1. Complexity Analysis $(1, n, n^2, 2^n, n!)$

$f(n) = O(g(n))$ when $g(n) \ge f(n)$

2. Master theorem, Recurrence

$$T(n) = aT\left(\frac{n}{b}\right) + n^c, \quad T(n) = \begin{cases} \theta(n^{\log_b a}) & a > b^c \\ \theta(n^c \log_2 n) & a = b^c \\ \theta(n^c) & a < b^c \end{cases}$$

3. Arrays. Linked list

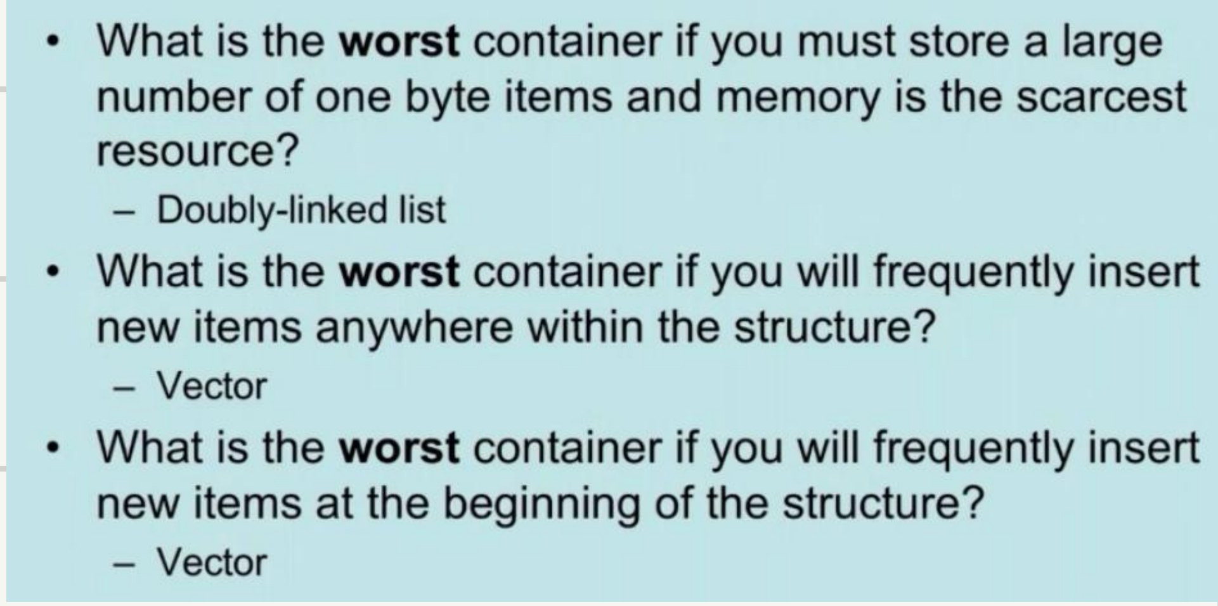
Random in O(1) / O(n), Sequential O(1) / O(1)

Insert in O(n) / O(n), Append O(1) / O(n)

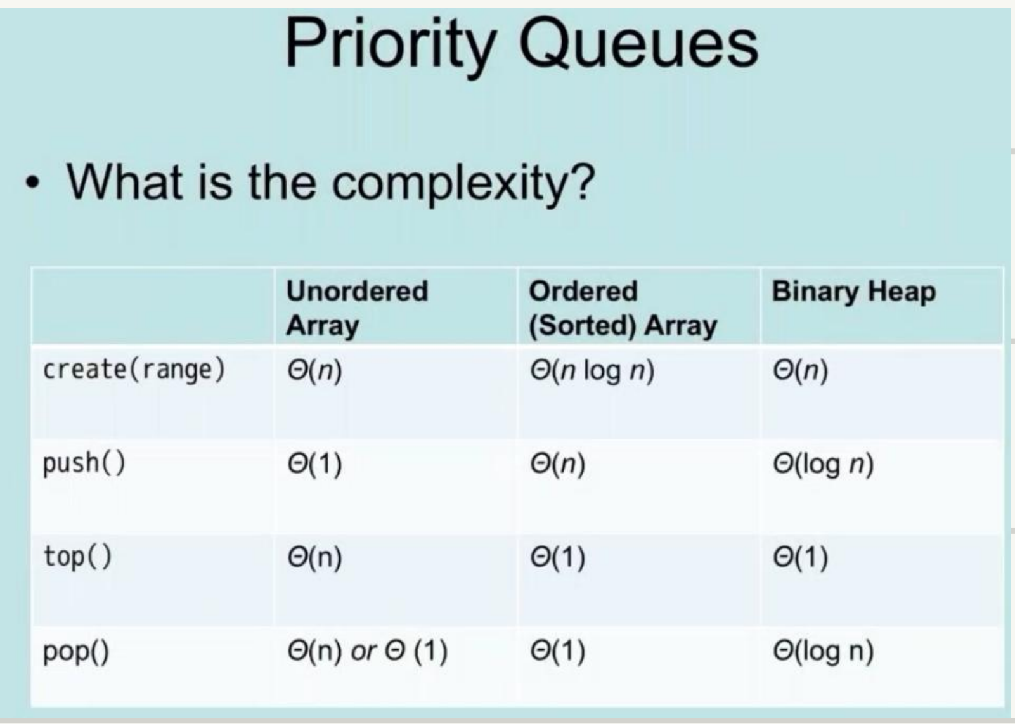
Ordered	[]	insertAfter	delete
Array	O(1)	O(n)	O(n)
Linked List	O(n)	O(1)	O(1)

Sorted	[]	find()	delete
Array	O(1)	O(logn)	O(n)
Linked List	O(n)	O(n)	O(1)

Priority Queue is not iterable / searchable / ordered



- What is the **worst** container if you must store a large number of one byte items and memory is the scarcest resource?
  - Doubly-linked list
- What is the **worst** container if you will frequently insert new items anywhere within the structure?
  - Vector
- What is the **worst** container if you will frequently insert new items at the beginning of the structure?
  - Vector



Priority Queues

- What is the complexity?

	Unordered Array	Ordered (Sorted) Array	Binary Heap
create(range)	Θ(n)	Θ(n log n)	Θ(n)
push()	Θ(1)	Θ(n)	Θ(log n)
top()	Θ(n)	Θ(1)	Θ(1)
pop()	Θ(n) or Θ (1)	Θ(1)	Θ(log n)

6. Sorting Algorithms (Elementry / Advanced)

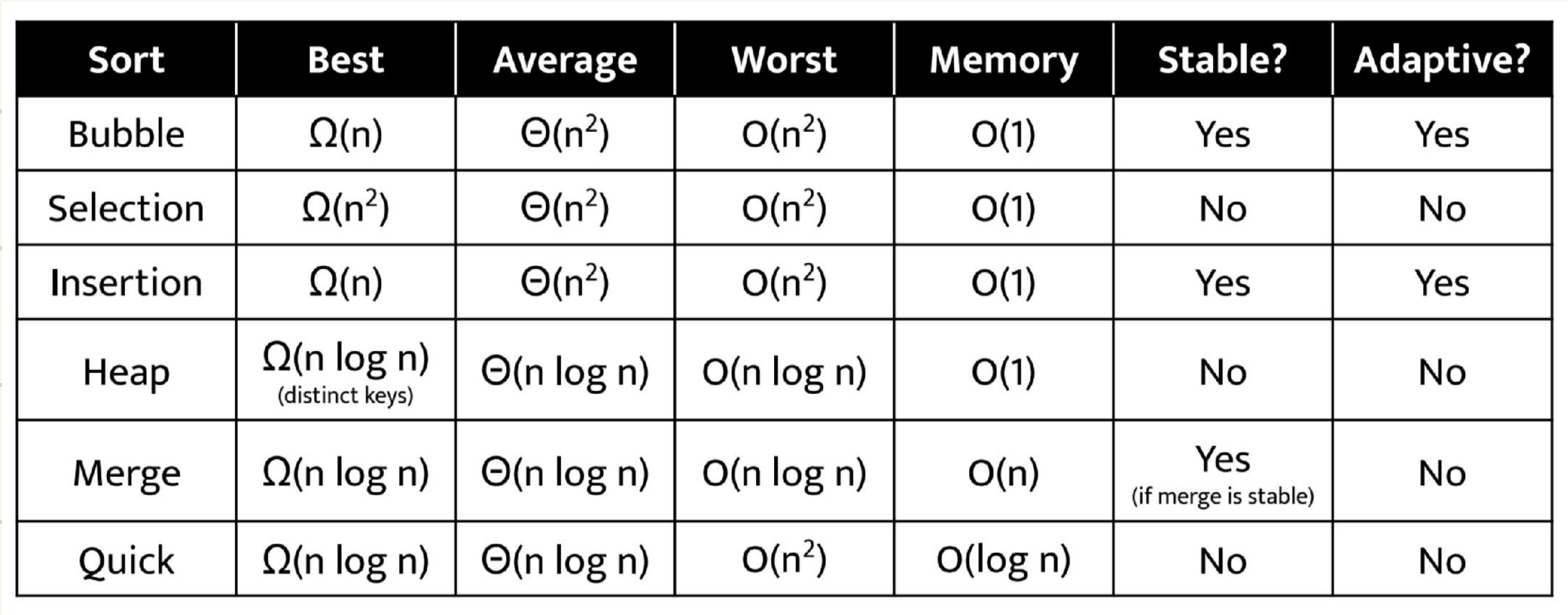
Bubble Sort : 옆자리와 비교하면서 (compare two) largest element to top

Selection Sort : 제일 작은것 앞으로, 그다음, 그다음 ... 반복

Insertion Sort : 범위 하나씩 늘려가면서 sort

→ Best on small input, good for nearly sorted

Quick Sort : pick pivot left pointer 더크면, right pointer 작으면 swap



Sort	Best	Average	Worst	Memory	Stable?	Adaptive?
Bubble	Ω(n)	Θ(n²)	O(n²)	O(1)	Yes	Yes
Selection	Ω(n²)	Θ(n²)	O(n²)	O(1)	No	No
Insertion	Ω(n)	Θ(n²)	O(n²)	O(1)	Yes	Yes
Heap	Ω(n log n) (distinct keys)	Θ(n log n)	O(n log n)	O(1)	No	No
Merge	Ω(n log n)	Θ(n log n)	O(n log n)	O(n)	Yes (if merge is stable)	No
Quick	Ω(n log n)	Θ(n log n)	O(n²)	O(log n)	No	No

std::sort → Θ(nlogn)

7. Priority Queues / Heaps (priority q.push : Θ(logn))

Priority Queues implemented with Binary heaps

max-heap (std::less), min-heap (std::greater)

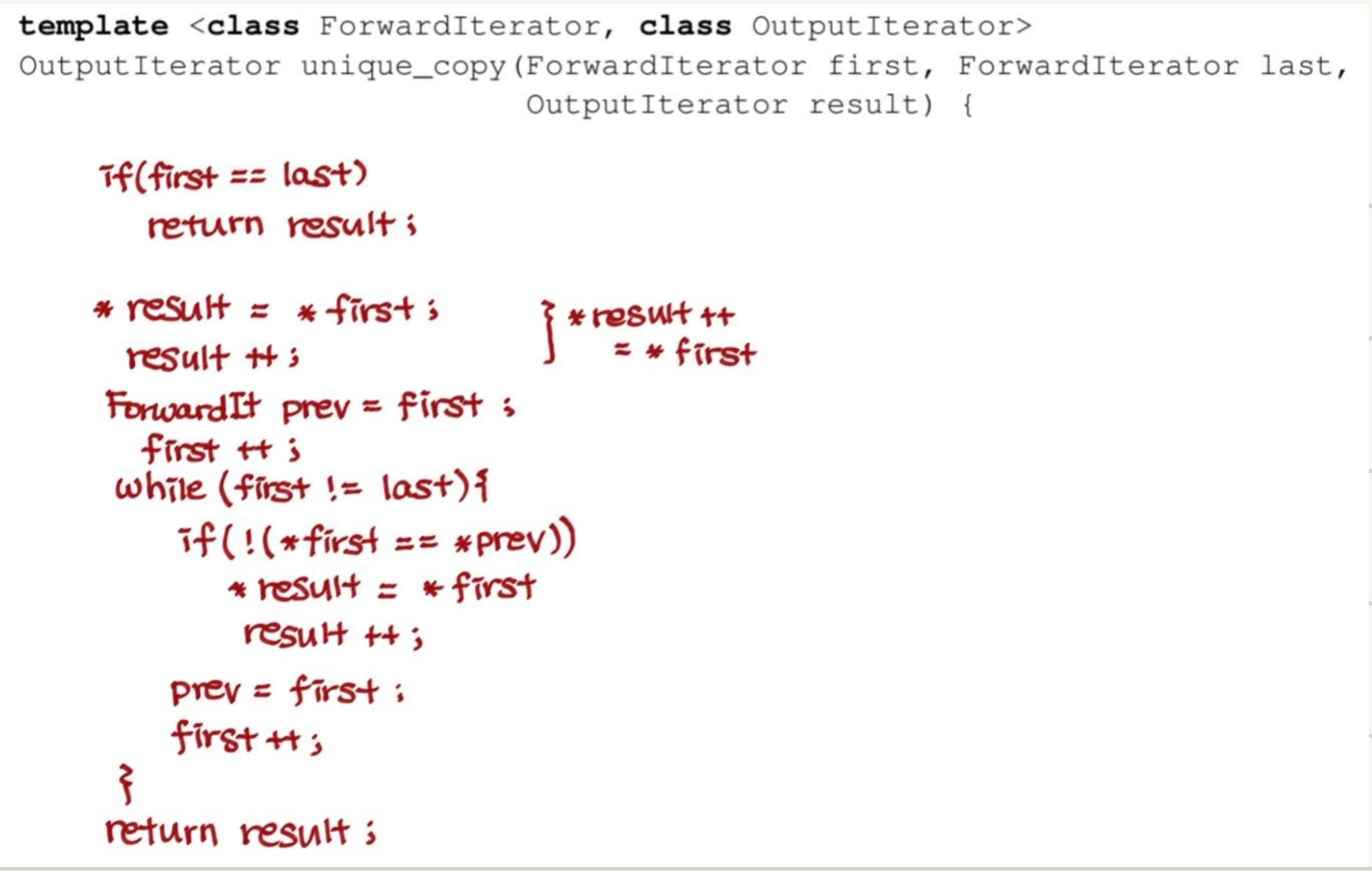
fixUp, fixDown, pop → O(logn)

Heapify → bottom up, fixDown() → O(n) (fixUp → O(nlogn))

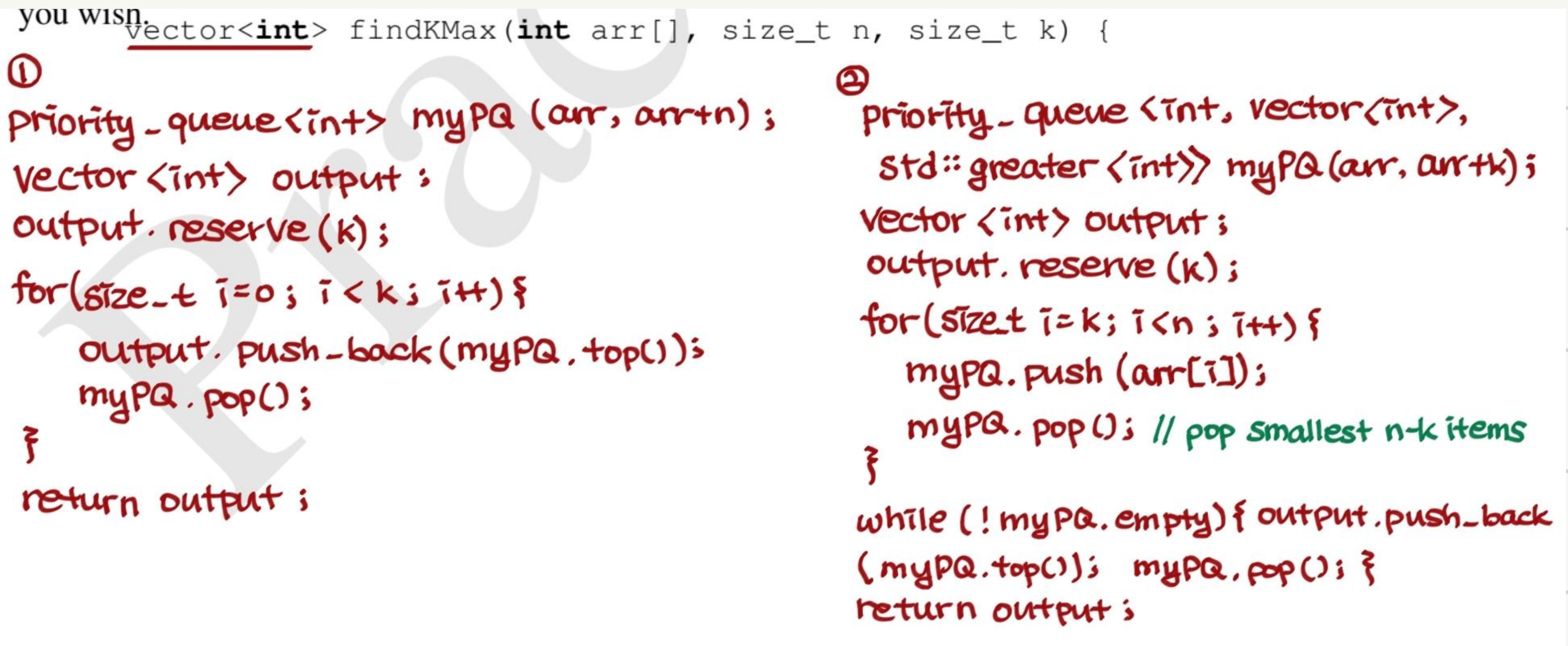
heapsort

- make maxheap (O(n)) and swap with the last element then fixDown() again and again (O(nlogn)) → complexity O(nlogn + n) = O(nlogn)

memory : O(1) needed

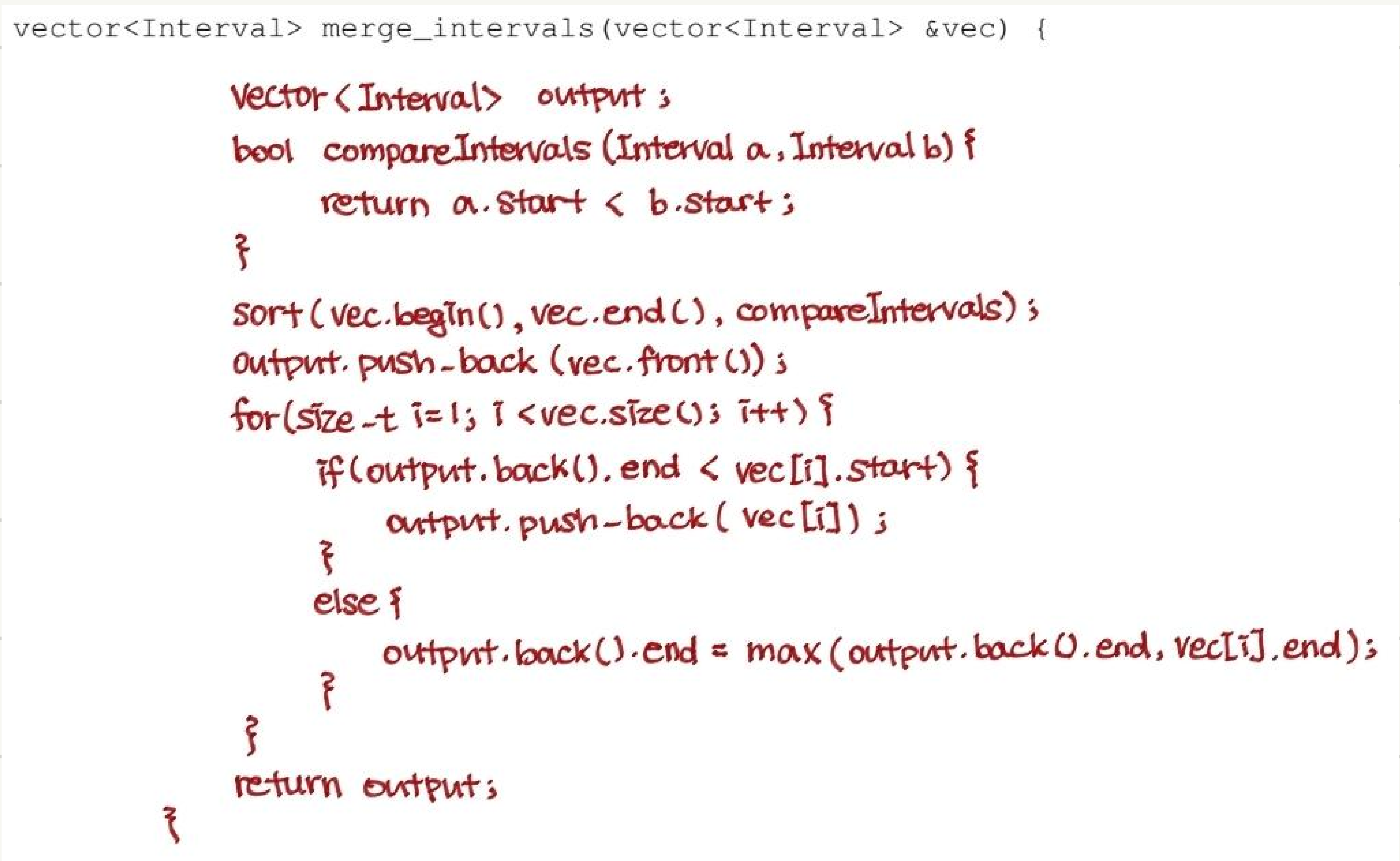


```
template <class ForwardIterator, class OutputIterator>
OutputIterator unique_copy(ForwardIterator first, ForwardIterator last,
                           OutputIterator result) {

    if(first == last)
        return result;

    *result = *first;          } *result ++
     result ++;                      = * first
    ForwardIt prev = first;
     first ++;
    while (first != last){
        if(!(*first == *prev))
            *result = *first
             result ++;
        prev = first;
        first ++;
    }
    return result;
```

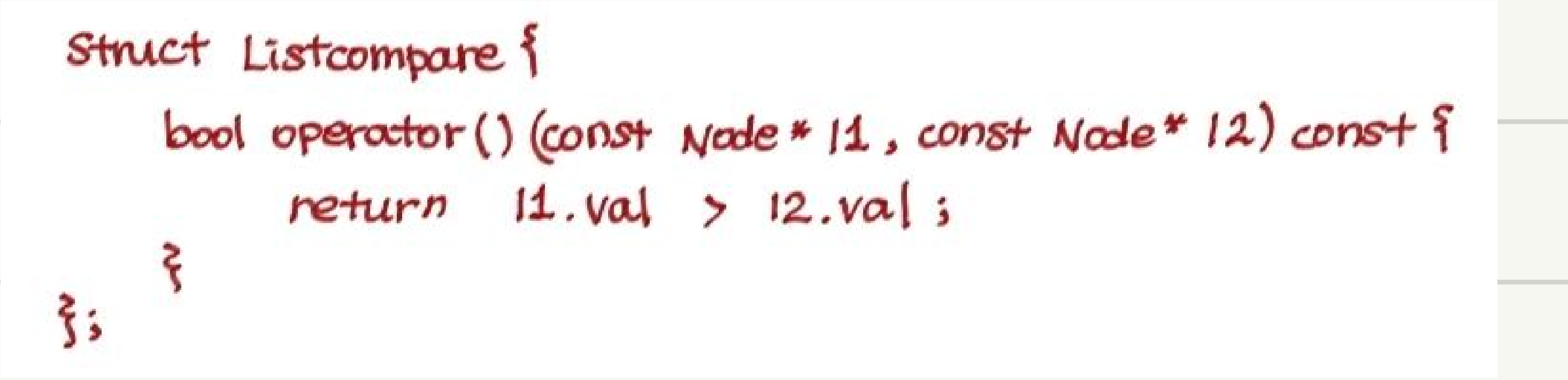


you wisn.

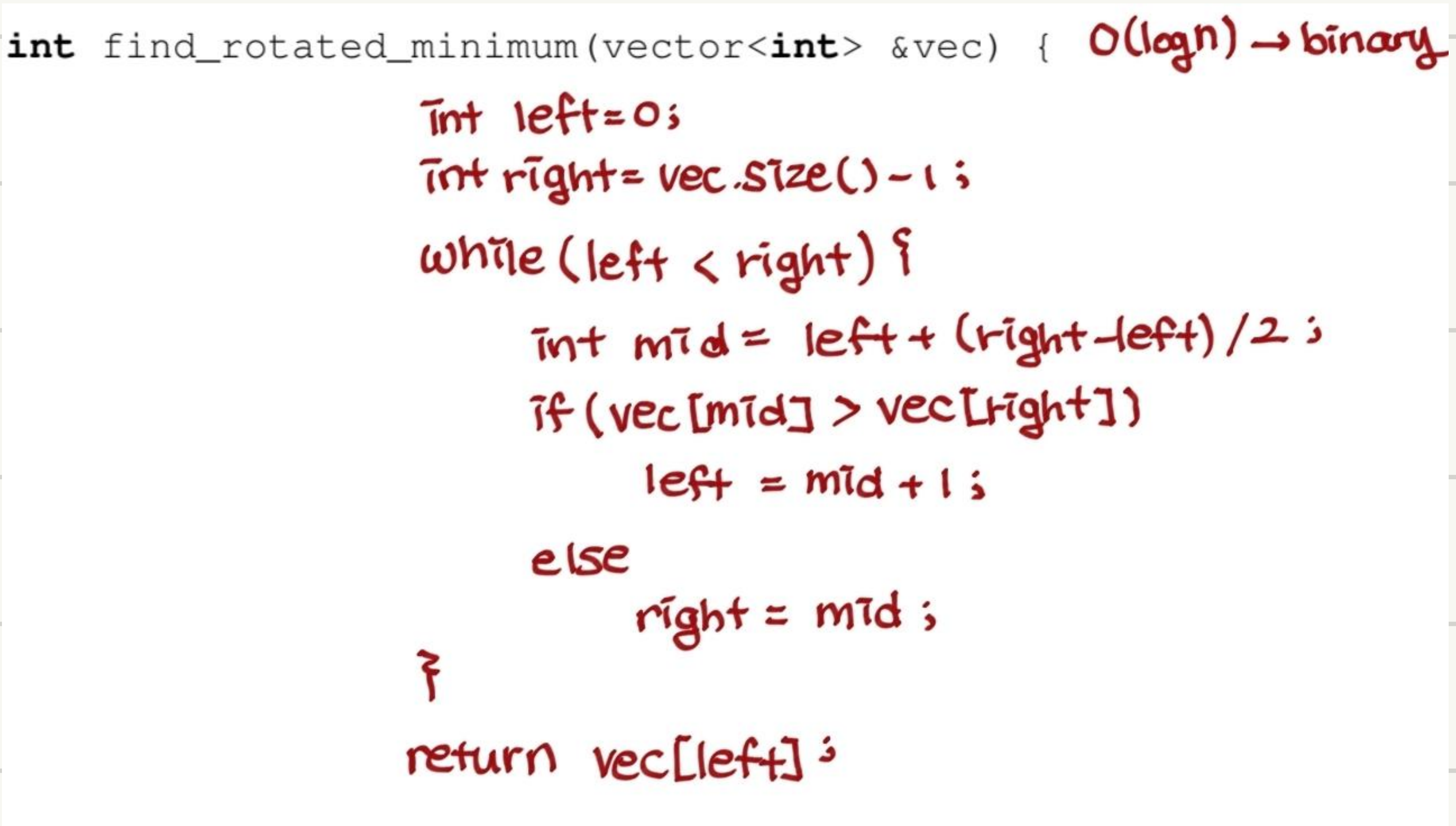
```
vector<int> findKMax(int arr[], size_t n, size_t k) {
①
priority_queue<int> myPQ (arr, arr+n);
vector <int> output;
output. reserve(k);
for(size_t i=0; i<k; i++){
    output. push_back(myPQ.top());
    myPQ.pop();
}
return output;

②
priority_queue <int, vector<int>,
 std::greater <int>> myPQ(arr, arr+k);
vector <int> output;
output. reserve (k);
for(size_t i=k; i<n; i++){
    myPQ.push (arr[i]);
    myPQ. pop(); // pop smallest n-k items
}
while (!myPQ.empty){ output.push_back
(myPQ.top()); myPQ.pop(); }
return output;
```



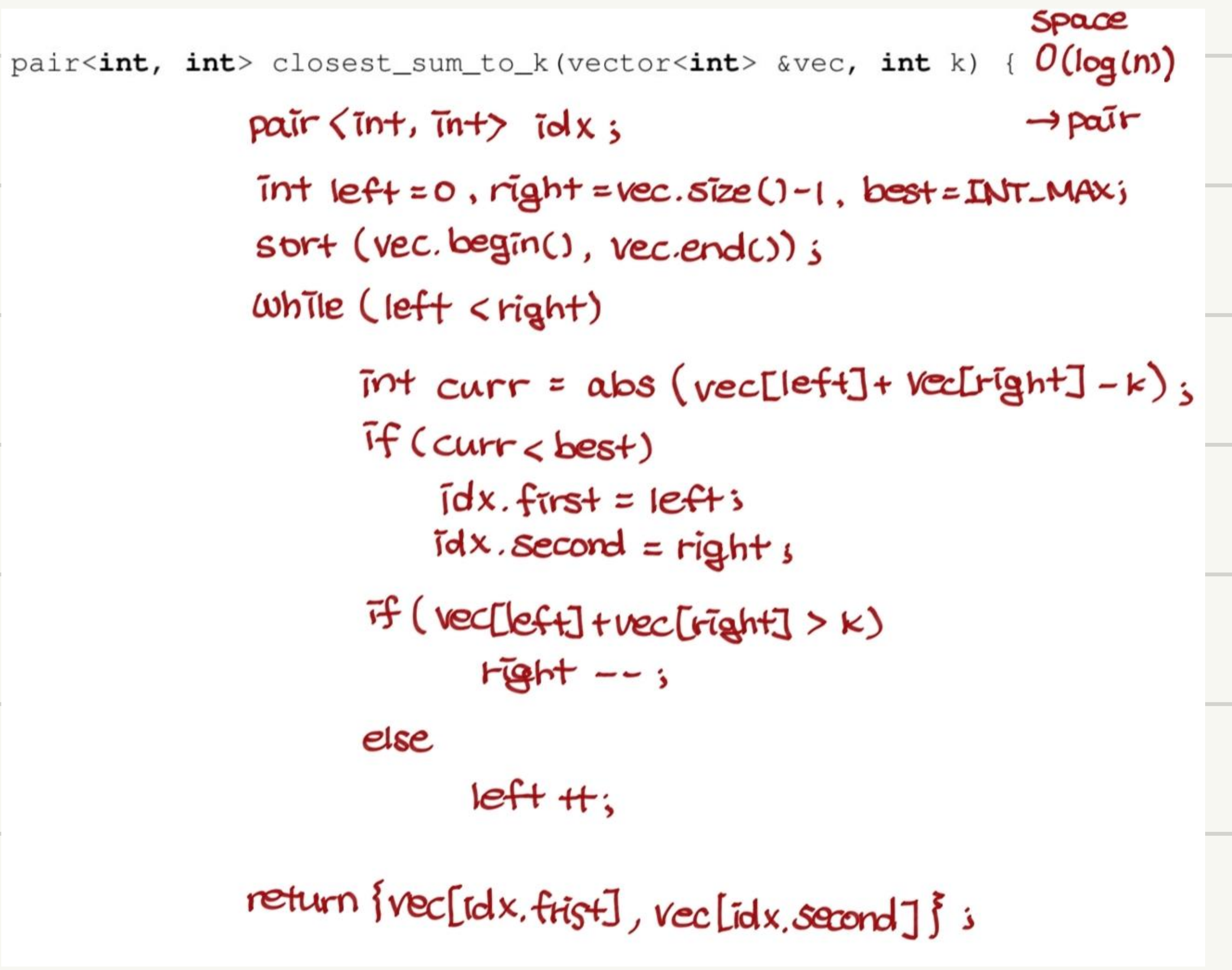
```
vector<Interval> merge_intervals(vector<Interval> &vec) {

        vector<Interval>  output;
        bool  compareIntervals (Interval a, Interval b) {
              return a.start < b.start;
        }
        sort (vec.begin(), vec.end(), compareIntervals);
        output. push_back (vec.front());
        for(size_t i=1; i<vec.size(); i++) {
              if(output.back().end < vec[i].start) {
                    output. push_back( vec[i]);
              }
              else {
                    output.back().end = max(output.back().end, vec[i].end);
              }
        }
        return output;
}
```



```
Struct Listcompare {
    bool operator() (const Node* l1, const Node* l2) const {
        return  l1.val > l2.val;
    }
};
```



```
int find_rotated_minimum(vector<int> &vec) {   O(logn) → binary
            int left=0;
            int right= vec.size()-1;
            while (left < right) {
                  int mid = left + (right-left)/2;
                  if (vec[mid] > vec[right])
                        left = mid + 1;
                  else
                        right = mid;
            }
            return vec[left];
```



```
                                                                      space
pair<int, int> closest_sum_to_k(vector<int> &vec, int k) {        O(log(n))
            pair<int, int> idx;                                     → pair
            int left=0, right=vec.size()-1, best=INT_MAX;
            sort (vec.begin(), vec.end());
            while (left <right)

                  int curr = abs (vec[left]+ vec[right] - k);
                  if (curr < best)
                        idx.first = left;
                        idx.second = right;

                  if (vec[left]+vec[right] > k)
                        right --;
                  else
                        left ++;

            return {vec[idx.frist], vec[idx.second]};
```