CO 330 LECTURE 25 SUMMARY

FALL 2026

Summary

We started random generation. By random generation we mean the following (at least for now).

Let $\mathcal{C}$ be a combinatorial class. An algorithm which takes as input a nonnegative integer n and retuns as output an element $c \in \mathcal{C}_n$ such that every element of $\mathcal{C}_n$ has probability $1/c_n$ of being produced is known as a *uniform generation algorithm*.

We'll assume we can generate random numbers uniformly in $[0, 1)$ with a function we'll call `rand()` and so then we can generate a random integer in $\{0, \ldots, n-1\}$ via $\lfloor$`n*rand()`$\rfloor$.

Some things we can generate directly, like binary strings. To uniformly generate a random binary string of length n just pick each bit uniformly. Some things we can generate via a bijection, like subsets: we have a bijection between subsets of $\{1, \ldots, n\}$ and binary strings of length n via the indicator vector, so to generate a random subset, generate a random binary string and then take it through the bijection.

We can generate a random permutation in the same kind of way. In the q-counting section we talked about a bijection $\mathsf{S}_n \leftrightharpoons [n] \times [n-1] \times \cdots \times [1]$. On the right hand side we can generate uniformly at random in each coordinate, and then take it back over to get a permutation.

That's all fine and good but lets do something more systematic. Let's use the information of a combinatorial specification in order to do *recursive random generation*. Sometimes we can look inside the structure of a class in more detail and do better, but this is something we can do whenever we have a specification.

The specification itself is a tree (or a forest if the specification is a system), and we will follow this tree structure. We just need to know how to deal with each of these operations and will call recursively on the children if any. $\mathcal{Z}$ and $\mathcal{E}$ are straightforward.

```
def genZ
  input n (the desired size of the output object)
  if n=1
    return Z
  else
    return null

def genE
  input n
  if n=0
    return E
  else
    return null
```

Next suppose $\mathcal{A} = \mathcal{B} \cup \mathcal{C}$ with $\mathcal{B} \cap \mathcal{C} = \emptyset$. How do we pick a random element of size n? We should pick from $\mathcal{B}_n$ $\frac{b_n}{a_n}$ of the time and we should pick from $\mathcal{C}_n$ $\frac{c_n}{a_n}$ of the time in order to be uniform over all. This gives

```
def genA=BcupC
  input n
  x = rand()
  if x < b_n/a_n
    return genB(n)
  else
    return genC(n)
```

Note that we're cheating slightly. The input is more than just n, we also need b_n and a_n (or b_n and c_n). For some classes we have closed forms for the b_n and a_n so we can just put these in the function. Otherwise we should precalculate them and put them in a table. We can do this by iterating the specification so this is reasonably fast.

Also, you should think of this as acting on the expression tree for the specification, so when we are acting on a $\cup$ vertex then the need the two subtrees (the $\mathcal{B}$ and $\mathcal{C}$) so we can call recursively on them.

The next step is $\times$, which is where we'll start on Monday, but we already tried it at the end in an example just in plain python and in the enumeration.ca notes you can see more sophisticated sage code, which we will look at on Monday as well.

Next time

We'll finish recursive random generation, and then on to Boltzmann samplers.

References

- https://enumeration.ca/asymptotics/random-generation/