JFP **16** (2): 129–136, 2006. © 2003 Cambridge University Press
doi:10.1017/S095679680300474X First published online 31 July 2003 Printed in the United Kingdom

FUNCTIONAL PEARL
Marble mingling

S. A. CURTIS[*]
University of Stirling, Stirling FK9 4LA, UK
(*e-mail:* s.curtis@cs.stir.ac.uk)

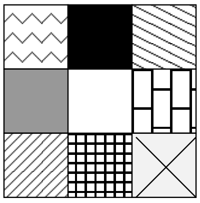
1 Introduction

A bag (literally and mathematically!) of marbles is deemed to be *mingled* if all the colours of the marbles are different. Given a positive integer k and a collection of m marbles, our objective is to try and extract as many mingled bags as possible from the collection, where each bag contains exactly k marbles.

For example, consider a collection of four red, three green, three blue and two yellow marbles. If $k = 3$, then we may select at most four bags: two RGB selections, one RGY selection, and one RBY selection. Alternatively, if $k = 4$, then we may select at most two bags, both with $RGBY$ selections, leaving one green, one blue and two red marbles left over.

A special case of this problem was considered in Curtis (1996) by a programmer wishing to plan usage of drugs packages in a double-blind medical trial: the different types of drugs corresponded to the marble colours, the packages were assigned in pairs because of the double-blind criterion, so that $k = 2$, and to obtain minimum wastage of unused packages, the maximum possible number of marble selections was needed.

More general examples of this problem occur in craft work, where items are constructed from many differently-coloured pieces. For example, a patchwork maker may have a large pile of fabric squares, out of which are to be produced as many "ninepatch blocks" as possible, where the fabrics in each block are all different. In this case, $k = 9$:



In this pearl, we look at this problem in two ways. Firstly, a greedy algorithm is used to produce a listing of as many mingled marble selections as possible. Secondly, attention is restricted to the maximum *number* of selections possible, for which we find an even more efficient algorithm.

[*] Now at: Department of Computing, Oxford Brookes University, Oxford, OX33 1HX. e-mail: sharoncurtis@brookes.ac.uk

2 A greedy solution

Not all marble selection strategies will lead to an optimal solution. For example, with the marble collection from the previous page, and with $k = 3$, choosing two selections of GBY ensures that only one further selection of RGB can be made, leaving three red marbles left over. Only three selections are obtained, not as many as the four given previously.

One intuitive guess at a successful strategy is that which at each step chooses the k most common marbles of those remaining. To put it another way: suppose some kind person has sorted our marble collection into jars, so that all the red marbles are in one jar, all the blue marbles are in another jar, and so on. At each step of the algorithm, k marbles are chosen from the k fullest jars at that time. Using our example marble collection to illustrate, with $k = 3$, the steps followed by this greedy algorithm would be

Marbles Remaining	*Selections*
4R 3G 3B 2Y	
	RGB
3R 2G 2B 2Y	
	RGB
2R 1G 1B 2Y	
	RGY
1R 1B 1Y	
	RBY

This strategy does indeed produce an optimal solution:

Proof
It is sufficient to show that the first greedy step of the algorithm can lead to an optimal solution, and then the same argument may be applied to the remaining steps.

If there are at least k different colours of marbles in the collection to begin with, then the greedy strategy dictates choosing the selection of marbles with colours $\{c_1, \dots c_k\}$, where the colours of marbles have been labelled as follows: c_1 is the most common marble colour, c_2 the most common after c_1, and so on.

Let S be an optimal solution to the problem, consisting of a listing of mingled marble selections, together with any marbles left over. We obtain another optimal solution S_k containing a marble selection with colours $\{c_1, \dots c_k\}$, and to do this, we use induction on i with the following induction hypothesis:

For $1 \leqslant i \leqslant k$, the marbles in S may be rearranged to obtain another optimal solution S_i, which contains a marble selection including colours $c_1 \dots c_i$.

Base case: either a c_1-coloured marble is already included in one of the selections of S, in which case $S_1 = S$, or all the c_1-coloured marbles are left over, one of which may be safely swapped into any mingled marble selection of S to form S_1.

Inductive case: using the induction hypothesis, we rearrange S to form S_i, which includes a selection s containing marbles of colours $c_1 \dots c_i$. If s already contains a c_{i+1}-coloured marble, then $S_{i+1} = S_i$ and we are done. If not, we must find a c_{i+1}-coloured marble elsewhere in S_i to swap with one of the other marbles in s.

If one of the left over marbles in S_i is coloured c_{i+1}, it may be swapped safely with one of the other marbles in s. Otherwise, all the c_{i+1}-coloured marbles must be in other selections of S_i, and we must swap one of them with some c_m-coloured marble (where $m > i+1$) in s.

If it is not possible to perform this swap whilst keeping the selections mingled, then the following must be the case: not only is there a c_m-coloured marble in s, but any selection with a c_{i+1}-coloured marble also contains a c_m-coloured marble, contradicting the fact that c_{i+1}-coloured marbles are at least as common as c_m-coloured marbles. Thus we can choose a selection in S_i containing a c_{i+1}-coloured marble and safely swap it with the c_m-coloured marble in s, to form another optimal solution S_{i+1}. □

2.1 Implementation

We now translate the above algorithm into a functional program to produce a listing of as many marble selections as possible.

A jar of marbles can be represented by the quantity of marbles in the jar, together with their colour:

```
type JarC = (Int, Colour)
```

Thus the program's input has type [*JarC*]. We further make the assumption that the input data is appropriate: that is, no jar contains less than one marble, and no two jars hold marbles of the same colour.

We introduce a sampling function that removes marbles from jars, and also removes any newly-emptied jars from the list of jars:

```
sampleC :: Int → [JarC] → ([Colour] , [JarC])
sampleC i js = (map snd js, concat (map (removeC i) js))

removeC :: Int → JarC → [JarC]
removeC i (n, c) | n == i  =  [ ]
                 | n > i   =  [(n − i, c)]
```

The above is more general than currently needed, as it offers the choice of how many marbles should be removed from each jar: this choice will be useful later on. For now, we use *sampleC* 1 to sample precisely one marble from each jar.

After sorting the initial list of jars into descending order of fullness, the greedy step of taking one marble from each of the k fullest jars of the list *js* can be performed by *sampleC* 1 (*take k* (*sort js*)). It would be inefficient to sort the jars at every step, so the jars are sorted once at the beginning of the algorithm. Then, as a sampling of a sorted list of jars returns a list of jars which is also sorted, merging the list of sampled jars with the unsampled jars safely retains the sorted property of the list of jars at each step. Thus a program to list as many mingled selections as possible is the function *marblesList*:

```
marblesList   :: Int → [JarC] → [[Colour]]
marblesList k  =  mList k  .  sort

mList  :: Int → [ JarC] → [[Colour]]
mList k js  | length js  <  k  =  [ ]
            | otherwise        =  s : mList k (merge js′ (drop k js))
            where (s, js′)     =  sampleC 1 (take k js)
```

This functional program is still inefficient, because it calculates *length js* at each step, but it is possible to improve this by calculating *length js* once at the beginning of the algorithm, and rewriting the *sampleC* function so that it also returns the new list length at each step. With this alteration (omitted above, for clarity), the complexity of the algorithm is $O(j\ log\ j+m)$, for j jars containing m marbles in total. This complexity comes from $O(j\ log\ j)$ for sorting efficiently, and $O(m)$ for *mList*, as a maximum of $\lfloor m/k \rfloor$ selections are possible, each taking $O(k)$ to perform. In cases when the distribution of marbles is not sparse amongst the jars, so that $j \ll m$, the algorithm is of complexity $\Theta(m)$, and this is optimal: simply listing the marble selections must take m steps.

For practical use in real-world problems, marble listings are required, and the above program is efficient at producing these. If instead, we do not care about listings, and turn our interest to solely the maximum *number* of mingled selections that can be obtained, further efficiency improvements are possible.

3 An even greedier solution

We begin with the straightforward translation of the above program into one that simply counts the number of marble selections. First, the colour information is omitted, so that our datatype for jars is now

```
    type Jar  =  Int
```

The sampling may now simply remove the marbles from the jars, as no colour listings are required:

```
sample  ::  Int → [Jar] → [Jar]
sample i  =  concat  .  map (remove i)

remove  :: Int → Jar → [Jar]
remove i n  | n == i  =  [ ]
            | n > i   =  [n − i]
```

Similarly to how *mList* listed the marble selections, we use a function *mCount* to count the selections:

```
marblesCount  :: Int → [Jar] → Int
marblesCount k  =  mCount k  .  sort
```

```
mCount  :: Int → [Jar] → Int
mCount k js  | length js < k = 0
             | otherwise     = 1 + mCount k js′
             where js′       = merge (sample 1 (take k js)) (drop k js)
```

By now, the reader might be frustrated by the apparently slow progress of this algorithm, which chooses just one selection at each step. It may seem that removing as many marbles as possible at one step is more advantageous, but the correctness of the algorithm needs to be maintained. Note what happens with this strategy when used on our example marble collection, with $k = 3$:

Marbles Remaining	*Selections*
4R 3G 3B 2Y	
	RGB, RGB, RGB
1R 2Y	

By taking as many as possible at the first step, we get only three marble selections, whereas the original correct greedy algorithm above gives the optimal number, 4. This shows the importance of re-evaluating at each step which jars are fullest.

However, there is a way to reduce the number of steps required: when the k jars which are fullest remain so after the sampling of one marble from each, then these same jars will be those sampled from next. Combining multiple samplings of one marble into one sampling of multiple marbles increases the efficiency of the algorithm, but how many can we sample at once? Intuition suggests we may sample i marbles from each jar, where i is one more than the difference in quantities between the least full of the k fullest jars, and the fullest jar of those remaining. Indeed a straightforward (but unilluminating, so omitted) calculation can be performed on *mCount*, leading to this result. Here is the transformed version:

```
mCount2  :: Int → [Jar] → Int
mCount2 k js  | length js < k = 0
              | otherwise     = i + mCount2 k js′
              where  i  = js!!(k−1) − js!!k + 1
                     js′ = merge (sample i (take  k  js)) (drop  k  js)
```

We now have an accelerated version of the same algorithm. Again using the same example, we can see its effect on the number of steps performed:

Marbles Remaining	*Selections*
4R 3G 3B 2Y	
	RGB, RGB
2R 2Y 1G 1B	
	RYG
1R 1Y 1B	
	RYB

The resulting efficiency improvement is not great, because although the first step of the algorithm may result in a large sampling of marbles, subsequent samplings consist of only one or two marbles, as either $js!(k-1) - 1 = js!k$ or $js!(k-1) = js!k$

at each subsequent step. Thus overall, the worst case complexity remains the same. However, progress towards greater efficiency can be made by looking carefully at how this algorithm performs with assorted marble collections.

4 A more than greedy solution

We observed earlier that in any bag of m marbles, the maximum number of selections of size k obtainable is $\lfloor m/k \rfloor$. Examination of the previous algorithm reveals a clear pattern as to when this maximum is *not* achieved (for the sake of this explanation we assume that the red marbles are those most numerous): on the occasions when there are at least k marbles left over after running the algorithm, there were too many red marbles to use up, even though a red marble was selected at each step. Here is a typical example of too many red marbles, with $k=3$:

Marbles Remaining	*Selections*
8R 4G 3B 2Y	
	RGB, RGB, RGB
5R 2Y 1G	
	RYG
4R 1Y	

Conversely, if there are not too many red marbles, they are easy to use up, leaving fewer than k marbles left over, and so the full complement of $\lfloor m/k \rfloor$ selections is obtained. (See previous pages for examples of marble collections with sufficiently few red marbles.) This inspires a conjecture:

If the fullest jar contains no more than $\lfloor m/k \rfloor$ marbles, where m is the total number of marbles, then it is possible to extract $\lfloor m/k \rfloor$ mingled marble selections of size k from the jars.

This turns out to be true, and the proof is by induction on k:

Proof
Base case: when $k=1$, $\lfloor m/k \rfloor = m$, and trivially we obtain m mingled selections with just one marble in each.

Inductive case: we first set aside the first $\lfloor m/k \rfloor$ marbles from the ordered list of jars. That is, first we set aside as many as we can of the marbles in the first jar (those of the most common colour), up to $\lfloor m/k \rfloor$ of them, then if necessary use marbles from the second jar, and so on, until we have $\lfloor m/k \rfloor$ marbles set aside. Assume without loss of generality that the $\lfloor m/k \rfloor$th marble set aside was yellow.

Before dealing with the remaining jars, we note that

$$\left\lfloor \frac{m - \lfloor \frac{m}{k} \rfloor}{k-1} \right\rfloor = \left\lfloor \frac{m}{k} \right\rfloor + \left\lfloor \frac{m \bmod k}{k-1} \right\rfloor.$$

We now observe that the remaining jars contain $m - \lfloor m/k \rfloor$ marbles in total, and we know already that no individual jar contains more than $\lfloor m/k \rfloor$ marbles. From the above equation, we have that $\lfloor m/k \rfloor \leqslant \lfloor (m - \lfloor m/k \rfloor)/(k-1) \rfloor$, so that none of the remaining jars may contain more than $\lfloor (m - \lfloor m/k \rfloor)(k-1) \rfloor$ marbles. This enables us to use the induction hypothesis on the remaining jars, to obtain $\lfloor (m - \lfloor m/k \rfloor)(k-1) \rfloor$ mingled marble selections of size $k-1$. As $\lfloor (m - \lfloor m/k \rfloor)(k-1) \rfloor \geqslant \lfloor m/k \rfloor$, we may take $\lfloor m/k \rfloor$ of these selections, and to each, we add one of the $\lfloor m/k \rfloor$ marbles set aside earlier.

Care must be taken with the yellow marbles, as this is the only colour which, as well as having been set aside earlier, may also appear in the selections of size $k-1$. As there were no more than $\lfloor m/k \rfloor$ yellow marbles in the first place, it is possible to avoid adding a yellow marble to a selection already containing a yellow marble, thus obtaining $\lfloor m/k \rfloor$ mingled marble selections of size k. $\square$

We have dealt with the case where the fullest jar is not too full, but we still need to consider the case when the fullest jar contains more than $\lfloor m/k \rfloor$ (red) marbles. In this case, note that the number of possible mingled marble selections is exceeded by the number of red marbles. This means that any optimal solution is equivalent to one which uses a red marble in every mingled selection (selections without red marbles may have one of their marbles swapped with a leftover red marble). Thus the maximum possible number of mingled selections of size k is equal to the maximum number we can obtain of size $k-1$, without the red marbles. We thus arrive at the following algorithm:

```
marblesCount3  :: Int → [Jar] → Int
marblesCount3 k js  =  mCount3 k (sum js) (sort  js)


mCount3  :: Int → Int → [Jar] → Int
mCount3 k m js  | k == 1      =  m
                | j ⩽ d       =  d
                | otherwise   =  mCount3 (k − 1) (m − j) (tail  js)
                where j  =  head js
                      d  =  m ‘div’ k
```

4.1 Lazy improvement

Laziness can improve the efficiency of this algorithm: in cases where the fullest jar is not too full, sorting of the whole list is not required, as only the number of marbles in the fullest jar and the total number of marbles are needed. In the worst case, it is still necessary to sort the whole list of jars.

By using a heap of jars, the algorithm is made more efficient. Initially, the heap is constructed in linear time (see Brodal & Okasaki (1996) and Okasaki (1996)), and then at each step, the fullest jar is removed from the heap. Our final program is thus:

```
marblesCount4  :: Int → [Jar] → Int
marblesCount4 k js  =  mCount4 k (sum js) (makeHeap  js)
```

```
mCount4  :: Int → Int → Heap Jar → Int
mCount4  k m jh | k == 1     = m
                | j ⩽ d      = d
                | otherwise  = mCount4 (k − 1) (m − j) jh′
                  where d    = m ‘div’ k
                     (j, jh′) = deleteMax jh
```

This gives a time complexity of $O(j)$ (for j jars) in the best case, and $O(j \ log \ j)$ in the worst. For the average case, we consider the distribution of m marbles into j non-empty jars by first assigning one marble to each jar, then randomly choosing a jar for each of the remaining $m - j$ marbles. The larger the number of jars to distribute the marbles between, the closer to 0 the chances of the fullest jar containing more than $\lfloor m/k \rfloor$ marbles (see Freeman (1979) for more details of this distribution). Hence this algorithm has average $O(j)$ complexity.

5 Conclusion

The final linear algorithm is a considerable improvement over the $O(j \ log \ j + m)$ of our first algorithm. However, the first algorithm does have its uses. Although the final algorithm could be adapted for the production of marble listings, the first algorithm is more suitable for a human to implement manually: the repeated sampling from the k fullest jars is a simple step-by-step algorithm which can easily be used in practical applications.

Acknowledgements

Thanks are due especially to Peter Harding, the original programmer for the drugs trial, and also to Jeremy Gibbons, who made helpful suggestions.

References

Brodal, G. S. and Okasaki, C. (1996) Optimal purely functional priority queues. *J. Functional Program.*, **6**(6), 839–857.

Curtis, S. (1996) *A relational approach to optimization problems.* DPhil thesis, Technical Monograph PRG-122, Computing Laboratory, Oxford.

Freeman, P. R. (1979) Exact distribution of the largest multinomial frequency. *J. Appl. Stat.*, **28**(3), 333–336.

Okasaki, C. (1996) The role of lazy evaluation in amortized data structures. *Proc. ACM SIGPLAN International Conference on Functional Programming (ICFP ’96)*, vol. 31(6), pp. 62–72.