**Министерство науки и высшего образования РФ**

**ФГБОУ ВО «РХТУ им. Д. И. Менделеева»**

**ОТЧЕТ**

ПО ЛАБОРАТОРНОЙ РАБОТЕ №2

**ВЫПОЛНИЛ:** студент группы МК-20 Краснов М.В.

**ПРОВЕРИЛ:** к.т.н. Красильников И.В.

**Москва**

**2021**

Постановка задачи

Даны 2 алгоритма сортировки: сортировка слиянием и быстрая сортировка, каждый из которых нужно реализовать как рекурсивный и как итерационный.

quickSort\_rec – классический алгоритм быстрой сортировки в рекурсионном виде

quickSort\_iter – перестройка алгоритма в итерационный вид

mergesort\_rec – алгоритм сортировки вставками в рекурсионном виде

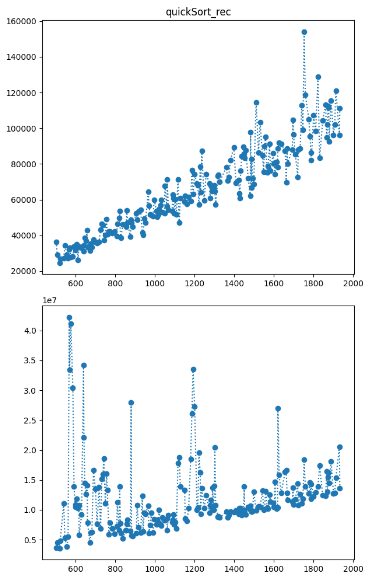
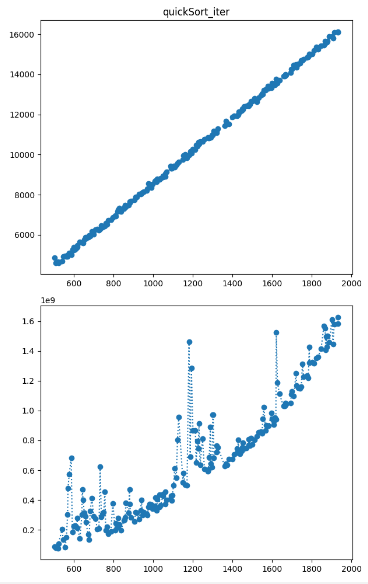
merge\_iter - алгоритм сортировки вставками в итерационном виде

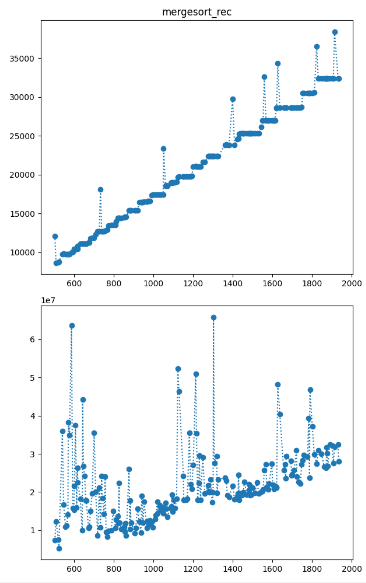
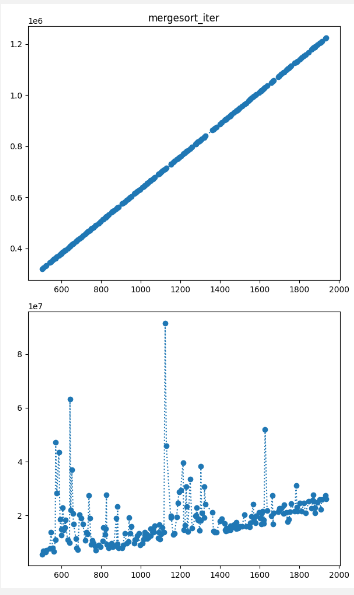
На каждой итерации исследования создается список (массив) простых чисел размером n элементов, распределенных не по порядку. Производится замер затрат памяти и времени исполнения для каждой реализации.

Всего производится 100 итераций с вычислением среднего времени и средних пиковых затрат памяти (с учетом затрат на хранение результатов исследовния).

Результаты исследования.

При n0=500, nmax = 1900, получены следующие рузультаты:

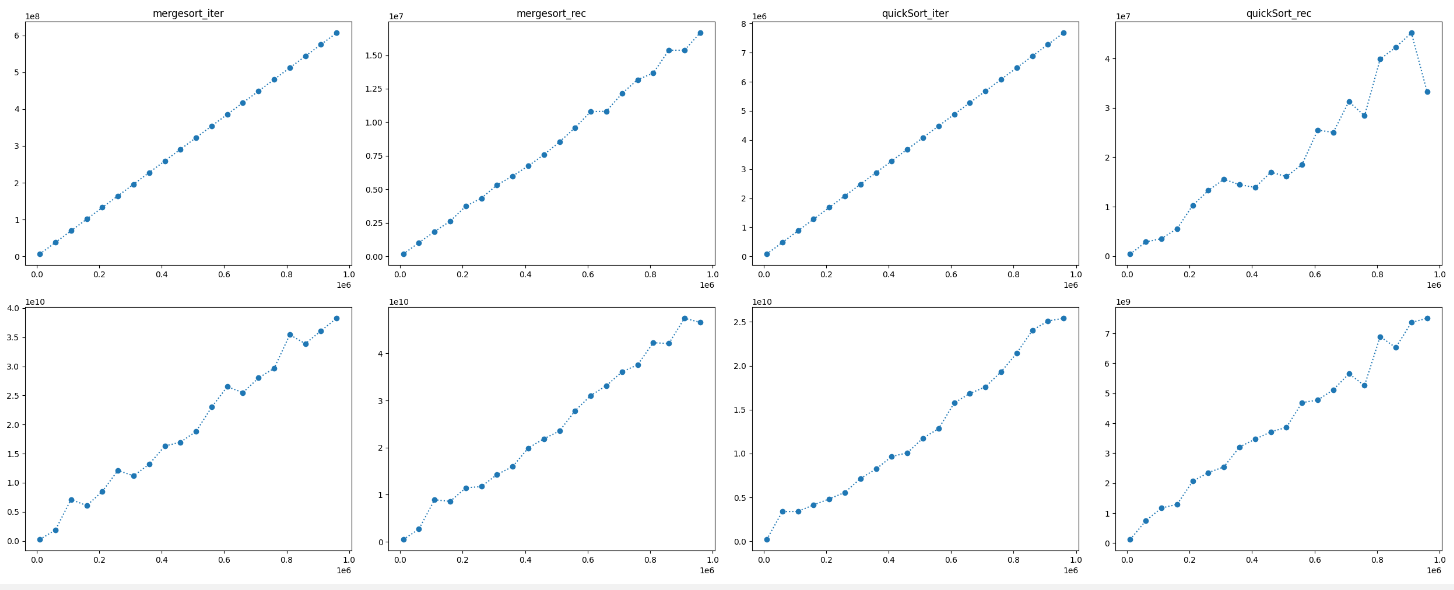
 

Видно, что по затратам времени быстрая сортировка рекурсией на два порядка опережает показатели быстрой сортировки итерацией. В то же время, итерационный вариант на порядок экономней по памяти.

Сортировка вставками с помощью рекурсии на 2 порядка выгодней по памяти, чем итерационный вариант. Достоверных различий по времени двух вариантов исполнения выявлено не было.

При n0=10000, nmax = 1000000 (шаг 10000), картина практически не меняется.



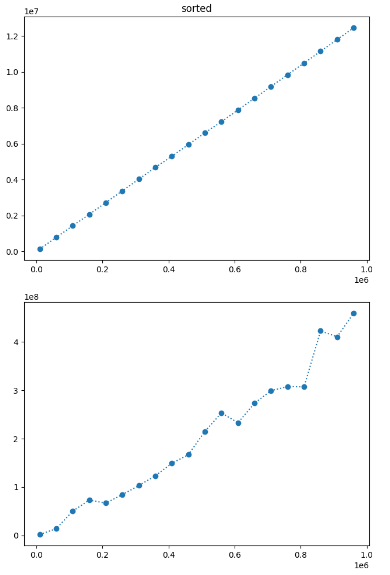
Выводы:

Рекурсионные варианты обоих алгоритмов показали более эффективное использование машинного времени на исполнение. Алгоритм быстрой сортировки ожидаемо выгодней по используемой памяти в виде итерационного варианта. Однако, рекурсионный алгоритм сортировки вставками оказался выгодней по памяти, чем итерационный вариант. Думаю, это связано с тем, что сам алгоритм сортировки вставками изначально имеет свою «идеальную» реализацию с помощью рекурсии и его итерационная версия не эфективна.

Кроме того, итерационный алгорим быстрой сортировки оказался более эффективным по паямяти, чем встроенный оптимизированный сложный алгоритм сортировки языка (sorted()) , а рекурсионный алгоритм сортировки вставками лишь смог приблизиться по порядку к показателям встроенного алгоритма.

По скорости же ни один из представленных для исследования алгоритмов не смог приблизиться к встроенному, а рекурсионная быстрая сортировка – на целый порядок отставала от него, хоть и была лидером среди других алгоритмов.

Это связан с тем, что встроенная функция определяет наилучшее решение для осуществления сортировки в зависимости от размера списка и типа его элементов и оптимизирует скорость сортировки сложными механизмами.



Код программы

*import* tracemalloc

*import* time

*import* pandas *as* pd

*import* numpy *as* np

*import* matplotlib.pyplot *as* plt

*import* sys

*import* math

*import* random

*import* sympy

def prepare\_n(*nLimit*,*nstart*):

    m = nstart

    n\_primes = []

    i=0

*while* i < nLimit:

        n = sympy.nextprime(m)

        n\_primes.append(n)

        m = n

        i=i+1

*return* n\_primes

*#Функция разделения для нужд "быстрой сортировки"*

def partition(*arr*, *low*, *high*):

    i = (low - 1)  *# индекс меньшего элемента*

    pivot = arr[high]

*for* j *in* range(low, high):

*# Если текущий элемент меньше*

*# чем или равно Pivot*

*if* arr[j] <= pivot:

*# индекс приращения*

*# меньший элемент*

            i += 1

            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]

*return* (i + 1)

*# Основная функция, реализующая QuickSort*

*# Классический рекурсивный вариант*

def quickSort\_rec(*nums*):

*if* len(nums) <= 1:

*return* nums

*else*:

        q = random.choice(nums)

    l\_nums = [n *for* n *in* nums *if* n < q]

    e\_nums = [q] \* nums.count(q)

    b\_nums = [n *for* n *in* nums *if* n > q]

*return* quickSort\_rec(l\_nums) + e\_nums + quickSort\_rec(b\_nums)

*# Рекурсивный алгоритм для сортировки слиянием*

def merge\_rec(*left*, *right*):

*if* not len(left) or not len(right):

*return* left or right

    result = []

    i, j = 0, 0

*while* (len(result) < len(left) + len(right)):

*if* left[i] < right[j]:

            result.append(left[i])

            i += 1

*else*:

            result.append(right[j])

            j += 1

*if* i == len(left) or j == len(right):

            result.extend(left[i:] *or* right[j:])

*break*

*return* result

def quickSort\_iter(*arr*, *l*=False, *h*=False):

*if* h==False:

        l=0;h=len(arr)-1

*# Создать вспомогательный стек*

    size = h - l + 1

    stack = [0] \* (size)

*# инициализировать вершину стека*

    top = -1

*# поместь начальные значения l и h в стек*

    top = top + 1

    stack[top] = l

    top = top + 1

    stack[top] = h

*# Продолжать забирать из стека, пока не пусто*

*while* top >= 0:

        h = stack[top]

        top = top - 1

        l = stack[top]

        top = top - 1

*# отсортированный массив*

        p = partition(arr, l, h)

*# Если есть элементы на левой стороне оси*

*if* p - 1 > l:

            top = top + 1

            stack[top] = l

            top = top + 1

            stack[top] = p - 1

*# Если есть элементы на правой стороне оси*

*if* p + 1 < h:

            top = top + 1

            stack[top] = p + 1

            top = top + 1

            stack[top] = h

def mergesort\_rec(*list*):

*if* len(list) < 2:

*return* list

    middle = math.ceil(len(list) / 2)

    left = mergesort\_rec(list[:middle])

    right = mergesort\_rec(list[middle:])

*return* merge\_rec(left, right)

*# Итеративная функция сортировки слиянием*

*from* collections *import* deque

def atomize(*l*):

*return* deque(

        map(

            lambda *x*: deque([x]),

            l *if* l *is* *not* None *else* []

        )

    )

def merge\_iter(*l*, *r*):

    res = deque()

*while* (len(l) + len(r)) > 0:

*if* len(l) < 1:

            res += r

            r = deque()

*elif* len(r) < 1:

            res += l

            l = deque()

*else*:

*if* l[0] <= r[0]:

                res.append(l.popleft())

*else*:

                res.append(r.popleft())

*return* res

def mergesort\_iter(*l*):

    atoms = atomize(l) *# O(n)*

*while* len(atoms) > 1: *# O(n - 1)*

        atoms.append(merge\_iter(atoms.popleft(), atoms.popleft()))

*return* list(atoms[0])

def runAlg(*alg*, *tracer*, *arges*, *repeats*):

    n=arges[0]

    array = np.random.randint(200000, *size*=(n)).tolist()

    tracer.clear\_traces()

    t1 = time.perf\_counter\_ns()

*for* i *in* range(0, repeats):

        alg(array)

    t2 = time.perf\_counter\_ns()

    size, peak = tracemalloc.get\_traced\_memory()

*return* {"time": (t2 - t1) / repeats, "size": size, "peak": peak, }

def main():

    tracemalloc.start()

    alges = {

        "mergesort\_iter": mergesort\_iter,

        "mergesort\_rec": mergesort\_rec,

        "quickSort\_iter": quickSort\_iter,

        "quickSort\_rec": quickSort\_rec,

    }

    alg\_arges = ['n']

    repeats = 10

    nlimit = 200;

    nstart = 500;

    results = {}

*for* alg *in* alges.keys():

        results[alg] = {'time': [], 'peak': []}

*for* arg *in* alg\_arges:

            results[alg][arg] = []

    n\_primes = prepare\_n(nlimit,nstart)

*for* n *in* n\_primes:

*for* alg *in* alges.keys():

            it = runAlg(alges[alg], tracemalloc, [n], repeats)

*# print(alg)*

*# print(f"n={n}")*

*# print(array)*

*# print(it)*

*# print("----------------------------------")*

            listSize = sys.getsizeof(results)

            results[alg]['n'].append(n)

            results[alg]['time'].append(it['time'])

            results[alg]['peak'].append(it['peak'] - listSize)

    dFResults = {}

*for* alg *in* alges.keys():

        dFResults[alg] = pd.DataFrame(results[alg])

    fig = plt.figure(*figsize*=(25, 10))

    axes = []

    axPlace = 0

*for* alg *in* alges.keys():

        axes.append({})

        axPlace = axPlace + 1

        axes[-1] = fig.add\_subplot(240 + axPlace)

        x = dFResults[alg]['n']

        y = dFResults[alg]['peak']

        axes[-1].plot(x, y, ':o')

        axes[-1].set\_title(alg)

        axes.append({})

        axes[-1] = fig.add\_subplot(244 + axPlace)

        y = dFResults[alg]['time']

        axes[-1].plot(x, y , ':o')

    fig.show()

*for* alg *in* alges.keys():

        dFResults[alg].to\_excel(alg + '.xlsx')

*if* \_\_name\_\_ == '\_\_main\_\_':

    main()