题目描述

静态扫描可以快速识别源代码的缺陷，静态扫描的结果以扫描报告作为输出：

1、文件扫描的成本和文件大小相关，如果文件大小为N，则扫描成本为N个 金币

2、扫描报告的缓存成本和文件大小无关，每缓存一个报告需要M个金币

3、扫描报告缓存后，后继再碰到该文件则不需要扫描成本，直接获取缓存结果

给出源代码文件标识序列和文件大小序列，求解采用合理的缓存策略，最少需要的金币数。

输入描述

第一行为缓存一个报告金币数M，L<= M <= 100

第二行为文件标识序列：F1,F2,F3,....,Fn。

第三行为文件大小序列：S1,S2,S3,....,Sn。

备注：

- 1 <= N <= 10000
- 1 <= Fi <= 1000
- 1 <= Si <= 10

输出描述

采用合理的缓存策略，需要的最少金币数

用例

输入	5 1 2 2 1 2 3 4 1 1 1 1 1 1 1
输出	7
说明	文件大小相同，扫描成本均为1个金币。缓存任意文件均不合算，因而最少成本为7金币。

输入	5 2 2 2 2 2 5 2 2 2 3 3 3 3 3 1 3 3 3
输出	9
说明	无

题目解析

简单的贪心思维 逻辑题，解析请看代码注释。

JavaScript算法源码

```
/* 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 = [];
let m, f, s;
rl.on("line", (line) => {
  lines.push(line);

  if (lines.length === 3) {
    m = lines[0] - 0;
    f = lines[1].split(" ").map(Number);
    s = lines[2].split(" ").map(Number);
```

题目解析

简单的贪心思维 逻辑题，解析请看代码注释。

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 = [];
let m, f, s;
rl.on("line", (line) => {
  lines.push(line);

  if (lines.length === 3) {
    m = lines[0] - 0;
    f = lines[1].split(" ").map(Number);
    s = lines[2].split(" ").map(Number);

    console.log(getResult(m, f, s));

    lines.length = 0;
  }
});

function getResult(m, f, s) {
  // count用于保存每个文件出现的次数
  const count = {};
  // size用于保存文件的大小，即扫描成本
  const size = {};
  for (let i = 0; i < f.length; i++) {
    // k是文件标识
    const k = f[i];
    count[k] ? count[k]++ : (count[k] = 1);
    if (!size[k]) {
      size[k] = s[i];
    }
  }

  let ans = 0;
  for (let k in count) {
    // 选择每次都重新扫描的成本  和  扫描一次+缓存的成本  中最小的
    ans += Math.min(count[k] * size[k], size[k] + m);
  }

  return ans;
}
```

Java算法源码

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.Scanner;

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

    int m = Integer.parseInt(sc.nextLine());
    Integer[] f =
        Arrays.stream(sc.nextLine().split(" ")).map(Integer::parseInt).toArray(Integer[]::new);
    Integer[] s =
        Arrays.stream(sc.nextLine().split(" ")).map(Integer::parseInt).toArray(Integer[]::new);

    System.out.println(getResult(m, f, s));
  }

  public static int getResult(int m, Integer[] f, Integer[] s) {
    // count用于保存每个文件出现的次数
    HashMap<Integer, Integer> count = new HashMap<>();
    // size用于保存文件的大小，即扫描成本
    HashMap<Integer, Integer> size = new HashMap<>();

    for (int i = 0; i < f.length; i++) {
      // k是文件标识
      Integer k = f[i];
      count.put(k, count.getOrDefault(k, 0) + 1);
      size.putIfAbsent(k, s[i]);
    }

    int ans = 0;
    for (Integer k : count.keySet()) {
      // 选择每次都重新扫描的成本  和  扫描一次+缓存的成本  中最小的
      ans += Math.min(count.get(k) * size.get(k), size.get(k) + m);
    }

    return ans;
```

Java算法源码

```java
import java.util.Arrays;
import java.util.HashMap;
import java.util.Scanner;

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

    int m = Integer.parseInt(sc.nextLine());
    Integer[] f =
        Arrays.stream(sc.nextLine().split(" ")).map(Integer::parseInt).toArray(Integer[]::new);
    Integer[] s =
        Arrays.stream(sc.nextLine().split(" ")).map(Integer::parseInt).toArray(Integer[]::new);

    System.out.println(getResult(m, f, s));
  }

  public static int getResult(int m, Integer[] f, Integer[] s) {
    // count用于保存每个文件出现的次数
    HashMap<Integer, Integer> count = new HashMap<>();
    // size用于保存文件的大小，即扫描成本
    HashMap<Integer, Integer> size = new HashMap<>();

    for (int i = 0; i < f.length; i++) {
      // k是文件标识
      Integer k = f[i];
      count.put(k, count.getOrDefault(k, 0) + 1);
      size.putIfAbsent(k, s[i]);
    }

    int ans = 0;
    for (Integer k : count.keySet()) {
      // 选择每次都重新扫描的成本  和  扫描一次+缓存的成本  中最小的
      ans += Math.min(count.get(k) * size.get(k), size.get(k) + m);
    }

    return ans;
  }
}
```

Python算法源码

```python
# 输入获取
m = int(input())
f = list(map(int, input().split()))
s = list(map(int, input().split()))


# 算法入口
def getResult(m, f, s):
    # count用于保存每个文件出现的次数
    count = {}
    # size用于保存文件的大小，即扫描成本
    size = {}

    for i in range(len(f)):
        # k是文件标识
        k = f[i]
        if count.get(k) is None:
            count[k] = 1
        else:
            count[k] += 1

        if size.get(k) is None:
            size[k] = s[i]

    ans = 0
    for k in count.keys():
        # 选择每次都重新扫描的成本  和  扫描一次+缓存的成本  中最小的
        ans += min(count[k] * size[k], size[k] + m)
    return ans


print(getResult(m, f, s))
```