**Examples:**

[38, 6, 23, 2, 62, 4, 72, 7, 83, 5, 23, 6, 24, 2]

Input k: 5

The value of the k-th number is 6

[38, 6, 23, 2, 62, 4, 72, 7, 83, 5, 23, 6, 24, 2]

Input k: 6

The value of the k-th number is 6

[38, 6, 23, 2, 62, 4, 72, 7, 83, 5, 23, 6, 24, 2]

Input k: 8

The value of the k-th number is 23

[100, 99, 98, 97, 96, 95, 94, 93, 92, 91, 90, 89, 88, 87, 86, 85, 84, 83, 82, 81, 80, 79, 78, 77, 76, 75, 74, 73, 72, 71, 70, 69, 68, 67, 66, 65, 64, 63, 62, 61, 60, 59, 58, 57, 56, 55, 54, 53, 52, 51, 50, 49, 48, 47, 46, 45, 44, 43, 42, 41, 40, 39, 38, 37, 36, 35, 34, 33, 32, 31, 30, 29, 28, 27, 26, 25, 24, 23, 22, 21, 20, 19, 18, 17, 16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

Input k: 85

The value of the k-th number is 85

**Message:**

I made a Message class, which has 5 variables: sign, pivot, side, getPivot, and count.

Sign and Side are emuns: ***PARTITION***, ***CHOOSE***, ***REQUEST*** and ***LEFT***, ***RIGHT.***

A message has what content depend on which constructor programmer invoke.

The reason why I use emun is it can help my code more readable:)

**public** **enum** Sign {

***PARTITION***, ***CHOOSE***, ***REQUEST***

}

**public** **enum** Side {

***LEFT***, ***RIGHT***

}

**public** **class** Message **implements** Serializable{

**private** **static** **final** **long** ***serialVersionUID*** = 1L;

Sign sign;

**int** pivot;

Side side;

Boolean getPivot;

**int** count;

**public** Message(Sign sign) {

**super**();

**this**.sign = sign;

}

**public** Message(Sign sign, **int** pivot) {

**super**();

**this**.sign = sign;

**this**.pivot = pivot;

}

**public** Message(Sign sign, Side side) {

**super**();

**this**.sign = sign;

**this**.side = side;

}

**public** Message(**int** count) {

**super**();

**this**.count = count;

}

**public** Message(Boolean getPivot, **int** pivot) {

**super**();

**this**.getPivot = getPivot;

**this**.pivot = pivot;

}

**public** Message(Boolean getPivot) {

**super**();

**this**.getPivot = getPivot;

}

}

**Pseudo-code:**

**Initiator**:

create array;

send subarray to node1 and node2;

split array into two pieces;

input k;

kIndex = k – 1;

**while** (**true**) {

ask node1 and node2 to ***PARTITION;***

receive count1 and count2;

pivotIndex = count1 + count2;

**if** (pivotIndex > kIndex) {

ask node1 and node2 keep left(***CHOOSE***);

} **else** **if** (pivotIndex < kIndex) {

ask node1 and node2 keep left(***CHOOSE***);

kIndex = kIndex - pivotIndex - 1;

} **else** {

**break**;

}

ask node1 a pivot(***REQUEST***);

if(getPivot==false)

ask node2 a pivot(***REQUEST***);

}

**Node**:

Sequential Node and Parallel Node are also the same. The only different is sequential nodes invoke sequentialPartition(); parallel nodes invoke parallelPartition() at first. When the size of subarray is less than 4, it will invoke sequentialPartition(), because parallelPartition() will spilt the array into 4 pieces. If the size is less than 4, it won’t work well. Both sequentialPartition() and parallelPartition() are as the same as the question 5 of last group project.

create two nodes;

receive subarray;

end= array.length – 1;

start = 0;

**while** (end >= start) {

receive msg;

**switch** (msg.sign) {

**case** ***PARTITION***:

pivot = msg.pivot;

count = *sequentialPartition*();

send count to Initiator;

**break**;

// The parallel nodes will do operations below.

// if (end - start >= 4) {

// count = parallelPartition();

// } else {

// count = sequentialPartition();

// }

// send count to Initiator;

// break;

**case** ***CHOOSE***:

**if** (msg.side == Side.***LEFT***) {

end = start + count - 1;

} **else** {

start = start + count;

}

**break**;

**case** ***REQUEST***:

send a[start] as pivot to Initiator;

start++;

**break**;

}

}

**while** (!initiatorSocket.isClosed()) {

inFromInitiator.readObject();

outToInitiator.writeObject(new Message(false));

}

partition(int[] a, int pivot, int start, int end) {

int i = start, j = end;

int temp = a[i];

while (i < j) {

while (i < j && pivot <= a[j]) {

j--;

}

a[i] = a[j];

while (i < j && pivot >= a[i]) {

i++;

}

a[j] = a[i];

}

a[i] = temp;

if (a[i] < pivot)

return i - start + 1;

else

return i - start;

}

**Code:**

**package** Project\_2;

**import** java.io.\*;

**import** java.net.\*;

**import** java.util.Arrays;

**import** java.util.Scanner;

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**public** **class** Initiator {

**int** nodePort1;

**int** nodePort2;

ExecutorService pool;

**public** Initiator(**int** nodePort1, **int** nodePort2) **throws** Exception {

**super**();

**this**.nodePort1 = nodePort1;

**this**.nodePort2 = nodePort2;

**this**.pool = Executors.*newFixedThreadPool*(4);

}

ObjectInputStream inFromNode1;

ObjectOutputStream outToNode1;

ObjectInputStream inFromNode2;

ObjectOutputStream outToNode2;

**public** **static** **void** main(String[] args) **throws** Exception {

// **TODO** Auto-generated method stub

**new** Initiator(10000, 10001).start();

}

**public** **void** start() **throws** Exception {

// **TODO** Auto-generated method stub

**int**[] a = **new** **int**[] { 38, 6, 23, 2, 62, 4, 72, 7, 83, 5, 23, 6, 24, 2 };

// int[] a = NumberGenerator.randomArray(10000000, 10000000);

// int[] a = NumberGenerator.reverseOrderedArray(10000, 100);

System.***out***.println(Arrays.*toString*(a));

**int**[] a1 = **new** **int**[a.length / 2];

**int**[] a2 = **new** **int**[a.length - a1.length - 1];

System.*arraycopy*(a, 1, a1, 0, a1.length);

System.*arraycopy*(a, a1.length + 1, a2, 0, a2.length);

Socket node1Socket = **new** Socket("localhost", nodePort1);

outToNode1 = **new** ObjectOutputStream(node1Socket.getOutputStream());

inFromNode1 = **new** ObjectInputStream(node1Socket.getInputStream());

Socket node2Socket = **new** Socket("localhost", nodePort2);

outToNode2 = **new** ObjectOutputStream(node2Socket.getOutputStream());

inFromNode2 = **new** ObjectInputStream(node2Socket.getInputStream());

Future<?> array1Send = pool.submit(**new** SocketSend(outToNode1, a1));

Future<?> array2Send = pool.submit(**new** SocketSend(outToNode2, a2));

Scanner input = **new** Scanner(System.***in***);

System.***out***.print("Input k: ");

**int** k = input.nextInt();

array1Send.get();

array2Send.get();

**int** kValue = findk(a, k, 0, a.length - 1);

System.***out***.println("The value of the k-th number is " + kValue);

node1Socket.close();

node2Socket.close();

input.close();

pool.shutdown();

}

**public** **int** findk(**int**[] a, **int** k, **int** start, **int** end) **throws** Exception {

**int** kIndex = k - 1;

**int** pivot = a[0];

**while** (**true**) {

pool.submit(**new** SocketSend(outToNode1, **new** Message(Sign.***PARTITION***, pivot)));

pool.submit(**new** SocketSend(outToNode2, **new** Message(Sign.***PARTITION***, pivot)));

Future<Message> msg1Future = pool.submit(**new** SocketReceive(inFromNode1));

Future<Message> msg2Future = pool.submit(**new** SocketReceive(inFromNode2));

**int** pivotIndex = msg1Future.get().count + msg2Future.get().count;

Future<?> node1ChooseSide;

Future<?> node2ChooseSide;

a

node1ChooseSide.get();

pool.submit(**new** SocketSend(outToNode1, **new** Message(Sign.***REQUEST***)));

Message msg = pool.submit(**new** SocketReceive(inFromNode1)).get();

**if** (msg.getPivot) {

pivot = msg.pivot;

} **else** {

node2ChooseSide.get();

pool.submit(**new** SocketSend(outToNode2, **new** Message(Sign.***REQUEST***)));

pivot = pool.submit(**new** SocketReceive(inFromNode2)).get().pivot;

}

}

**return** pivot;

}

**public** **class** SocketReceive **implements** Callable<Message> {

ObjectInputStream input;

**public** SocketReceive(ObjectInputStream input) {

**super**();

**this**.input = input;

}

@Override

**public** Message call() **throws** Exception {

// **TODO** Auto-generated method stub

**return** (Message) input.readObject();

}

}

**public** **class** SocketSend **implements** Runnable {

ObjectOutputStream output;

Object data;

**public** SocketSend(ObjectOutputStream output, Object data) {

**super**();

**this**.output = output;

**this**.data = data;

}

@Override

**public** **void** run() {

// **TODO** Auto-generated method stub

**try** {

output.writeObject(data);

} **catch** (IOException e) {

// **TODO** Auto-generated catch block

e.printStackTrace();

}

}

}

}

**package** Project\_2;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**import** java.util.Arrays;

**public** **class** Node **implements** Runnable {

**int** port;

**public** Node(**int** port) {

**super**();

**this**.port = port;

}

**public** **static** **void** main(String[] args) **throws** Exception {

**new** Thread(**new** Node(10000)).start();

**new** Thread(**new** Node(10001)).start();

}

ObjectInputStream inFromInitiator;

ObjectOutputStream outToInitiator;

ServerSocket serverSocket;

**public** **void** run() {

// **TODO** Auto-generated method stub

**try** {

serverSocket = **new** ServerSocket(port);

Socket initiatorSocket = serverSocket.accept();

inFromInitiator = **new** ObjectInputStream(initiatorSocket.getInputStream());

outToInitiator = **new** ObjectOutputStream(initiatorSocket.getOutputStream());

**int**[] a = (**int**[]) inFromInitiator.readObject();

**int** start = 0;

**int** end = a.length - 1;

**int** pivot, count = 0;

**while** (end >= start) {

Message msg = (Message) inFromInitiator.readObject();

**switch** (msg.sign) {

**case** ***PARTITION***:

pivot = msg.pivot;

count = *partition*(a, pivot, start, end);

outToInitiator.writeObject(**new** Message(count));

**break**;

**case** ***CHOOSE***:

**if** (msg.side == Side.***LEFT***) {

end = start + count - 1;

} **else** {

start = start + count;

}

**break**;

**case** ***REQUEST***:

outToInitiator.writeObject(**new** Message(**true**, a[start]));

start++;

**break**;

}

}

**while** (!initiatorSocket.isClosed()) {

inFromInitiator.readObject();

outToInitiator.writeObject(**new** Message(**false**));

}

serverSocket.close();

} **catch** (Exception e) {

}

}

**public** **static** **int** partition(**int**[] a, **int** pivot, **int** start, **int** end) {

**int** i = start, j = end;

**int** temp = a[i];

**while** (i < j) {

**while** (i < j && pivot <= a[j]) {

j--;

}

a[i] = a[j];

**while** (i < j && pivot >= a[i]) {

i++;

}

a[j] = a[i];

}

a[i] = temp;

**if** (a[i] < pivot)

**return** i - start + 1;

**else**

**return** i - start;

}

}

**public** **int** parallelPartition(**int**[] a, **int** pivot, **int** start, **int** end) **throws** Exception {

**int** mid = (start + end) / 2;

**int** quarter = (start + mid) / 2;

**int** threeQuarter = (mid + end) / 2;

**int**[] temp = **new** **int**[end - start + 1];

Future<Integer> future1 = pool.submit(**new** Partition(a, pivot, start, quarter));

Future<Integer> future2 = pool.submit(**new** Partition(a, pivot, quarter + 1, mid));

Future<Integer> future3 = pool.submit(**new** Partition(a, pivot, mid + 1, threeQuarter));

Future<Integer> future4 = pool.submit(**new** Partition(a, pivot, threeQuarter + 1, end));

**int** result1 = future1.get();

**int** result2 = future2.get();

**int** result3 = future3.get();

**int** result4 = future4.get();

**int** sum = result1 + result2 + result3 + result4;

ArrayList<Callable<Boolean>> tasks = **new** ArrayList<Callable<Boolean>>();

tasks.add(**new** ArrayCopy(a, start, temp, 0, result1));

tasks.add(**new** ArrayCopy(a, quarter + 1, temp, result1, result2));

tasks.add(**new** ArrayCopy(a, mid + 1, temp, result1 + result2, result3));

tasks.add(**new** ArrayCopy(a, threeQuarter + 1, temp, result1 + result2 + result3, result4));

tasks.add(**new** ArrayCopy(a, start + result1, temp, sum, quarter - start - result1 + 1));

tasks.add(**new** ArrayCopy(a, quarter + result2 + 1, temp, sum + quarter - start - result1 + 1,

mid - quarter - result2));

tasks.add(**new** ArrayCopy(a, mid + result3 + 1, temp, result3 + result4 - start + mid + 1,

threeQuarter - mid - result3));

tasks.add(**new** ArrayCopy(a, threeQuarter + result4 + 1, temp, result4 - start + threeQuarter + 1,

end - threeQuarter - result4));

pool.invokeAll(tasks);

System.*arraycopy*(temp, 0, a, start, temp.length);

**return** sum;

}

**public** **class** Partition **implements** Callable<Integer> {

**private** **int**[] a;

**private** **int** pivot;

**private** **int** start;

**private** **int** end;

**public** Partition(**int**[] a, **int** pivot, **int** start, **int** end) {

**this**.a = a;

**this**.pivot = pivot;

**this**.start = start;

**this**.end = end;

}

**public** Integer call() {

**int** i = start, j = end;

**int** temp = a[i];

**while** (i < j) {

**while** (i < j && pivot <= a[j]) {

j--;

}

a[i] = a[j];

**while** (i < j && pivot >= a[i]) {

i++;

}

a[j] = a[i];

}

a[i] = temp;

**if** (temp < pivot) {

**return** i - start + 1;

} **else** {

**return** i - start;

}

}

}

**public** **class** ArrayCopy **implements** Callable<Boolean> {

**private** **int**[] a;

**private** **int** starta;

**private** **int**[] b;

**private** **int** startb;

**private** **int** length;

**public** ArrayCopy(**int**[] a, **int** starta, **int**[] b, **int** startb, **int** length) {

**this**.a = a;

**this**.starta = starta;

**this**.b = b;

**this**.startb = startb;

**this**.length = length;

}

**public** Boolean call() {

System.*arraycopy*(a, starta, b, startb, length);

**return** **true**;

}

}

**package** Project\_2;

**import** java.io.ObjectInputStream;

**import** java.io.ObjectOutputStream;

**import** java.net.ServerSocket;

**import** java.net.Socket;

**import** java.util.ArrayList;

**import** java.util.concurrent.Callable;

**import** java.util.concurrent.ExecutorService;

**import** java.util.concurrent.Executors;

**import** java.util.concurrent.Future;

**public** **class** ParallelNode **implements** Runnable {

**int** port;

ExecutorService pool;

**public** ParallelNode(**int** port) {

**super**();

**this**.port = port;

pool = Executors.*newFixedThreadPool*(4);

}

**public** **static** **void** main(String[] args) **throws** Exception {

**new** Thread(**new** ParallelNode(10000)).start();

**new** Thread(**new** ParallelNode(10001)).start();

}

ObjectInputStream inFromInitiator;

ObjectOutputStream outToInitiator;

ServerSocket serverSocket;

**public** **void** run() {

// **TODO** Auto-generated method stub

**try** {

serverSocket = **new** ServerSocket(port);

Socket initiatorSocket = serverSocket.accept();

inFromInitiator = **new** ObjectInputStream(initiatorSocket.getInputStream());

outToInitiator = **new** ObjectOutputStream(initiatorSocket.getOutputStream());

**int**[] a = (**int**[]) inFromInitiator.readObject();

// System.out.println(Arrays.toString(a));

**int** start = 0;

**int** end = a.length - 1;

**int** pivot, count = 0;

**while** (end >= start) {

Message msg = (Message) inFromInitiator.readObject();

**switch** (msg.sign) {

**case** ***PARTITION***:

pivot = msg.pivot;

**if** (end - start >= 4) {

count = parallelPartition(a, pivot, start, end);

} **else** {

count = sequentialPartition(a, pivot, start, end);

}

outToInitiator.writeObject(**new** Message(count));

**break**;

**case** ***CHOOSE***:

**if** (msg.side == Side.***LEFT***) {

end = start + count - 1;

} **else** {

start = start + count;

}

**break**;

**case** ***REQUEST***:

outToInitiator.writeObject(**new** Message(**true**, a[start]));

start++;

**break**;

}

}

**while** (!initiatorSocket.isClosed()) {

inFromInitiator.readObject();

outToInitiator.writeObject(**new** Message(**false**));

}

serverSocket.close();

pool.shutdown();

} **catch** (Exception e) {

}

}

**public** **int** sequentialPartition(**int**[] a, **int** pivot, **int** start, **int** end) {

**int** i = start, j = end;

**int** temp = a[i];

**while** (i < j) {

**while** (i < j && pivot <= a[j]) {

j--;

}

a[i] = a[j];

**while** (i < j && pivot >= a[i]) {

i++;

}

a[j] = a[i];

}

a[i] = temp;

**if** (a[i] < pivot)

**return** i - start + 1;

**else**

**return** i - start;

}

**public** **int** parallelPartition(**int**[] a, **int** pivot, **int** start, **int** end) **throws** Exception {

**int** mid = (start + end) / 2;

**int** quarter = (start + mid) / 2;

**int** threeQuarter = (mid + end) / 2;

**int**[] temp = **new** **int**[end - start + 1];

Future<Integer> future1 = pool.submit(**new** Partition(a, pivot, start, quarter));

Future<Integer> future2 = pool.submit(**new** Partition(a, pivot, quarter + 1, mid));

Future<Integer> future3 = pool.submit(**new** Partition(a, pivot, mid + 1, threeQuarter));

Future<Integer> future4 = pool.submit(**new** Partition(a, pivot, threeQuarter + 1, end));

**int** result1 = future1.get();

**int** result2 = future2.get();

**int** result3 = future3.get();

**int** result4 = future4.get();

**int** sum = result1 + result2 + result3 + result4;

ArrayList<Callable<Boolean>> tasks = **new** ArrayList<Callable<Boolean>>();

tasks.add(**new** ArrayCopy(a, start, temp, 0, result1));

tasks.add(**new** ArrayCopy(a, quarter + 1, temp, result1, result2));

tasks.add(**new** ArrayCopy(a, mid + 1, temp, result1 + result2, result3));

tasks.add(**new** ArrayCopy(a, threeQuarter + 1, temp, result1 + result2 + result3, result4));

tasks.add(**new** ArrayCopy(a, start + result1, temp, sum, quarter - start - result1 + 1));

tasks.add(**new** ArrayCopy(a, quarter + result2 + 1, temp, sum + quarter - start - result1 + 1,

mid - quarter - result2));

tasks.add(**new** ArrayCopy(a, mid + result3 + 1, temp, result3 + result4 - start + mid + 1,

threeQuarter - mid - result3));

tasks.add(**new** ArrayCopy(a, threeQuarter + result4 + 1, temp, result4 - start + threeQuarter + 1,

end - threeQuarter - result4));

pool.invokeAll(tasks);

System.*arraycopy*(temp, 0, a, start, temp.length);

**return** sum;

}

**public** **class** Partition **implements** Callable<Integer> {

**private** **int**[] a;

**private** **int** pivot;

**private** **int** start;

**private** **int** end;

**public** Partition(**int**[] a, **int** pivot, **int** start, **int** end) {

**this**.a = a;

**this**.pivot = pivot;

**this**.start = start;

**this**.end = end;

}

**public** Integer call() {

**int** i = start, j = end;

**int** temp = a[i];

**while** (i < j) {

**while** (i < j && pivot <= a[j]) {

j--;

}

a[i] = a[j];

**while** (i < j && pivot >= a[i]) {

i++;

}

a[j] = a[i];

}

a[i] = temp;

**if** (temp < pivot) {

**return** i - start + 1;

} **else** {

**return** i - start;

}

}

}

**public** **class** ArrayCopy **implements** Callable<Boolean> {

**private** **int**[] a;

**private** **int** starta;

**private** **int**[] b;

**private** **int** startb;

**private** **int** length;

**public** ArrayCopy(**int**[] a, **int** starta, **int**[] b, **int** startb, **int** length) {

**this**.a = a;

**this**.starta = starta;

**this**.b = b;

**this**.startb = startb;

**this**.length = length;

}

**public** Boolean call() {

System.*arraycopy*(a, starta, b, startb, length);

**return** **true**;

}

}

}