Algorithm Design 2.

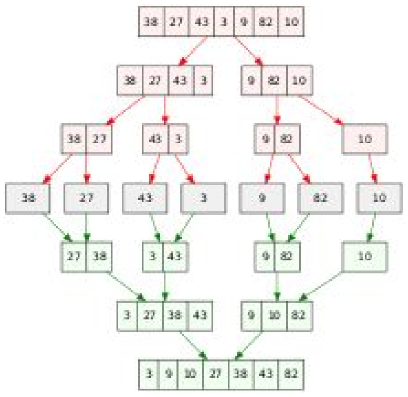
Ortiz Vega Angelo
Course Code: CE2103
Name: Algorithms and Data Structures II,
Academic Area of Computer Engineering.
Cartago, Costa Rica.

-Keywords: Algorithm Design, Genetic Algorithms, Greedy Algorithms, Dynamic Programming, Backtracking, Divide and Conquer.

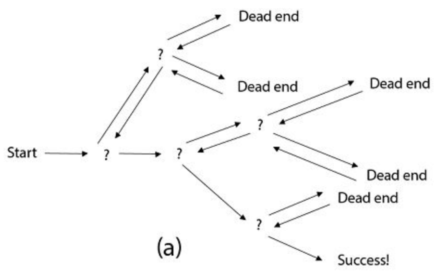
-Content:

Techniques to design algorithms:
~ Probabilistic Algorithms.
~ Genetic Algorithms.
~ Greedy Algorithms.
~ Dynamic Programming.
~ Backtracking.
~ Divide and Conquer.

Divide and Conquer.
Most basic technique with which you learn to program. Divide the problem into subproblems. Conquer the problems through recursion. Optionally combines the results of the subproblems.
Two states are identified:
1. Base Case: The problem is very simple and small.
2. Recursive Case: The problem can not be solved yet.

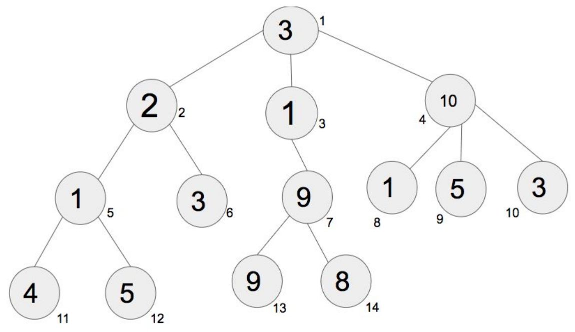


Backtracking:
Test all possible solutions, usually recursive algorithms. It decomposes the solution search process into partial tasks, if one task does not lead to a test solution with another.
Features:
1. Exhaustive: All possible solutions are sought until one is valid or all are finished.
2. Return: If a road does not lead to the global solution, it goes back and tries another.



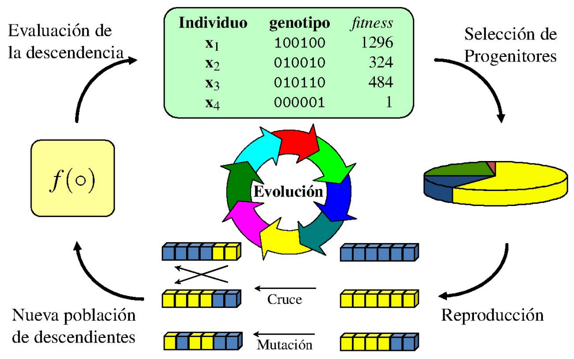
The eight queens puzzle is based on the classic strategy games problem which is in this case putting eight chess queens on an 8×8 chessboard such that none of them is able to capture any other using the standard chess queen's moves. The color of the queens is meaningless in this puzzle, and any queen is assumed to be able to attack any other. Thus, a solution requires that no two queens share the same row, column, or diagonal. The eight queens puzzle is an example of the more general *n* queens puzzle of placing *n*8 queens on an *n*×*n* chessboard:
http://www.brainmetrix.com/8-queens/

Dynamic Programming:
Method to solve problems of optimization by means of divisions of smaller problems (similar to divide and conquer).
It is based on the idea that the subproblems contribute to each other to solve the big problem.



Genetic Algorithms:
Given a specific problem to solve, the entry of the Genetic Algorithm (AG) is a set of potential solutions to that problem, coded in some way, and a metric called aptitude function, or fitness, that allows quantitatively evaluate each candidate solution. These candidates may be solutions that are already known to work, with the aim of improving the AG, but are often generated randomly.
From there, AG evaluates each candidate according to the aptitude function. Of course, it should be borne in mind that these first randomly generated candidates will have minimal efficiency with respect to the resolution of the problem, and most will not work at all. However, by pure chance, a few can be promising, showing some characteristics that show, even if only in a weak and imperfect way, a certain capacity to solve the problem.
These promising candidates are preserved and allowed to reproduce. Multiple copies of them are made, but these copies are not perfect, but they are introduced some random changes during the copying process, in the way of mutations that can suffer the descendants of a population. Then, this digital offspring continues with the next generation, forming a new set of candidate solutions, and they are again submitted to a round of aptitude assessment. Candidates who have worsened or have not improved with the changes in their code are eliminated again; but again, by pure chance, the

random variations introduced in the population may have improved some individuals, making them better solutions to the problem, more complete or more efficient. The process repeats the iterations that are necessary, until we obtain sufficiently good solutions for our purposes.


Evaluación de la descendencia
Selección de Progenitores
$f(\circ)$
Evolución
Cruce
Mutación
Nueva población de descendientes
Reproducción

Individuo	genotipo	*fitness*
x_1	100100	1296
x_2	010010	324
x_3	010110	484
x_4	000001	1

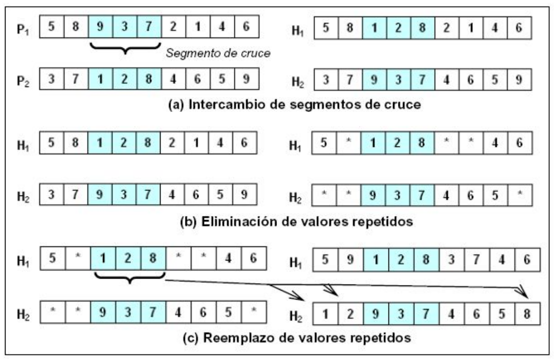


Figura 6. Operación de cruce

Compression Algorithms:

Compression: generate a new representation for data that requires less space.

There are two types of compression algorithms:

1. Lossless: there is no loss of data.
2. Lossy: there is data loss.

Huffman Algorithm:

Huffman coding is a lossless data compression algorithm. In this algorithm, a variable-length code is assigned to input different characters. The code length is related to how frequently characters are used. Most frequent characters have the smallest codes and longer codes for least frequent characters.

There are mainly two parts. First one to create a Huffman tree, and another one to traverse the tree to find codes.

Steps to build Huffman Tree:

Input is an array of unique characters along with their frequency of occurrences and output is Huffman Tree.

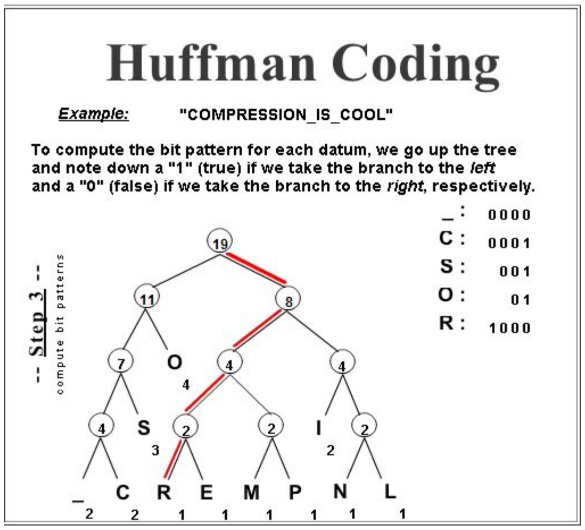
1. Create a leaf node for each unique character and build a min heap of all leaf nodes (Min Heap is used as a priority queue. The value of frequency field is used to compare two nodes in min heap. Initially, the least frequent character is at root)

2. Extract two nodes with the minimum frequency from the min heap.

3. Create a new internal node with a frequency equal to the sum of the two nodes frequencies. Make the first extracted node as its left child and the other extracted node as its right child. Add this node to the min heap.

4. Repeat steps#2 and #3 until the heap contains only one node. The remaining node is the root node and the tree is complete

Example of Huffman Compression:

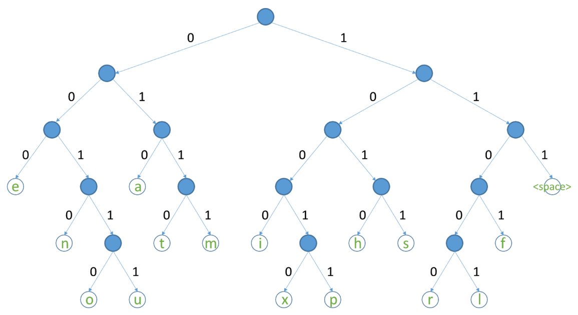


Huffman Coding

Example: "COMPRESSION_IS_COOL"

To compute the bit pattern for each datum, we go up the tree and note down a "1" (true) if we take the branch to the *left* and a "0" (false) if we take the branch to the *right*, respectively.

_ :	0000
C :	0001
S :	001
O :	01
R :	1000


e
a
<space>
n
t
m
i
h
s
f
o
u
x
p
r
l