# Appendix B

# Algorithms

This appendix is about algorithms.

## B.1 Matrix Algorithms

When describing algorithms for manipulating matrices, we will use the notation $\mathbf{x}[i]$ for the $i$-th element of a vector (row or column), and $\mathbf{A}[i, j]$ for the entry at the $i$-th row and the $j$-th column of matrix $\mathbf{A}$.

It is important to understand that, depending on the programming language and the situation, obtaining the value $\mathbf{A}[i, j]$ may just correspond to getting a value in the memory, or executing a more or less complex function of $\mathbf{A}$, $i$ and $j$. In particular, the field of "values" contained in the matrices is not specified. In most cases, values are real or complex numbers. But in some cases, they may be polynomials, functions or even... matrices.

When evaluating the algorithmic complexity, we will always consider that obtaining a "value", multiplying or adding two values are "elementary" operations.

The indices of elements of vectors, matrices or tables will be assumed to start from 1, even though most programming languages of today have indices start from 0.

Final preliminary observation: we shall not make exagerate attempts at saving bits of memory or eliminating instructions here and there. We prefer that algorithms be clear, at the possible (negligible) cost of memory or time. Modern compilers perform automatically many optimizations on memory reuse anyway.

### B.1.1 Matrix/Matrix multiplication

The first generic algorithm is for computing the matrix product: $\mathbf{C} = \mathbf{A} \times \mathbf{B}$. The matrices may be of any compatible dimensions: if $\mathbf{A} \in \mathcal{M}_{n\times m}$ and $\mathbf{B} \in \mathcal{M}_{m\times p}$, then $\mathbf{C} \in \mathcal{M}_{n\times p}$.

The algorithmic complexity of this algorithm is $nmp$ multiplications and additions. In short: $O(nmp)$. In particular, if all matrices are square $n \times n$, the complexity is $O(n^3)$.

### B.1.2 Vector/Matrix multiplication

Algorithms performing vector/matrix multiplications may of course be adapted to matrix/vector multiplication by transposition.

**Algorithm 13:** Matrix multiplication

```
Data: Two matrices $\mathbf{A} \in \mathcal{M}_{n\times m}$ and $\mathbf{B} \in \mathcal{M}_{m\times p}$
Result: A matrix $\mathbf{C} \in \mathcal{M}_{n\times p}$ such that $\mathbf{C} = \mathbf{AB}$
begin
    for $i$ from 1 to $n$ do
        for $j$ from 1 to $p$ do
            $val \leftarrow 0$
            for $k$ from 1 to $m$ do
                $val \leftarrow val + \mathbf{A}[i,k] \times \mathbf{B}[k,j]$
            $\mathbf{C}[i,j] \leftarrow val$
```

We present two algorithms. The first one is the special case of Algorithm 13, where the first matrix has only one row and the second one is square.

**Algorithm 14:** Vector/Full Matrix multiplication

```
Data: One row vector $\mathbf{x} \in \mathcal{M}_{1\times n}$ and one square matrix $\mathbf{A} \in \mathcal{M}_{n\times n}$
Result: A vector $\mathbf{y} \in \mathcal{M}_{1\times n}$ such that $\mathbf{y} = \mathbf{xA}$
begin
    for $i$ from 1 to $n$ do
        $val \leftarrow 0$
        for $k$ from 1 to $n$ do
            $val \leftarrow val + \mathbf{x}[k] \times \mathbf{A}[k,i]$
        $\mathbf{y}[i] \leftarrow val$
```

The complexity of this algorithm is $O(n^2)$.

The second algorithm takes advantage of the fact that, in practice, matrices may contain a large number of zeroes. Such matrices are called *sparse*.

In that case, it is possibly advantageous to store matrices in a special data structure. We describe this structure as adapted to vector/matrix multiplications. Remember that things must be transposed for matrix/vector multiplications.

For every column $i$ of matrix $\mathbf{A}$, define:

- `nb`$[i]$ as the number of non-zero elements in the column;
- `idx`$[i,j]$ as the index of the $j$-th non-zero element in the column; this is a list/array of `nb`$[i]$ integer numbers;
- `val`$[i,j]$ as the value of the $j$-th non-zero element in the column.

The amount of memory needed to store a matrix this way is of order $m$ integer numbers and values, where $m$ is the number of non-zero entries in the matrix. This is potentially a large improvement with respect to the storage of full matrices which requires $O(n^2)$ values.

Multiplication is performed with Algorithm 15. The complexity of this algorithm is $O(m)$. Again, this is possibly a large improvement with respect to the complexity of Algorithm 14.

**Algorithm 15:** Vector/Sparse Matrix multiplication

```
Data: One row vector $\mathbf{x} \in \mathcal{M}_{1\times n}$ and one square matrix $\mathbf{A} \in \mathcal{M}_{n\times n}$, stored in sparse
      form
Result: A vector $\mathbf{y} \in \mathcal{M}_{1\times n}$ such that $\mathbf{y} = \mathbf{xA}$
begin
    for $i$ from 1 to $n$ do
        $val \leftarrow 0$
        for $j$ from 1 to $nb[i]$ do
            $k \leftarrow \texttt{idx}[i, j]$
            $val \leftarrow val + \mathbf{x}[k] \times \mathbf{A}[k, i]$
        $\mathbf{y}[i] \leftarrow val$
```

## B.2 Random Numbers

When simulating Markov chains and other random processes with a computer, it is necessary to have the computer produce *random* numbers.

Programming Languages such as `C`, `C++`, `Java` or `Maple` all have basic functions to generate random numbers. Since these are produced by a deterministic procedure, they are called *pseudo-random*.

There exist several methods to produce a sequence of pseudo-random numbers. Their detailed description is beyond the scope of this document. Standard references for this are [14, 3].

The starting point is always a procedure which provides pseudo-random numbers *uniformly distributed* in the intervall $[0, 1]$. C'est assez facile dans la plupart des langages de programmation. In the `C` and `C++` languages, the function `random()`. In the Java language, one uses the function `Math.random()` which returns a double value with a positive sign, greater than or equal to 0.0 and less than 1.0.

### B.2.1 Exponentially distributed random numbers

The simulation of CTMC involves random variables with an exponential distribution. These can be generated using Algorithm 16.

**Algorithm 16:** Generation of exponentially distributed random numbers

```
Data: A parameter $\lambda > 0$
Result: A pseudo-random number with distribution $\sim \text{Exp}(\lambda)$
return $-\dfrac{\texttt{Unif}([0,1])}{\lambda}$
```

### B.2.2 The naive algorithm for General Discrete Distributions

A general finite discrete distribution is described by a sequence of nonnegative numbers $p_1, \ldots, p_n$ such that $p_1 + \ldots + p_n = 1$ (the probabilities), together with a sequence of values $v_1, \ldots, v_n$.

They represent the distribution of a random variable $X$ such that for all $i = 1..n$,

$$\mathbb{P}(X = v_i) \; = \; p_i \; . \tag{B.1}$$

The values $v_i$ need not be distinct, although in this case it is more efficient to "group" the probabilities and remove duplicates in the list of values.

The following algorithm generates samples of this distribution. It is quite intuitive and straightforward to program: we call it the "naive" algorithm. Indeed, its performance can be quite bad, and Walker's method, to be described in Section B.2.3, is much better.

**Algorithm 17:** The naive algorithm for sampling from a discrete distribution

```
Data: An array of $n$ probabilities $p_i$ and values $v_i$
Result: A random number equal to v[$k$] with probability $p_k$
begin
    $U \leftarrow$ Unif([0, 1])
    $j \leftarrow 1$
    while $U >$ p[$j$] do
        $U \leftarrow U -$ p[$j$]
        $j \leftarrow j + 1$
return v[$j$]
```

The principal trouble with this algorithm is that the number of loops needed to get the result is itself a random value. In the worst case, it is $n$. In average, it is: $\sum_i ip_i$.

### B.2.3 Walker's Algorithm for General Discrete Distributions

Walker's method, or the method of aliases, is a technique for generating random numbers according to a discrete distribution, in constant time. This requires a special data structure, and a pre-processing of the distribution, which we describe next. A good reference for this algorithm is [14, §3.4.1].

The starting point is a finite discrete distribution, as described in Section B.2.2:

$$\mathbb{P}(X = v_i) \; = \; p_i \; . \tag{B.2}$$

The purpose of the method is to construct a sequence of numbers $P[j] \in [0, 1]$, $j = 1, \dots, n$ and of *aliases* $Y[j]$, $j = 1, \dots, n$ such that Algorithm 19 below returns a random value with the distribution in (B.2).

The algorithm which constructs the arrays `P` and `Y` is Algorithm 18. The algorithm uses a function `SortList`, which acts on lists of the form $[(p_1, a_1), \dots, (p_k, a_k)]$, and sorts them in increasing order of their first element. The algorithmic complexity of this algorithm is of order $O(k^2)$. Indeed, sorting initially the list of $k$ elements can be performed in $O(k \log(k))$ operations and sorting the subsequent lists can be done in $O(k)$ because the lists are already partially sorted. Since there are $k$ loops, the result is $O(k^2)$.

As an illustration, this algorithm is executed on the Binomial distribution characterized by the $k = 5$ pairs of probabilities and values:

$$(\frac{1}{16}, A), (\frac{4}{16}, B), (\frac{6}{16}, C), (\frac{4}{16}, D), (\frac{1}{16}, E).$$

**Algorithm 18:** Construction of the alias tables for Walker's method

```
Data: An array of $n$ probabilities $p_i$ and values $v_i$
Result: Two arrays: the levels P and the aliases Y
begin
    set $q \leftarrow$ the list of couples $(p_i, i)$, $i = 1..n$
    for $k$ from $n$ to $1$ do
        $q \leftarrow$ SortList($q$)
        // The result is a sequence $(q_1, a_1), \ldots, (q_k, a_k)$
        P[$a_1$] $\leftarrow kq_1$
        Y[$a_1$] $\leftarrow v_{a_k}$
        $q_k \leftarrow q_k - (1/k - q_1)$
        remove the first element of the list $q$
```

For each of the 5 loops, the table below displays the value of the list $q$ just after the sorting instruction, and at the end of the loop. There, the first element is removed but shown as a "$-$", and the last element is modified. When showing the pairs $(q_j, a_j)$, we have preferred to show the *value* v[$a_j$] instead of the index $a_j$. The values of the tables P and Y are shown as well. Observe that the last element in Y at the end of the algorithm is irrelevant. The graphical interpretation of the values P and the aliases is given in FigureB.2.3. Algorithm 19 consists in picking a point at random, uniformly in the larger rectangle, and returning the label corresponding to the area.

| Step | Value of $q$ | Table P | Table Y |
|---|---|---|---|
| 1 | $[(\frac{1}{16},A),(\frac{1}{16},E),(\frac{4}{16},B),(\frac{4}{16},D),(\frac{6}{16},C)]$ | | |
| | $[\ -\ ,(\frac{1}{16},E),(\frac{4}{16},B),(\frac{4}{16},D),(\frac{19}{80},C)]$ | $[\frac{5}{16},-,-,-,-]$ | $[C,-,-,-,-]$ |
| 2 | $[(\frac{1}{16},E),(\frac{19}{80},C),(\frac{4}{16},B),(\frac{4}{16},D)]$ | | |
| | $[\ -\ ,(\frac{19}{80},C),(\frac{4}{16},B),(\frac{9}{80},D),]$ | $[\frac{5}{16},\frac{5}{16},-,-,-]$ | $[C,D,-,-,-]$ |
| 3 | $[(\frac{9}{80},D),(\frac{19}{80},C),(\frac{4}{16},B),]$ | | |
| | $[\ -\ ,(\frac{19}{80},D),(\frac{13}{80},C)]$ | $[\frac{5}{16},\frac{5}{16},\frac{9}{16},-,-]$ | $[C,D,B,-,-]$ |
| 4 | $[(\frac{13}{80},C),(\frac{19}{80},D)]$ | | |
| | $[\ -\ ,(\frac{19}{80},D)]$ | $[\frac{5}{16},\frac{5}{16},\frac{9}{16},\frac{13}{16},-]$ | $[C,D,B,C,-]$ |
| 5 | $[(\frac{1}{5},C)]$ | | |
| | $[\ -\ ]$ | $[\frac{5}{16},\frac{5}{16},\frac{9}{16},\frac{13}{16},1]$ | $[C,D,B,C,-]$ |

### B.2.4 The rejection method

The rejection method is used to sample uniformly an object in a set $\mathcal{A}$, if there exists a method to sample uniformly in a larger set $\mathcal{B}$.

If Unif($\mathcal{B}$) is a given procedure, then Unif($\mathcal{A}$) is obtained as Algorithm 20. In order for this algorithm to work, it is necessary to have a method for sampling uniformly in $\mathcal{B}$ (function represented as Unif($\mathcal{B}$)) and a method for deciding if the object sampled belongs to $\mathcal{A}$. Depending on the situation, the second method may be more costly than the first one.

The algorithmic complexity of this method is interesting to study. Each time it is called, this algorithm will perform a number of loops which is a random variable $L$. The probability that $L = \ell$ is the probability that the test fails $\ell - 1$ times and succeeds at the last attempt.

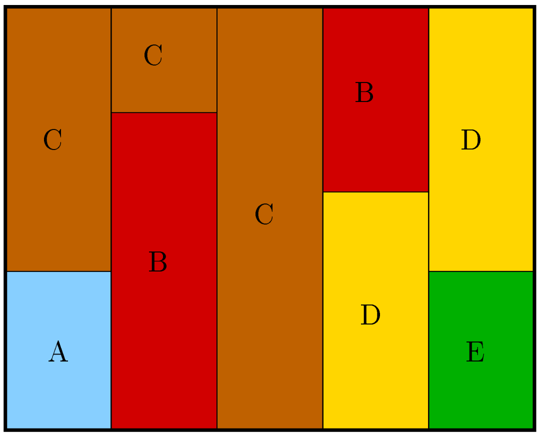

Figure B.1: Illustration of the alias table on the binomial example

**Algorithm 19:** Sampling of a discrete distribution using Walker's method

```
Data: One array of $n$ values $\texttt{v}[i]$, $i = 1..n$
Data: Two arrays of $n$ numbers $\texttt{P}[i]$ and aliases $\texttt{Y}[i]$, obtained from $\texttt{v}$ and some list of
      probabilities $p_k$ using Algorithm 18
Result: A random number equal to $\texttt{v}[k]$ with probability $p_k$
begin
    $V \leftarrow \texttt{Unif}([0,1])$
    $J \leftarrow \texttt{Unif}(\{1,\dots,n\})$
    if $V < P[J]$ then
        $res \leftarrow v_J$
    else
        $res \leftarrow Y[j]$
return $res$
```

**Algorithm 20:** Function $\texttt{Unif}(\mathcal{A})$

```
Result: A sample of the uniform distribution on $\mathcal{A}$
begin
    repeat
        $x \leftarrow \texttt{Unif}(\mathcal{B})$
    until $x \notin A$;
    return $x$
```

The probability that the test succeeds is

$$\sigma \;=\; \frac{\#\mathcal{A}}{\#\mathcal{B}} \,,$$

and the probability that it fails is $1-\sigma$. Consequently, and since the different samplings using `Unif`$(\mathcal{B})$ are supposed to be independent, the probability of exactly $\ell$ loops is: $(1-\sigma)^{\ell-1}\sigma$. This is a *geometric* distribution on $\{1,2,\ldots\}$. The average number of loops is then $1/\sigma = \#\mathcal{B}/\#\mathcal{A}$. If this ratio is large, the algorithm may look for a long time for a needle in a haystack.