

## Randomized algorithms

**Definition:** A *randomized algorithm* is an algorithm that can make calls to a random number generator during the execution of the algorithm.

These calls will be of the form  $x := \text{Random}(a, b)$ , where  $a, b$  are integers,  $a \leq b$ .

A call to  $\text{Random}(a, b)$  returns an integer in  $[a, b]$ , with every integer in the range being equally likely to occur.

Successive calls to  $\text{Random}(*, *)$  are assumed to be mutually independent.

**Definition:** The *expected running time of a randomized algorithm on a given input* is the average running time of the algorithm over all outcomes of calls to the random number generator.

The *expected running time of a randomized algorithm for inputs of length  $n$*  is the maximum expected running time of the algorithm over all inputs of length  $n$ .

A randomized algorithm terminates in time  $t(n)$  with probability at least  $p$  if, for every input of length  $n$ , the probability that the algorithm terminates in time  $t(n)$  is at least  $p$ .

# Generating a Random Permutation

The need to generate a random permutation arises in many situations, so let us look at a simple randomized algorithm to generate a random permutation.

---

RANDOMIZE-IN-PLACE( $A$ )

**Input.** An array  $A[1..n]$  of  $n$  elements.

**Output.** A rearrangement of the elements of array  $A$ , with every permutation of the  $n$  elements equally likely to occur.

**for**  $i := 1$  **to**  $n$  **do**

    swap  $A[i]$  and  $A[\text{RANDOM}(i, n)]$

**rof**

---

In order to prove correctness of this algorithm we need the following definition:

**Definition.** A  $k$ -permutation of an  $n$ -element set  $S$  is a sequence of  $k$  elements from the set  $S$ .

How many  $k$ -permutations of an  $n$ -element set are there?

We claim there are  $\frac{n!}{(n-k)!}$  of them. (Why?)

**Proof of correctness** of RANDOMIZE-IN-PLACE.

We use the following **loop invariant**:

- At the start of the  $i$ th iteration of the **for** loop, the subarray  $A[1..i-1]$  contains each  $(i-1)$ -permutation of the elements of the array  $A$  with probability  $\frac{(n-i+1)!}{n!}$ .

**Initialization.** For  $i = 1$  the statement refers to the empty subarray and the 0-permutation, and is trivially true.

**Maintenance.** Assume the loop invariant holds at the start of iteration  $i$ . We will now show that it must hold at the start of the  $(i+1)$ th iteration. To see this, consider any  $i$ -permutation  $X = \langle x_1, x_2, \dots, x_i \rangle$  of the input elements.

Let  $E_1$  be the event that the first  $i-1$  iterations of the **for** loop placed the sequence  $\langle x_1, x_2, \dots, x_{i-1} \rangle$  in subarray  $A[1..(i-1)]$ .

Let  $E_2$  be the event that the  $i$ th iteration of the **for** loop placed  $x_i$  in  $A[i]$ .

Since the  $i$ th iteration of the **for** loop does not examine any element in subarray  $A[1..(i-1)]$ , the sequence  $X$  will appear in subarray  $A[1..i]$  at the start of the  $(i+1)$ st iteration if and only if both events  $E_1$  and  $E_2$  occur.

Thus, probability that  $X$  occurs in  $A[1..i]$  at the start of the  $(i+1)$ st iteration is  $Pr(E_1 \cap E_2)$ .

By the induction hypothesis,  $Pr(E_1) = (n-i+1)!/n!$ .

Also, the conditional probability  $Pr(E_2 | E_1) = 1/(n-i+1)$  since the element  $x_i$  is chosen randomly from the subarray  $A[i..n]$ .

Hence, using the fact definition of conditional probability we have (since  $Pr(E_1) \neq 0$ ):

$$Pr(E_1 \cap E_2) = Pr(E_2 | E_1) \cdot Pr(E_1) = \frac{(n-i+1)!}{n!} \cdot \frac{1}{(n-i+1)} = \frac{(n-i)!}{n!}$$

This restores the loop invariant at the start of the  $(i+1)$ st iteration.

**Termination.** The algorithm terminates at the end of the  $n$ th iteration of the **for** loop, i.e., at the start of the  $(n+1)$ st iteration. The loop invariant tells us that at the start of the  $(n+1)$ st iteration the array  $A[1..n]$  contains each  $n$ -permutation of the elements of the input array with probability  $1/n!$ . Since there are  $n!$  permutations of an  $n$ -element set, this proves that RANDOMIZE-IN-PLACE rearranges the elements in the input array such that every permutation of the elements is equally likely to occur.

**Running time.** Although RANDOMIZE-IN-PLACE is a randomized algorithm, it returns the correct answer in (deterministic)  $O(n)$  time, independent of the random bits generated during execution. This is not typical of randomized algorithms, and occurs in this case because randomization arose only through the problem specification (i.e., we need to generate a *random* permutation).

In the next lecture we will study a randomized version of QUICKSORT, where the running time is a random variable.