Санкт-Петербургский политехнический университет Петра великого
Институт машиностроения, материалов и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Программирование на языках высокого уровня

Тема: Алгоритм Timsort

Разработал:
студент гр. 3331506/70401 Деникин В.А.
Преподаватель Ананьевский М.С.

«____» _____ 2020 г.

Санкт-Петербург
2020

Оглавление

Введение

Timsort, в отличии от «пузырьков» и «вставок», изобретен был недавно в 2002 году Тимом Петерсом. С тех пор он уже стал стандартным алгоритмом сортировки в Python, OpenJDK 7 и Android JDK 1.5.

Алгоритм построен на той идее, что в реальном мире сортируемый массив данных часто содержат в себе упорядоченные (не важно, по возрастанию или по убыванию) подмассивы. На таких данных Timsort преобладает над всеми остальными алгоритмами.

В данной работе будет разобран принцип работы алгоритма, его реализации на C++, а также произведен анализ сложности и численный анализ алгоритма.

Описание алгоритма

Алгоритм находит подпоследовательности данных, которые уже упорядочены (выполняется), и использует их для более эффективной сортировки остатка. Это делается путем слияния прогонов до тех пор, пока не будут выполнены определенные критерии.

Основная идея алгоритма:

- По специальному алгоритму входной массив разделяется на подмассивы.
- Каждый подмассив сортируется сортировкой вставками.
- Отсортированные подмассивы собираются в единый массив с помощью модифицированной сортировки слиянием.

Принципиальные особенности алгоритма в деталях, а именно в алгоритме разделения и модификации сортировки слиянием.

Реализация алгоритма

Шаг 0. Вычисление minrun.

Minrun — это минимальный размер подмассива. Хорошо бы, чтобы $N \backslash minrun$ было степенью числа 2 (или близким к нему). Это требование обусловлено тем, что алгоритм слияния подмассивов наиболее эффективно работает на подмассивах примерно равного размера.

Шаг 1. Разбиение на подмассивы и их сортировка.

1. Ставим указатель текущего элемента в начало входного массива.
2. Начиная с текущего элемента, ищем во входном массиве run (упорядоченный подмассив). Если получившийся подмассив упорядочен по убыванию — переставляем элементы так, чтобы они шли по возрастанию.
3. Если размер текущего run'а меньше чем minrun — берём следующие за найденным run-ом элементы в количестве minrun — size(run).
4. Применяем к данному подмассиву сортировку вставками. Так как размер подмассива невелик и часть его уже упорядочена — сортировка работает быстро и эффективно.
5. Ставим указатель текущего элемента на следующий за подмассивом элемент.
6. Если конец входного массива не достигнут — переход к пункту 2, иначе — конец данного шага.

Шаг 2. Слияние.

Теперь нам нужно объединить эти подмассивы для получения результирующего, полностью упорядоченного массива. Два требования:

1. Объединять подмассивы примерно равного размера (эффективнее).
2. Сохранить стабильность алгоритма — т.е. не делать бессмысленных перестановок (например, не менять два последовательно стоящих одинаковых числа местами).

Для этого:

1. Создаем пустой стек пар <индекс начала подмассива>-<размер подмассива>. Берём первый упорядоченный подмассив.

2. Добавляем в стек пару данных <индекс начала>-<размер> для текущего подмассива.

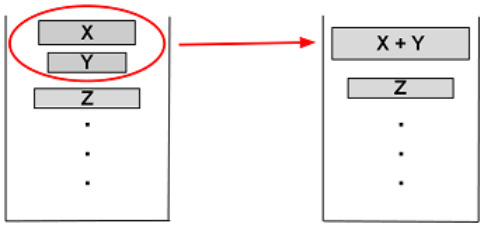
3. Определяем, нужно ли выполнять процедуру слияния текущего подмассива с предыдущими. Для этого проверяется выполнение 2 правил (пусть X, Y и Z — размеры трёх верхних в стеке подмассивов):

X > Y + Z

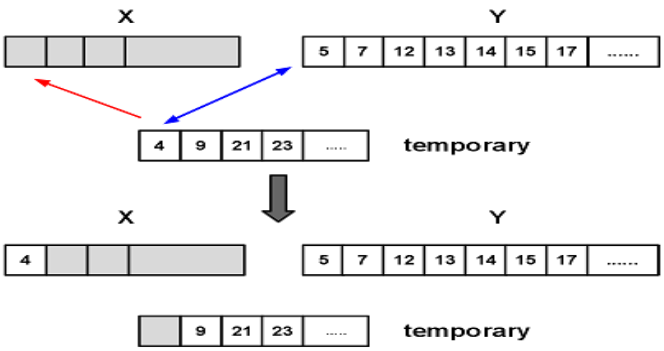
Y > Z

4. Если одно из правил нарушается — массив Y сливается с меньшим из массивов X и Z. Повторяется до выполнения обоих правил или полного упорядочивания данных.

5. Если еще остались не рассмотренные подмассивы — берём следующий и переходим к пункту 2. Иначе — конец.



Процедура слияния подмассивов



Мы всегда соединяем 2 последовательных подмассива. Для их слияния используется дополнительная память.

1. Создаём временный массив в размере меньшего из соединяемых подмассивов.
2. Копируем меньший из подмассивов во временный массив
3. Ставим указатели текущей позиции на первые элементы большего и временного массива.
4. На каждом следующем шаге рассматриваем значение текущих элементов в большем и временном массивах, берём меньший из них и копируем его в новый отсортированный массив. Перемещаем указатель текущего элемента в массиве, из которого был взят элемент.
5. Повторяем 4, пока один из массивов не закончится.
6. Добавляем все элементы оставшегося массива в конец нового массива.

Модификация процедуры слияния подмассивов

Алгоритм Timsort предлагает модификацию, которую он называет «галоп». Суть в следующем:

1. Начинаем процедуру слияния, как было показано выше.

2. На каждой операции копирования элемента из временного или большего подмассива в результирующий запоминаем, из какого именно подмассива был элемент.

3. Если уже некоторое количество элементов было взято из одного и того же массива — предполагаем, что и дальше нам придётся брать данные из него. Чтобы подтвердить эту идею, мы переходим в режим «галопа», т.е. бежим по массиву-претенденту на поставку следующей большой порции данных бинарным поиском текущего элемента из второго соединяемого массива.

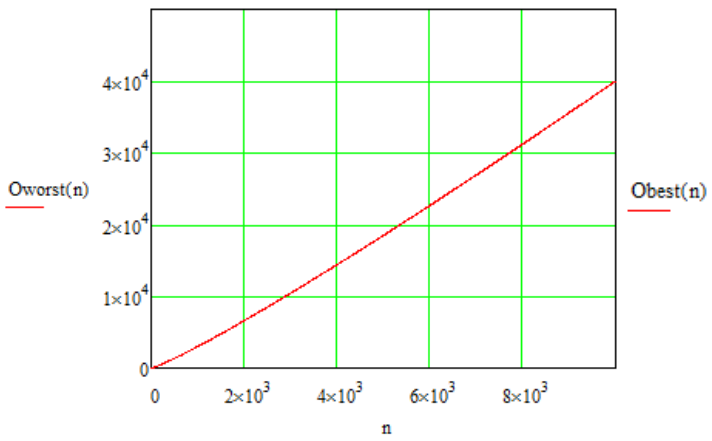
4. Найдя момент, когда данные из текущего массива-поставщика нам больше не подходят (или дойдя до конца массива), мы можем, наконец, скопировать их все разом (что может быть эффективнее копирования одиночных элементов).

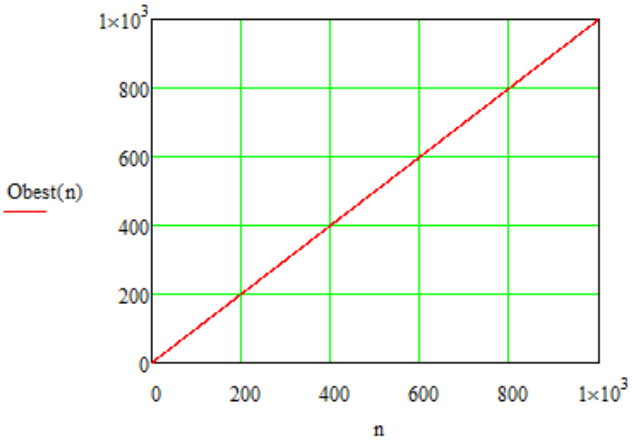
Анализ алгоритма

Name	Best	Average	Worst	Memory	Stable
Timsort	n	$n \log n$	$n \log n$	n	Yes

В худшем случае Тимсорт принимает *O(nlogn)* сравнения для сортировки массива из *n* элементов. В лучшем случае, который происходит, когда входные данные уже отсортированы, он выполняется за линейное время, что означает, что это алгоритм адаптивной сортировки.

Каждый подмассив *runi* может участвовать в не более *O(logn)* операций слияния, а значит и каждый элемент будет задействован в сравниниях не более *O(logn)* раз. Элементов n, откуда получаем оценку в *O(nlogn)*.





Преимущество по сравнению с Quicksort для сортировки ссылок на объекты или указателей, потому что они требуют дорогостоящего косвенного обращения к памяти для доступа к данным и выполнения сравнений, а преимущества согласованности кэша Quicksort значительно уменьшаются.

Область применения

Timsort - стандартный алгоритм сортировки в Python, начиная с версии 2.3. Он также используется для сортировки массивов, не примитивного типа в Java SE 7 , на платформе Android , в GNU Octave и Google Chrome .

Timsort – «адаптивная, стабильная, естественное слияние». Он обладает сверхъестественными характеристиками для многих типов частично упорядоченных массивов (требуется меньше, чем lg (N!) сравнений, и всего лишь N-1). Тимсорт специально предназначен для обнаружения и использования частично отсортированных подпоследовательностей во входе, которые часто встречаются в реальном мире, что в сравнении намного дороже, чем замена элементов в списке, поскольку обычно просто переопределяют указатели, что очень часто делает timsort отличным выбором.

Заключение

В ходе выполнения курсовой работы был рассмотрен алгоритм сортировки Timsort, реализация алгоритма которого была выполнена на языке C++.

Так же был проверен анализ сложности алгоритма и его численный анализ, была рассмотрена область применения данной сортировки и ее особенности.

Список литературы

1. Peter McIlroy "Optimistic Sorting and Information Theoretic Complexity", Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms, ISBN 0-89871-313-7, Chapter 53, pp 467-474, January 1993.

2. Magnus Lie Hetland. Python Algorithms: Mastering Basic Algorithms in the Python Language. — Apress, 2010. — 336 с.

3. Питерс, Тим. "Сортировка [Python-Dev]"

Приложение 1 – Исходный код

```
#include"stdc++.h"

using namespace std;
const int RUN = 32;

// this function sorts array from left index to
// to right index which is of size atmost RUN
void insertionSort(int arr[], int left, int right)
{
    for (int i = left + 1; i <= right; i++)
    {
        int temp = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > temp)
        {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = temp;
    }
}

// merge function merges the sorted runs
void merge(int arr[], int l, int m, int r)
{
    // original array is broken in two parts
    // left and right array
    int len1 = m - l + 1, len2 = r - m;

    //int left[len1], right[len2];
    int* left = new int[len1];
    int* right = new int[len2];

    for (int i = 0; i < len1; i++)
        left[i] = arr[l + i];
    for (int i = 0; i < len2; i++)
        right[i] = arr[m + 1 + i];

    int i = 0;
    int j = 0;
    int k = l;

```

```
    // after comparing, we merge those two array
    // in larger sub array
    while (i < len1 && j < len2)
    {
        if (left[i] <= right[j])
        {
            arr[k] = left[i];
            i++;
        }
        else
        {
            arr[k] = right[j];
            j++;
        }
        k++;
    }

    // copy remaining elements of left, if any
    while (i < len1)
    {
        arr[k] = left[i];
        k++;
        i++;
    }

    // copy remaining element of right, if any
    while (j < len2)
    {
        arr[k] = right[j];
        k++;
        j++;
    }

    delete [] left;
    delete [] right;
}
// iterative Timsort function to sort the
// array[0...n-1] (similar to merge sort)
void timSort(int arr[], int n)
{
    // Sort individual subarrays of size RUN
    for (int i = 0; i < n; i += RUN)
        insertionSort(arr, i, min((i + 31), (n - 1)));

    // start merging from size RUN (or 32). It will merge
    // to form size 64, then 128, 256 and so on ....
    for (int size = RUN; size < n; size = 2 * size)
    {
        // pick starting point of left sub array. We
        // are going to merge arr[left..left+size-1]
        // and arr[left+size, left+2*size-1]
        // After every merge, we increase left by 2*size
        for (int left = 0; left < n; left += 2 * size)
        {
            // find ending point of left sub array
            // mid+1 is starting point of right sub array
            int mid = left + size - 1;
            int right = min((left + 2 * size - 1), (n - 1));

            // merge sub array arr[left.....mid] &
            // arr[mid+1....right]
            merge(arr, left, mid, right);
        }
    }
}
```

```
// utility function to print the Array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    cout << endl;
}

// Driver program to test above function
int main()
{
    int arr[] = { 5, 21, 7, 23, 19 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << "Given Array is \n";
    printArray(arr, n);

    timSort(arr, n);

    cout << "After Sorting Array is \n";
    printArray(arr, n);
    return 0;
}

```