题目描述

火车站附近的货物中转站负责将到站货物运往仓库，小明在中转站负责调度2K辆中转车（K辆干货中转车，K辆湿货中转车）。

货物由不同供货商从各地发来，各地的货物是依次进站，然后小明按照卸货顺序依次装货到中转车，一个供货商的货只能装到一辆车上，不能拆装，但是一辆车可以装多家供货商的货；

中转车的限载货物量由小明统一指定，在完成货物中转的前提下，请问中转车的统一限载货物数最小值为多少。

输入描述

- 第一行 length 表示供货商数量 1 <= length <= 10^4
- 第二行 goods 表示供货数数组 1 <= goods[i] <= 10^4
- 第三行 types表示对应货物类型，types[i]等于0或者1，其中0代表干货，1代表湿货
- 第四行 k表示单类中转车数量 1 <= k <= goods.length

输出描述

运行结果输出一个整数，表示中转车统一限载货物数

备注

中转车最多跑一趟仓库

用例

输入	4 3 2 6 3 0 1 1 0 2
输出	6
说明	货物1和货物4为干货，由2辆干货中转车中转，每辆车运输一个货物，限载为3 货物2和货物3为湿货，由2辆湿货中转车中转，每辆车运输一个货物，限载为6 这样中转车统一限载货物数可以设置为6（干货车和湿货车限载最大值），是最小的取值

输入	4 3 2 6 8 0 1 1 1 1
输出	16
说明	货物1为干货，由1辆干货中转车中转，限载为3 货物2、货物3、货物4为湿货，由1辆湿货中转车中转，限载为16 这样中转车统一限载货物数可以设置为16（干货车和湿货车限载最大值），是最小的取值

题目解析

~~根据用例意思，干货只能由干货车运输，湿货只能由湿货车运输，而干、湿货车都各有k辆。~~

~~如果 某类货数量 <= 某类车数量，那么该类车的最大限载就是该类货物中最重的。~~

~~如果 某类货数量 > 某类车数量，那么说明，无法为 每个供货商的货 都安排一辆车，必然由多个供货商的货，被安排到一辆车上。~~

~~此时，要实现：统一最小限载，则只能将比较轻的两个供货商的货物放到一辆车上，这样才能实现最小限载，比如有如下供货商干货：~~

~~10 20 30 40~~

~~但是只有三辆干货车，那么如何安放货物才能实现最小限载呢？~~

~~肯定是将最轻的两个货10、20放到一辆车上，此时相当于每辆干货车承载货物重量为：30、30、40，此时最小~~

题目解析

根据用例意思，干货只能由干货车运输，湿货只能由湿货车运输，而干、湿货车都各有k辆。

如果 某类货数量 <= 某类车数量，那么该类车的最大限载就是该类货物中最重的。

如果 某类货数量 > 某类车数量，那么说明，无法为 每个供货商的货 都安排一辆车，必然由多个供货商的货，被安排到一辆车上。

此时，要实现：统一最小限载，则只能将比较轻的两个供货商的货物放到一辆车上，这样才能实现最小限载，比如有如下供货商干货：

10 20 30 40

但是只有三辆干货车，那么如何安放货物才能实现最小限载呢？

肯定是将最轻的两个货10，20放到一辆车上，此时相当于每辆干货车承载货物重量为：30，30，40，此时最小限载为40

如果将10和30放在一辆车上，或者将10和40放在一辆车上，都可能造成最小限载超过40。

我的解题思路是：

定义一个干货数组dry，和一个湿货数组wet，然后将输入的goods根据types将干货存入dry，将湿货存入wet。

如果dry的数量 <= k，则不需要将多个货物放到一辆车上

如果dry的数量 > k，那么需要将多个货物放到一辆车上，此时根据贪心思维，我们应该将最轻的连两个货物放到一辆车上。

那么如何找出最轻的两个货物呢？

我这里直接先将dry降序排序了，那么dry尾部两个货物就是最轻的，因此直接将dry尾部两个货物弹出，然后相加，得到一个组合货物重量。

此时需要将组合重量再次加入dry，但是为了保证dry的有序，这里我采用二分查找O(logN)后插入，这要比重新排序O(n)性能更好。

然后再看dry的数量 是否 > k，如果还大于，则继续上面逻辑，直到dry数量 <= k。

对于wet的处理同dry。

2023.02.04 根据网友指正，上面的解题逻辑没有考虑一个重要的点：

> 货物由不同供货商从各地发来，各地的货物是依次进站，然后小明按照卸货顺序依次装货到中转车，一个供货商的货只能装到一辆车上，不能拆装，但是一辆车可以装多家供货商的货；

这意味着，我们不难改变输入的顺序，要按照输入的顺序去装车。并且干货只能装到干货车，湿货只能装到湿货车。比如用例1中：

```
4
3 2 6 3
0 1 1 0
2
```

第1个货物是：干货，因此装到干货车。

第2个货物是：湿货，由于湿货不难装到干货车，因此第一辆车只装一个货物就必须走。第二个货物需要装到一辆湿货车上。

第3个货物是：湿货，前一步的湿货车还没走，此时我们有两种选择：

- 将第3个货物继续装到前面的湿货车上，即和第2个货物装一起，但是会导致最低限载被提高
- 将第3个货物装到下一辆湿货车上

第4个货物是：干货，因此只能重新要一辆干货车，继续装载。

说到这里，大家可能已经感觉出来，连续的0（干货）可以用一辆车运，也可以分成多辆车运。而一旦连续0（干货）被中断，即出现了1（湿货），则前面的干货车必须走，这个湿货需要申请一辆湿货车来运。

根据上面逻辑，我们可以将连续干货 和 连续湿货 对应的区间段统计出来：

连续干货：[3]、[3]

连续湿货：[2, 6]

- 如果连续干货的区间段数 刚好等于 k，则刚好每段连续干货，都可以分配一辆干货车。
- 如果连续干货的区间段数 < k，则出了必要的给每段分配一辆干货车外，还有多余的干货车可以继续分配：此时，我们应该将多段连续干货，按照和大小降序，优先给 和最大 的连续干货段多分配一辆车，即相当于将一个区间一分为二，这样最低限载就降低了，并且多出的干货车数量减1。如果多余的干货车数量不等于0，则继续按上面逻辑，分配一辆车给 和最大 的连续干货段。直到多余的干货车用完。
- 如果连续干货的区间段数 > k，此时是无法完成运输任务的，应该视为异常情况，但是本题没有说明异常处理，因此可以认为此场景不存在。

上面逻辑中，每次将区间一份为二后，都要重新按照 区间和大小降序，这里可以用 优先队列 。

JavaScript算法源码

```
/* JavaScript Node ACM模式 控制台输入获取 */
const readline = require("readline");
```

JavaScript算法源码

```javascript
/* JavaScript Node ACM模式 控制台输入获取 */
const readline = require("readline");

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout,
});

const lines = [];
rl.on("line", (line) => {
  lines.push(line);

  if (lines.length === 4) {
    const n = lines[0] - 0;
    const goods = lines[1].split(" ").map(Number);
    const types = lines[2].split(" ").map(Number);
    const k = lines[3] - 0;

    console.log(getResult(n, goods, types, k));
    lines.length = 0;
  }
});

function getResult(n, goods, types, k) {
  const drys = new PriorityQueue((a, b) => b[0] - a[0]);
  const wets = new PriorityQueue((a, b) => b[0] - a[0]);

  let isDry = true;
  let tmp = [];
  let sum = 0;

  for (let i = 0; i < n; i++) {
    if (types[i] == 0) {
      if (isDry) {
        tmp.push(goods[i]);
        sum += goods[i];
      } else {
        tmp.unshift(sum);
        wets.offer(tmp);

        tmp = [];
        sum = 0;

        tmp.push(goods[i]);
        sum += goods[i];

        isDry = true;
      }
    } else {
      if (isDry) {
        tmp.unshift(sum);
        drys.offer(tmp);

        tmp = [];
        sum = 0;

        tmp.push(goods[i]);
        sum += goods[i];

        isDry = false;
      } else {
        tmp.push(goods[i]);
        sum += goods[i];
      }
    }
  }

  tmp.unshift(sum);
  if (isDry) {
    drys.offer(tmp);
  } else {
    wets.offer(tmp);
  }

  return Math.max(divide(drys, k), divide(wets, k));
}

function divide(goods, k) {
  while (goods.size() < k) {
    const top = goods.poll();

    if (!top) break;

    const dry_sum = top.shift();

    if (top.length <= 1) {
      return dry_sum;
    }

    let splitIdx = 0;
    let splitHalf = 0;
    let half = 0;
```

```
    drys.offer(tmp);
  } else {
    wets.offer(tmp);
  }

  return Math.max(divide(drys, k), divide(wets, k));
}

function divide(goods, k) {
  while (goods.size() < k) {
    const top = goods.poll();

    if (!top) break;

    const dry_sum = top.shift();

    if (top.length <= 1) {
      return dry_sum;
    }

    let splitIdx = 0;
    let splitHalf = 0;
    let half = 0;
    let min = dry_sum;

    for (let i = 1; i < top.length; i++) {
      const val = top[i - 1];
      half += val;

      const max = Math.max(half, dry_sum - half);
      if (max < min) {
        min = max;
        splitIdx = i;
        splitHalf = half;
      }
    }

    const halfDry = [splitHalf, ...top.slice(0, splitIdx)];
    goods.offer(halfDry);

    const otherhalfDry = [dry_sum - splitHalf, ...top.slice(splitIdx)];
    goods.offer(otherhalfDry);
  }

  return goods.peek()[0];
}

// 基于堆实现优先队列
class PriorityQueue {
  constructor(cpr) {
    this.queue = [];
    this.cpr = cpr;
  }

  swap(a, b) {
    const tmp = this.queue[a];
    this.queue[a] = this.queue[b];
    this.queue[b] = tmp;
  }

  // 上浮
  swim() {
    let c = this.queue.length - 1;

    while (c >= 1) {
      const f = Math.floor((c - 1) / 2);

      if (this.cpr(this.queue[c], this.queue[f]) < 0) {
        this.swap(c, f);
        c = f;
      } else {
        break;
      }
    }
  }

  // 入队
  offer(val) {
    this.queue.push(val);
    this.swim();
  }

  // 下沉
  sink() {
    let f = 0;

    while (true) {
      let c1 = 2 * f + 1;
      let c2 = c1 + 1;

      let c;
      let val1 = this.queue[c1];
      let val2 = this.queue[c2];
      if (val1 && val2) {
        c = this.cpr(val1, val2) < 0 ? c1 : c2;
```

```
    let c = this.queue.length - 1;

    while (c >= 1) {
      const f = Math.floor((c - 1) / 2);

      if (this.cpr(this.queue[c], this.queue[f]) < 0) {
        this.swap(c, f);
        c = f;
      } else {
        break;
      }
    }
  }

  // 入队
  offer(val) {
    this.queue.push(val);
    this.swim();
  }

  // 下沉
  sink() {
    let f = 0;

    while (true) {
      let c1 = 2 * f + 1;
      let c2 = c1 + 1;

      let c;
      let val1 = this.queue[c1];
      let val2 = this.queue[c2];
      if (val1 && val2) {
        c = this.cpr(val1, val2) < 0 ? c1 : c2;
      } else if (val1 && !val2) {
        c = c1;
      } else if (!val1 && val2) {
        c = c2;
      } else {
        break;
      }

      if (this.cpr(this.queue[c], this.queue[f]) < 0) {
        this.swap(c, f);
        f = c;
      } else {
        break;
      }
    }
  }

  // 出队
  poll() {
    this.swap(0, this.queue.length - 1);
    const res = this.queue.pop();
    this.sink();
    return res;
  }

  peek() {
    return this.queue[0];
  }

  size() {
    return this.queue.length;
  }
}
```

Java算法源码

```
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int n = sc.nextInt();

    int[] goods = new int[n];
    for (int i = 0; i < n; i++) {
      goods[i] = sc.nextInt();
    }

    int[] types = new int[n];
    for (int i = 0; i < n; i++) {
      types[i] = sc.nextInt();
    }

    int k = sc.nextInt();

    System.out.println(getResult(n, goods, types, k));
  }
```

Java算法源码

```java
import java.util.LinkedList;
import java.util.PriorityQueue;
import java.util.Scanner;

public class Main {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    int n = sc.nextInt();

    int[] goods = new int[n];
    for (int i = 0; i < n; i++) {
      goods[i] = sc.nextInt();
    }

    int[] types = new int[n];
    for (int i = 0; i < n; i++) {
      types[i] = sc.nextInt();
    }

    int k = sc.nextInt();

    System.out.println(getResult(n, goods, types, k));
  }

  public static int getResult(int n, int[] goods, int[] types, int k) {
    PriorityQueue<LinkedList<Integer>> drys = new PriorityQueue<>((a, b) -> b.get(0) - a.get(0));
    PriorityQueue<LinkedList<Integer>> wets = new PriorityQueue<>((a, b) -> b.get(0) - a.get(0));

    boolean isDry = true;
    LinkedList<Integer> tmp = new LinkedList<>();
    int sum = 0;

    for (int i = 0; i < n; i++) {
      if (types[i] == 0) {
        if (isDry) {
          tmp.add(goods[i]);
          sum += goods[i];
        } else {
          tmp.addFirst(sum);
          wets.offer(tmp);

          tmp = new LinkedList<>();
          sum = 0;

          tmp.add(goods[i]);
          sum += goods[i];

          isDry = true;
        }
      } else {
        if (isDry) {
          tmp.addFirst(sum);
          drys.offer(tmp);

          tmp = new LinkedList<>();
          sum = 0;

          tmp.add(goods[i]);
          sum += goods[i];

          isDry = false;
        } else {
          tmp.add(goods[i]);
          sum += goods[i];
        }
      }
    }

    tmp.addFirst(sum);
    if (isDry) {
      drys.offer(tmp);
    } else {
      wets.offer(tmp);
    }

    int dry_min = divide(drys, k);
    int wet_min = divide(wets, k);

    return Math.max(dry_min, wet_min);
  }

  public static int divide(PriorityQueue<LinkedList<Integer>> goods, int k) {
    while (goods.size() < k) {
      LinkedList<Integer> top = goods.poll();

      if (top == null) break;

      int dry_sum = top.removeFirst();

      if (top.size() == 1) {
```

```
          sum = 0;

          tmp.add(goods[i]);
          sum += goods[i];

          isDry = false;
        } else {
          tmp.add(goods[i]);
          sum += goods[i];
        }
      }
    }

    tmp.addFirst(sum);
    if (isDry) {
      drys.offer(tmp);
    } else {
      wets.offer(tmp);
    }

    int dry_min = divide(drys, k);
    int wet_min = divide(wets, k);

    return Math.max(dry_min, wet_min);
  }

  public static int divide(PriorityQueue<LinkedList<Integer>> goods, int k) {
    while (goods.size() < k) {
      LinkedList<Integer> top = goods.poll();

      if (top == null) break;

      int dry_sum = top.removeFirst();

      if (top.size() == 1) {
        return dry_sum;
      }

      int splitIdx = 0;
      int splitHalf = 0;
      int half = 0;
      int min = dry_sum;

      for (int i = 1; i < top.size(); i++) {
        int val = top.get(i - 1);
        half += val;

        int max = Math.max(half, dry_sum - half);
        if (max < min) {
          min = max;
          splitIdx = i;
          splitHalf = half;
        }
      }

      LinkedList<Integer> halfList = new LinkedList<>();
      halfList.add(splitHalf);
      halfList.addAll(top.subList(0, splitIdx));
      goods.offer(halfList);

      LinkedList<Integer> otherHalfList = new LinkedList<>();
      otherHalfList.add(dry_sum - splitHalf);
      otherHalfList.addAll(top.subList(splitIdx, top.size()));
      goods.offer(otherHalfList);
    }

    return goods.peek().get(0);
  }
}
```

Python算法源码

```
import queue

# 输入获取
n = int(input())
goods = list(map(int, input().split()))
types = list(map(int, input().split()))
k = int(input())

class Obj:
    def __init__(self, sumV, goods):
        self.sumV = sumV
        self.goods = goods

    def __lt__(self, other):
        return self.sumV > other.sumV

# 算法入口
def getResult(n, goods, types, k):
    drys = queue.PriorityQueue()
    wets = queue.PriorityQueue()

```

Python算法源码

```python
import queue

# 输入获取
n = int(input())
goods = list(map(int, input().split()))
types = list(map(int, input().split()))
k = int(input())

class Obj:
    def __init__(self, sumV, goods):
        self.sumV = sumV
        self.goods = goods

    def __lt__(self, other):
        return self.sumV > other.sumV

# 算法入口
def getResult(n, goods, types, k):
    drys = queue.PriorityQueue()
    wets = queue.PriorityQueue()

    isDry = True
    tmp = []
    sumV = 0

    for i in range(n):
        if types[i] == 0:
            if isDry:
                tmp.append(goods[i])
                sumV += goods[i]
            else:
                wets.put(Obj(sumV, tmp))

                tmp = []
                sumV = 0

                tmp.append(goods[i])
                sumV += goods[i]

                isDry = True
        else:
            if isDry:
                drys.put(Obj(sumV, tmp))

                tmp = []
                sumV = 0

                tmp.append(goods[i])
                sumV += goods[i]

                isDry = False
            else:
                tmp.append(goods[i])
                sumV += goods[i]

    if isDry:
        drys.put(Obj(sumV, tmp))
    else:
        wets.put(Obj(sumV, tmp))

    return max(divide(drys, k), divide(wets, k))


def divide(goods, k):
    while goods.qsize() < k:
        top = goods.get()

        if top is None:
            break

        dry_sum = top.sumV

        if len(top.goods) <= 1:
            return dry_sum

        splitIdx = 0
        splitHalf = 0
        half = 0
        minV = dry_sum

        for i in range(1, len(top.goods)):
            val = top.goods[i - 1]
            half += val

            maxV = max(half, dry_sum - half)
            if maxV < minV:
                minV = maxV
                splitIdx = i
                splitHalf = half

        halfDry = Obj(splitHalf, top.goods[:splitIdx])
        goods.put(halfDry)
```

```
                drys.put(Obj(sumV, tmp))

                tmp = []
                sumV = 0

                tmp.append(goods[i])
                sumV += goods[i]

                isDry = False
            else:
                tmp.append(goods[i])
                sumV += goods[i]

    if isDry:
        drys.put(Obj(sumV, tmp))
    else:
        wets.put(Obj(sumV, tmp))

    return max(divide(drys, k), divide(wets, k))


def divide(goods, k):
    while goods.qsize() < k:
        top = goods.get()

        if top is None:
            break

        dry_sum = top.sumV

        if len(top.goods) <= 1:
            return dry_sum

        splitIdx = 0
        splitHalf = 0
        half = 0
        minV = dry_sum

        for i in range(1, len(top.goods)):
            val = top.goods[i - 1]
            half += val

            maxV = max(half, dry_sum - half)
            if maxV < minV:
                minV = maxV
                splitIdx = i
                splitHalf = half

        halfDry = Obj(splitHalf, top.goods[:splitIdx])
        goods.put(halfDry)

        otherHalfDry = Obj(dry_sum - splitHalf, top.goods[splitIdx:])
        goods.put(otherHalfDry)

    return goods.queue[0].sumV


# 算法调用
print(getResult(n, goods, types, k))
```