Iter != End; ++Iter)
    ++Cnt;
  cout << "Number of 2: " << Cnt << '\n';
  return 0;
}
```

自訂比較

```cpp
struct CMP {
  bool operator()(int a, int b) { return a > b; }
};
int main() {
  set<int> A{1, 2, 3, 4, 5};
  set<int, CMP> B{1, 2, 3, 4, 5};

  cout << "A: " << *A.begin() << ' ' << *A.rbegin() << '\n';
  cout << "B: " << *B.begin() << ' ' << *B.rbegin() << '\n';
  return 0;
}
```

無關聯性容器 unordered_set unordered_map

有multi系列，只是名子太長懶得放

放棄了排序使用雜湊，元素的加入/查詢都是**平均**最差

用法與set/map差不多，只是自訂雜湊很難寫。

近來因為優秀的複雜度，有被**圍剿**的跡象，小心使用。

```
#include <unordered_set>
#include <unordered_map>
```