МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ

УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ «МИСИС»

ИНСТИТУТ КОМПЬЮТЕРНЫХ НАУК (ИКН)

Курсовая работа на тему «Алгоритм Timsort»

Выполнил: студент группы БИВТ-23-4
Дарьютин Даниил Денисович

Москва, 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Сортировка данных - одна из важнейших задач как в прикладном программировании, так и в теоретической информатике. Она лежит в основе множества процессов: от обработки данных в базах и отображения информации в пользовательских интерфейсах до внутренних механизмов работы операционных систем и поисковых сервисов. Эффективный алгоритм сортировки может существенно повлиять на общую производительность системы.

Среди множества известных подходов особое место занимает алгоритм Timsort. Он был разработан в 2002 году Тимом Петерсом специально для языка Python, а позже принят и в другие экосистемы, включая Java и Android. Это говорит о его эффективности и практической ценности.

Рис 1. Тим Петерсон

Алгоритм Timsort представляет собой гибрид двух классических методов - сортировки вставками и сортировки слиянием. Особенно хорошо он справляется с массивами, которые уже частично отсортированы - это довольно типичный случай в реальных задачах. Главным преимуществом Timsort считается его адаптивность и устойчивость. Он не просто сортирует, но ещё и анализирует структуру данных, подстраиваясь под неё

Класс задач:

- Сортировка списков/массивов объектов по одному или нескольким ключам
- Упорядочивание данных для дальнейшего поиска или отображения
- Форматирование отчетов или таблиц

Пример задачи:

Дан список сотрудников, каждый из которых представлен в виде пары (имя, возраст). Отсортируйте сотрудников по возрасту по возрастанию, сохраняя оригинальный порядок среди сотрудников с одинаковым возрастом.

ОСНОВНАЯ ЧАСТЬ

Ключевые этапы алгоритма Timsort

Алгоритм Timsort состоит из двух основных фаз: выделение и сортировка run’ов и слияние отсортированных run’ов. Работа алгоритма основана на наблюдении, что в реальных данных часто встречаются уже отсортированные участки - Timsort умеет эффективно использовать это свойство.

Timsort - это гибридный алгоритм сортировки, который объединяет преимущества сортировки вставками и сортировки слиянием. Он отличается адаптивностью, устойчивостью и оптимизирован для работы с реальными данными, которые часто бывают частично упорядоченными.

Алгоритм выполняется в несколько этапов, каждый из которых играет ключевую роль в обеспечении как высокой производительности, так и корректности сортировки.

Шаг 1. Определение минимального размера блока (minrun)

На начальном этапе определяется минимальный размер подмассива (run), который будет использоваться на следующих шагах алгоритма.

Число minrun определяется на основе длины исходного массива, исходя из следующих принципов:

- Оно не должно быть слишком большим, поскольку к подмассиву размера minrun будет в дальнейшем применена сортировка вставками, а она эффективна только на небольших массивах
- Оно не должно быть слишком маленьким, поскольку чем меньше подмассив - тем больше итераций слияния подмассивов придётся выполнить на последнем шаге алгоритма.
- Хорошо бы, чтобы N \ minrun было степенью числа 2 (или близким к нему). Это требование обусловлено тем, что алгоритм слияния подмассивов наиболее эффективно работает на подмассивах примерно равного размера.

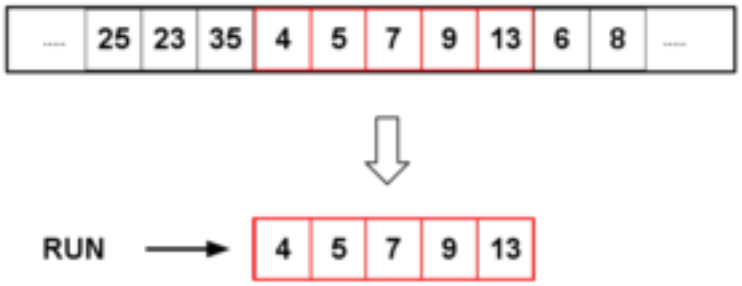

Рис 2. Пример run'а

Шаг 2. Поиск и подготовка run'ов

На данном этапе у нас есть входной массив, его размер n и вычисленное число minrun. Обратим внимание, что если данные изначального массива достаточно близки к случайным, то размер упорядоченных подмассивов близок к minrun. Но если в изначальных данных были упорядоченные диапазоны, то упорядоченные подмассивы могут иметь размер, превышающий minrun.

Шаги:

0. Указатель текущего элемента ставится в начало входного массива.
1. Начиная с текущего элемента, идет поиск во входном массиве упорядоченного подмассива run. По определению, в run однозначно войдет текущий элемент и следующий за ним. Если получившийся подмассив упорядочен по убыванию, то после вычисления run для текущего массива элементы переставляются так, чтобы они шли по возрастанию.
2. Если размер текущего run меньше minrun, тогда выбираются следующие за найденным подмассивом run элементы в количестве minrun-size(run). Таким образом, на выходе будет получен подмассив размером большим или равным minrun, часть которого (в лучшем случае - он весь) упорядочена.
3. К данному подмассиву применяем сортировку вставками. Так как размер подмассива невелик и часть его уже упорядочена - сортировка работает эффективно.
4. Указатель текущего элемента ставится на следующий за подмассивом элемент.
5. Если конец входного массива не достигнут - переход к шагу 1.

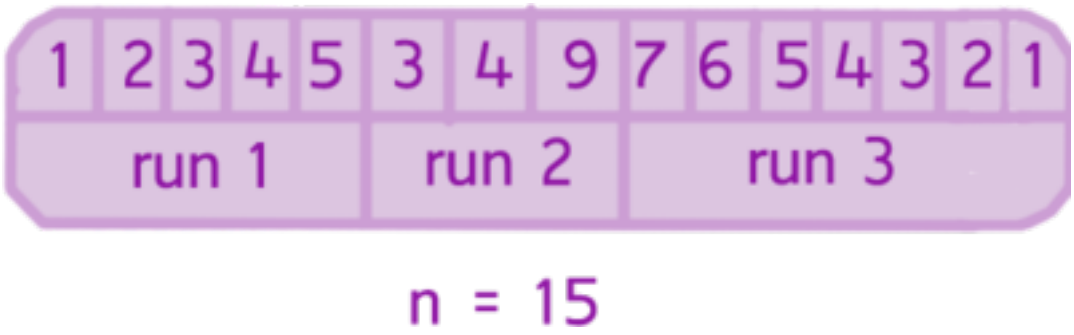

Рис 3. Разбиение массива на run'ы

Шаг 3. Слияние с использованием Galloping Merge

После того как массив был разбит на отсортированные подмассивы (run'ы), начинается этап их последовательного слияния в один отсортированный массив. Данный шаг реализует ключевую идею алгоритма Merge Sort, но с важной оптимизацией, заимствованной из оригинальной реализации Timsort - Galloping Merge, или «режим выбега».

Для демонстрации эффективности этой техники рассмотрим слияние следующих массивов:

A = 1, 2, 3, …, 9999, 10000

B = 20000, 20001, …, 29999, 30000

При обычном поэлементном слиянии потребуется 10000 сравнений и 10000 копирований. Однако Timsort использует в таких ситуациях режим галопа (gallop), позволяющий значительно сократить число операций.

Шаги:

0. Начинается процедура слияния.
1. На каждой операции копирования элемента из временного или большего подмассива в результирующий запоминается, из какого именно подмассива был элемент.
2. Если уже некоторое количество элементов (например, в JDK 7 это число равно 7) было взято из одного и того же массива - предполагается, что и дальше придётся брать данные из него. Чтобы подтвердить эту идею, алгоритм переходит в режим галопа, то есть перемещается по массиву-претенденту на поставку следующей большой порции данных бинарным поиском (массив упорядочен) текущего элемента из второго соединяемого массива.

3. В момент, когда данные из текущего массива-поставщика больше не подходят (или был достигнут конец массива), данные копируются целиком.

Шаг 4. Завершение алгоритма

После выполнения всех итераций слияния массив становится полностью отсортированным. На этом работа алгоритма завершается. Благодаря использованию адаптивной сортировки вставками, последовательного слияния и оптимизации Galloping Merge, Timsort обеспечивает высокую производительность и стабильность даже на больших объёмах данных.

Анализ алгоритма Timsort

Корректность алгоритма

Алгоритм Timsort всегда завершает работу корректно, так как:

- Все подмассивы (run’ы) сортируются с использованием проверенной сортировки вставками
- Процесс слияния гарантирует, что каждый элемент попадёт в нужную позицию
- Слияние выполняется только между отсортированными блоками, и в ходе него не теряются и не дублируются элементы

Благодаря стабильному слиянию сохраняется порядок одинаковых элементов - это важно для задач мультисортировки и обработки структурированных данных.

Асимптотическая сложность

Name	Best	Average	Worst	Memory	Stable
Quicksort	$n \log n$	$n \log n$	n^2	$\log n$	Depends
Merge sort	$n \log n$	$n \log n$	$n \log n$	Depends	Yes
In-place Merge sort	_	_	$n (\log n)^2$	1	Yes
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Insertion sort	n	n^2	n^2	1	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No
Selection sort	n^2	n^2	n^2	1	Depends
Timsort	n	$n \log n$	$n \log n$	n	Yes
Shell sort	n	$n(\log n)^2$	$O(n\log^2 n)$	1	No
Bubble sort	n	n^2	n^2	1	Yes
Binary tree sort	n	$n \log n$	$n \log n$	n	Yes
Cycle sort	—	n^2	n^2	1	No
Library sort	—	$n \log n$	n^2	n	Yes
Patience sorting	—	—	$n \log n$	n	No
Smoothsort	n	$n \log n$	$n \log n$	1	No
Strand sort	n	n^2	n^2	n	Yes
Tournament sort	—	$n \log n$	$n \log n$		
Cocktail sort	n	n^2	n^2	1	Yes
Comb sort	—	—	n^2	1	No
Gnome sort	n	n^2	n^2	1	Yes
Bogosort	n	$n \cdot n!$	$n \cdot n! \to \infty$	1	No

Рис 4. Анализ алгоритма Timsort

Лучший случай (O(n)) - наблюдается, когда массив уже почти отсортирован. Алгоритм быстро находит длинные run'ы и использует сортировку вставками, не переходя к массивным слияниям.

Средний и худший случаи (O(n log n)) - алгоритм ведёт себя как улучшенный Merge Sort, эффективно разбивая и объединяя run'ы.

Сложность по памяти

Timsort использует временные массивы для хранения копий run'ов во время слияния. Общий объём выделяемой дополнительной памяти зависит от длины наибольшего сливаемого блока, но в худшем случае может достигать размера входного массива - O(n).

Стабильность

Алгоритм является стабильным: он сохраняет порядок элементов с равными ключами. Это критически важно, например, при сортировке массивов объектов

по нескольким признакам (мультисортировка), где результат одной сортировки должен сохраняться при последующих.

Сравнительный анализ

Для более полного понимания преимуществ и ограничений алгоритма Timsort целесообразно сравнить его с другими популярными алгоритмами сортировки, такими как Quicksort, Mergesort. Сравнение будет проводиться по следующим критериям: асимптотическая сложность, стабильность, адаптивность, использование памяти и применимость в реальных задачах.

1. Сравнение с Quicksort

Quicksort в среднем работает с эффективностью O(n log n), но в худшем случае, например, на почти отсортированных данных, её сложность ухудшается до O(n^2). Timsort, напротив, гарантирует O(n log n) даже в худшем случае благодаря механизму слияния.

Timsort является стабильным алгоритмом, сохраняющим порядок элементов с одинаковыми ключами, тогда как Quicksort в стандартной реализации нестабилен, что ограничивает его в задачах мультисортировки.

Timsort также адаптивен, эффективно обрабатывая частично отсортированные массивы, в отличие от Quicksort, который не использует эту особенность.

Однако Quicksort требует меньше дополнительной памяти - O(log n) для стека вызовов, тогда как Timsort может использовать до O(n).

2. Сравнение с Mergesort

Mergesort, как и Timsort, стабилен и имеет сложность O(n log n) во всех случаях.

Однако Timsort превосходит Mergesort за счёт адаптивности: он оптимизирован для частично упорядоченных данных, часто встречающихся в реальных задачах. Mergesort делит массив на равные части, что может привести к избыточным слияниям, тогда как Timsort минимизирует их, используя естественные run’ы и Galloping Merge.

По памяти оба алгоритма схожи: оба требуют O(n) для временных массивов, но Timsort может быть экономичнее в лучшем случае, если данные уже частично отсортированы.

Реализация

В моей курсовой работе данный алгоритм я реализовал на языке JavaScript

1. Функция getMinRun

Функция getMinRun вычисляет значение минимального размера подмассива, называемого minrun, который используется в алгоритме Timsort для разбивки исходного массива на небольшие отсортированные участки (run’ы).

```
//minrun
function getMinRun(arr) {
  let n = arr.length
  let r = 0
  while (n >= 64) {
    r |= n & 1
    n >>= 1
  }
  return n + r
}
```

Рис 5. Функция getMinRun

Функция принимает на вход массив и возвращает целое число в диапазоне от 32 до 64. Алгоритм вычисления следующий:

- Изначально n - длина массива, r - флаг, равный 0.
- Пока n не станет меньше 64, происходит циклический сдвиг n вправо на один бит (деление на 2) с накоплением информации о последнем бите в переменную r.
- В результате возвращается сумма n + r, которая гарантирует, что минимальный размер блока будет достаточно большим для эффективной сортировки вставками, но не слишком большим, чтобы не усложнять процесс слияния.

2. Функция insertionSort

На этапе обработки каждого run’а в алгоритме Timsort используется классическая сортировка вставками. Эта сортировка особенно эффективна на

небольших и частично отсортированных массивах, что и является случаем для run’ов.

```
// Insertion Sort
function insertionSort(arr, left, right) {
  for (let i = left + 1; i <= right; i++) {
    let key = arr[i]
    let j = i - 1
    while (j >= left && arr[j] > key) {
      arr[j + 1] = arr[j]
      j--
    }
    arr[j + 1] = key
  }
}
```

Рис 6. Функция insertionSort

Принцип работы:

- Начинается проход с элемента, следующего за left.
- Текущий элемент key сравнивается с элементами слева от него.
- Пока элементы слева больше key, они сдвигаются вправо.
- Когда находится позиция для key, он вставляется на своё место.

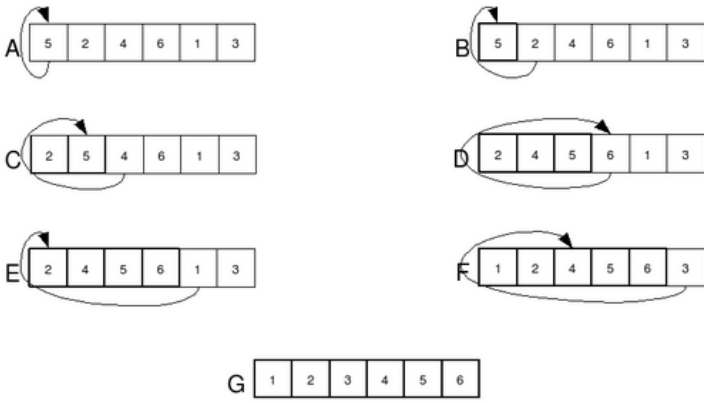

Рис 7. Визуализация сортировки вставками

3. Функция exponentialSearch

Функция exponentialSearch реализует поиск позиции вставки элемента key в отсортированном подмассиве arr, начиная с индекса start и длиной length. Это

улучшенный вариант бинарного поиска, называемый экспоненциальным поиском.

```
// ExponentialSearch (экспоненциальный поиск)
function exponentialSearch(key, arr, start, length) {
  let offset = 1
  let lastOffset = 0

  if (length === 0) return start

  while (offset < length && arr[start + offset] < key) {
    lastOffset = offset
    offset = (offset << 1) + 1
  }

  let lo = start + lastOffset
  let hi = Math.min(start + offset, start + length)

  while (lo < hi) {
    let mid = lo + ((hi - lo) >> 1)
    if (key > arr[mid]) {
      lo = mid + 1
    } else {
      hi = mid
    }
  }

  return lo
}
```

Рис 8. Функция exponentialSearch

Принцип работы:

- Сначала экспоненциально увеличиваем смещение offset, чтобы быстро приблизиться к области, где должен быть вставлен key.
- Этот этап позволяет определить границы для последующего бинарного поиска, который работает уже на суженном диапазоне.

- Затем выполняется классический бинарный поиск для точного определения позиции вставки.
- Функция возвращает индекс, куда следует вставить элемент key, чтобы сохранить отсортированность массива.

4. Функция gallopingMergeSort

Функция gallopingMergeSort реализует слияние двух отсортированных подмассивов left и right обратно в исходный массив arr. Это важный этап алгоритма Timsort, который улучшает классическое слияние, используя режим «галопа» (galloping).

```
// Galloping Merge Sort
function gallopingMergeSort(arr, l, m, r) {
  const minGallop = 7
  let left = arr.slice(l, m + 1)
  let right = arr.slice(m + 1, r + 1)
  let i = 0, j = 0, k = l
  let countLeft = 0, countRight = 0

  while (i < left.length && j < right.length) {
    if (left[i] <= right[j]) {
      arr[k++] = left[i++]
      countLeft++
      countRight = 0
    } else {
      arr[k++] = right[j++]
      countRight++
      countLeft = 0
    }

    if (countLeft >= minGallop) {
      let gallopEnd = exponentialSearch(right[j], left, i, left.length - i)
      while (i < gallopEnd) arr[k++] = left[i++]
      countLeft = 0
    } else if (countRight >= minGallop) {
      let gallopEnd = exponentialSearch(left[i], right, j, right.length - j)
      while (j < gallopEnd) arr[k++] = right[j++]
      countRight = 0
    }
  }

  while (i < left.length) arr[k++] = left[i++]
  while (j < right.length) arr[k++] = right[j++]
}
```

Рис 9. Функция gallopingMergeSort

- minGallop - пороговое значение, при достижении которого происходит переключение в режим галопа.
- left и right - копии двух подмассивов, которые нужно слить.
- Счётчики countLeft и countRight отслеживают, сколько элементов подряд взято из соответствующего подмассива.

Принцип работы:

1. Сначала элементы из left и right сравниваются по одному, и меньший копируется в результирующий массив.
2. Если подряд взято много элементов из одного подмассива (не менее minGallop), слияние переключается в режим галопа.
3. В режиме галопа используется экспоненциальный поиск (exponentialSearch), который позволяет быстро найти блок элементов, которые можно скопировать подряд, минуя покомпонентное сравнение.
4. После копирования блока из режима галопа счётчики сбрасываются и процесс продолжается поэлементным сравнением.

5. Функция Timsort

Функция timSort реализует основные этапы алгоритма Timsort, которые мы разбирали ранее.

```
// Timsort
function timSort(arr) {
  const n = arr.length
  const minrun = getMinRun(arr)
  console.log(`minrun: ${minrun}`)

  let i = 0
  while (i < n) {
    let runStart = i
    let runEnd = i + 1

    // Поиск направления run
    if (runEnd < n && arr[runEnd] < arr[runEnd - 1]) {
      while (runEnd < n && arr[runEnd] < arr[runEnd - 1]) {
        runEnd++
      }
      arr.splice(runStart, runEnd - runStart, ...arr.slice(runStart, runEnd).reverse())
    } else {
      while (runEnd < n && arr[runEnd] >= arr[runEnd - 1]) {
        runEnd++
      }
    }

    // Расширение до minrun
    if ((runEnd - runStart) < minrun) {
      runEnd = Math.min(runStart + minrun, n)
    }

    // Сортировка вставками
    insertionSort(arr, runStart, runEnd - 1)
    i = runEnd
  }

  // Слияние run'ов
  for (let size = minrun; size < n; size *= 2) {
    for (let left = 0; left < n; left += 2 * size) {
      let mid = Math.min(left + size - 1, n - 1)
      let right = Math.min(left + 2 * size - 1, n - 1)
      if (mid < right) {
        gallopingMergeSort(arr, left, mid, right)
      }
    }
  }
}
```

Рис 10. Функция timSort

1. Вызывается функция, которая вычисляет минимальный размер блока (run), оптимальный для эффективной сортировки вставками.
2. Цикл перебирает массив, разделяя его на отсортированные (по возрастанию или убыванию) подмассивы - run’ы. Таким образом, мы получаем последовательность отсортированных подмассивов размером не меньше minrun.

3. На следующем этапе начинается последовательное слияние отсортированных run’ов с использованием улучшенного алгоритма слияния gallopingMergeSort

Визуализация

Для лучшего понимания работы алгоритма Timsort была реализована визуализация его шагов прямо в коде. В процессе сортировки алгоритм выводит в консоль информацию о каждом этапе:

- когда найден упорядоченный подмассив (run);
- когда он разворачивается (если убывает);
- когда run расширяется до размера minrun;
- когда применяется сортировка вставками;
- когда начинается слияние двух run’ов;
- когда активируется режим Galloping Merge;
- какие элементы копируются при слиянии.

1. Без включения Galloping Merge

```
let arr = [
  582, 913, 124, 764, 432, 88, 905, 210, 667, 358,
  471, 103, 556, 780, 312, 690, 894, 34, 608, 776,
  421, 973, 641, 528, 712, 880, 60, 150, 820, 109,
  540, 294, 135, 822, 509, 618, 737, 267, 449, 302,
  730, 173, 961, 553, 804, 738, 215, 380, 946, 674,
  722, 279, 493, 847, 195, 257, 803, 622, 981, 438,
  761, 988, 571, 698, 386, 826, 100, 665, 983, 728,
  413, 670, 476, 238, 459, 614, 548, 317, 721, 602,
  945, 531, 135, 679, 94, 820, 423, 998, 296, 386,
  511, 94, 534, 614, 680, 722, 521, 299, 703, 505
]
```

Рис 11. Пример массива для визуализации

```
minrun: 50
Обнаружен возрастающий run: [0…1]
Расширение run: [0…1] -> [0…49]
--- Сортировка вставками: [0…49]
Обнаружен убывающий run: [50…51]
Расширение run: [50…51] -> [50…99]
--- Сортировка вставками: [50…99]

 Слияние подмассивов: [0…49] + [50…99]
Время сортировки Timsort: 0.549ms
```

Рис 12. Изображение логов выполнения

Режим Galloping не включается, так как порядок сравнения часто чередуется.

2. С включением Galloping Merge

```
let arr = [
  1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
  11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
  21, 22, 23, 24, 25, 26, 27, 28, 29, 30,
  31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
  41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
  100, 101, 102, 103, 104, 105, 106, 107, 108, 109,
  110, 111, 112, 113, 114, 115, 116, 117, 118, 119,
  120, 121, 122, 123, 124, 125, 126, 127, 128, 129,
  130, 131, 132, 133, 134, 135, 136, 137, 138, 139,
  140, 141, 142, 143, 144, 145, 146, 147, 148, 149
]
```

Рис 13. Пример массива для визуализации

```
minrun: 50
Обнаружен возрастающий run: [0…99]
--- Сортировка вставками: [0…99]

 Слияние подмассивов: [0…49] + [50…99]
Переход в galloping-режим (слева) 7
Поиск вставки (галоп) для 100 в диапазоне [7…49]
Время сортировки Timsort: 0.312ms
```

Рис 14. Изображение логов выполнения

Здесь Timsort активирует режим Galloping, обнаружив, что весь левый подмассив меньше правого. Это позволяет быстро найти точку вставки через экспоненциальный поиск и выполнить блочное копирование.

Тестирование

Протестируем наш алгоритм с разным количеством чисел, а также проанализируем время сортировки

1) Возьмём 10 чисел

```
//Тестовый массив
let arr = [42, 7, 19, 3, 25, 8, 15, 11, 2, 30]

console.log("Исходный массив:", arr)
console.time("Время сортировки Timsort")
timSort(arr)
console.timeEnd("Время сортировки Timsort")
console.log("Отсортированный массив:", arr)
```

Рис 15. Массив из 10 чисел

```
Исходный массив: [
  42,  7, 19, 3, 25,
   8, 15, 11, 2, 30
]
minrun: 10
Время сортировки Timsort: 0.392ms
Отсортированный массив: [
   2,  3,  7,  8, 11,
  15, 19, 25, 30, 42
]
```

Рис 16. Тест с 10 числами

Как мы видим на рисунке 16, наш алгоритм отработал без ошибок за 0.392 мс

2) Возьмём 365 чисел

```
//Тестовый массив
let arr = [
  582, 913, 124, 764, 432, 88, 905, 210, 667, 358,
  471, 103, 556, 780, 312, 690, 225, 894, 34, 608,
  776, 421, 973, 641, 528, 349, 712, 880, 60, 150,
  366, 247, 820, 109, 540, 294, 135, 976, 822, 346,
  509, 618, 81, 331, 737, 1000, 267, 88, 449, 154,
  482, 302, 730, 406, 173, 961, 370, 132, 621, 553,
  804, 410, 738, 519, 215, 380, 689, 946, 255, 94,
  800, 123, 674, 369, 722, 841, 328, 279, 493, 63,
  847, 195, 566, 257, 689, 37, 803, 324, 491, 622,
  147, 981, 217, 438, 126, 402, 194, 761, 277, 988,
  915, 283, 571, 722, 698, 651, 242, 74, 386, 355,
  401, 965, 520, 826, 590, 100, 665, 99, 431, 983,
  205, 614, 728, 413, 288, 670, 122, 330, 476, 238,
  615, 459, 908, 614, 53, 689, 334, 147, 329, 548,
  623, 317, 235, 307, 721, 303, 476, 800, 602, 771,
  945, 177, 531, 701, 135, 288, 193, 679, 808, 94,
  820, 590, 423, 973, 87, 320, 291, 413, 998, 66,
  155, 296, 450, 310, 202, 386, 973, 751, 177, 511,
  619, 94, 100, 874, 534, 141, 614, 217, 861, 1000,
  429, 680, 356, 722, 388, 521, 732, 659, 101, 299,
  909, 333, 227, 801, 132, 703, 651, 429, 547, 101,
  763, 505, 192, 417, 150, 790, 328, 625, 284, 306,
  464, 888, 195, 455, 614, 650, 140, 770, 933, 514,
  802, 460, 300, 503, 612, 405, 899, 498, 322, 741,
  181, 932, 716, 101, 602, 842, 492, 55, 368, 122,
  445, 981, 487, 793, 299, 216, 327, 711, 470, 376,
  349, 628, 234, 519, 661, 399, 973, 564, 774, 655,
  803, 910, 84, 327, 688, 743, 444, 589, 723, 90,
  537, 296, 152, 824, 498, 928, 110, 470, 500, 651,
  211, 943, 765, 392, 439, 727, 208, 482, 333, 781,
  148, 618, 941, 211, 354, 689, 587, 227, 378, 755,
  602, 719, 496, 431, 287, 845, 96, 998, 360, 647,
  512, 488, 168, 308, 667, 337, 425, 349, 285, 789,
  612, 228, 464, 905, 493, 373, 721, 315, 1000
];
```

Рис 17. Массив из 365 чисел

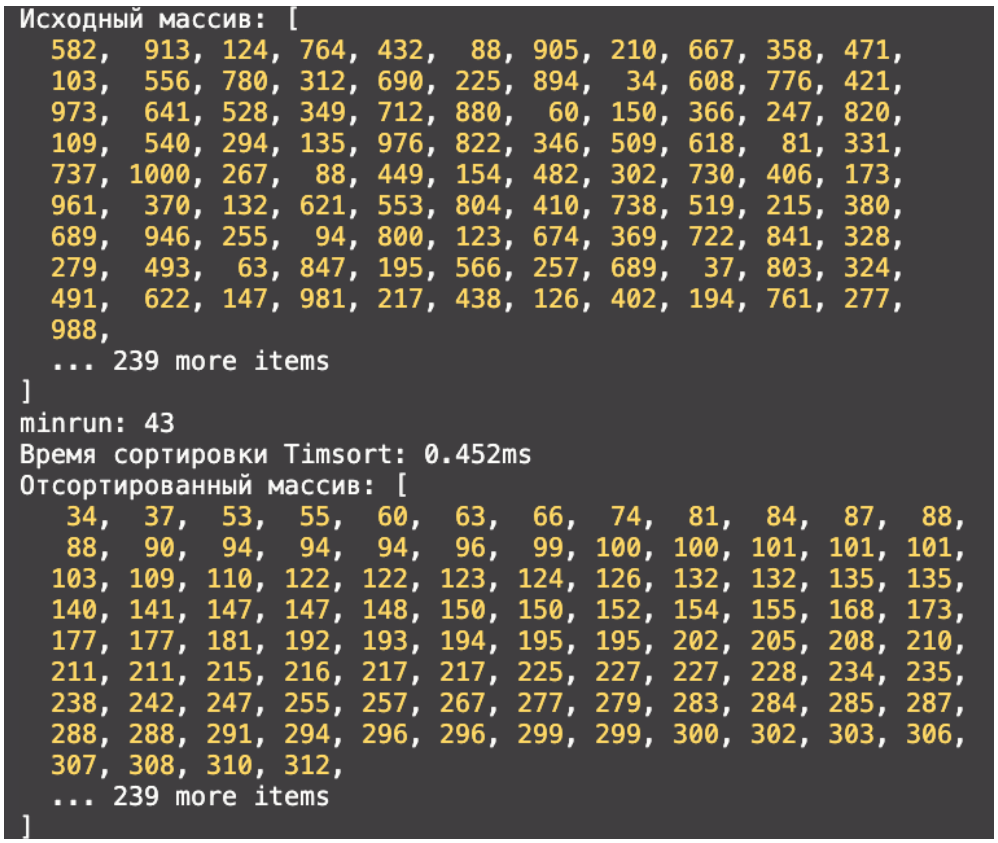

Рис 18. Тест с 365 числами

Как мы видим на рисунке 18, наш алгоритм отработал без ошибок за 0.452 мс

3) Возьмём частично отсортированный массив

```
let arr = [1, 3, 5, 2, 4, 6, 8, 7]
```

Рис 19. Частично отсортированный массив

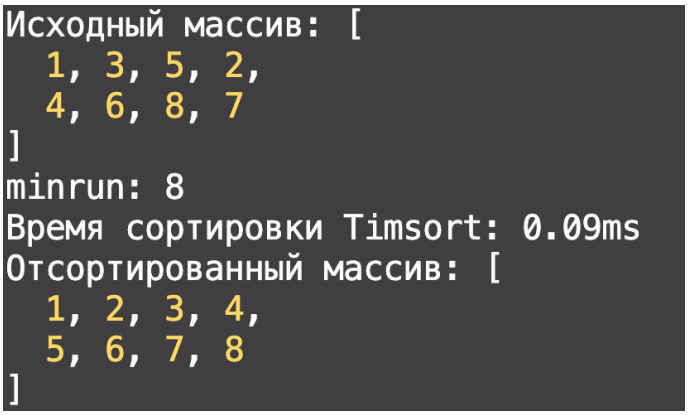

Рис 20. Тест с частично отсортированным массивом

Как мы видим на рисунке 20, наш алгоритм успешно отсортировал частично упорядоченный массив за 0.09 мс. Это время существенно быстрее, чем время сортировки полностью случайного массива аналогичного размера (0.392 мс, см. Рис 16), что демонстрирует адаптивные преимущества Timsort при работе с частично структурированными данными.

ЗАКЛЮЧЕНИЕ

Timsort является ярким примером того, как можно объединить классические подходы к сортировке данных с современными требованиями к производительности и адаптивности. Благодаря сочетанию сортировки вставками и слиянием, а также использованию техник типа Galloping Merge, алгоритм демонстрирует высокую эффективность в реальных условиях, где данные часто частично упорядочены.

В ходе работы были рассмотрены основные этапы алгоритма: определение минимального размера подмассива (minrun), разбиение массива на упорядоченные run’ы, сортировка каждого из них вставками и эффективное слияние с использованием режима «галопа». Также был проведён анализ временной и пространственной сложности, рассмотрены достоинства и недостатки по сравнению с другими алгоритмами сортировки, такими как Quicksort и Merge Sort.

Практическая часть курсовой работы включала реализацию алгоритма на языке JavaScript, его визуализация и тестирование. Полученные результаты подтвердили корректность и производительность реализации.

Таким образом, алгоритм Timsort представляет собой надёжное и адаптивное решение задачи сортировки. Благодаря сочетанию двух эффективных методов - сортировки вставками и слиянием - он демонстрирует высокую производительность на практике, особенно при работе с частично отсортированными данными. Его устойчивость, стабильность и оптимизация под реальные сценарии объясняют широкое распространение в таких языках программирования, как Python, Java и других. Всё это делает Timsort не просто алгоритмом, а стандартом сортировки в современных программных системах.

ИСТОЧНИКИ

1. https://habr.com/ru/companies/infopulse/articles/133303/
2. https://neerc.ifmo.ru/wiki/index.php?title=Timsort
3. https://ru.wikipedia.org/wiki/Timsort

Репозиторий - https://github.com/DaniilDarjutin/Timsort_DariutinDD_MISIS