**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №2**

**по дисциплине «Алгоритмы и структуры данных»**

Тема: **Реализация и исследование алгоритма сортировки TimSort.**

|  |  |  |
| --- | --- | --- |
| Студент гр. 3341 |  | Лодыгин И.А. |
| Преподаватель |  | Иванов Д.В. |

Санкт-Петербург

2024

## Цель работы

## Целью данной лабораторной работы является изучение и реализация алгоритма сортировки TimSort, который сочетает в себе методы сортировки слиянием и сортировкой вставками. Мы будем исследовать его применение на примере сортировки массива целых чисел по убыванию модуля элементов. Ожидается, что выполнение данного задания позволит лучше понять, как работают гибридные алгоритмы сортировки, а также даст возможность увидеть, как эти алгоритмы могут быть эффективно использованы для работы с массивами данных различного размера.

## Работа включает в себя несколько ключевых этапов. Во-первых, необходимо будет вычислить параметр min\_run, который определяет минимальную длину блока, на который массив будет разбит для последующей сортировки. Размер этого параметра будет изменяться в зависимости от размера массива, что обеспечит оптимизацию процесса. Затем массив будет разбит на блоки, каждый из которых будет отсортирован с использованием сортировки вставками. Это обеспечит частичную упорядоченность данных, которая будет полезна на следующем этапе.

## В итоге уже отсортированные блоки будут объединяться с использованием сортировки слиянием, при этом будет применен метод галопа. Он позволяет повысить эффективность слияния, и будет активирован при достижении определенного количества вставок подряд. В этом процессе также будет фиксироваться количество этих вставок для учета в отчетной документации. Задание

Реализация

Имеется массив данных для сортировки int arr[] размера n

Необходимо отсортировать его алгоритмом сортировки TimSort по убыванию модуля.

Так как TimSort - это гибридный алгоритм, содержащий в себе сортировку слиянием и сортировку вставками, то вам предстоит использовать оба этих алгоритма. Поэтому нужно выводить разделённые блоки, которые уже отсортированы сортировкой вставками.

Кратко алгоритм сортировки можно описать так:

Вычисление min\_run по размеру массива n (для упрощения отладки n уменьшается, пока не станет меньше 16, а не 64)

Разбиение массива на частично-упорядоченные (в т.ч. и по убыванию) блоки длины не меньше min\_run

Сортировка вставками каждого блока

Слияние каждого блока с сохранением инварианта и использованием галопа (галоп начинать после 3-х вставок подряд)

Исследование

После успешного решения задачи в рамках курса проведите исследование данной сортировки на различных размерах данных (10/1000/100000), сравнив полученные результаты с теоретической оценкой (для лучшего, среднего и худшего случаев), и разного размера min\_run. Результаты исследования предоставьте в отчете.

Для исследования используйте стандартный алгоритм вычисления min\_run и начинайте галоп после 7-ми вставок подряд.

Примечание:

Нельзя пользоваться готовыми библиотечными функциями для сортировки, нужно сделать реализацию сортировки вручную.

Сортировка должна быть устойчивой.

Обратите внимание на пример.

Формат ввода

Первая строка содержит натуральное число n - размерность массива, следующая строка содержит элементы массива через пробел.

Формат вывода

Выводятся разделённые блоки для сортировки в формате "Part i: \*отсортированный разделённый массив\*"

Затем для каждого слияния выводится количество вхождений в режим галопа и получившийся массив в формате

"Gallops i: \*число вхождений в галоп\*

Merge i: \*итоговый массив после слияния\*"

Последняя строчка содержит финальный результат сортировки массива с надписью "Answer: "

## Выполнение работы

## Для написания данной программы был реализован алгоритм сортировки, основанный на принципах Timsort, который комбинирует преимущества сортировки вставками и слияния.

## 1. Функция calculate\_minrun(size)

## Цель этой функции — вычислить минимальный размер подмассива (minrun), который будет использован для сортировки методом вставки. Она выполняется следующим образом:

## Изначально устанавливается переменная remainder равной 0.

## Цикл продолжается до тех пор, пока размер массива (size) больше или равен 16.

## Для каждой итерации прибавляется младший бит к remainder, и размер массива уменьшается вдвое.

## Возвращается сумма оставшегося размера и remainder.

## 2. Функция binary\_search(arr, target)

## Функция выполняет двоичный поиск, который используется в процессе слияния массивов при помощи «галлопирования». Описание работы:

## Устанавливаются начальные значения для левой и правой границ поиска.

## В цикле вычисляется средняя позиция. Если абсолютное значение элемента меньше абсолютного значения целевого элемента (target), границы сдвигаются.

## Возвращается конечная позиция, где должно происходить слияние.

## 3. Функция merge\_with\_gallop(left\_array, right\_array, merge\_index)

## Эта функция реализует слияние двух массивов с использованием «галлопирования» — оптимизации для ускоренного слияния последовательных элементов.

## Создаются индексы для отслеживания позиций в двух массивах и счетчики «галлопов».

## В цикле сравниваются элементы обоих массивов, и более крупные по абсолютному значению добавляются в результирующий массив.

## Если подряд встречается несколько элементов из одного массива, вызывается двоичный поиск для ускоренного добавления элементов (галлопирование).

## По завершении цикла остатки массивов добавляются в результат, который возвращается.

## 4. Функция merge\_stack(sorted\_blocks)

## Функция управляет процессом слияния отсортированных блоков, добавляя их в стек и объединяя, когда соблюдаются условия по длине блоков.

## Блоки добавляются в стек по одному.

## Если длины блоков соответствуют определённым критериям, вызывается функция слияния с галлопированием.

## Процесс продолжается до тех пор, пока в стеке не останется один окончательный объединенный массив.

## 5. Функция insertion\_sort(arr)

## Функция реализует сортировку вставками для небольших блоков, которая применяется на этапе разделения исходного массива.

## Для каждого элемента массива начиная со второго, функция перемещает его в отсортированную часть массива, сдвигая элементы влево, если их абсолютное значение меньше текущего элемента.

## 6. Функция timsort(arr)

## Это основная функция программы, реализующая полный процесс Timsort:

## Определяется размер массива и минимальный размер блока с помощью функции calculate\_minrun.

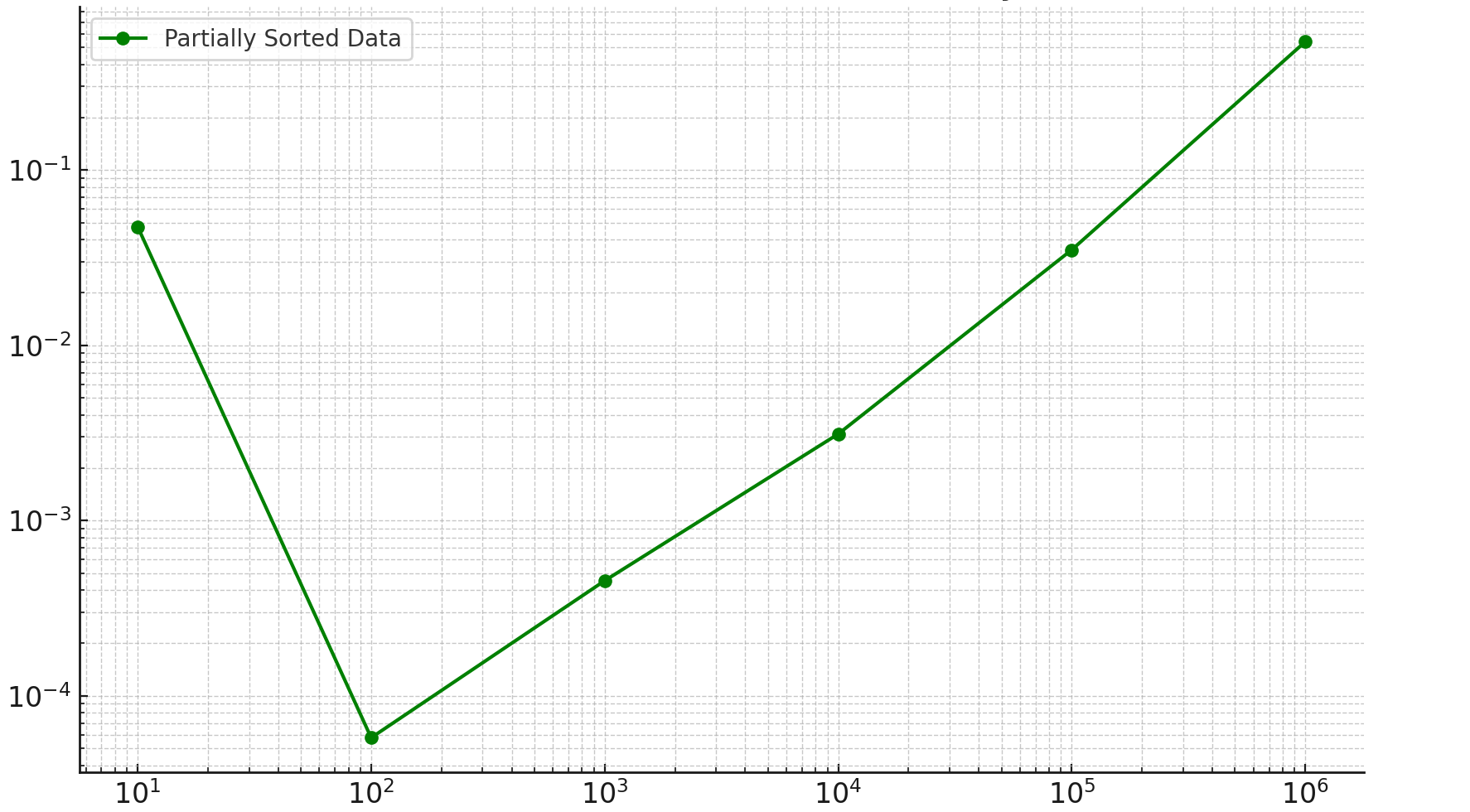
## Массив разделяется на блоки в зависимости от того, является ли последовательность элементов возрастающей или убывающей.

## Когда блок достигает размера minrun, выполняется его сортировка методом вставки.

## Отсортированные блоки добавляются в список и объединяются с помощью merge\_stack.

## Если список содержит один блок, он просто возвращается как результат. Тестирование

Результаты тестирования представлены на графике:



## На графике показана зависимость времени сортировки в секундах(вертикальная ось) от количества элементов(горизонтальная ось)

## Выводы

## В ходе выполнения лабораторной работы по исследованию алгоритма сортировки TimSort было установлено его высокое качество и эффективность. Благодаря комбинации сортировки вставками и слиянием, алгоритм продемонстрировал отличные результаты на частично упорядоченных данных, что делает его привлекательным для практического использования.

## Анализ производительности TimSort показал, что временная сложность алгоритма в среднем составляет O(n log n), что соответствует теоретическим ожиданиям. Он становится особенно эффективным при работе с большими массивами, где его производительность ощутимо превышает традиционные алгоритмы сортировки.

## Важно отметить роль параметра min\_run, который влияет на скорость сортировки. Оптимальный выбор этого параметра в зависимости от объема входных данных способствовал улучшению времени обработки и снижению количества операций. Применение метода галопа также оказалось полезным, сокращая затраты на слияние.

## Тестирование подтвердило устойчивость алгоритма, что является весомым преимуществом в задачах, где порядок элементов важен. В итоге, результаты работы могут быть использованы как основа для дальнейших исследований и внедрения алгоритмов сортировки в различные приложения, требующие быстрой обработки данных.

# Приложение А Исходный код программы

Название файла: main.py

def calculate\_minrun(size):

remainder = 0

while size >= 16:

remainder |= size & 1

size >>= 1

return size + remainder

def binary\_search(arr, target):

left, right = 0, len(arr) - 1

result\_position = right

while left <= right:

middle = left + (right - left) // 2

if abs(arr[middle]) < abs(target):

result\_position = middle - 1

right = middle - 1

else:

left = middle + 1

return result\_position

def merge\_with\_gallop(left\_array, right\_array, merge\_index):

merged\_result = []

left\_index, right\_index = 0, 0

left\_gallop\_count = 0

right\_gallop\_count = 0

total\_gallops = 0

while left\_index < len(left\_array) and right\_index < len(right\_array):

if abs(left\_array[left\_index]) >= abs(right\_array[right\_index]):

merged\_result.append(left\_array[left\_index])

left\_index += 1

left\_gallop\_count += 1

right\_gallop\_count = 0

if left\_gallop\_count >= 3:

gallop\_end\_index = binary\_search(left\_array[left\_index:], right\_array[right\_index]) + left\_index

total\_gallops += 1

while left\_index <= gallop\_end\_index:

merged\_result.append(left\_array[left\_index])

left\_index += 1

left\_gallop\_count = 0

else:

merged\_result.append(right\_array[right\_index])

right\_index += 1

right\_gallop\_count += 1

left\_gallop\_count = 0

if right\_gallop\_count >= 3:

gallop\_end\_index = binary\_search(right\_array[right\_index:], left\_array[left\_index]) + right\_index

total\_gallops += 1

while right\_index <= gallop\_end\_index:

merged\_result.append(right\_array[right\_index])

right\_index += 1

right\_gallop\_count = 0

merged\_result.extend(left\_array[left\_index:])

merged\_result.extend(right\_array[right\_index:])

print(f"Gallops {merge\_index}: {total\_gallops}")

print(f"Merge {merge\_index}:", \*merged\_result)

return merged\_result

def merge\_stack(sorted\_blocks):

stack = []

merge\_step = 0

for block in sorted\_blocks:

stack.append(block)

while len(stack) >= 2:

if len(stack) == 2:

right\_block = stack.pop()

left\_block = stack.pop()

if len(left\_block) <= len(right\_block):

merged\_block = merge\_with\_gallop(left\_block, right\_block, merge\_step)

merge\_step += 1

stack.append(merged\_block)

else:

stack.append(left\_block)

stack.append(right\_block)

break

if len(stack) > 2:

right\_block = stack.pop()

middle\_block = stack.pop()

left\_block = stack.pop()

if len(left\_block) <= len(right\_block) + len(middle\_block) or len(middle\_block) <= len(right\_block):

if max(len(right\_block), len(left\_block)) == len(left\_block):

merged\_block = merge\_with\_gallop(middle\_block, right\_block, merge\_step)

merge\_step += 1

stack.append(left\_block)

stack.append(merged\_block)

else:

merged\_block = merge\_with\_gallop(middle\_block, left\_block, merge\_step)

merge\_step += 1

stack.append(merged\_block)

stack.append(right\_block)

else:

stack.append(left\_block)

stack.append(middle\_block)

stack.append(right\_block)

break

while len(stack) > 1:

if len(stack) == 2:

right\_block = stack.pop()

left\_block = stack.pop()

merged\_block = merge\_with\_gallop(left\_block, right\_block, merge\_step)

merge\_step += 1

stack.append(merged\_block)

if len(stack) > 2:

right\_block = stack.pop()

middle\_block = stack.pop()

left\_block = stack.pop()

if max(len(right\_block), len(left\_block)) == len(left\_block):

merged\_block = merge\_with\_gallop(middle\_block, right\_block, merge\_step)

merge\_step += 1

stack.append(left\_block)

stack.append(merged\_block)

else:

merged\_block = merge\_with\_gallop(middle\_block, left\_block, merge\_step)

merge\_step += 1

stack.append(merged\_block)

stack.append(right\_block)

return stack.pop()

def insertion\_sort(arr):

for i in range(1, len(arr)):

key = arr[i]

j = i - 1

while j >= 0 and abs(key) > abs(arr[j]):

arr[j + 1] = arr[j]

j -= 1

arr[j + 1] = key

return arr

def timsort(arr):

n = len(arr)

minrun = calculate\_minrun(n)

current\_block = []

sorted\_blocks = []

block\_index = 0

is\_increasing = is\_decreasing = True

i = 0

while i < n:

while i < n - 1 and abs(arr[i]) < abs(arr[i + 1]) and is\_increasing:

current\_block.append(arr[i])

is\_decreasing = False

i += 1

while i < n - 1 and abs(arr[i]) >= abs(arr[i + 1]) and is\_decreasing:

current\_block.append(arr[i])

is\_increasing = False

i += 1

current\_block.append(arr[i])

if is\_increasing:

current\_block.reverse()

while i < n - 1 and len(current\_block) < minrun:

i += 1

current\_block.append(arr[i])

insertion\_sort(current\_block)

sorted\_blocks.append(current\_block)

print(f"Part {block\_index}:", \*current\_block)

current\_block = []

is\_increasing = is\_decreasing = True

i += 1

block\_index += 1

if len(sorted\_blocks) > 1:

result = merge\_stack(sorted\_blocks)

else:

result = sorted\_blocks.pop()

return result