Algorithms and data structures test questions

How does the Divide and Conquer method work?

:r1 It divides the problem into partial tasks, which must be independent.

:r2 It divides the problem into partial tasks that may be dependent.

:r3 It divides the tasks between multiple computers

:r4 It does not look for a specific solution, but just some appropriate approximation

:r1 ok 2

--

How does a greedy algorithm work?

:r1 It calls itself until it finds a solution

:r2 Makes decisions based on random (pseudo-random) elections

:r3 Seeks local extrema to find extreme global

:r4 It divides the problem into partial tasks that are dependent.

:r3 ok 2

--

What is ADT - Abstract data type?

:r1 Abstract data that cannot be implemented

:r2 Designation for data types that are independent of their own implementation

:r3 General data types like Integer, Real, Boolean ...

:r4 Specific implementation of data type

:r2 ok 2

--

Basic operations with ADT are:

:r1 Constructor, Selector, Modifier

:r2 Constructor, Destructor

:r3 Constructor, Selector, Modifier, Destructor

:r4 Selector, Destructor

:r1 ok 2

--

Experimental analysis of time-consuming algorithms:

:r1 Take place in an environment where the program (algorithm) runs

:r2 It does not depend on the environment where the program is running

:r3 It does not require algorithm implementation

:r4 It focuses on the simplest (least difficult) case

:r1 ok 2

--

Pseudo-code:

:r1 It depends on the programming language

:r2 It reveals the problems of a specific implementation

:r3 Is a higher description level, more structured than a classic description, less detailed than implementation

:r4 A lower description level, less structured than a classic description, requires knowledge of a specific programming language

:r3 ok 2

--

Primitive operations

:r1 Depending on the specific programming language, it cannot be identified in the pseudocode

:r2 Another name for a specific implementation of the procedure (methods, algorithm)

:r3 A program block that includes several operations that are made on a single call basis

:r4 Basic operations performed by an algorithm such as evaluating an expression, assigning a value to a variable, calling a procedure ...

:r4 ok 2

--

If the algorithm is the number of 2n² -3, the time-consuming and Big O notation is:

:r1 Exponential O (c^n -3)

:r2 Logarithmic, O (log n)

:r3 Constant, O (1)

:r4 Quadratic O (n^2)

:r4 ok 2

--

ADT Stack

:r1 Inserting and deleting is performed using the FIFO principle

:r2 The basic operations are: push (Object), findMin (), findMax (), deleteAll ()

:r3 It never throws an exception

:r4 It is controled by the LIFO principle and the basic operations are: push (Object), pop ()

:r4 ok 2

--

If the Stack is based on an array,

:r1 Then, time required for each operation is O (n^2 + n), where n number of elements in the stack

:r2 At the beginning, we need to define the stack size

:r3 Any number of elements can be stored, you can simply enlarge the stack

:r4 When adding an element to the full stack, the EmptyStackException exception occurs

:r2 ok 2

--

ADT Queue

:r1 Inserting and deleting is performed using the FIFO principle

:r2 The basic operations are: push (Object), findMin (), findMax (), deleteAll ()

:r3 Insertion and deletion occurs using the LIFO principle

:r4 It never throws an exception

:r1 ok 2

--

Queue

:r1 Stores a collection of items where the item is an ordered pair (priority-value), which maintains the order in the queue

:r2 We use a circular array for implementation

:r3 The basic operations are: enqueue (object), dequeue (), first (), last (), order()

:r4 The time complexity for any implementation it is best to O (N^2 log N)

:r2 ok 2

--

Vector

:r1 Defines the relationship before / after between positions

:r2 Elements are ordered by size

:r3 Expands the concept of array to store sequence of arbitrary objects

:r4 Allows you to store objects even on negative positions (negative index)

:r3 ok 2

--

List

:r1 Dynamic data structure, size varies based on element addition (deletion)

:r2 Static data structure, size does not change

:r3 The element can be read, inserted and removed by determining its order

:r4 Basic, generally usable data type for storing an ordered set of elements

:r1 ok 2

--

Single linked list

:r1 Contains a link to the previous and next nodes

:r2 It can be used to implement stack and queues with linear memory difficulty and constant time requirement for each basic operation

:r3 Static data structure

:r4 Defines the relationship before / after between positions

:r2 ok 2

--

Sequence

:r1 Combination of Vector and List ADT, access to elements by order and position

:r2 Elements cannot be accessed by determining their order

:r3 Access to elements based on the FIFO principle

:r4 Access to elements based on the LIFO principle

:r1 ok 2

5 – 8

Tree

:r1 Linear static data structure

:r2 It contains nodes, with the ancestor-child relationship (parent-child)

:r3 The external node is called the root

:r4 The internal node is called a leaf

:r2 ok 2

--

The pre-order traversal of tree shown in the figure gives the sequence:

:r1 A, B, C, D, E, F, G, H, I

:r2 A. C, E, D, F, G, H, I

:r3 F, B, A, D, C, E, G, I, H

:r4 C, E, H, A, D, I, B, G, F

:r3 ok 2

--

The in-order passage through the tree shown in the figure gives the sequence:

:r1 A, B, C, D, E, F, G, H, I

:r2 A, C, E, D, B, H, I, G, F

:r3 F, B, A, D, C, E, G, I, H

:r4 C, E, H, A, D, I, B, G, F

:r1 ok 2

--

Binary tree

:r1 It contains two roots

:r2 It contains just two sheets

:r3 Each internal node has at least two offspring, at least one of which is a leaf

:r4 Each internal node has just two offspring that form an ordered pair (left child, right child)

:r4 ok 2

--

Priority queue

:r1 It introduces the Comparator ADT, which allows you to compare two objects

:r2 Two different elements must always have a different priority (key)

:r3 A set of elements must be defined on a set of elements

:r4 Basic operations with a priority queue are: insertItem (k, o), removeMin (), upheap (), downheap ()

:r1 ok 2

--

Priority queue

:r1 Stored elements can be accessed based on their queue index

:r2 Keys are any objects that can be defined in order, two different elements may have the same key

:r3 Linear static data structure

:r4 The element with the smallest priority is always taken first

:r2 ok 2

--

Heap

:r1 It is represented as a binary tree that keeps the keys as internal nodes, for which the child key is always greater than or equal to the parent keys

:r2 Root contains the largest key

:r3 The last node leaf is the outermost node located at the leftmost

:r4 The downheap () function re-arranges the heap when inserting a new element

:r1 ok 2

--

Heap

:r1 Access to elements based on the FIFO principle

:r2 Access to elements based on the LIFO principle

:r3 After inserting a node, you need to call the upheap () function to restore the stack

:r4 Inserting and removing any element from the heap will not disrupt its arrangement

:r3 ok 2

--

Dictionary

:r1 Size must always be set when creating a dictionary, regardless of implementation

:r2 Requires that in the key-value pair, both the key and the value can be ordered

:r3 Oredered key-value pairs, where the keys are unique, values are not

:r4 We always remove the element with the smallest key

:r3 ok 2

--

Log File

:r1 Cannot be used if the set of values is not defined in the layout

:r2 Used for small dictionaries or applications where search is the most common operation

:r3 It is used where the most frequent operation is insertion, while searching and removal is rare.

:r4 The elements are always inserted in the middle of the sequence

:r3 ok 2

--

Lookup table

:r1 We always use a disordered dictionary to implement it

:r2 Used for small dictionaries or applications where the most common operation is searching

:r3 It is used where the most frequent operation is insertion, while searching and removal is rare.

:r4 Access to elements based on the LIFO principle

:r2 ok 2

--

Hash table

:r1 For a given key type, it has a hash function that assigns an integer value to the key, and a table of size N

:r2 Cannot be used if the hash function returns the same value to different keys (there is a collision)

:r3 Requires that in the key-value pair, both the key and the value can be ordered

:r4 It organizes the values according to the key, independent of the hash function, which only serves to control the stored data

:r1 ok 2

--

Sorting algorithms

:r1 We designate them as stable if they retain the order of inserting items with the same key

:r2 Sorts a dataset by value, not by key

:r3 They require a defined order of values

:r4 Cannot be used if the data contains several keys with the same value

:r1 ok 2

--

Bubble sort

:r1 It works by comparing the first and last element of the series

:r2 Requires extra memory for the size of the sorted data file

:r3 Only one pass is sufficient to order any data set

:r4 Versatile, the most value elements bubble at the end of the list

:r4 ok 2

--

Heap sort

:r1 It is a stable sorting algorithm using the ADT List

:r2 From the array, it creates a heap where the smallest element serves as a root that we then remove

:r3 Demanding the minimum time complexity is O (N^3)

:r4 It cannot sort data on the spot, it requires extra memory for the size of the data file

:r2 ok 2

--

Insertion sort

:r1 Compares two adjacent elements in a sequence

:r2 It fails if the set is partially ordered

:r3 Stable, efficient on partially ordered sets, can sort data online (as they come to input)

:r4 Unstable, it is used for a large amount of data

:r3 ok 2

9 – 12

Mergesort

:r1 Unstable, cannot be paralleled, the extra memory required is O (1)

:r2 Stable, well parallelizable, worst time complexity is O (nlog n)

:r3 Only one pass is sufficient to compare any data set

:r4 It works by comparing the first and last element of the series

:r2 ok 2

--

Quicksort

:r1 Stable, able to order any data set in a single pass

:r2 It fails if the set is partially aligned

:r3 Versatile, the most value elements bubble at the end of the list

:r4 On average, it is the fastest known algorithm, unstable, uses pivot for sorting

:r4 ok 2

--

Selection sort

:r1 Algorithm complicated for implementation, but very fast, for ordering any data set is sufficient only one pass

:r2 Universal, local, unstable, suitable for small amounts of data

:r3 Requires additional memory of O (n^2)

:r4 The minimum and average time complexity is O (n)

:r2 ok 2

--

Sorting algorithms:

:r1 In the worst-case scenario, the time complexity of Heapsort and Mergesort O (nlog n)

:r2 Heapsort, Quicksort, and Insertion sorts are all unstable

:r3 Heapsort, Quicksort, Merge sort, and Insertion sort sorts data in-place (in-place, extra memory is O (1))

:r4 Bubble sort and Insertion sort ideally have time complexity constant - O (1)

:r1 ok 2

--

Pattern matching

:r1 It means finding a pattern P in a given sequence T (searching for a substring in a string)

:r2 Compare two objects based on their size

:r3 Finding repeating parts of the text

:r4 Finding the sequence that can encode the whole text

:r1 ok 2

--

Pattern matching - Brute force

:r1 Passes the text from right to left (back), not all positions

:r2 Compares P with T for all possible positions

:r3 It is a simple and time-consuming algorithm, its time complexity with longer text decreases

:r4 Removing characters from the text until the text T remains a string of the same length as P

:r2 ok 2

--

Pattern matching - KMP algorithm

:r1 It searches text from left to right, like Brute-Force algorithm makes all comparisons

:r2 If it encounters a disagreement, it moves the search for more than one letter, does not make all possible comparisons

:r3 It is a simple and time-consuming algorithm, its time complexity with longer text decreases

:r4 Compare strings from behind, shifts only by one character

:r2 ok 2

--

Trie

:r1 The node order at that level indicates the order of the letter in the word

:r2 If Trie is compressed, each node contains just one letter

:r3 The text-processing structure where each node has one letter or word

:r4 Cannot store the entire text

:r3 ok 2

--

Theory of graphs

:r1 If the graph is oriented, it contains at least one oriented edge (not all edges must be oriented)

:r2 The path is the sequence of vertices such that there are edges between consecutive vertices, and the vertices are not repeated

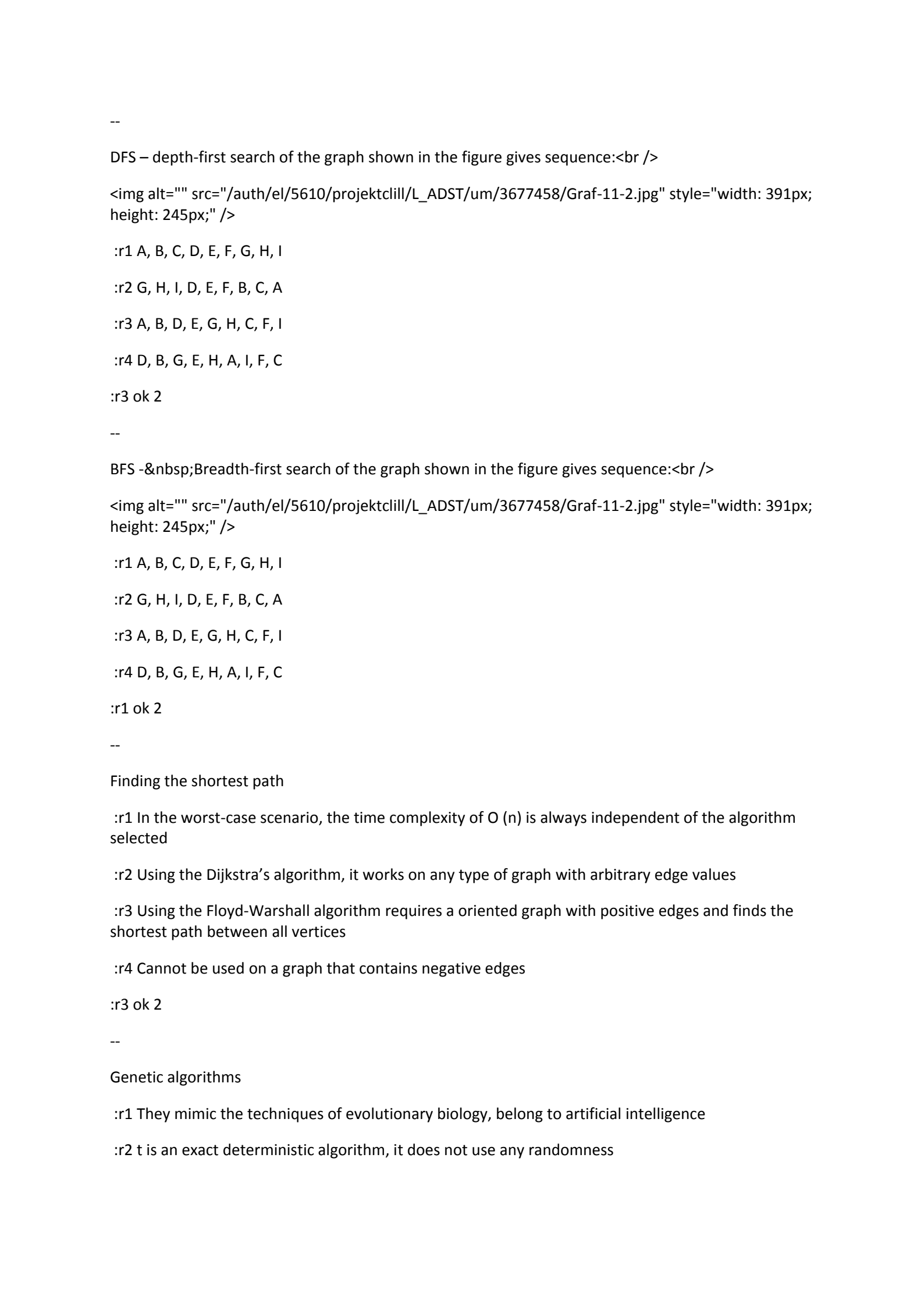
:r3 The edge is determined by more than two vertices, weight and direction

:r4 The graph is an ordered pair V, E, where V is a set of vertices and E is a set of edges

:r4 ok 2

--

DFS – depth-first search of the graph shown in the figure gives sequence:



:r1 A, B, C, D, E, F, G, H, I

:r2 G, H, I, D, E, F, B, C, A

:r3 A, B, D, E, G, H, C, F, I

:r4 D, B, G, E, H, A, I, F, C

:r3 ok 2

--

BFS - Breadth-first search of the graph shown in the figure gives sequence:

:r1 A, B, C, D, E, F, G, H, I

:r2 G, H, I, D, E, F, B, C, A

:r3 A, B, D, E, G, H, C, F, I

:r4 D, B, G, E, H, A, I, F, C

:r1 ok 2

--

Finding the shortest path

:r1 In the worst-case scenario, the time complexity of O (n) is always independent of the algorithm selected

:r2 Using the Dijkstra’s algorithm, it works on any type of graph with arbitrary edge values

:r3 Using the Floyd-Warshall algorithm requires a oriented graph with positive edges and finds the shortest path between all vertices

:r4 Cannot be used on a graph that contains negative edges

:r3 ok 2

--

Genetic algorithms

:r1 They mimic the techniques of evolutionary biology, belong to artificial intelligence

:r2 t is an exact deterministic algorithm, it does not use any randomness

:r3 The method of describing (coding) individuals does not affect the success or failure of solving a specific task

:r4 If they are triggered several times in a row, they always give the same result

:r1 ok 2

--

Parents selection

:r1 It is done so that parents become the individuals with the lowest fitness value

:r2 We use our fitness value when choosing our parents randomly

:r3 It is done to get the best individuals for the next generation

:r4 It does not affect the success or failure of a specific task

:r3 ok 2

--

Crossover

:r1 There is no way to crossover more than two individuals in one generation

:r2 Parents will exchange part of their genetic code, two new individuals are created at the one-point crossover

:r3 Every individual in each generation must become a parent

:r4 The parent always extinguishes and his / her place is taken by a descendant, this descendant will have the same fitness value as his / her parent

:r2 ok 2

--

Mutation

:r1 Describes a random change in the genome of an individual and it is done with very small probability

:r2 It never brings new features that would not exist in the original generation

:r3 Every single individual from every generation mutates at least in one place

:r4 e mutations always take place at one and the same place in the genome in the same generation

:r1 ok 2