Random Permutations

Michael Goodrich
CS 165

Trees with snow on branches, "Half Dome, Apple Orchard, Yosemite," 1933. Ansel Adams. U.S. government image. U.S. National Archives and Records Administration.

Generating Random Permutations

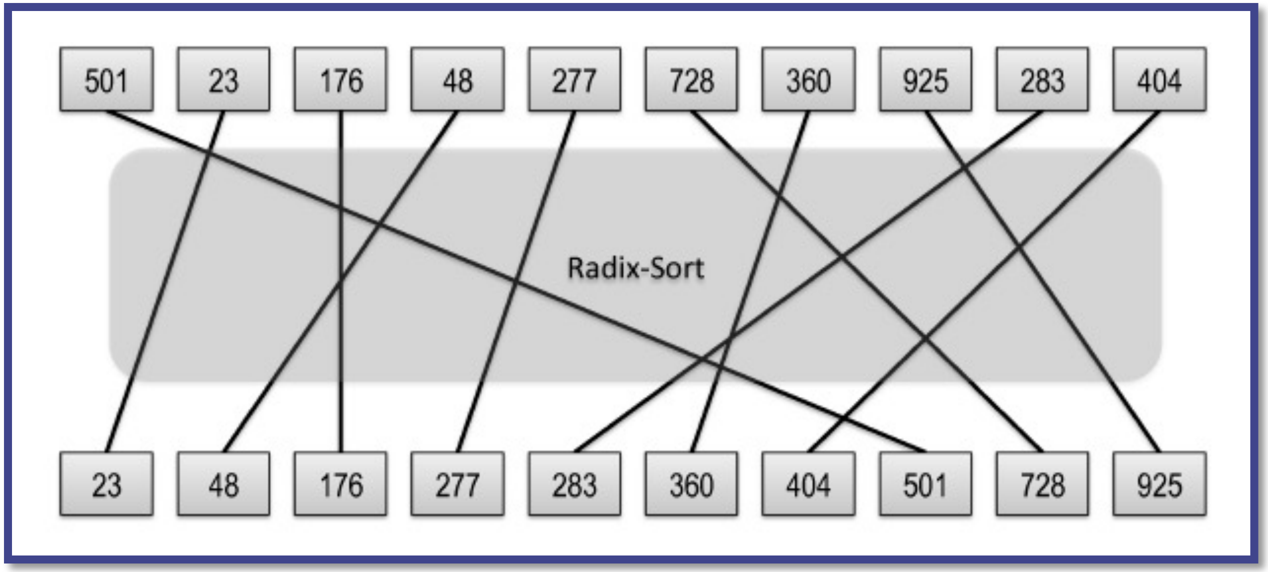
- The input to the random permutation problem is a list, $X = (x_1, x_2, \ldots, x_n)$, of n elements, which could stand for playing cards or any other objects we want to randomly permute.
- The output is a reordering of the elements of X, done in a way so that all permutations of X are equally likely.
- We can use a function, **random**(k), which returns an integer in the range $[0, k-1]$ chosen uniformly and independently at random.

Applications: Simple Algorithms and Card Games

- A **randomized algorithm** is an algorithm whose behavior depends, in part, on the outcomes of random choices or the values of random bits.
- The main advantage of using randomization in algorithm design is that the results are often simple and efficient.
- In addition, there are some problems that need randomization for them to work effectively.
- For instance, consider the problem common in computer games involving playing cards—that of randomly shuffling a deck of cards so that all possible orderings are equally likely.

Algorithm 1: Random Sort

- This algorithm simply chooses a random number for each element in X and sorts the elements using these values as keys.



Analysis of Random-Sort

- To see that every permutation is equally likely to be output by the random-sort method, note that each element, x_i, in X has an equal probability, 1/n, of having its random r_i value be the smallest.
- Thus, each element in X has equal probability of 1/n of being the first element in the permutation.
- Applying this reasoning recursively, implies that the permutation that is output has the following probability of being chosen:

$$\left(\frac{1}{n}\right) \cdot \left(\frac{1}{n-1}\right) \cdots \left(\frac{1}{2}\right) \cdot \left(\frac{1}{1}\right) = \frac{1}{n!}$$

- That is, each permutation is equally likely to be output.
- There is a small probability that this algorithm will fail, however, if the random values are not unique.

Fisher-Yates Shuffling

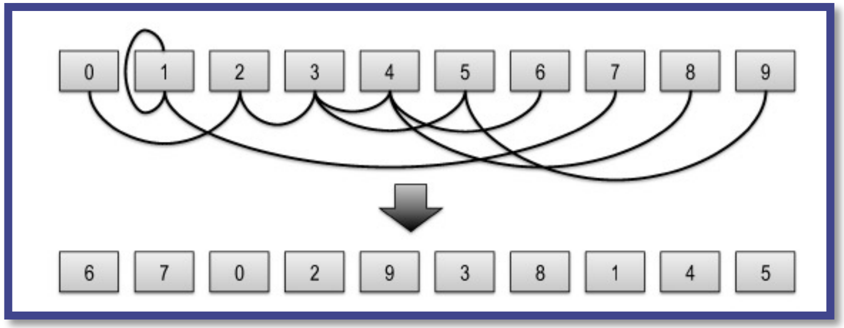
- There is a different algorithm, known as the Fisher-Yates algorithm, which always succeeds.

Algorithm FisherYates(X):

Input: An array, X, of n elements, indexed from position 0 to $n-1$

Output: A permutation of X so that all permutations are equally likely

```
for k = n − 1 downto 1 do
    Let j ← random(k + 1)        // j is a random integer in [0, k]
    Swap X[k] and X[j]           // This may "swap" X[k] with itself, if j = k
return X
```



Analysis of Fisher-Yates

- This algorithm considers the items in the array one at time from the end and swaps each element with an element in the array from that point to the beginning.
- Notice that each element has an equal probability, of 1/n, of being chosen as the last element in the array X (including the element that starts out in that position).
- Applying this analysis recursively, we see that the output permutation has probability

$$\left(\frac{1}{n}\right) \cdot \left(\frac{1}{n-1}\right) \cdots \left(\frac{1}{2}\right) \cdot \left(\frac{1}{1}\right) = \frac{1}{n!}$$

- That is, each permutation is equally likely.