Load Balancing

Clearly it is possible to rebalance the jobs if and only if total number of jobs currently assigned to processors is divisible by number of processors **N**. Now let's suppose that it is (otherwise the solution is **-1**). Let **d** denote the number of jobs to be assigned to each processor after rebalancing and $\mathbf{P_i}$ denote the number of jobs currently assigned to processor **i**.

Let's compute for each processor **i** the values $\mathbf{r_i}$ and $\mathbf{l_i}$ denoting number of jobs to be transferred to left and right neighbour of **i**-th processor respectively. Clearly the solution will be the maximum of values $\mathbf{r_i}$ and $\mathbf{l_i}$ over all $\mathbf{1<=i<=N}$. Let's consider the first processor. If the number of jobs currently assigned to this processor is less than **d** this processor must receive $\mathbf{d-P_1}$ jobs from processor **2**. No other jobs need to be transferred from **2** to **1**(solution transferring more jobs from **2** to **1** cannot be better). So $\mathbf{l_1=r_1=0}$ and $\mathbf{l_2=d-P_1}$. If $\mathbf{P_1>d}$ then similarly $\mathbf{l_1=0,r_1=P_1-d,l_2=0}$. We can compute the values $\mathbf{r_i}$ and $\mathbf{l_{i+1}}$ for other **i** in similar way. If $\mathbf{V_i=P_i-l_i+r_{i-1}<d}$ then $\mathbf{r_i=0}$ and $\mathbf{l_{i+1}=d-V_i}$, if $\mathbf{V_i>d}$ then $\mathbf{r_i=V_i-d}$ and $\mathbf{l_{i+1}=0}$.

Candies

One possible solution to this problem is using Microsoft Excel (or a Unix awk and sort command, which can be even faster).

The idea is that for each type of candy, we only need to consider at most three candidate bags, namely three bags with maximum number of candies of that type.

Thus I found a solution to both inputs like this:

1. Import the input file into Excel; you get a nice three column table.
2. Attach bag numbers to each record, for example using a macro like this:

```
Sub my_simple_macro()
    For i = 1 To 500 Step 1
      Cells(i, 4).Value = i
    Next i
End Sub
```

3. Sort the rows by the first, second and third column in descending order. Always record the first three lines. For the easy input you will get something like this:

```
Chocola Strawbe Banana  Bag#
10771   7194    4593    388    (Sorted by Chocolate)
10734   10472   9841    180
10732   5926    9957    372
          ....

4712    10749   10748   411    (Sorted by Strawberry)
4531    10745   10352   458
8063    10700   6794    253
          ....

4712    10749   10748   411    (Sorted by Banana)
```

```
7792    2500    10696   59
3313    4136    10682   424
          ....
```

4. Now look at what you've got. Bag 388 contains most Chocolate candies and is not in the first three places for Strawberry nor Banana candies. Thus we can safely assign bag 388 to Chocolate. Bag 411 is the most popular for both Strawberry and Banana. Since the difference between the first and second option (bag 411 vs. 458) for Strawberry is smaller than the difference between the first and second option (411 vs. 59) for Banana, we assign bag 411 to Banana and bag 458 to Strawberry.

In fact, what we've done in the previous step is called minimum-cost (or rather maximum-cost) matching. As you see, small instances like this can be solved pretty efficiently just by common sense.

P.S. Something for those more Unix inclined: the same output as shown above can be achieved by this (rather lengthy) command:

```
awk 'BEGIN{i=0} {if (i>0) printf "%5d %5d %5d %4d\n",\
$1,$2,$3,i; i++}' c1.in | sort -k1 -nr -g | head -3
```

This prints the first three, i.e. Chocolate rows, if you want the Strawberry and Banana rows, you need to change the `-k1` parameter of `sort` to `-k2` and `-k3` respectively.

Gossipers

Since this problem is rather easy a straightforward solution is good enough. So we will just simulate the meetings as they was happening. For each gossip we will traverse through the meetings and count the number of all gossipers that got to know this gossip.

To make the solution a bit faster, we can put gossipers' names into a hash table. That makes us able to put a name into the table and also to find a certain name in the table in **O(L)** time, where **L** is length of the name.

Dependency Problems

This problem was probably the easiest one. One possible solution works as follows. Consider the input as a graph with vertices corresponding to the packages and edges corresponding to the dependencies. An edge leading from **u** to **v** denotes that **u** depends on **v**. Each vertex is colored white or black, white meaning the package is available. The out-degree of a vertex is the number of edges leaving it, e.g. the number of dependencies the corresponding package has. We may install a package iff the corresponding vertex is white and has out-degree 0.

While reading the input, we compute the out-degree and color for each vertex. Also for each vertex **v** we store a list **D(v)** containing all vertices **u** such that there is an edge from **u** to **v** in our graph. (E.g. a list of all packages depending on **v**.) We will simply simulate the installation process. If there is no white vertex with out-degree 0, we are done. Otherwise we pick any such vertex **v**, install the corresponding package and remove **v** from the graph. By

removing **v** the out-degree of vertices in **D(v)** decreases by 1. In this way new installable packages may arise.

We will maintain all currently installable packages in a queue. In each step we pop a package from the queue, install it and decrease the corresponding out-degrees. If we decreased the out-degree of a white vertex to 0, we push it into the queue. The algorithm ends when there are no more packages in the queue.

Both time and space complexity of this algorithm are linear in the size of the given graph. Note that an optimal algorithm would use a hash table to store the names of the packages. However, the input files were small enough and so this approach wasn't necessary.

Folding the Paper

Let **Page[i]** be the number of the page printed at the position **i** and **Pos[i]** the position of page number **i** (also **Pos[Page[i]]=i** and **Page[Pos[i]]=i**). Let **Left[i]** and **Right[i]** be the left and right neighbors of page number **i** (also **Left[i]=Page[Pos[i]-1]**, **Right[i]=Page[Pos[i]+1]**).

How does the folded paper look like (in case it is possible to fold it)? On the top there is the page number **1**, under the page number **1** is the page number **2**, ... and at the bottom is the page number **N**. Each page **i** is connected with the pages **Left[i]** and **Right[i]**. If the printed side of some page **i** is facing up then the connection to **Left[i]** is on the left side and connection to **Right[i]** is on the right side of the resulting column of pages. Moreover, the printed sides of the pages **Left[i]** and **Right[i]** are facing down. If the printed side of the page **i** is facing down, the whole situation is reversed.

The rules stated above determine exactly how should the resulting column of pages look like. E.g. consider the example input from the problem statement. There were 5 pages, printed in the order 3-1-5-4-2. Suppose the printed side of the page number 1 is facing up. From this we may deduce that 1 and 3 have to be connected on the left side, 1 and 5 on the right side, 5 and 4 on the left side and pages 4 and 2 have to be connected on the right side. Clearly it is possible to fold the paper this way if and only if no two connections intersect each other. But how to check this?

Imagine a horizontal sweeping plane moving from the top to the bottom of our column of paper. At each moment the sweeping plane intersects some of the connections, some of them are completely above and some are below the sweeping plane. We will call the intersected connections **open** and all other connections **closed**. Suppose that the sweeping plane is between pages **i-1** and **i**. What happens when it moves below the page **i**? If the page **i** is connected to some page **j** such that **i<j**, a connection from **i** to **j** is opened. If **i>j**, the connection from **j** to **i** is closed. Clearly the connections don't intersect if and only if for each two connections **c** and **d** on the same side (e.g. left or right) the following condition holds: If **c** is opened before **d** then **c** is closed after **d**.

This idea leads to the following algorithm: Suppose that the pages printed at odd positions are facing up. (The other case is symmetric with this one.) We will use two stacks, one for the left side and one for the right one. In these stacks we will store the currently open connections. The most recently opened connection will be at the top of the stack. At the beginning both stacks are empty. We will iterate through all **i** from **1** to **N**, in the **i**-th iteration the sweeping

plane will pass over the page number **i**: We check all (at most two) connections involving page number **i**. (Note that these connections are determined by the values **Pos**, **Page**, **Left** and **Right** defined above.) If a connection is opened, we push it onto the corresponding stack. If a connection is closed, we check whether it is on the top of the corresponding stack. If yes, we pop it from the stack. Otherwise the algorithm terminates and the answer is "NO". If we finish all the iterations without finding two intersecting connections the answer is "YES". Both time and space complexity of this algorithm are **O(N)**.