Задача № 2/5

1. Напишите алгоритмы самых быстрых (по тикам процессора), на ваш взгляд, сортировок массивов и списков. Объясните свой выбор, в случае затруднения возьмите случайный алгоритм с логарифмической сложностью.

def mergeSort(nlist):

    print("Splitting ",nlist)

    if len(nlist)>1:

        mid = len(nlist)//2

        lefthalf = nlist[:mid]

        righthalf = nlist[mid:]

        mergeSort(lefthalf)

        mergeSort(righthalf)

        i=j=k=0

        while i < len(lefthalf) and j < len(righthalf):

            if lefthalf[i] < righthalf[j]:

                nlist[k]=lefthalf[i]

                i=i+1

            else:

                nlist[k]=righthalf[j]

                j=j+1

            k=k+1

        while i < len(lefthalf):

            nlist[k]=lefthalf[i]

            i=i+1

            k=k+1

        while j < len(righthalf):

            nlist[k]=righthalf[j]

            j=j+1

            k=k+1

    print("Merging ",nlist)

nlist = [14,46,43,27,57,41,45,21,70]

mergeSort(nlist)

print(nlist)

1. Проанализируйте алгоритм, протестировав его работу на нескольких массивах и списках, сравните результаты работы с двумя другими случайно выбранными алгоритмами. В качестве результата представьте таблицу, генерирующуюся во время сортировок, с количеством перестановок и (желательно) временем работы программы, а также количеством элементов в массиве/списке.
2. Реализуйте последовательный поиск элемента в массиве/списке.

def sequential\_search(alist, item):

    position = 0

    while position < len(alist):

        if alist[position] == item:

            return True

        position = position + 1

    return False

testlist = [1, 2, 32, 8, 17, 19, 42, 13, 0]

sequential\_search(testlist, 50)

1. Реализуйте поиск на основе хеша.

import pprint

class Hashtable:

    def \_\_init\_\_(self, elements):

        self.bucket\_size = len(elements)

        self.buckets = [[] for i in range(self.bucket\_size)]

        self.\_assign\_buckets(elements)

    def \_assign\_buckets(self, elements):

        for key, value in elements:

            hashed\_value = hash(key)

            index = hashed\_value % self.bucket\_size

            self.buckets[index].append((key, value))

    def get\_value(self, input\_key):

        hashed\_value = hash(input\_key)

        index = hashed\_value % self.bucket\_size

        bucket = self.buckets[index]

        for key, value in bucket:

            if key == input\_key:

                return(value)

        return None

    def \_\_str\_\_(self):

        return pprint.pformat(self.buckets)

if \_\_name\_\_ == "\_\_main\_\_":

     capitals = [

        ('France', 'Paris'),

        ('United States', 'Washington D.C.'),

        ('Italy', 'Rome'),

        ('Canada', 'Ottawa')

    ]

hashtable = Hashtable(capitals)

print(f"The capital of Italy is {hashtable.get\_value('Italy')}")

1. (по желанию) Реализуйте бинарное дерево поиска.

class BinaryTreeNode:

  def \_\_init\_\_(self, data):

    self.data = data

    self.leftChild = None

    self.rightChild=None

def insert(root,newValue):

    if root is None:

        root=BinaryTreeNode(newValue)

        return root

    if newValue<root.data:

        root.leftChild=insert(root.leftChild,newValue)

    else:

        root.rightChild=insert(root.rightChild,newValue)

    return root

def search(root,value):

    if root==None:

        return False

    elif root.data==value:

        return True

    elif root.data <value:

        return search(root.rightChild,value)

    else:

        return search(root.leftChild,value)

root= insert(None,15)

insert(root,10)

insert(root,25)

insert(root,6)

insert(root,14)

insert(root,20)

insert(root,60)

print(search(root,14))

print(search(root,60))