[BASIC](#basics) =

[асимптотика и algorithm complexity](#algorithm_complexity)

устойчивость сортировки [link](https://ru.wikipedia.org/wiki/Устойчивая_сортировка)

NP-полные алгоритмы

приближенные алгоритмы

жадные алгоритмы

динамическое программирование

линейной программирование

[DATA STRUCTURES](#data_structures) =

[data on disk](#data_on_disk)

[multiple indexing](#multiple_indexing)

data types vs. structures

[priority queue / min\_ and max\_heap](#priority_queue_min_max_heap)

array

list

hash table

inverted index

trees

[tree and tree tasks](#tree_and_tasks)

[full, complete and perfect trees](#full_complete_perfect_trees)

[traversals](#traversals)

self\_balansed (binary, red-black)

[B and B+ trees](#b_tplus_trees)

cartesian tree

[randomize tree](#randomize_search_tree)

[lru-cache](#lru_cache)

hash-tables

[SORTING ALGORITHMS](#sorting_algorithms) (a lot of here - [link](https://habr.com/ru/post/335920/)) =

[selection sort](#selection_sort)

[quick sort](#quicksort)

[merge sort](#merge_sort)

[heap sort](#heap_sort)

summary table

[SEARCHING ALGORITHMS](#searching_algorithms) =

binary search

two pointers

[finding m elements](#finding_m_elements)

dfs, bfs – [here](#tree_and_tasks)

dijkstra

a-star [link](https://habr.com/ru/post/331192/)

bellman-ford

summary table

[OTHER ALGORITHMS](#other_algorithms) =

[karatsuba’s multiplying](#karatsubas)

k nearest neighbour

map\_reduce algorithm

bloom filter [link](https://habr.com/ru/company/otus/blog/541378/)

diffie–hellman crypto

[levenstein distance](#levenst_distance)

[catalan number](#cataln_number)

dynamic [link](https://www.youtube.com/watch?v=ey4_Fm1os54) [link](https://habr.com/ru/post/191498/)

С++ STL =

containers summary table

algorithms summary table

**BASICS**

more about: …

**\*\*\*\*\*** **ALGORITHM COMPLEXITY \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

АСИМПТОТИКА = характер изменения функции при изменении аргумента

O = обозначение для сложности алгоритма. Сложность означает время выполнения алгоритма при всех прочих равных и расчитывается в зависимости только от количества элементов, над которыми производится действие алгоритма.

Пусть есть накоторый набор чисел, где n = количество элементов,

тогда следующая запись:

f(n) = O(g(n))

означает, что сложность некоторого алгоритма f(n) при увеличении количества элементов не будет превышать сложности алгоритма g(n) умноженной на некоторую константу.

Основные классы сложности:

O(1) = константная

O(log(n)) = логарифмическая

O(n) = линейная

O(n \* log(n)) = квазилинейная

O(n^m) = полиномиальная

O(2^n) = экспоненциальная

Как правило, оценка дается по двум параметрам:

средняя время, худшее время

СЛОЖЕНИЕ АСИМПТОТИК

1. если известны обе асимптотики, то большая поглащает меньшую

2. если неизвестно их соотношение, то сложность сумиируется

O(n) + O(log(n)) = O(log(n))

O(X) + O(Y) = O(X + Y), если X ? Y

УМНОЖЕНИЕ АСИМПТОТИК

1. умножение асимптотик описывается произведением

2. умножение на константу дает ту же асимптотику

константа не имеет значения при сравненнии оразных асимптотик

O(n) \* O(log(n)) = O(n \* log(n))

с \* O(n) = O(n)

**\*\*\*\*\* TIME EVALUATION \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

ОТОБРАЖЕНИЕ АСИМПТОТИКИ ВО ВРЕМЯ

1. вычислить асимптотику

2. рассмотреть худший случай

3. вычислить количество операций при максимальном n

4. умножить на константу ~ 10-50

5. считать, что за одну секунду выполняется 10^9 операций

**DATA STRUCTURES**

more about: …

**\*\*\*\*\*** **DISK STRUCTURE and HOW DATA is STORED on disk \*\*\*\*\*\*\*\*\*\*\*\***

0 511

**TRACK**

**SECTOR**

**BLOCK**

most common block size = 512 bytes

**ADRESS = (TRACK, SECTOR) + OFFSET**

**OFFSET**

Data on disk dealing is about DBMS (database management system). Data cant be used directly in the disk: they should be previously loaded to the RAM. In RAM data presented as Data Structures.

**\*\*\*\*\*** **MULTIPLE INDEXING \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

INDEX

idx + ptr

16 bytes

32 i. / bl.

**32** bl. total

DATA

1000 objects of

128 bytes each

4 obj. / block

**250** bl. total

**250**

**search, total blocks =**

DATA

100 objects of

128 bytes each

4 obj. / block

25 bl. total

**33**

**search through idx blocks**

**ptr**

DATA

100 objects of

128 bytes each

4 obj. / block

25 bl. total

**3**

2\_INDEX

idx + ptr

16 bytes

32 i. / bl.

32 bl. total

1\_INDEX

i0 + ptr

i32 + ptr

…

i992 + ptr

32 of 16 bytes

**1** bl. total

**ptr**

**ptr**

**search**

**\*\*\*\*\* DATA TYPES and STRUCTURES \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

ADT (abstract data type) = абстрактный тип данных = описательная модель / набор требований, соответствие которым определяет, является ли конкретная структура тем или иным типом данных.

STRUCTURE = implementation of one or several-in-one ADT

[1]

**ADT** = priority queue

**STRUCTURE** = min\_heap, max\_heap

PRIORITY QUEUE - is\_empty( )

- peek( ) = считать первый элемент

- pop( ) = убрать первый элемент

- insert( ) with priority

MIN\_ , MAX\_HEAP

- complete tree, следовательно, может быть

- представлен в виде обыкновенного массива

- каждое поддерево также является min\_ , max\_heap

- соотношение узлов разных поддеревьев не регламентировано

- пусть idx = индекс узла, тогда:

parent = (idx – 1) // 2

left child = idx \* 2 + 1

right child = idx \* 2 + 2

shift\_down = просеивание вниз shift\_up = просеивание вверх

пусть n = количество элементов в куче, тогда h = высота = log(n)

тогда сложность формирования кучи:

начать с листьев

идти в сторону корня

использовать shift\_down

O(n) = 0 \* n/2 + 1 \* n/4 … + h \* 1

начать с корня

идти в сторону листьев

использовать shift\_up

O(nlog(n)) = 0 \* 1 … + h/2 \* n/4 + h \* n/2

**\*\*\*\*\*** **TREE and TREE TASKS** **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

СТРУКТУРА ДЕРЕВА

inner node (узел) = любой узел, который имеет потомков

root node (корень) = исходный узел (level 0)

leaf node (лист) = конечный узел без потоков

edge = ребро, соединяющее узлы

degree = количество дочерних узлов

breadth = количество листьев

level = сумма ребер к узлу, образующих уникальный путь

width = количество узлов конкретного level

depth = максимальное значение level

1. class TREE: # пример бинарного дерева
2. def \_\_init\_\_(self, v, l, r): # конструктор
3. self.value = v # значение узла
4. self.left = l # левое поддерево
5. self.right = r # правое поддерево
7. obj = TREE(1, TREE(2, None, None), TREE(3, None, None))

ТИПЫ **СБАЛАНСИРОВАННЫХ** ДЕРЕВЬЕВ

Сбалансированное дерево = дерево,

- каждое из поддеревьев которого сбалансировано

- высота левого и правого поддерева отличается максимум на 1

АВЛ-деревья, красно-черные деревья = гарантия сбалансированости

Декартовы деревья = вероятностность сбалансированности

**ДАЛЕЕ** будут описаны типичные задачи работы с деревьями.

Комментариии к ним находятся в этом документе, а пример кода для решения каждой из них – в отдельном файле с кодом.

SEARCH( )

Поиска ответа на вопрос, существует ли конкретный узел дерева.

Две стратегии реализации поиска:

- dfs\_search = поиск в глубину (depth-first search)

- bfs\_search = поиск в ширину (breadth-first search)

WHERE( )

Если ответа на вопрос о существовании узла недостаточно (например, необходимо найти несколько узлов по какому-то признаку и вернуть в качестве результата других характеристики этих же узлов), можно модифицировать функцию поиска.

SERIALIZE( )

Процесс перевода структуры данных в последовательность байтов. Например, представление дерева в виде строки.

STRUCTURALIZE( )

Процесс восстановления структуры данных из последовательности байт. Например, восстановление дерева их строки. Этот метод можно также использовать в качестве помощника конструктору.

FROM\_BOTTOM( )

Решения задач могут основываться как на циклах, так и на рекурсии. Иногда необходимо производить поиск от корня к листьям – иногда, наоборот, от листьев к корню. Эта функция демонстрирует подход, как с помощью рекурсии можно добраться сначала до листьев и лишь после этого, при обратном раскручивании, выполнять к-л действия.

LCA( )

Lowest common ancentor = задача поиска узла, являющегося ближайшим родительским узлом для двух других узлов.

ALLND( )

All nodes distance k = задача поиска узлов, удаленных от заданного на расстояние k ребер. Для решения необходимо учитывать также и родителей. В классическом варианте нельзя дважды проходиться по одному и тому же узлу (например, при k = 2 не допускается перейти к соседнему узлу и затем снова вернуться в исходный).

**\*\*\*\*\*** **FULL, COMPLETE and PERFECT TREES** **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

FULL = дерево, в котором у каждого из узлов:

- либо **dergee** потомков

- либо совсем нет потомков

COMPLETE = дерево, которое заполнено последовательно (например, в котором заполнение производилось сверху вниз и слева направо – если в таком дереве появляется узел, нарушающий этот порядок заполнения, дерево перестает быть complete)

PERFECT = дерево, в котором на каждом существующем уровне каждый из узлов имеет ровно **dergee** потомков. А на последнем уровне, никакой из узлов не имеет ни одного потомка.

**\*\*\*\*\*** **TRAVERSALS** **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**RECURSIVE TRAVERSALS** PRE\_ORDER = **node** – left - right

IN\_ORDER = left – **node** - right

POST\_ORDER = left – right - **node**

1

2

5

3

4

6

7

4

2

6

1

3

5

7

7

3

6

1

2

4

5

PRE\_ORDER

(same as DFS)

IN\_ORDER

POST\_ORDER

**NON-RECURSIVE TRAVERSALS** BREADTH-FIRST SEARH

**\*\*\*\*\* AVL \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*** **B and B+ TREES** **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

The main thing we want from multi-indexing is =

SELF-MANAGED MULTI-LEVEL INDEXING (tree)

B and B+ trees are self-balanced, but don't included into the category because of defferent balancing approach.

**M-WAY SEARCH TREES**

M-way search tree propeties:

1. every node might have M childs

2. every node might have M - 1 keys

3. all children keys are fit into the distance between parent’s

k1 k2 … km-1

1

2

M

...

**B TREES**

B tree = M-way tree which meets the following requirements

1. every node has minimum (M / 2) children

2. root node has at least 2 children

3. all leafs are at the same level

4. is filled from bottom to top

● refs to data are with their keys in corresponding nodes

Построение дерева, а также его само-балансировка осуществляются благодаря правилу **разбиения** переполненного узла на два и **поднятия** одного из значений в узел-родитель. Пример:

B tree (4-degree) = 10 20 30 40 50 60 70

10 20 30

40

10 20

40

30

50, 60

При переполнении узла = разбиение

- не\_крайнее значение наверх

- два новых узла на уровне

30 60

10 20

40 50

70

10 20

30

70

40 50 60

REFS to DATA are with their keys in corresponding nodes

30 60

10 20

40 50

70

**DATA**

**B+ TREES**

B+ tree = B tree in which

1. every key duplicates in the appropriate leaf

2. refs to data are NOT with their keys in corresponding nodes

3. refs to data are with their dupliceted keys in leafs

4. all leafs are linked as list

**30** **60**

10 20 **30**

40 50 **60**

70

**DATA**

**\*\*\*\*\* CARTESIAN TREE** **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*** **RANDOMIZE SEARCH TREE** **\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Рандомизированное дерево поиска – дерево, в операциях:

вставки элемента

удаления элемента

которого используется фактор случайности.

Этот фактор призван решить проблему сбалансированности дерева

ФОРМУЛИРОВКА ПРОБЛЕМЫ

Опасность добавления в дерево в том, что если на вход попадется последовательность из постоянно возрастающих / убывающих чисел, то дерево окажется сильно несбалансированным.

СПОСОБ РЕШЕНИЯ

При этом известно, что если все числа, подаваемые на вход, все-таки очень хорошо "перемешать", дерево окажется сбалансировано удовлетворительно:

сбалансированное бинарное дерево поиска f(n) = O(log2n)

рандомизированное дерево поиска f(n) = O(2\*log2n)

Иными словами, любое подаваемое на вход число может с некоторой вероятностью оказаться как родителем, так и ребенком. При этом вероятность оказаться родителем

= 1 / (n + 1) ,

где n количество узлов, уже добавленных в дерево.

Задача – смоделировать "перемешанность" входных данных.

Для этого необходимо иметь две функции:

обыкновенная вставка элемента 1 – 1 / (n + 1)

вставка элемента в корень 1 / (n + 1)

и применять их с соответствующими вероятностями

Вставка элемента в корень:

рекурсивно спуститься до места вставки

вставить элемент на его место, как в обычной вставке

выполнить поворот, чтобы узел оказался на месте своего корня

УДАЛЕНИЕ ЭЛЕМЕНТОВ

1. удалить элемент

2. выполнить рекурсивное слияние двух его поддеревьев

размер левого / правого поддерева = i / j

левый узел становится корнем с вероятностью = i / (i + j)

правый узел становится корнем с вероятностью = j / (i + j)

**\*\*\*\*\*** **LRU CACHE** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

LRU = least recently used = кэш, который хранит данные, обращение к которым происходит чаще всего.

ПРАВИЛА РАБОТЫ

1. запрос данных у функции

2. проверка, есть ли эти данные в кэше

если есть если нет

- переместить на первое место - удалить данные из хвоста

- вернуть ответ из кэша - посчитать новый ответ

- добавить на первое место

3. вернуть ответ из функции

helper(args)

CACHE c

if (c.ifexists(args))

return c.get(args)

else

**data** = evaluate(args)

c.push(data)

return data

main( )

**args** = …

result = helper(args)

evaluate(args)

**CACHE**

**HASH-TABLE**

**key** = **input args**

value = key position in list

**METHODS**

ifexists( )

get( )

push( )

NODE

tail = ptr to less recently

head = ptr to more recently

**data** = **cached data**

node\_most node\_i node\_j node\_least

**2LINKED LIST**

**SORTING ALGORITHMS**

more about: …

**\*\*\*\*\* SELECTION SORT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

1. def selection\_sort(A):
2. B = [ ] # empty array
3. while A:
4. B.append(min(A)) # add min from A to B
5. A.pop(index(min(A))) # remove min A from A

m f(n) = O(n^2)

w f(n) = O(n^2)

**\*\*\*\*\*** **QUICK SORT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Суть быстрой сортировки в том, чтобы

- выбрать опорный элемент

- поместить его в конечное положение

Особенности

- не является устойчивой

- не задействует дополнительную память

Порядок действий:

1. выбирается опорный элемент = pivot

2. swap( arr[pivot], arr[end-1] ) ; pivot = end - 1

3. i = -1, j = 0

4. if arr[j] < pivot : j += 1

5. if arr[j] < pivot : swap( arr[j], arr[i+1] ); i += 1; j += 1

6. if j == end : swap( arr[pivot], arr[i+1] ) ; pivot = i + 1

теперь таким образом

- выбранный элемент помещен на его конечное место

- pivot = индекс этого элемента

- все элементы слева = меньше опорного и не упорядочены

- все элементы справа = больше опорного и не упорядочены

7. повторить пункты 1-6 для левого … и правого подмассивов

**\*\*\*\*\*** **MERGE SORT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Суть сортировки слиянием в том, чтобы

- отсортировать две половины

- сравнивая левые элементы, сложить половины в новый массив

Особенности

- является устойчивой

- задействует дополнительную память

1. def merge(left, right):
2. # left is already sorted
3. # right as well
4. # temp = []
5. # temp += min(left[i], right[j])
7. def split(container):
8. pivot = len(container) // 2
9. if len(container) > 1:
10. left = split(container[:pivot:])
11. right = split(container[pivot::])
12. else: return container
13. return merge(left, right)

**\*\*\*\*\*** **HEAP SORT \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Суть в постоянном swap(root, last) в структуре данных "куча"

дано : массив arr = [3, 4, 1, 5, 2]

сделать : отсортировать его по возрастанию

Порядок действий:

1. idx = arr.size( ) – 1 = 4

2. преобразуем arr в max\_heap = [5, 3, 1, 4, 2]

3. swap(root, last) = swap(arr[0], arr[idx]) = [2, 3, 1, 4, 5]

4. –-idx;

5. if (idx >= 0) повторить шаги 2 - 4

**SUMMARY TABLE**

**SEARCHING ALGORITHMS**

more about: …

**\*\*\*\*\* TWO PINTERS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Суть в использовании двух указателей на элементы контейнера:

- один последовательно перебирает все элементы

- второй перемещается по собственному правилу

Пример задачи, которая решается методом двух указателей:

дано : строка str = "abcbacadefegdfhijhkli"

найти : разбить строку так, чтобы ни одна буква не входила

в две подстроки: abcbaca – defegde – hijhklij

Порядок действия:

1. ptr\_1 перебирает все элементы
2. ptr\_2 = последнее вхождение элемента ptr\_1
3. if ptr\_1 == ptr\_2 : подстрока найдена
4. if ptr\_2 == str.end( ) – 1 :

последняя подстрока найдена

конец алгоритма

**a** b c b a c **a**

**1 2**

a **b** c **b** a c a

**1 2**

a b **c** b a **c** a

**1 2**

a b c **b** a c a

**1 2**

a b c b **a** c **a**

**1 2**

a b c b a **c** a

**1 2**

**ptr\_1 == ptr\_2**

**d** e f e g **d** f

**1 2**

d **e** f **e** g d f

**1 2**

d e **f** e g d **f**

**1 2**

d e f **e** g d f

**1 2**

d e f e **g** d f

**1 2**

d e f e g **d** f

**1 2**

**ptr\_1 == ptr\_2**

**h** i j **h** k l i

**1 2**

h **i** j h k l **i**

**1 2**

**ptr\_2 == str.end( ) - 1**

**\*\*\*\*\*** **FINDING M ELEMENTS \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Суть в использовании стурктуры данных "куча"

дано : неотсортированный массив arr, n = 100

найти : m наибольших элементов

Алгоритм имеет смысл для малых значений m

Порядок действий:

1. кладем первые m элементов массива arr в min\_heap

2. итерируемся по оставшимся элементам массива arr

3. сравниваем каждый элемент el с корнем кучи min\_heap

4. if el > root : root = el; shift\_down(root)

Если необходимо, можно отсортировать получившуюся min\_heap. Тогда алгоритм поиска будет доработан до алгоритма частичной сортировки = partial sort.

**OTHER ALGORITHMS**

more about: …

**\*\*\*\*\* KARATSUBA'S MULTIPLYING \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Рекуррентный алгоритм для умножения двух чисел

Сложность обычного умножения f(n) = O(n^2) = O(n^log24)

Сложность метода Карацубы f(n) = O(n^log34)

Пример: 1144 \* 7788 , n = 4

11 = a , 44 = b , 77 = c , 88 = d

1144 = a \* 10^(n/2) + b

7788 = c \* 10^(n/2) + d

пусть 10^(n/2) = x, тогда

aabb \* ccdd = (ax + b)(cx + d) = **ac** \* x^2 + (**ad** + **bc**) \* x + **bd**

сложность O(n^log24) так как умножение производится **четыре раза**

(ad + bc) = (a + b)(c + d) – ac – bd, при этом **ac** и **bd** уже есть

aabb \* ccdd = **ac** \* x^2 + (**(**a + b)(c + d**)** – ac – bd) \* x + **bd**

сложность O(n^log23) так как умножение производится **три раза**

1. def mult(x, y):
2. xl = len(str(x))
3. yl = len(str(y))
4. if xl == 1 or yl == 1: return x \* y
5. half = max(xl, yl) // 2
6. n = 10\*\*half
7. a = int(x // n) ; b = int(x % n)
8. c = int(y // n) ; d = int(y % n)
9. m0 = mult(a, c)
10. m1 = mult(a + b, c + d)
11. m2 = mult(b, d)
12. m = n\*\*2 \* m0 + n \* (m1 - m0 - m2) + m2
13. return m

**\*\*\*\*\*** **LEVENSTEIN DISTANCE \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Алгоритм вычисления наименьшего количества изменений, которые необходимо внести в одну строку, чтобы привести ее к другой.

Дано: строка A длина I

строка B длина J

Допустимые операции: удалить элемент из строки

вставить элемент в строку

заменить элемент

ЛОГИКА РЕШЕНИЯ

Пусть мы сравниваем две подстроки Ai и Bj. Каждый раз, когда мы находим несоответствие этих строк, мы:

- применяем одну из допустимых операций

- увеличиваем счетчик расстояния на 1

- сокращаем длину одной (обеих строк в случае замены) на 1

Пусть мы приводим строку A к строке B. Тогда примеры операций:

Ai Bj счетчик итог

**⤬**

удаление xxxA xxx +1 A(i-1) = xxx A vs Bj

**⤬**

вставка xxx xxxB +1 Ai vs B(j-1) = xxx B

замена xxxA xxxB +1 A(i-1) vs B(j-1)

Остановка сравнения происходит, если:

1) либо в какой-то момент Ai == Bj. Тогда текущий счетчик и есть ответ, так как оставшиеся подстроки совпадают

2) либо один из коэффициентов i or j становится равен 0. Значит, одна из построк теперь пустая и чтобы привести подстроки друг к другу необходимо произвести вставку / удаление ровно столько раз, сколько элементов осталось в непустой подстроке. Следовательно, конечный ответ = счетчик + ненулевой коэффициент.

РЕШЕНИЕ ДИНАМИЧЕСКИМ ПРОГРАММИРОВАНИЕМ

A = oxxo , I = 4 , подстрока i

B = zoo , J = 3 , подстрока j

Сначала = создать матрицу M размером (I + 1) x (J + 1)

0 1 2 3 4 i, если j придет к 0

1 x x x x

2 x x x x

3 x x x x

j, если i придет к 0

Заполнять эту матрицу начиная к клетки [1][1]

- последовательность = поэлементно строка за строкой

- правило:

если A[i] равно B[j] то M[i][j] = M[i-1][j-1]

иначе M[i][j] = 1 (так как нужно изменение)

+ min ( M[ i ][j-1]

M[i-1][ j ]

M[i-1][j-1] ) (кратчайший путь)

Конечный ответ будет находиться в M[I][J]

1. def levenstein(A, B):
2. # create dynamic matrix
3. M = [[ i + j if not i \* j else 0 for j … ] for i … ]
4. # dynamic matrix traversal
5. for i in range(1, len(A) + 1):
6. for j in range(1, len(B) + 1):
7. if A[i - 1] == B[j - 1]: M[i][j] = M[i - 1][j - 1]
8. else: M[i][j] = 1 + min( … )
10. return M[len(A)][len(B)]

1 2 3 4

A = **o x x o**

B = **z** o o M

**0** 1 2 3 4 [1][1] = [o][z] = 1 + min (0) = 1

1 **1** **2** **3** **4** [2][1] = [x][z] = 1 + min (1) = 2

2 x x x x [3][1] = [x][z] = 1 + min (2) = 3

3 x x x x [4][1] = [o][z] = 1 + min (3) = 4

1 2 3 4

A = **o x x o**

B = z **o** o M

0 1 2 3 4 [1][2] = [o][o] = equal = 1

**1** 1 2 **3** 4 [2][2] = [x][o] = 1 + min (1) = 2

2 **1** **2** **3** **3** [3][2] = [x][o] = 1 + min (2) = 3

3 x x x x [4][2] = [o][o] = equal = 3

1 2 3 4

A = **o x x o**

B = z o **o** M

0 1 2 3 4 [1][3] = [o][o] = equal = 2

1 1 2 3 4 [2][3] = [x][o] = 1 + min (1) = 2

**2** **1** 2 **3** 3 [3][3] = [x][o] = 1 + min (2) = 3

3 **2** **2** **3** **3** [4][3] = [o][o] = equal = 3

**\*\*\*\*\*** **CATALAN NUMBER \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Числа из некоторых задач на комбинаторику:

подсчет количества

- уникальных бинарных деревьев поиска для n чисел

- монотонных путей в одной из половин квадрата n на n

- треугольников в выпуклом многоугольнике n + 2 сторон

Общая формула числа: Сn = (2n)! / (n! \* (n+1)!)

В случае рекуррентного подхода считатется, что C0 = С1 = 1

Визуализация рекуррентного подхода:

C(n)

F(1, n) + F(2, n) + ... + F(i, n) i = от 1 до n

C(0) \* C(n-1) ... ... C(i-1) \* C(n-i)

ПРИМЕР ДЛЯ ДЕРЕВА n = 2

lst = [1,2] C(2) = F(1, 2) + F(2, 2)

21

1

1

2

C(0) \* C(1) C(1) \* C(0)

ПОСЛЕДОВАТЕЛЬНОСТЬ РАССУЖДЕНИЙ:

Задача должна быть разбита на части: чтобы получить конечный ответ, нужно суммировать ответы для каждого из случаев, когда **в корень помещается очередное число** последовательности.

Как только мы выделяем такую задачу, она может быть разбита на две части – так как все оставшиеся числа могут оказаться либо только слева от выбранного, либо только справа. **Решаем исходную задачу для каждого из поддеревьев**.

F(i, n) = очередное число, помещенное в корень

С(j) = подзадача, эквивал. очередному левому/правому поддереву