COUNTING WITH CODE *

Saad Mneimneh and Alexey Nikolaev
Department of Computer Science
Hunter College
The City University of New York
New York NY 10065
saad@hunter.cuny.edu

ABSTRACT

Counting problems pose a difficult challenge for a majority of students. The struggle typically lies in setting up the problem and figuring out what counting rules to apply. To overcome this hurdle, we developed a programming language for counting with an emphasis on representation. The premise of the approach is that a standard representation for counting problems not only provides a framework for setting up the problem, but also allows counting rules to be applied with less ambiguity. The language itself is kept simple, so that the programming aspect would not hinder the mathematical imagination of the students and their ability to engage in abstract thought.

INTRODUCTION

When learning about counting, students discover the need for systematic methods to count things like the number of 16-bit patterns with 4 ones. Even then, they still stumble on the techniques. This is mainly due to their lack of a correct representation of what is being counted; they resort to examine order and/or repetition, but typically in a very ad-hoc way. For instance, on the one hand, the order of bits within a pattern is important, and on the other hand, if we specify which bits are ones, we can do so with no particular order.

The same can be said about repetition. It is obvious that bits repeat, but once we have 4 ones, the ones can no longer repeat. Without a clear framework of thought, counting problems are tricky to reason about. A programming language can provide a mechanism for setting up the problem; moreover, if the syntax promotes standard representations, counting rules for order and repetition become less ambiguous.

Our approach may be described as adding programming to the mathematical learning experience. We are not alone in that regard. Support for combinatorics exist in many libraries. However, these mainly provide the functionality to generate permutations and combinations and the like. While the ability to list alternatives has been found to be important in learning how to count [3], the use of such software assumes (and requires) prior knowledge of the subject. The work in [5] adds a visual component to the generation of permutations and combinations, but has only done so for a small set of predefined problems, and thus lacks the element of creative thinking when it comes to solving general problems that are not stated explicitly in terms of permutations and combinations.

An interesting take in [1] uses a finite state automaton (FSA) to count the number of strings of a some length n in a given regular language, which explores the fact that the number of paths to an accepting state in the corresponding

FSA is precisely the count we seek. The emphasis there is on setting up recurrences (dynamic programming) to calculate that number and avoid the exponential blowup.

Aside from the above, there are individual efforts that present pedagogy for tackling specific counting problems, e.g [8,2,9], but these do not transcend to a general and systematic approach.

Finally, [6,7] provide an argument for combining functional programming and discrete mathematics. While the two can illuminate each other, the goal there was to rescue functional programming as a curriculum trend in decline by weaving it into discrete mathematics. In our case, we are proposing to use a programming language specially for counting, while the focus remains not on how to program but on how to count.

A SYNTAX INSPIRED BY MATHEMATICAL NOTIONS

An effective tool to teach about counting should expose the underlying mathematics. Therefore, we envision a syntax that is inspired by the mathematical notions that we encounter in counting problems: A set is an unordered (unless indicated otherwise) collection of elements. Elements in a set have the same "type", so we call a set "homogeneous". A set is expressed using the notation $\{elem_1, elem_2, \ldots, elem_n\}$. A tuple is an ordered collection of elements not necessarily of the same type. So a tuple may or may not be homogeneous. A tuple is expressed using the notation $(elem_1, elem_2, \ldots, elem_n)$. We rely almost entirely on the following counting principle:

Definition 1 *(Product Rule) If a task consists of n phases, and the i^{th} phase can be carried out in α_i ways, irrespective of how the previous phases are carried out, then the entire task can be carried out in $\alpha_1 \alpha_2 \ldots \alpha_n$ ways.*

For instance, in New York city, every taxi cab has a plate number that consists of a digit, followed by a letter, followed by two digits. In counting the number of taxi cabs, one can model this problem by imagining the task of making a single plate. This task consists of 4 phases. The first phase can be carried out in 10 ways because we have 10 possible digits. Similarly, the second, third, and fourth phases can be carried out in 26, 10, and 10 ways respectively. Therefore, we can have $10 \times 26 \times 10 \times 10 = 26000$ taxi cabs.

In general, however, every problem will look different. Therefore, to handle the multiplicity of carrying out the phases in a standard way, we will say that each phase is an act made by choosing one element from a given set. Thus, the number of ways a phase can be carried out is equal to the size of that set (sometimes, however, choosing an element from a set reduces its size by 1. But the general idea remains the same). For the taxi problem, the sets are {0, 1, …, 9} and {a, b, …, z}. To represent phases, set and tuple expressions can list for each phase which set is used to make the choice. In a set representation, the phases can be carried out in any order (they can be permuted). In a tuple representation, the phases are carried out in the specified order. In both cases, the size of the set or tuple representation determines the number of phases.

From Representations to Programs: Consider the taxi problem described above. If we use *digit*={0, 1, …, 9} and *letter*={a, b, …, z}, then the phases for the taxi problem are: choose any element from *digit*, choose any element from *letter*, choose any element from *digit*, and choose any element from *digit*, carried out in that order. This can be represented by a tuple *(?digit, ?letter, ?digit, ?digit)* where *?* means *any*. There is no indication that elements cannot be reused; we say that the sets are *reusable*. The corresponding program is below:

```
digit = reusable {0,1,2,3,4,5,6,7,8,9}
#we can simply specify the size of the set
letter = reusable 26
count (?digit, ?letter, ?digit, ?digit)
```

Depending on the setting, making a choice from a set can remove the corresponding element permanently; in this case, we say that the set is *nonreusable*, and thus *?S* generally means *any available* element in *S*. Let's revisit the 16-bit problem. Creating a 16-bit pattern with 4 ones consists of choosing four distinct positions for the 4 ones. Therefore, if we assume a nonreusable set of positions *pos*={1, 2, …,16}, we repeatedly choose *any* available element from *pos* in four phases carried out in no particular order. This can be represented by a set {*?pos, ?pos, ?pos, ?pos*}. In order to enforce the homogeneity of a set representation, we will impose the syntax {*?S:n*}, where *S* is a set and *n* is an integer (for convenience, we also allow (*?S:n*) to denote a homogeneous tuple). Here's the program:

```
pos = nonreusable 16
count {?pos:4}
```

This time, however, and since the set is nonreusable, each choice reduces the size of the set by 1. By the product rule, the number of 16-bit patterns with 4 ones is 16×15×14×13. This is obviously wrong because the 4 phases can be permuted and the correct answer is obtained by dividing the above by the number of permutations 4!=24, resulting in 1820. This is automatically detected by the language (see Appendix), but in the initial stages, the essential goal is to teach the students how to come up with good representations, which is discussed in detail in Assisted Thinking.

Nested Representations: Sets and tuples may be nested. Consider for instance the problem of seating 3 people on 3 chairs. First we assume the two sets *person* and *chair*. We then start thinking about the task (in phases) of generating a single seating. To do so, we choose any person and any chair to make our first assignment, then the same to make the second, and finally the third, a total of six phases. This can be represented as: {(*?person, ?chair*):3}. The three elements in the nested set representation have the same

type given by the pair (*person, chair*), thus satisfying set homogeneity. The full programs is given below.

```
person = nonreusable {a,b,c}
chair = nonreusable {1,2,3}
count {(?person, ?chair):3}
```

Another problem is to count the number of ways we can make 3 teams of 2 given 6 people. The representation and program follow: {{?person:2}:3}.

```
person = nonreusable {a,b,c,d,e,f}
count {{?person:2}:3}
```

More language constructs: The notion of order can be made more explicit, so we introduce another notation, !S, which signifies making a choice by choosing the *next* available element from set *S*. This imposes an order on the elements of the set, and as such, !S can be carried out in only 1 way. As a design choice, ! cannot appear within { } because ! is about order and { } is not.

If we revisit the problem of making teams described above, we can find that the following representation also works (see Assisted Thinking): ((*!person, ?person)*:3). Since *person* is a nonreusable set and *!person* can be carried out in only 1 way, by the product rule, the number of ways we can make the teams is (1×5)×(1×3)×(1×1)=5×3×1=15.

What is the importance of introducing *next*? It offers some alternative ways for creating representations. This is very insightful because it helps discover identities by counting the same thing in different ways. For instance, the previous representation {{*?person*:2}:3} leads to (6×5/2!)×(4×3/2!)×(2×1/2!)/3!, which must also be 15, and one could generalize that (2n-1)(2n-3)…1= (2n)!/(2nn!).

We also introduce two additional kinds of sets: *identical* where all the elements are the same, and *ordered* where elements are assumed to be ordered; the use of these sets with ? and ! is described in Table 1.

Table 1: The use of *any* and *next*

	$?S$	$!S$	count	side effect (initially, $size = \|S\|$)
nonreusable	✓	✓	$size$ with ?	
			$\min(1, size)$ with !	$size = \max(0, size - 1)$
reusable	✓	✗	$size$	none
identical	✓	✗	$\min(1, size)$	$size = \max(0, size - 1)$
ordered	✗	✓	$\min(1, size)$	$size = \max(0, size - 1)$

An identical set can be used for the problem of distributing 5 dollar bills among 3 kids (all gifts must go); the seating problem has also been reformulated below in terms of an ordered set.

```
gift = identical 5                person = ordered 3
kid = reusable 3                  chair = nonreusable 3
count {(?gift, ?kid):5}           count ((!person, ?chair):3)
```

With Table 1 in mind, students can be gradually acquainted with the rules of counting and the algorithm presented in the Appendix.

ASSISTED THINKING (AND TWO GOLDEN QUESTIONS)

The most crucial aspect of obtaining a correct count is to create a good representation of the phases using sets and tuples following the procedure outlined previously. The representations are typically not unique. For instance, we could have used the representation (*?digit, ?digit, ?digit, ?letter*) for the taxi problem (choose the 3 digits then the letter). On the other hand, using (*?pos*:4) for the 16-bit problem would have been wrong. So when is a representation "good"? With the representation acting as a "template", we view a *configuration* as follows:

Definition 2 *(Configuration) A configuration is a string obtained by replacing ?S and !S with an element from S while obeying set and selection properties, e.g. elements replacing ?S must be distinct if S is nonreusable, and elements replacing !S must be chosen in some assumed order.*

We then need the notion of "equivalent" configurations, which we motivate by an example. Consider the handshake problem: how many possible handshakes can we count given a group of people? It is clear that one handshake (configuration) can be generated by a choice of two people. Given the set person={a, b, …}, students will typically be confronted to make a choice of representation between {*?person*:2} (correct) and (*?person*:2) (wrong). Either way, the configurations {*a, b*} and {*b, a*} are equivalent, and the same is true for (*a, b*) and (*b, a*) (in all cases persons *a* and *b* are chosen to shake hands).

To ensure that a representation is good, the students are encouraged to ask **two golden questions**:

- can the representation generate all valid (and only the valid) configurations?
- do "equivalent" configurations correspond to permutations of unordered choices (within { }), and vice-versa?

While "equivalent" relies on one's understanding of the problem (as in the handshake problem), observe that these two questions are tightly coupled to the syntax and the semantics of the language, and provide a mechanistic way of checking one's reasoning. For instance, the notion of *overcounting* is embodied in the fact that some equivalent configurations cannot be obtained from one another by permutation, e.g. the two equivalent configurations (a, b) and (b, a) above; which imply an overcounting by a factor of 2 (the basis of the handshake lemma). In contrast, {a, b} and {b, a} can be obtained from one another by permutation, making {*?person*:2} a good representation. As argued in [3], a significant challenge with solving counting problems is to convince oneself that each of the desirable outcomes have been counted exactly once. The goal of the two golden questions is exactly that! Let's put the two golden questions to the test:

16-bit problem: It is not hard to see that the representation {*?pos*:4} can generate all valid 16-bit patterns with 4 ones by replacing every occurrence of *?pos* with a distinct element from {1, 2, …, 16}. Given two equivalent configurations, say {1, 2, 3, 4} and {4, 2, 1, 3} (both represent the pattern 1111000000000000), it is obvious that one can be obtained from the other by permuting the unordered choices. Similarly, permuting the unordered choices results in equivalent configurations. Observe that if the wrong representation (*?pos*:4) is used instead, we would still be able to generate all valid configurations, but equivalent configurations such as (1, 2, 3, 4) and (4, 2, 1, 3) no longer correspond to a permutation of unordered choices, simply because there are none.

Taxi problem: It is easy to verify the first question. In addition, the configuration (1, a, 2, 3), for instance, is only equivalent to itself. Moreover, the choices in a tuple cannot be permuted.

Seating people on chairs: One may conceive the following representation (wrong) with three pairs (*?person, ?chair, ?person, ?chair, ?person, ?chair*). If we think of {*a, b, c*} as the set of people and {1, 2, 3} as the set of chairs, we observe that (a, 1, b, 2, c, 3) and (b, 2, c, 3, a, 1) are equivalent configurations but cannot be obtained from one another by permutation (no { } exist). However, the elements of a set must be homogeneous, so we cannot mix people and chairs as in {*?person, ?chair, ?person, ?chair, ?person, ?chair*}. In fact, one should not expect to be able to permute people and chairs. We observe that only certain classes of permutations give equivalent configurations, namely those that preserve the pairs of people and chairs. Hence, the following nested representation is needed: {(*?person, ?chair*):3}. Permuting the unordered choices results in equivalent configurations, and all equivalent configurations can be obtained by permutations. Observe that ((*!person, ?chair*):3) is also a good representation (where every configuration is only equivalent to itself), and is needed if person is changed to an ordered set (recall from Table 1 that ! must be used with ordered sets and ! is forbidden within { }).

Making teams: It is not hard to verify that {{*?person*:2}:3} is a good representation. Let's verify why ((*!person, ?person*):3) is a good representation. One has to assume an order on the people in {a, b, c, d, e, f}, e.g alphabetical order. Then every configuration of teams can be uniquely generated from the representation that repeatedly chooses the "smallest" available person first, then any available partner. Therefore, our representation passes the two golden questions. It is interesting to verity that ((*?person, !person*):3) does not!

Incomplete representations: When seating people on chairs, or making teams, it is tempting to come up with incomplete representations. For instance, if we seat two people, the third assignment of person to chair becomes automatic. Similarly, once we form the first two teams, the third team is implicit. As such, one might use the following representations, respectively: {(*?person, ?chair*):2} and {{*?person*:2}:2}. These "shortcuts" are wrong. Again, asking the two golden questions will clear up the issue. For instance, {(*a*, 1),(*b*, 2)} and {(*a*, 1),(*c*, 3)} are the same seating but cannot be obtained from one another by permutations. Similarly, {{*a, b*}, {*c, d*}} and {{*a, b*}, {*e, f*}} are equivalent teams.

In spite of the two golden questions, students can still make mistakes while reasoning about representations. The last line of defense is to actually check if the count is correct by considering smaller examples. For instance, one could reduce the digits and letters to {0, 1, 2} and {*a, b*} respectively, and check if the program for the taxi problem produces the correct count by explicitly generating all configurations manually. Therefore, given a counting problem, we advise the students to perform the procedure of Figure 1.

1. Define the sets and their properties (e.g. nonreusable vs. reusable).
2. Think about the task of generating one configuration. This task consists of phases and each phase chooses an element from a set (in a way consistent with the set).
3. Make a representation of the phases using sets and tuples (this becomes the template for configurations).
4. Answer the two golden questions about configurations (need a "Yes" for both).
5. Try the program on smaller examples and compare the answer to the number obtained by explicit enumeration (go back to 1 if needed).

Figure 1: Assisted Thinking procedure.

PROGRAM TRANSFORMATIONS THAT MIMIC ABSTRACTIONS

The four kinds of sets, i.e. nonreusable, reusable, identical, and ordered (and their rules of use as described in Table 1), can create alternative representations for a given counting problem, or different settings thereof. These representations provide new perspectives and insights to the problem, and as it turns out, in many cases systematically transform the program in ways that mimic the type of abstractions we perform often without giving enough careful thought. Therefore, we believe our programming language is a great tool for making the students more conscious of their abstractions, hence less prone to mistakes.

Consider the problem of distributing 3 gifts among 5 kids, where all gifts must go. We can think of a set of kids and a set of gifts. A typical task to generate one configuration would proceed in six phases where we first choose any kid and then choose any gift to make a pair, and repeat this three times (with no particular order), thus giving the representation {(*?kid*, *?gift*):3}. In the first setting, we will assume that we can give at most 1 gift per kid. This means that the set of kids is nonreusable. In the second setting, we assume unlimited gifts per kid. This is simply captured by making the set of kids reusable.

```
#at most 1 gift/kid
kid = nonreusable 5
gift = nonreusable {1,2,3}
count {(?kid, ?gift):3}
```

```
#unlimited gifts/kid
kid = reusable 5
gift = nonreusable {1,2,3}
cout {(?kid, ?gift):3}
```

The counting algorithm will produce (5×3)×(4×2)×(3×1)/3! = 5×4×3 = 60, and (5×3)×(5×2)×(5×1)/3! = 53 = 125, respectively.

Both settings can be redone by making the set of gifts ordered. This will force the use of ! when choosing a gift. But since ! cannot appear within { }, students will have to seek through abstract thought another representation (and verify the two golden questions). Indeed, one can systematically show, given the counting algorithm, that for a nonreusable set S of size n, the following representations are equivalent in terms of their count: {(*?S*, …):*n*} ≡ ((*!S*, …):*n*).

```
#at most 1 gift/kid                #unlimited gifts/kid
kid = nonreusable 5                kid = reusable 5
gift = ordered {1,2,3}             gift = ordered {1,2,3}
count ((?kid, !gift):3)            count ((?kid, !gift):3)
```

Note that making kid an ordered set instead of gift, and using ((!kid, ?gift):3) fails the first golden question. More importantly, knowing that !S contributes 1 to the product rule, the ordered set gift can be entirely eliminated from the program to yield for both settings:

```
#at most 1 gift/kid                #unlimited gifts/kid
kid = nonreusable 5                kid = reusable 5
count (?kid:3)                     count (?kid:3)
```

This boils down to choosing 3 kids with order (without and with repetition, respectively). This is a new way of looking at the problem. The kids order determines completely who gets what. We therefore managed to abstract away the gifts, something we typically do without a systematic thought process.

For both settings, if the gifts are identical, say $1 bills, then the programs are transformed as follows:

```
#identical gifts              #identical gifts
#at most 1 gift/kid                #unlimited gifts/kid
kid = nonreusable 5                kid = reusable 5
gift = identical 3                 gift = identical 3
count {(?kid, ?gift):3}            count {(?kid, ?gift):3}
```

The counting algorithm will produce (5×1)×(4×1)×(3×1)/3! = 5×4×3/3! = 10, and C(5+3-1,3)=35, respectively. Again, by observing that ?S contributes 1 to the product rule for identical sets, the identical set *gift* can be entirely eliminated from the program to yield for both settings:

```
#identical gifts                   #identical gifts
#at most 1 gift/kid                #unlimited gifts/kid
kid = nonreusable 5                kid = reusable 5
count {?kid:3}                     count {?kid:3}
```

This boils down to choosing 3 kids without order (without and with repetition, respectively). We have thus reproduced the four kinds of selection by systematically transforming the program, which provides mathematical "hooks" to where changes in the result are carried out by those abstractions.

CONCLUSIONS

We presented a language for counting. Programming with this language does not obscure the mathematical notions encountered in counting problems, on the contrary it emphasizes them by embedding them into the syntax, which stands behind why our framework can handle general counting problems; this is novel given the literature on

teaching how to count. The syntax offers a natural way for students to reason correctly about counting problems through the two golden questions and Steps 1 to 5 in Figure 1. Program transformations offer insights about the abstractions behind them.

The features outlined in this paper have been implemented, and an interpreter is made available online [4] with explanations and example problems. The examples are based on transforming word problems into representations, and should have a natural progression that makes sense to learners. In addition, those who have prior knowledge about counting should find the syntax helpful for strengthening their understanding of the subject

REFERENCES

[1] E. Corwin and A. Logar. Counting and automata. *Journal of Computing Sciences in Colleges*, 20(1):187-196, 2004.

[2] E. Goode and G. Trajkovski. Napoleon's soldiers. *Journal of Computing Sciences in Colleges*, 18(5):193-197, 2003.

[3] E. Lockwood and B. R. Gibson. Combinatorial tasks and outcome listing: Examining productive listing among undergraduate students. *Educational Studies in Mathematics*, 91(2):247-270, 2016.

[4] Saad Mneimneh and Alexey Nikolaev. Count by writing code. www.cs.hunter.cuny.edu/~saad/count/, 2015.

[5] L. B. Sherrell, J. J. Robertson, and T. W. Sellers. Using software simulations as an aide in teaching combinatorics to high school students. *Journal of Computing Sciences in Colleges*, 20(6):108-117, 2005.

[6] T. VanDrunen. The case for teaching functional programming in discrete math. In *OOPSLA*, pages 81-86. ACM, 2011.

[7] T. VanDrunen. Discrete mathematics and functional programming. *Franklin, Beedle and Associates,* 2012.

[8] J.-F. Yao, C. Turner, Y. Liu, and T. Goette. Binary number applications. *Journal of Computing Sciences in Colleges,* 22(2):251-258, 2006.

[9] J.-F. Yao. Counting the number of rectangles in an m by n grid. *Journal of Computing Sciences in Colleges,* 28(5):100-106, 2013.

APPENDIX (THE COUNTING ALGORITHM)

Given a representation *rep*, we say that it is *distinct* iff it contains ?S or !S where S is nonreusable or ordered. The count can then be computed recursively using the product rule with adjustments:

- count ?S and count !S are done as described in Table 1 with the appropriate side effect.
- count {*rep:*0} and count ( ) are 1 (empty choice).

- count (rep_1, …, rep_k) is $_{i=1…k}$ count rep_i (ordered selection).
- count {$rep:k$} can be divided into two cases. Let n_i=count rep for $i=1…k$ (explicitly performed k times to produce the proper side effects).
  - o rep is distinct: count {$rep:k$} is $\prod i=1…k\ n_i/i$ (unordered selection).
  - o rep is not distinct: count {rep:k} is $C(n+k-1, k)$, where n=mini=$1…k\ n_i$ (unordered with repetition).