Beyond Counting

Owen Kaser

September 17, 2014

1 Introduction

Combinatorial objects such as permutations and combinations are frequently studied from a counting perspective. For instance, “How many distinct subsets are there of the set \{cat, dog, moose\}?”. Such counting questions are important to CS as well as other disciplines. In CS, however, there are some related issues that we also consider. These include enumeration and generation.\footnote{Knuth’s TAOCP volume 4 \cite{Knu11} lists “existence”, “construction”, “enumeration”, “generation” and “optimization” as things to consider about combinatorial arrangements. However, what Knuth calls “enumeration”, we call counting. What Knuth calls “construction” corresponds to our (random) generation, and Knuth’s “generation” is our enumeration. So much for consistent terminology!}

2 Enumeration

Enumeration refers to the listing, without repetition, of all the combinatorial arrangements, preferably in some reasonable order. The counting issue just asks, “how long will the list be?” So, from counting, we know there are \(\binom{3}{2}\) length-2 subsets of \{cat, dog, moose\}. Enumeration corresponds to the list \{cat, dog\}, \{cat, moose\}, \{dog, moose\}. Since this is CS, our interest is in having computers do the enumeration. So we need to find efficient algorithms that perform enumeration of each kind of arrangement.

Enumeration can be used to design a brute-force solution of many “optimization problems”. These problems typically describe some kind of combinatorial arrangement of the input, and ask for a/the best arrangement, given some objective function that measures the goodness or badness of a particular arrangement.

For instance, given a set of \(k\) different strings, find the subset of size two, such that the sum of the two strings’ lengths is as small as possible. Here, the “subsets of size two” describe the arrangements. The “sum of the lengths” describes the objective function, and “as small as possible” says that we have a minimization problem rather than a maximization problem.

A sane algorithm for this problem is to sort the \(k\) strings by increasing length, and take the first two strings in the sorted list. The brute-force solution (see Algorithm 1) is to enumerate all possible subsets of size two (and from
counting, we know there are \( \binom{k}{2} \) of them. For each, we use the objective function to see how good it is, and we keep running track of the “best seen yet” arrangement. Once all arrangements have been enumerated, we return the best-seen-yet one. The brute-force solution is usually viewed as a last resort because the enumeration process typically sifts through a huge number of alternatives.

Data: Items \( i_1 \) to \( i_n \)

Result: Some arrangement involving these items

Algorithm 1: Sketch of the brute-force approach (don’t try this at home)

In Algorithm 1, the enumeration corresponds to the FOR loop. Exactly what kind of arrangements (subsets, subsets of a given size, permutations) are to be enumerated would vary. In our example, we would want to enumerate the \( \binom{n}{2} \) subsets of our items.

Although an efficient algorithm for the “two strings whose total length is shortest” problem is obvious, there are many other problems where an efficient algorithm may be hard to find. If your input is small enough, maybe the brute-force approach will be good enough. At the higher level of programming competitions, the inputs are usually designed so that a brute-force approach won’t be fast enough. However, at a local or regional programming competition, brute force is worth considering!

2.1 Enumerating ranges

Given items \( n \) that are (totally) ordered, to enumerate a range requires only deciding on the starting and stopping items. Use something like

Algorithm 2: Enumerating all ranges

Counting tells us that there are \( \Theta(n^2) \) possible ranges and inspecting the
code shows that we will enter the body of the inner loop once for each of these ranges — so it costs \( \Theta(n^2) \) to do the enumeration.

Note that the algorithm above does not enumerate any empty ranges.

### 2.2 Enumerating subsets

Counting tells us that there are \( 2^{63} \) subsets of a 63-element set. Even if we can visit one subset per nanosecond, it will be many lifetimes before \( 2^{63} \) subsets are visited. Therefore, we can assume that nobody will want to enumerate all subsets if there are more than 63 items.

A common programming trick is to use the "bijection" (1-1 and onto relation) between subsets and numbers written in binary. Mathematically, we consider the "characteristic vector" of a subset \( S' \) to be a binary number. (Recall that the characteristic vector \( iyd j \) entry say whether or not \( i_j \in S' \).) So consider \( S = \{ \text{cat, dog, moose} \} \) with \( i_1 = \text{cat,} i_2 = \text{dog,} \text{ and } i_3 = \text{moose.} \) The subset \( S' \) of \{dog, moose\} has characteristic vector 011, which we view as the decimal number 3 (the binary number is 011).

To enumerate all subsets of \( i_1 \) to \( i_n \) we just enumerate all numbers from 0 to \( 2^n - 1 \). To figure out the elements of the corresponding subset, we can use bitwise operations, as found in most programming languages. Things work out a little nicer if we start counting at 0, so let's suppose we are interested in all subsets of \( i_0 \) to \( i_{n-1} \).

```plaintext
for bitcounter = 0 to \( 2^n - 1 \) do
    you have a subset \( S' \), but to use it you need to examine bitcounter;
end
```

**Algorithm 3:** Enumerating subsets. If \( n < 32 \) you can use an \texttt{int} in Java for \texttt{bitcounter}. But for \( 32 \leq n < 64 \) you need to use a \texttt{long}.

In the loop, to know whether \( i_k \in S' \), use code like

```plaintext
if (bitcounter & (1L << k) != 0) ...
```

to inspect the \( k \)th bit in your \texttt{long}.

For the time complexity, note that the loop iterates \( 2^n \) times. The code to check whether any particular item is present in \( S' \) takes \( \Theta(1) \) time. Of course, you may end up needing to inspect each of the \( n \) items within the inner loop, leading to a time complexity of \( \Theta(n2^n) \).

### 2.3 Enumerating Permutations

Again, since the number of permutations of \( n \) items is \( n! \in \Omega(2^n) \), it is only feasible to enumerate them if \( n \) is small. There is a known bijection between a range of numbers and a set of permutations. However, the formula is complicated.

Instead, it fairly common to use recursion to enumerate permutations. See the book by Skiena[Ski98] for further details.
2.3.1 Permutations via recursion

To use the recursive approach, call \texttt{en(1,2,...,n}\texttt{, new empty list}) in

\begin{algorithm}
\textbf{Data:} Set $S$, List \texttt{ans}
\begin{algorithmic}
  \If{$S = \emptyset$}
    \State \textbf{return} (or “visit”) \texttt{ans};
  \Else
    \For{\texttt{each item in } $S$}
      \State remove item from $S$;
      \State add item to end of \texttt{ans};
      \State recursively call \texttt{en} with parameters $S$ and \texttt{ans};
      \State remove item from end of \texttt{ans};
      \State add item back to $S$;
    \EndFor
  \EndIf
\end{algorithmic}
\end{algorithm}

\textbf{Algorithm 4:} Recursive method \texttt{en} enumerates permutations of $S$, appending them after the prefix \texttt{ans}.

Tracing it on input $1,2,3$ leads to the recursion trace shown in Figure 1. The approach could be proven correct by an induction argument on the size of the first parameter, but we skip this.

To analyze the runtime cost, suppose that the costs of adding and removing items in $S$ and \texttt{ans} could be implemented in $\Theta(1)$ time. Then the cost would be $\Theta$ of the number of recursive calls made. There is one node at the top of the recursion tree. It has $n$ children. Each of them has $n-1$ children. So there are $n(n-1)$ nodes at the second layer of the tree, $n(n-1)(n-2)$ at the third layer of the tree, and so forth. Totalling all $n$ layers of the tree, we have something (assuming we have $n > 3$) like

\[
\sum_{l=0}^{n} n!/l! = n! \sum_{l=0}^{n} 1/l! \\
= n!(1 + 1 + \frac{1}{2} + \frac{1}{6} + \sum_{l=4}^{n} 1/l!) \\
\leq n!(\frac{5}{3} + \sum_{l=4}^{n} 1/l^2) \text{ since } l! > l^2 \text{ for } l \geq 4 \\
\in \Theta(n!) \text{ since exercise A.2-1 says } \sum_{l=1}^{n} 1/l^2 \in \Theta(1)
\]
2.3.2 Pandita’s algorithm

A non-recursive method has been known since at least the time of Narayana Pandita, around 700 years ago, and has the nice property of enumerating permutations in sorted order. We start with the smallest permutation of 0, . . . , n − 1 in an array $a$:

```plaintext
for $i = 0$ to $n - 1$ do
    $a[i] = i$
end
```

To go from one permutation in $a$ to the next, we use the following algorithm:

1. find largest index $k$ such that $a[k] < a[k + 1]$;
2. find largest index $l$ such that $a[k] < a[l]$;
3. swap $a[k]$ and $a[l]$;
4. reverse the range $a[k + 1]$ to $a[n - 1]$;

In the worst case, these steps could cost $\Theta(n)$. However, most of the time it is cheap to go from one permutation to the next. Analyzing this requires fancy techniques that are beyond the scope of CS2383.

2.3.3 Other algorithms

Knuth’s TAOCP volume 4, Section 7.2.1.2, is devoted to many other methods. The best ones cost $c + O(1/n)$ per permutation visited, where $c$ would be small.

3 $r$-Permutations

Recall that an $r$-permutation of an $n$ element set is arrangement of $r$ of the $n$ members of the set, and the order of the arrangement matters.

The 2-permutations of \{cat, dog, fox\} are

1. cat, dog
2. dog, cat
3. cat, fox
4. fox, cat
5. dog, fox
6. fox, dog

Looking at the recursion tree for the recursive approach to permutation generation, we see that the $r$ permutations will all appear at the same level in the recursion tree. The base case of the recursive algorithm could be modified to visit an $r$ permutation when $|S| = n - r$ rather than waiting until $|S| = 0$. 
4 \( r \)-Combinations

We know there are \( \binom{n}{r} \) \( r \)-combinations, if we count them.

If \( r \) is a very small constant (e.g., \( r < 5 \)) we can use nested loops. For \( r = 3 \) we have

\[
\begin{align*}
\text{for} & \quad x_r = r - 1 \quad \text{to} \quad n - 1 \quad \text{do} \\
& \quad \text{for} \quad x_{r-1} = r - 2 \quad \text{to} \quad x_r - 1 \quad \text{do} \\
& \quad \quad \text{for} \quad x_{r-2} = r - 3 \quad \text{to} \quad x_{r-1} - 1 \quad \text{do} \\
& \quad \quad \quad \text{print or visit} \quad i_{x_r}, i_{x_{r-1}}, i_{x_{r-2}}; \\
& \quad \text{end} \\
& \text{end} \\
& \text{end}
\end{align*}
\]

It gets ugly when \( r \) gets bigger, plus it is not nice to hard-code your algorithm for a specific value of \( r \) — think of the program maintainer who has to handle a “trivial” change of looking for 3-combinations to looking for 5-combinations. One would not normally expect to have to add in extra levels of “for” loops.

A general way to do the same basic computation is given in Knuth, TAOCP volume 4, Algorithms 7.2.1.2L (and also a more efficient “Algorithm T” that we skip).

It uses a \( r + 2 \) element array, \( a \). The algorithm “looks wrong” at first glance because the two nested loops are messing with the same variable, \( j \). Trace it to see it work!

\[
\begin{align*}
\text{create a new array } & \text{a initialized to } 0, 1, 2, \ldots, r-1, n, 0; \\
& j = 0; \\
& \text{while } j < r \text{ do} \\
& \quad \text{visit / print } a[r - 1], a[r - 2], \ldots, a[0]; \\
& \quad j = 0; \\
& \quad \text{while } a[j] + 1 = a[j + 1] \text{ do} \\
& \quad \quad a[j] = j; \\
& \quad \quad j = j + 1; \\
& \quad \text{end} \\
& \quad a[j] = a[j] + 1; \\
& \text{end}
\end{align*}
\]

5 Generation

Generation refers to “get me one of . . . ”. E.g., given a set of \( n \) integers, we might want to find any arbitrary 2-permutation. For permutations, combinations, ranges, and the other kinds of arrangements were are considering, it is not very interesting.

So we consider the harder problem of selecting, at random, one of the possible arrangements. We typically want the probability of selecting any of the arrangements to be equal. Since CS2383 students are not guaranteed to have
finished their probability courses, there can be no fancy proofs of correctness. A terribly inefficient method would be to enumerate all the arrangements and store them (perhaps in an array). Then choose one random index in this array and return the stored result. We can do much better than this brute-force type of approach.

5.1 Generating Ranges

Assume the range is within 0 to \( n - 1 \) inclusive.

If the range has to be non-empty, then just generate two non-equal endpoints that are between 0 and \( n - 1 \). Let \( l \) be the smaller one and \( r \) be the bigger one. Use the range from \( l \) to \( r \).

5.2 Generating a Member of a Range

If you need a random member of the range from \( \text{low} \) to \( \text{high} \), use \( \text{low} + \lfloor \text{rand()} \ast (\text{high} - \text{low} + 1) \rfloor \) where \( \text{rand()} \) generates a random real number such that \( 0 \leq \text{rand()} < 1 \). In Java, you might use \texttt{Math.random()}.

5.3 Generating a Subset

Since there is a bijection between subsets of an \( n \) element set and the numbers 0 to \( 2^n - 1 \), we can just select an integer uniformly at random between 0 and \( 2^n - 1 \). When written in binary, its 1 bits tell us about elements in the subset, and its 0 bits tell us about elements that are not in the subset.

This works fine when \( n \) is small, less than 64. If \( n \) is larger, loop through the elements of \( n \). For each one, flip a coin (e.g., generate a random number between 0 and 1 and see whether it is greater than 0.5) to decide whether it belongs to the subset.

5.4 Generating a Permutation

Generating a random permutation of some items means shuffling them randomly. The best method is sometimes called RANDOMIZE-IN-PLACE. Traverse down an array and, for each position \( i \), swap the item with one of the items from \( i \) to the end of the array.

5.5 Generating an r-Combination

Generating a random collection of \( r \) items out of a larger collection of \( N \) is called “sampling without replacement” by statisticians. If you have the \( N \) items in an array, it would seem that you could just randomly generate \( k \) from 1 to \( N \), and if the \( k \)th item had not already been included in the sample, include it. Keep doing this until you have \( r \) items. The downside is that, if you are unlucky in your choice of random numbers, the approach might run forever.
Another approach could be to generate a random permutation and then take the first $r$ items in it. You actually don’t need to produce the entire permutation, so with $\Theta(r)$ random swaps you can get your sample. The approach also requires $\Theta(N)$ memory because you should store all $N$ items to support swapping. Perhaps $r \ll N$ and your computer has enough memory to store the sample, but not enough memory to store the $N$ items.

A better approach is to use the “Selection Sampling” algorithm by Fan et al., from 1962. It is described in Knuth’s TAOCP v2, Section 3.4.1 in the 3rd edition.

This approach is better if your data is coming in a stream, and there is so much of it that you cannot store it. You see the data item once, and have to decide, on the spot, whether the item belongs to the sample.

Taking each item with a $r/N$ probability seems to be an option. However, while you will, on average, wind up with $r$ items in your sample, you will usually not end up with exactly $r$ items in a particular sample. Selection sampling solves that. In the code below `rand()` generates a uniform random number over $[0, 1)$.

\[
S = \emptyset;
\]

\[
\text{for } t = 0 \text{ to } N - 1 \text{ do}
\]

\[
\text{if } (N - t)^*\text{rand}()< r - |S| \text{ then}
\]

\[
\text{add } A[t] \text{ to } S;
\]

\[
\text{exit if } |S| = r;
\]

\[\text{end}\]

The correctness of this algorithm is probably not apparent, but when this algorithm is applied to $N$ items, for any item the probability that it will be selected is $r/N$.

In the worst case, this algorithm runs in $O(N)$ time.

\section*{5.6 Generating an $r$-Permutation}

You could use the approach sketched earlier: generate a random permutation and take the first $r$ items in it. If you have enough memory to store all $N$ items, this would work well.

If you have enough memory to store $r$ items but not $N$, you need an alternative. A sensible approach to generating an $r$-permutation would be to generate an $r$-combination and then randomly choose the order of the items in the sample. In other words, first use Selection Sampling and then use RANDOMIZE-IN-PLACE on the $r$ items resulting.

\section*{References}