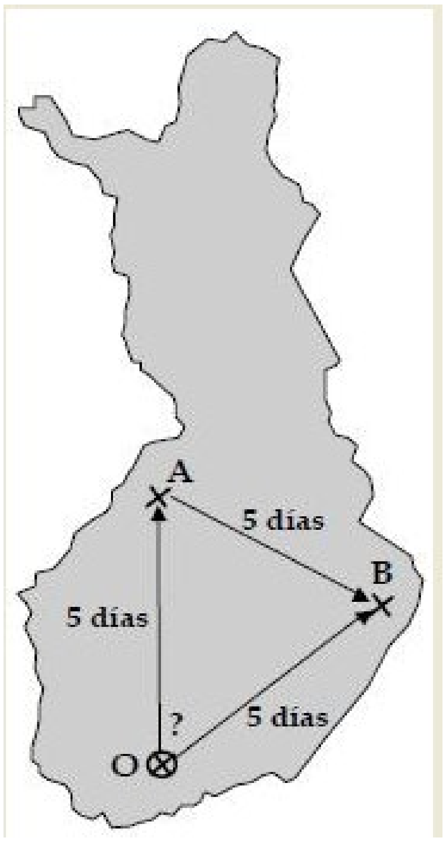
UNIT 6. NON DETERMINISTIC ALGORITHMS

1. Introduction.

A story about a treasure, a dragon, a computer, an elf and a doubloon. In A or B there is a treasure of x lingots of gold but I do not know in which of the two places it is. A dragon visits the treasure every and take y lingots. I know that if I stay 4 more days in O with my computer I will solve the mystery. An elf offers me a deal: he gives me the solution now if I pay him the equivalent of the amount the dragon would take in 3 nights.



What should I do? If I stay 4 more days in O until I solve the mystery, I can get to the treasure in 9 days, and get x-9y lingots. If I accept the deal with the elf, I get to the treasure in 5 days, I find there x-5y lingots from which I must pay 3y to the elf, and I get x-8y lingots. It is better to accept the deal but ... there is a better solution! Which one? Use the doubloon that remains in my pocket! I throw it into the air to decide which place I go first (A or B). If I hit the right place in the first attempt, I get x-5y lingots. If I do not hit, I go to the next site later and settle for x-10y lingots. The average expected benefit is x-7.5y.

Conclusion: in some algorithms in which a decision appears, it is sometimes preferable to choose randomly rather than to waste time calculating which alternative is the best. This occurs if the time required to determine the optimal choice is too much compared to the average obtained by taking the decision at random. This is the basic idea of a non-deterministic or probabilistic algorithm.

The fundamental characteristic of a probabilistic algorithm is that the same algorithm can behave differently applying the same data. The differences between the deterministic and probabilistic algorithms are the following:

- A deterministic algorithm is never allowed to end, make a division by 0, enter an infinite loop, etc. A probabilistic algorithm can be allowed if that happens with a very small probability for any data. If it happens, the algorithm is aborted and its execution is repeated with the same data.
- If there is more than one solution for given data, a deterministic algorithm always finds the same solution (unless it is programmed to find several or all). A probabilistic algorithm can find different solutions running several times with the same data.
- A deterministic algorithm is not allowed to calculate an incorrect solution for any data. A probabilistic algorithm can be wrong if this happens with a small probability for each input data. By repeating the execution, a sufficient number of times for the same data, the degree of confidence in obtaining the correct solution can be increased as much as desired.
- The analysis of the efficiency of a deterministic algorithm is sometimes difficult. The analysis of probabilistic algorithms is, very often, very difficult.

A comment on "chance" and "uncertainty": a probabilistic algorithm can be allowed to calculate a wrong solution, with a small probability. A deterministic algorithm that takes a long time to obtain the solution may suffer errors caused by hardware failures and obtain the wrong solution. That is to say, the deterministic algorithm does not always guarantee the certainty of the solution and it is also slower. Moreover, there are problems for which no algorithm is known (deterministic or probabilistic) that gives the solution with certainty and in a reasonable time (for example, the duration of the life of the programmer, or the life of the universe ...). A fast probabilistic algorithm that gives the correct solution with a certain probability of error is better. Example: decide if a number of 1000 figures is prime.

2. MonteCarlo algorithms.

Let p be a real number such that $0<p<1$. A Monte Carlo algorithm is p-correct if it returns a correct solution with probability greater than or equal to p, regardless of the input data. Sometimes, p will depend on the size of the input, but never on the data of the input itself.

A very important problem that can be solved with the Monte Carlo method is to determine if a number is prime. It is the best known Monte Carlo algorithm. This problem is especially useful for cryptographic techniques, which are based on factoring very large numbers into prime factors. There is no known deterministic algorithm that solves this problem in a reasonable time for very large numbers (hundreds of decimal digits). In recent years we have witnessed an unprecedented expansion in the use of public key cryptography, including in the field of institutional relations, such as the electronic ID card and electronic passport. These documents and many others need to use prime numbers (usually large) as basic elements to generate the keys or for other stages of the cryptographic protocol. The most used probabilistic algorithm is that of Miller-Rabin, which is based on Fermat's minor theorem (1640):

Let n be a prime number. Then $a^{n-1} \bmod n = 1$ for any integer a such that $1<=a<=n-1$. For example, let $n=7$ and $a=5$. Then, $a^{n-1}=5^6=15625=2232x7+1$. We will use the contrapositive version of this theorem: If n and are integers such that $1<=a<=n$ and $a^{n-1} \bmod n \neq 1$, then n is not a prime number. Although we have not seen it, there is an algorithm based on the Divide and Conquer technique that allows the modular exponentiation (solve a^n mod z) that is in O(log n). Using two elementary properties of modular arithmetic:

$$xy \bmod z = [(x \bmod z)\ \ (y \bmod z)] \bmod z$$
$$(x \bmod z)^y \bmod z = x^y \bmod z$$

we would have the following pseudocode:

```
Function expomod(a, n, z) // calculates $a^n$ mod z
    i=n
    r=1
    x=a mod z
    while i > 0 do
        if (i is odd) then
            r = rx mod z
            x =  $x^2$ mod z
            i =int(i /2)
        eif
    ewhile
    return r
efun
```

The problem is that to verify that a number n is prime, we will have to check that all the values a between 1 and n-1 satisfy that a^{n-1} mod n = 1. However, we could prove it with a single number chosen at random between 1 and n-1. For example, a first version of the probabilistic algorithm is:

```
Function Fermat(n)
    a = uniform(1..n-1)
    if expomod(a,n-1,n)=1 then return True
    else return False
efun
```

When Fermat (n) returns false we will be sure that n is not a prime. However, if Fermat (n) returns true we cannot conclude anything. This example demonstrates the way of working of the Monte Carlo type algorithms.

Another example of application of the Monte Carlo algorithms is to determine if a vector has a majority element. To do this, it is checked at random if any of these elements appears more than N/2 times. The algorithm would look like this:

```
Function MajorityMonteCarlo(v, N)
    x=v[random(1…N)]
    total=0
    for i=1 to n do
        if (v(i)=x) then total=total+1
    efor
    return (total>N/2)
efun
```

For non-majority vectors this method has probability 1 of hitting (since no element repeats more than N/2 times). For the vectors that do have a majority element, the method has a probability greater than 1/2, with probability p=(#occurrences of the majority element)/N. Then it fails with Pf<1/2. It could be improved by doing:

```
Function MajorityMonteCarlo2(v, N)
    if MajorityMonteCarlo(v, N) then
        Return True
    else
        Return MajorityMonteCarlo(v, N)
```

Now assuming that the vector is a majority, it will only fail if MajoritaryMonteCarlo fails twice. Therefore, MajoritaryMonteCarlo2 fails with probability $p<(1/2)\cdot(1/2)=1/4$. To generalize it with an error dimension $\varepsilon>0$:

```
Function MajorityMonteCarlo_ε(v, N, e)
    p=1
    m=False
    repeat
        p=p/2
        m=MajorityMonteCarlo(v,N)
    until m or (p<ε/2)
    return m
```

3. Las Vegas algorithms.

A Las Vegas algorithm differs from a Monte Carlo one, in that it never gives a false solution, but makes random decisions to find a solution before a deterministic algorithm and if it does not find a solution it admits it. There are two types of Las Vegas algorithms, based on the possibility of not finding a solution:

a) Those which always find a correct solution, even if random decisions are not successful and efficiency decreases.
b) Those which sometimes, due to unfortunate decisions, do not find a solution.

The first type applies to problems in which the deterministic version is much faster in the average case than in the worst case. For example, in Quicksort, where the worst cost is n^2 and the average cost is nlogn. Las Vegas algorithms can reduce or eliminate efficiency differences for different input data. Very likely, cases that require a lot of time with the deterministic method are now solved much faster. In cases where the deterministic algorithm is very efficient, they are now solved with more cost. In the average case, the cost is not improved, but an uniformization of the execution time is made for all entries of equal size.

The second type is acceptable if they fail with low probability. In that case, they are executed again with the same input data. They are used to solve problems for which efficient deterministic algorithms that solve them are not known. The execution time is not limited, but it is reasonable with a high probability. They are presented in the form of a procedure with a success variable that takes true value if a solution is obtained and false in another case. It would be LV(x, solution, success) where x represents the input data.

Let p(x) be the probability of success if the input is x. Las Vegas algorithms require that p(x)>0 for all x. Be the following function:

```
Function repeatLV(x)
    repeat
        LV(x, solution, success)
    until success
    return solution
efun
```

The number of passes of the loop is 1/p(x). Let t(x) be the expected time of repeating LV(x). The first call to LV succeeds after a time s(x) with probability p(x). The first call to LV fails after a time f(x) with probability 1-p(x). The total expected time in this case is f(x)+t(x), because after the call to LV fails we return to the starting point (and therefore we need a time t(x)). Thus, t(x)=p(x)s(x)+(1-p(x))(f(x)+t(x)). Solving the above equation you get:

$$t(x) = s(x) + \frac{1-p(x)}{p(x)} f(x)$$

This equation is the key to optimize the performance of this type of algorithms.