Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 2

Выполнил студент группы .....КС-30..... Лихолат Полина Николаевна
Ссылка на репозиторий: ..... https://github.com/MUCTR-IKT-CPP /Likholat_algorithms

Приняли: .....Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович
.....Лобанов Алексей Владимирович
.....Крашенинников Роман Сергеевич

Дата сдачи: 06.03.2023

Оглавление

Описание задачи.

Вариант 2

Необходимо реализовать метод сортировки слиянием.

Для реализованного метода сортировки необходимо провести серию тестов для всех значений N из списка (1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000), при этом:

в каждом тесте необходимо по 20 раз генерировать вектор, состоящий из N элементов

каждый элемент массива заполняется случайным числом с плавающей запятой от -1 до 1

При проведении сортировки, подсчитать количество дополнительной потребляемой памяти, под дополнительной понимается память, которая используется на хранение промежуточного результата сортировки. Построить график потребления памяти и сравнить его с функцией c * n.

При работе сортировки подсчитать количество вызовов рекурсивной функции, и высоту рекурсивного стека. Построить график худшего, лучшего, и среднего случая для каждой серии тестов.

Для серии тестов построить график худшего, лучшего и среднего случая.

Подобрать такую константу c, что бы график функции c1 * n * log(n) ограничивал получившийся на этапе 5 график сверху, а функция c2 * n * log(n) ограничивал получившийся на этапе 5 график снизу.

Проанализировать полученные графики и определить есть ли на них следы деградации метода относительно своей средней сложности.

Описание метода/модели.

Сортировка слиянием (англ. Merge sort) — алгоритм сортировки, использующий O(n) дополнительной памяти и работающий за O(nlog(n)) времени.

Принцип работы:

Алгоритм использует принцип «разделяй и властвуй»: задача разбивается на подзадачи меньшего размера, которые решаются по отдельности, после чего их решения комбинируются для получения решения исходной задачи. Конкретно процедуру сортировки слиянием можно описать следующим образом:

1. Если в рассматриваемом массиве один элемент, то он уже отсортирован — алгоритм завершает работу.
2. Иначе массив разбивается на две части, которые сортируются рекурсивно.
3. После сортировки двух частей массива к ним применяется процедура слияния, которая по двум отсортированным частям получает исходный отсортированный массив.

Время работы

Чтобы оценить время работы этого алгоритма, составим рекуррентное соотношение. Пускай T(n) — время сортировки массива длины n, тогда для сортировки слиянием справедливо $T(n)=2T(n/2)+O(n)$

O(n) — время, необходимое на то, чтобы слить два массива длины n. Распишем это соотношение:

$T(n)=2T(n/2)+O(n)=4T(n/4)+2O(n)=\cdots=T(1)+\log(n)O(n)=O(n\log(n))$.

Сравнение с другими алгоритмами

Достоинства:

- устойчивая,
- можно написать эффективную многопоточную сортировку слиянием,
- сортировка данных, расположенных на периферийных устройствах и не вмещающихся в оперативную память.

Недостатки:

- требуется дополнительно O(n) памяти, но можно модифицировать до O(1).

Выполнение задачи.

Реализация алгоритма выполнена с использованием языка C++.

Сортировка слиянием реализуется с помощью двух функций.

```
void merge(vector<double>& arr, int l, int m, int r, int& count, int& mem) {
```

Рассчитываем длину векторов

```
  int n1 = m - l + 1;
  int n2 = r - m;
```

Инициализируем два вектора с заданной длиной

```
  vector<double> L(n1);
  vector<double> R(n2);
```

Копируем значения в новые вектора и считаем кол-во памяти для временных векторов

```
  for (int i = 0; i < n1; i++) {
    L[i] = arr[l + i];
    mem += sizeof(L[i]); // Track memory usage
  }
  for (int j = 0; j < n2; j++) {
    R[j] = arr[m + 1 + j];
    mem += sizeof(R[j]); // Track memory usage
  }
```

Нам надо получить вектор с размером |L|+|R|. Для этого можно применить процедуру слияния.

Записываем в исходный вектор отсортированные значения

```
int i = 0, j = 0, k = l;
while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
        arr[k] = L[i];
        i++;
    }
    else {
        arr[k] = R[j];
        j++;
    }
    k++;
}
```

Если остались элементы только в одном векторе, записываем их по порядку в конец

```
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

// Copy the remaining elements of R[], if there are any
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
count++;
}
```

Эта процедура заключается в том, что мы сравниваем элементы массивов (начиная с начала) и меньший из них записываем в финальный. И затем, в массиве у которого оказался меньший элемент, переходим к следующему элементу и сравниваем теперь его. В конце, если один из массивов закончился, мы просто дописываем в финальный другой массив. После мы наш финальный массив записываем вместо двух исходных и получаем отсортированный участок.

Также с помощью инкремента переменной count мы будем засекать количество вызовов рекурсий, а для подсчета памяти мы добавляем к переменной mem величину sizeof(L/R[i]).

```
void mergeSort(vector<double>& arr, int l, int r, int& count, int depth, int& maxDepth, int& mem) {
```

Прописываем условие выхода из функции

```
    if (l >= r) {
        return;
    }
```

Находим границу разбиения по формуле middle = left + (right - left) / 2.

```
        int m = l + (r - l) / 2;
        depth++; //увеличиваем счётчик глубины рекурсии
```

Вызываем функцию для каждой половины исходного вектора

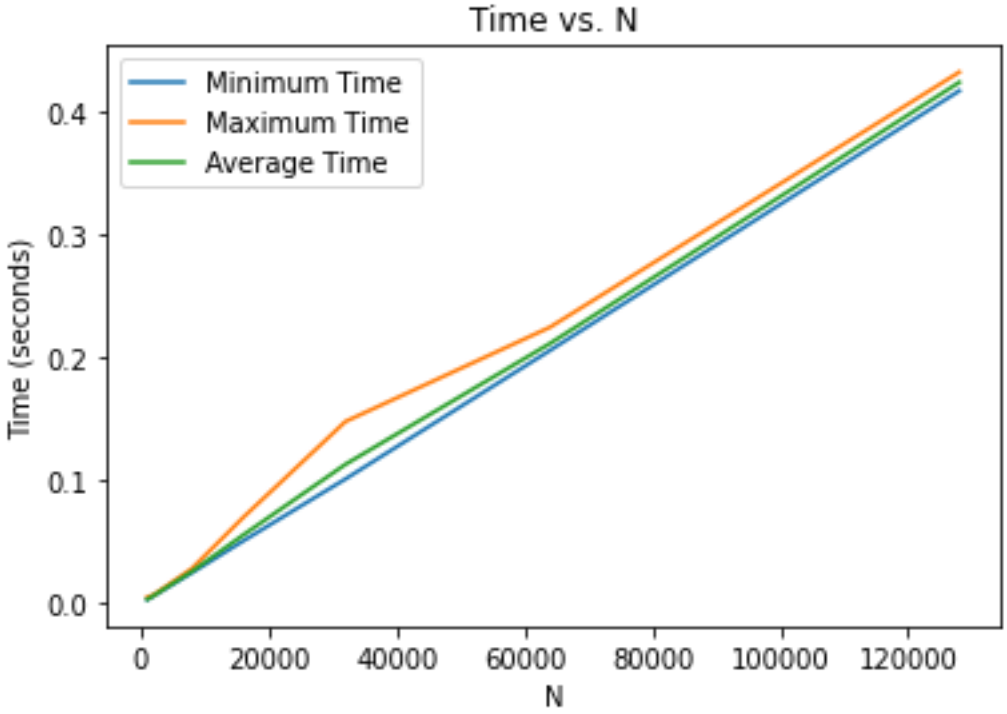
```
        mergeSort(arr, l, m, count, depth, maxDepth, mem);
        mergeSort(arr, m + 1, r, count, depth, maxDepth, mem);
        merge(arr, l, m, r, count, mem);

        depth--;
        maxDepth = max(maxDepth, depth);
    }
```

Данная функция реализует рекурсию и итоговое слияние. Подсчет глубины рекурсии делаем за счет инкремента к переменной depth в начале функции и декремента в конце.

Далее производим расчеты и анализируем данные.

Построим график времени по длине:

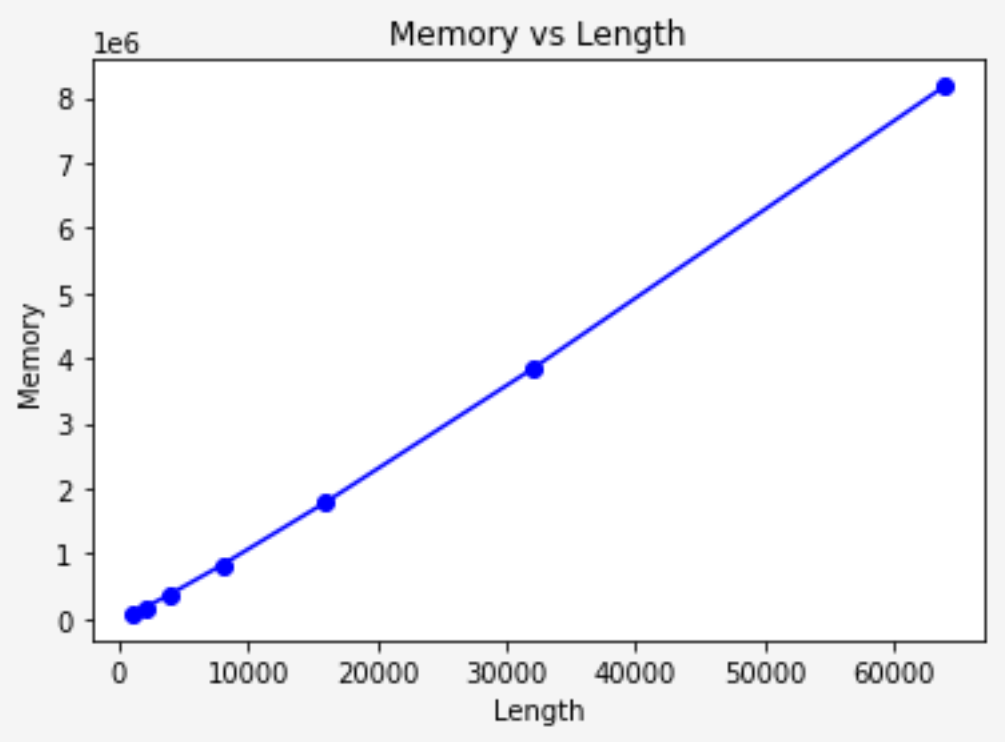
```
import matplotlib.pyplot as plt

# Insert the provided data into lists
min_time = [0.0031211, 0.0061127, 0.0124551, 0.0250709, 0.0509054, 0.1014,
0.206189, 0.417616]
max_time = [0.0051199, 0.0070261, 0.0141972, 0.0285147, 0.0700625, 0.148184,
0.225063, 0.432846]
avg_time = [0.00339089, 0.00638641, 0.0130413, 0.0261756, 0.0557156, 0.113326,
0.211825, 0.424537]
N = [1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000]

# Create a figure with a single subplot
fig, ax = plt.subplots()

# Plot the minimum, maximum, and average times vs. N
ax.plot(N, min_time, label='Minimum Time')
ax.plot(N, max_time, label='Maximum Time')
ax.plot(N, avg_time, label='Average Time')

# Add a title and axis labels
ax.set_title('Time vs. N')
ax.set_xlabel('N')
ax.set_ylabel('Time (seconds)')

# Add a legend and display the plot
ax.legend()
plt.show()
```



По этому графику мы видим, что данная зависимость сильно похожа на линейную (это подтверждает зависимость NlogN), а так же, что у нас имеется выброс.

Далее мы строим график потребляемой памяти от длины.

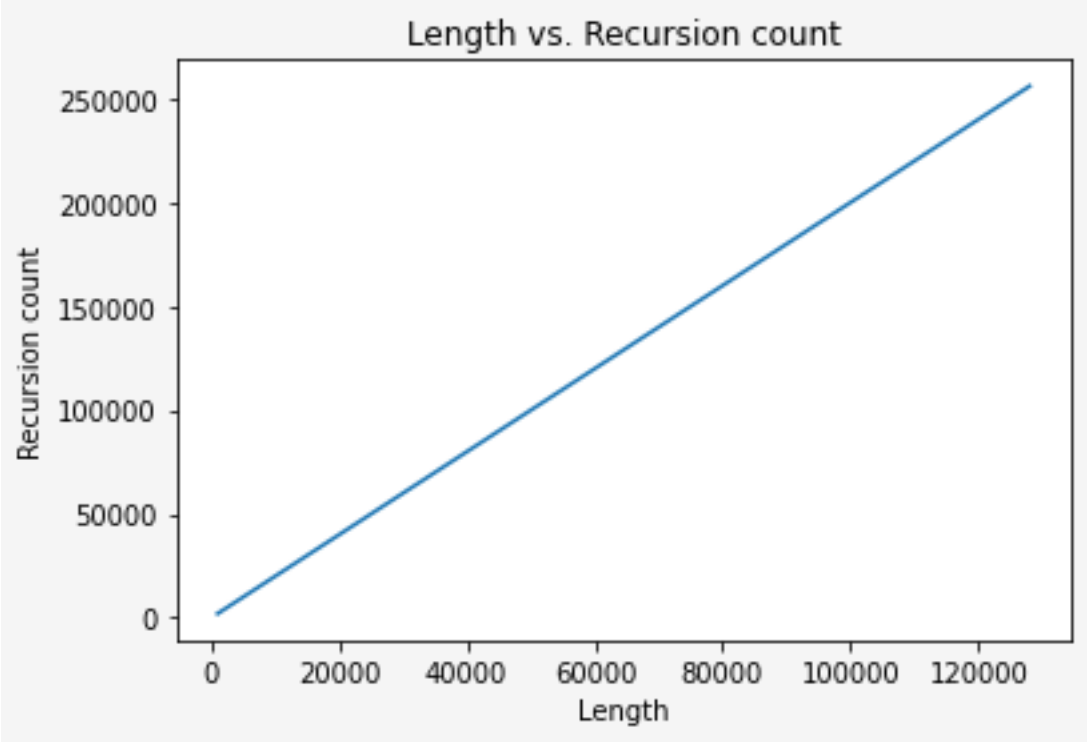
```
import matplotlib.pyplot as plt

# memory values
memory = [79808, 175616, 383232, 830464, 1788928, 3833856, 8179712]

# corresponding length values
length = [1000, 2000, 4000, 8000, 16000, 32000, 64000]

# plot the memory vs length graph
plt.plot(length, memory, 'bo-')
plt.title("Memory vs Length")
plt.xlabel("Length")
  plt.ylabel("Memory")
```



Мы видим, что память изменяется линейно (подтверждаем зависимость O(N)).

Далее строим график количества рекурсий от длины:

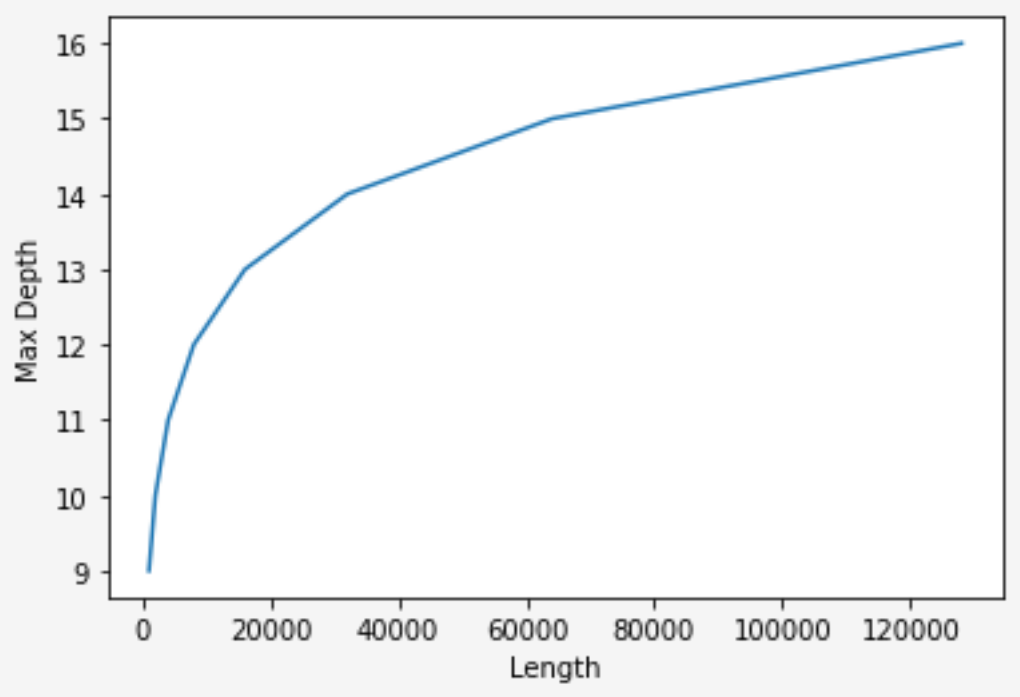
```
import matplotlib.pyplot as plt

# Length and Recursion count data
length_data = [1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000]
recursion_count_data = [1999, 3999, 7999, 15999, 31999, 63999, 127999, 255999]

# Create a figure and axis object
fig, ax = plt.subplots()

# Plot the data
ax.plot(length_data, recursion_count_data)

# Add labels and title
ax.set_xlabel('Length')
ax.set_ylabel('Recursion count')
ax.set_title('Length vs. Recursion count')

# Show the plot
  plt.show()
```



На нем мы видим, что количество рекурсии растет линейно, причем количество count = 2N – 1.

Далее мы строим график глубины рекурсии от длины:

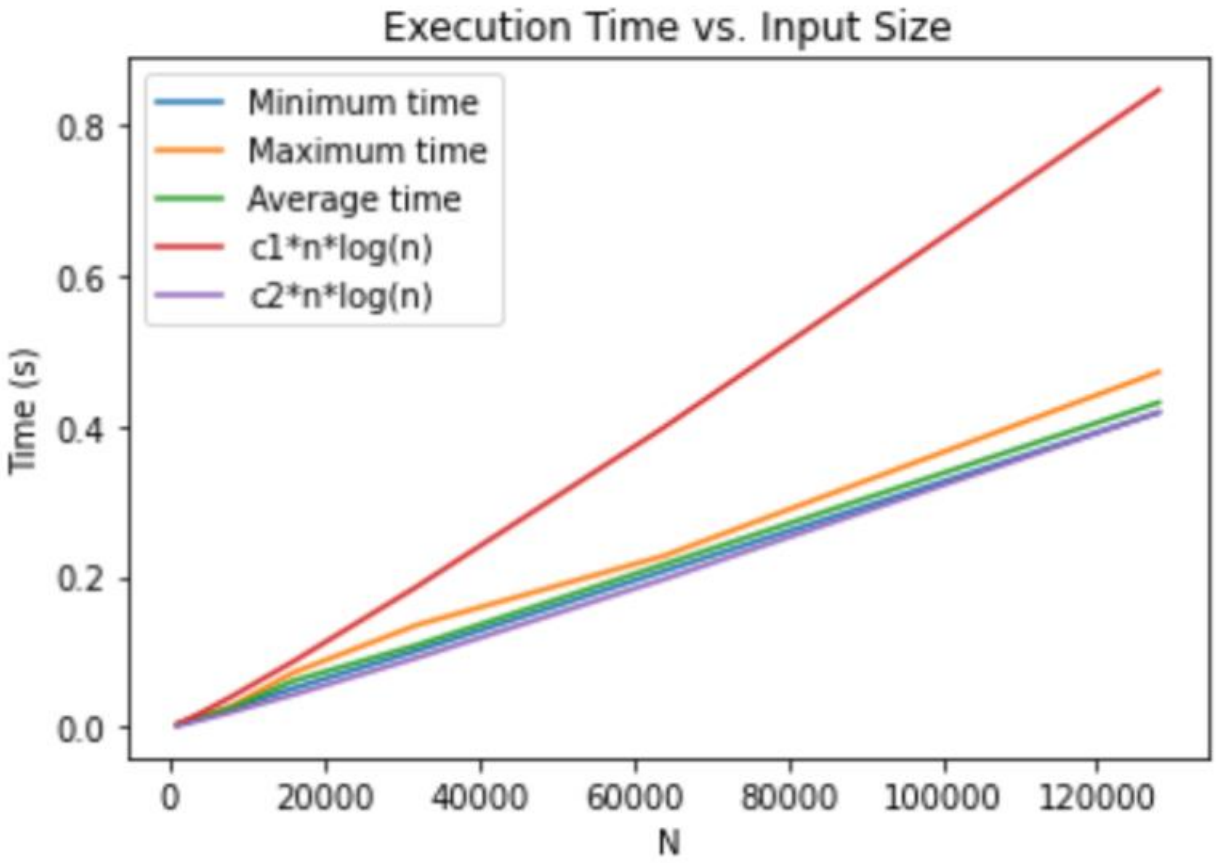
```
import matplotlib.pyplot as plt

# Data
lengths = [1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000]
max_depths = [9, 10, 11, 12, 13, 14, 15, 16]

# Create the figure and axes objects
fig, ax = plt.subplots()

# Set the x-axis and y-axis labels
ax.set_xlabel('Length')
ax.set_ylabel('Max Depth')

# Plot the data
ax.plot(lengths, max_depths)

# Show the plot
  plt.show()
```



Мы получили график, очень похожий на график логарифмической функции.

Далее мы находим константу C1, которая будет ограничивать график времени по длине сверху и C2, которая будет ограничивать график снизу и показываем график.

```
import matplotlib.pyplot as plt
import numpy as np

min_time = [0.0029739, 0.0061801, 0.0120995, 0.024662, 0.0515893, 0.101786, 0.207548,
0.418373]
max_time = [0.0042988, 0.0085568, 0.0139662, 0.0281054, 0.0725057, 0.135801, 0.227792,
0.47219]
avg_time = [0.00321153, 0.00641747, 0.0124895, 0.0256177, 0.0614313, 0.108992,
0.216409, 0.431519]
N = [1000, 2000, 4000, 8000, 16000, 32000, 64000, 128000]

# Calculate c1 and c2 constants
c1 = max_time[1] / (N[1] * np.log(N[1]))
c2 = min_time[-1] / (N[-1] * np.log(N[-1]))

# Create arrays for the c1*n*log(n) and c2*n*log(n) curves
x = np.array(N)
y1 = c1 * x * np.log(x)
y2 = c2 * x * np.log(x)

# Plot the actual data, c1*n*log(n), and c2*n*log(n) curves
fig, ax = plt.subplots()
ax.plot(x, min_time, label='Minimum time')
ax.plot(x, max_time, label='Maximum time')
ax.plot(x, avg_time, label='Average time')
ax.plot(x, y1, label='c1*n*log(n)')
ax.plot(x, y2, label='c2*n*log(n)')
ax.set_xlabel('N')
ax.set_ylabel('Time (s)')
ax.set_title('Execution Time vs. Input Size')
ax.legend()
  plt.show()
```



Так как у нас есть выброс на графике C1 заметно больше ожидаемого. Полученные значения:

```
C1 = 5.628805293546357e-07
C2 = 2.7794206370310053e-07
```

По графику видно, что график C1*N*logN будет сильно деградировать.

Заключение.

Сортировка слиянием удобна тем, что известно время её работы. Явным недостатком является задействование большого объема памяти, что усложняет её использование при работе с большими объёмами данных.