Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 3
по дисциплине ”Анализ алгоритмов”

Тема Сравнение методов сортировки

Студент Калашников С.Д.

Группа ИУ7-53Б

Преподаватель Волкова Л.Л. Строганов Ю.В.

Москва, 2022

СОДЕРЖАНИЕ

Введение

В информатике сортировкой называется процесс организации последовательности в упорядоченном порядке. Данная операция является одной из самых распространненых и важных в различных алгоритмах.

На данный момент существует огромное количество вариаций сортировок. Эти алгоритмы необходимо уметь сравнивать, чтобы выбирать наилучше подходящие в конкретном случае. Они оцениваются по следующим критериям:

- времени быстродействия;
- затратам памяти.

Целью данной лабораторной работы является анализ трудоемкости реализаций алгоритмов сортировки.

Для достижения поставленной цели требуется решить следующие задачи.

1. Изучить и реализовать алгоритмы сортировки: слиянием, плавная, блочная.
2. Провести сравнительный анализ трудоёмкости алгоритмов на основе теоретических расчетов.
3. Провести замеры процессорного времени работы реализаций выбранных сортировок.
4. Провести сравнительный анализ реализаций алгоритмов по затраченному процессорному времени и памяти.
5. Описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1. Аналитическая часть

1.1 Описание алгоритмов

В этом разделе будут рассмотрены следующие алгоритмы сортировки: плавная, слиянием, блочная.

1.1.1 Плавная сотрировка

Разработана известным ученым Э. Дейкстрой в 1981 году [1]. Является подвидом пирамидальной сортировки. Различие состоит в том, что в сортировке кучей используются бинарное дерево. А в плавной сортировке элеименты дерева сотоят из деревьев размером с одно из чисел Леонардо.

Числа Леонардо представляют собой последовательность, вычисляемую по формуле:

$$L(0) = 1;$$

$$L(1) = 1;$$

$$L(n) = L(n-1) + L(n-2) + 1;$$

Общие положения следующие.

1. Сортируемый массив делится на группу подмассивов. Каждый подмассив представляет собой структуру данных куча. Каждая куча имеет размер равный одному из чисел Леонардо. При этом левые элементы массива являются узлами самой большой возможной кучи. Оставшиеся элементы массива разбиваются на кучи по такому же жадному правилу. В дальнейшем эту группу подмассивов будем называть последовательность куч. При этом, элементы

этой последовательности будут строго уменьшаться в размере с ростом порядкового номера.

2. Не существует двух куч, имеющих одинаковый размер.

3. Никакие две кучи не будут иметь размер равным двум последовательным числам Леонардо. Исключением могут быть только две последние кучи. Если мы будем использовать подряд две кучи размерностью $L(x)$ и $L(x+1)$, то их можно будет заменить одной – размерностью $L(x+2)$

Следствия.

1. В вершине каждой кучи находится максимальный из узлов данной кучи.

2. Максимальный элемент массива нужно искать среди вершин сформированных куч.

Алгоритм состоит из двух этапов.

- Формирование последовательности куч: в начале последовательность куч пуста, последовательно добавляются элементы из исходного массива слева направо. Элемент, который будет добавляться в последовательность куч выступает либо в качестве вершины абсолютно новой кучи, либо в качестве вершины, которая объединит две последние кучи. Повторяем эту итерацию пока не закончатся элементы во входном массиве.

- Формирование отсортированного массива: на каждой итерации определяем текущий максимум массива, используя следствие 2. Ставим максимальный элемент на последнее место и исключаем его из последовательности куч. Повторяем это действие, пока последовательность куч не окажется пуста.

Замечание: при объединении двух куч в одну и и при обмене двух вершинных элементов нужно гарантировать сохранение свойства кучи, т.е. выполнять “просейку вниз” при необходимости.

1.1.2 Сортировка слиянием

Данная сортировка была разработана Джоном фон Нейманом в 1945 году и относиться к классу рекурсивных алгоритмов [2]. Алгоритм работает следующим образом.

- Массив рекурсивно разбивается пополам, и каждая из половин делится до тех пор, пока размер очередного подмассива не станет равным единице.
- Далее выполняется операция алгоритма, называемая слиянием. Два единичных массива сливаются в общий результирующий массив, при этом из каждого выбирается меньший элемент (сортировка по возрастанию) и записывается в свободную левую ячейку результирующего массива. После чего из двух результирующих массивов собирается третий общий отсортированный массив, и так далее. В случае, если один из массивов закончится, элементы другого дописываются в собираемый массив.
- В конце операции слияния, элементы перезаписываются из результирующего массива в исходный.

1.1.3 Блочная сортировка

Алгоритм сортировки, который работает путем распределения элементов массива по нескольким сегментам. Затем каждая корзина сортируется индивидуально, либо с использованием другого алгоритма сортировки, либо путем рекурсивного применения блочной сортировки [2].

Алгоритм работает следующим образом.

- Пусть l – минимальный, а r – максимальный элемент массива. Разобьем элементы на блоки, в первом будут элементы от l до $l + k$, во втором – от $l + k$ до $l + 2k$ и т.д., где $k = (r–l)/$количество блоков.
- Сортируем все получившиеся блоки другим алгоритмом сортировки.
- Выполним слияние блоков в единый массив.

2. Конструкторская часть

2.1 Трудоемкость алгоритмов

Введем модель трудоемкости для оценки алгоритмов.

1. Операции из списка (2.1) имеют трудоёмкость равную 1:

$$+, -, /, *, \%, =, +=, -=, *=, /=, \%=, ==$$

$$!=, <, >, <=, >=, [], ++, -- \tag{2.1}$$

2. Трудоёмкость оператора выбора if условие then A else B рассчитывается, как (2.2):

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.2}$$

3. Трудоёмкость цикла рассчитывается, как (2.3):

$$f_{for} = f_{\text{инициализации}} + f_{\text{сравнения}} + N(f_{\text{тела}} + f_{\text{инкремент}} + f_{\text{сравнения}}) \tag{2.3}$$

4. Трудоёмкость вызова функции равна 0.

Далее будет приведены оценки трудоемкости алгоритмов.

2.1.1 Сортировка слиянием

Лучший случай: массив отсортирован. Худший случай: массив отсортирован в обратном порядке.

1. Построчная оценка функции merge (Табл. 2.1).

Таблица 2.1 — Построчная оценка веса

Код	Вес
int i, j, k, c[50];	0
i = low;	1
k = low;	1
j = mid + 1;	2
while (i <= mid && j <= high)	3
{	0
if (arr[i] < arr[j])	3
{	0
c[k] = arr[i];	3
k++;	1
i++;	1
}	0
else	0
{	0
c[k] = arr[j];	3
k++;	1
j++;	1
}	0
}	0
while (i <= mid)	1
{	0
c[k] = arr[i];	3
k++;	1
i++;	1
}	0
while (j <= hight)	1
{	0

c[k] = arr[j];	3
k++;	1
j++;	1
}	0
for (i = low; i < k; i++)	3
{	0
arr[i] = c[i];	3
}	0
}	0

Трудоемкость: $1+1+2+n*(3+8)+k1*6+k2*6+n*3 = 14n+4+6(k1+k2) = O(n)$, где $k1 < n/2, k2 < n/2$

2. Проанализируем сложность mergeSort. Разобьём все “слияния”, выполненные в процессе сортировки массива на “слои”. Слияние двух частей изначального массива - первый слой, слияния частей каждой из этих двух частей (четвертей оригинального массива) - второй слой, и т.д. Необходимо заметить, что количество элементов в каждом слое равна N. Тогда из пункта 1 следует что на каждом слое все слияния выполняются за $O(N)$. Известно, что количество таких строк, называемое глубиной рекурсивного дерева, будет $log(N)$. Для наилучшего и наихудшего случая общая сложность mergeSort совпадает и равна: $O(n * log(n))$ [2].

2.1.2 Плавная сортировка

Лучший случай: массив отсортирован. Худший случай: массив отсортирован в обратном порядке.

Для определения сложности плавной сортировки необходимо для начала определить сложность пирамидальной сортировки. Основной структурой пирамидальной сортировки является бинарное дерево глубиной $k = log(n)$. Алгоритм состоит из двух частей.

1. Создание дерева.

2. Сортировка дерева.

На первом этапе перебирем n элементов. Само добавление в кучу обходится в $O(1)$, но затем для кучи нужно сделать просейку. Данная операция для каждого добавления обходится не более чем в k сравнений. Для второго этапа ситуация аналогичная. При обмене очередного максимума опять нужно просеять кучу, в корне которой он находился. На этой стадии сортировка выполняется для $n-1$ элементов с k сравнениеями. Таким образом общие затраты для пирамидальной сортировки составляют $k*n+k*(n-1)=O(n*log(n))$.

Перейдем к расчету сложности плавной сортировки. Для почти отсортированного массива строится дерево, удовлетворяющее первому следствию. Благодаря этому временная сложность просейки на первом этапе будет равна $O(1)$ так как просейка не опускается дальше первого элемента и общая сложность соответсвенно $O(n)$. Во время выполнения второго этапа результат трудоемкости остается таким же. Следовательно, общая трудоемкость для двух этапов будет $O(n)$.

Таким образом лучшая сложность плавной сортировки плавно стремится к $O(n)$ чем более упорядоченными будут входящие данные. А худшая сложность - $O(n*log(n))$ достигается, когда данные во входящем массиве не упорядоченны.

2.1.3 Блочная сортировка

Лучший случай: элементы входящего массива попали в разные корзины. Худший случай: элементы входящего массива попали в одну корзину.

Для наилучшего и наихудшего случаев сложность первого цикла будет одинакова и равна $O(n)$

Лучший случай: в каждый блок попадает по одному элементу. Сложность второго цикла равна $O(n)$, третьего - $O(n)*O(1)=O(n)$, четвертого аналогично. Тогда общая временная трудоемкость в наилучшем случае будет $O(n)$.

Худший случай: все элементы входного массива попадают в один блок. Сложность второго цикла равна $O(n)$, третьего - $(n-1)*O(1)+1*O(n*log(n))=O(n*log(n))$, четвертого - $(n-1)*O(2)+1*O(n)=O(n)$. Тогда общая временная трудоемкость в наихудшем случае будет $O(n*log(n))$.

2.2 Описание алгоритмов

В данном разделе будут рассмотрены схемы алгоритмов плавной сортировки (рис. 2.1), сортировки слиянием (рис. 2.9), блочной сортировки (рис. 2.12, 2.13). Схемы на рис. 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.10, 2.11 являются вспомогательными схемами.

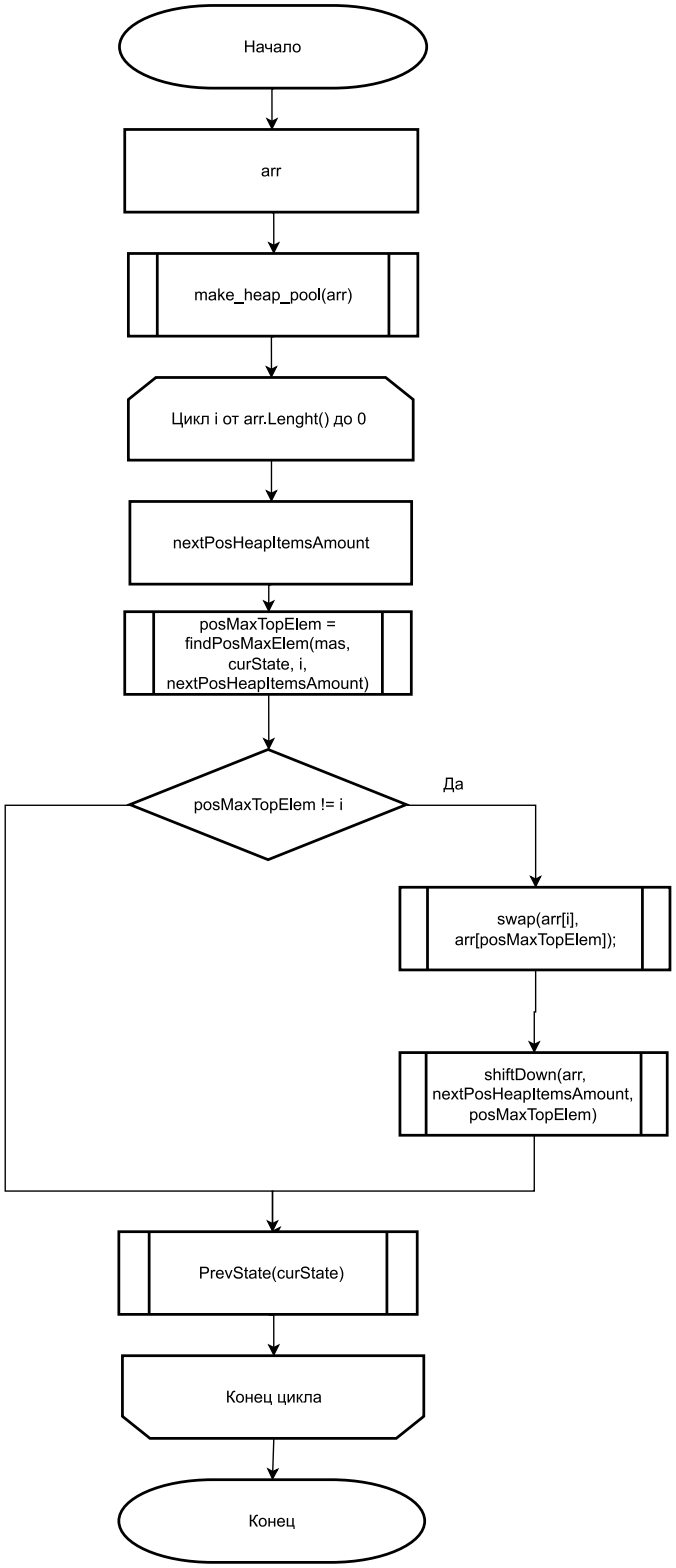

Рис. 2.1 – Схема алгоритма плавной сортировки

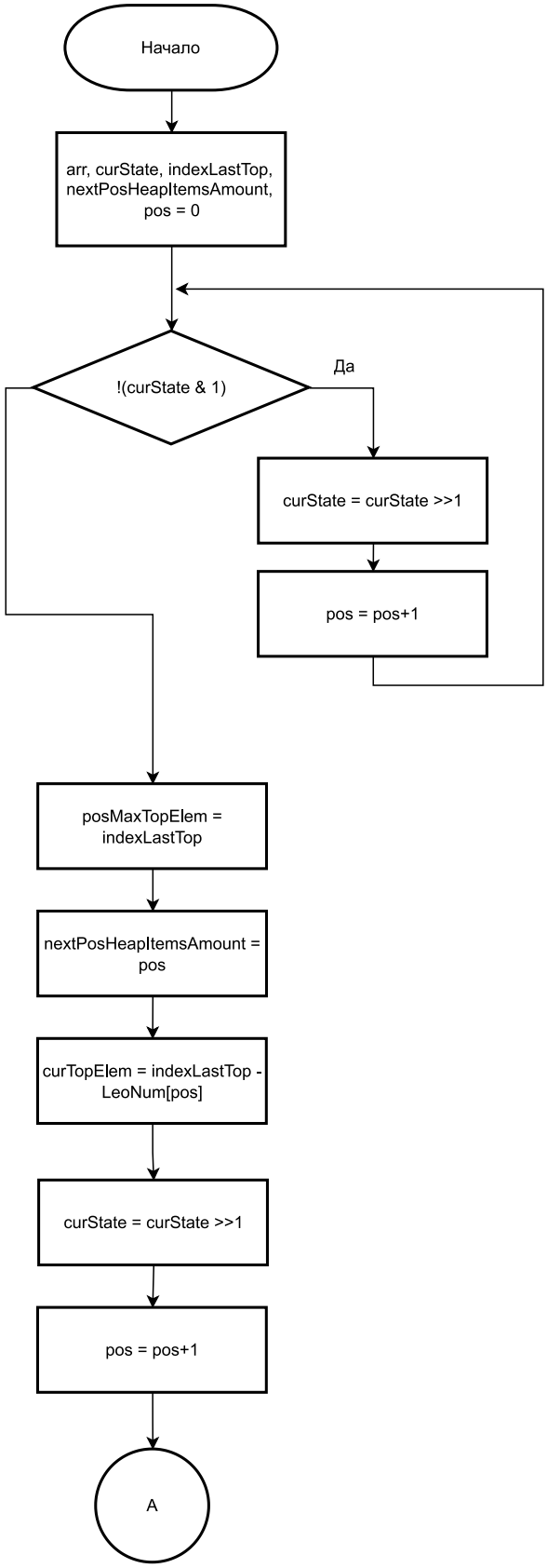

Рис. 2.2 – Схема функции findPosMaxElem, часть 1

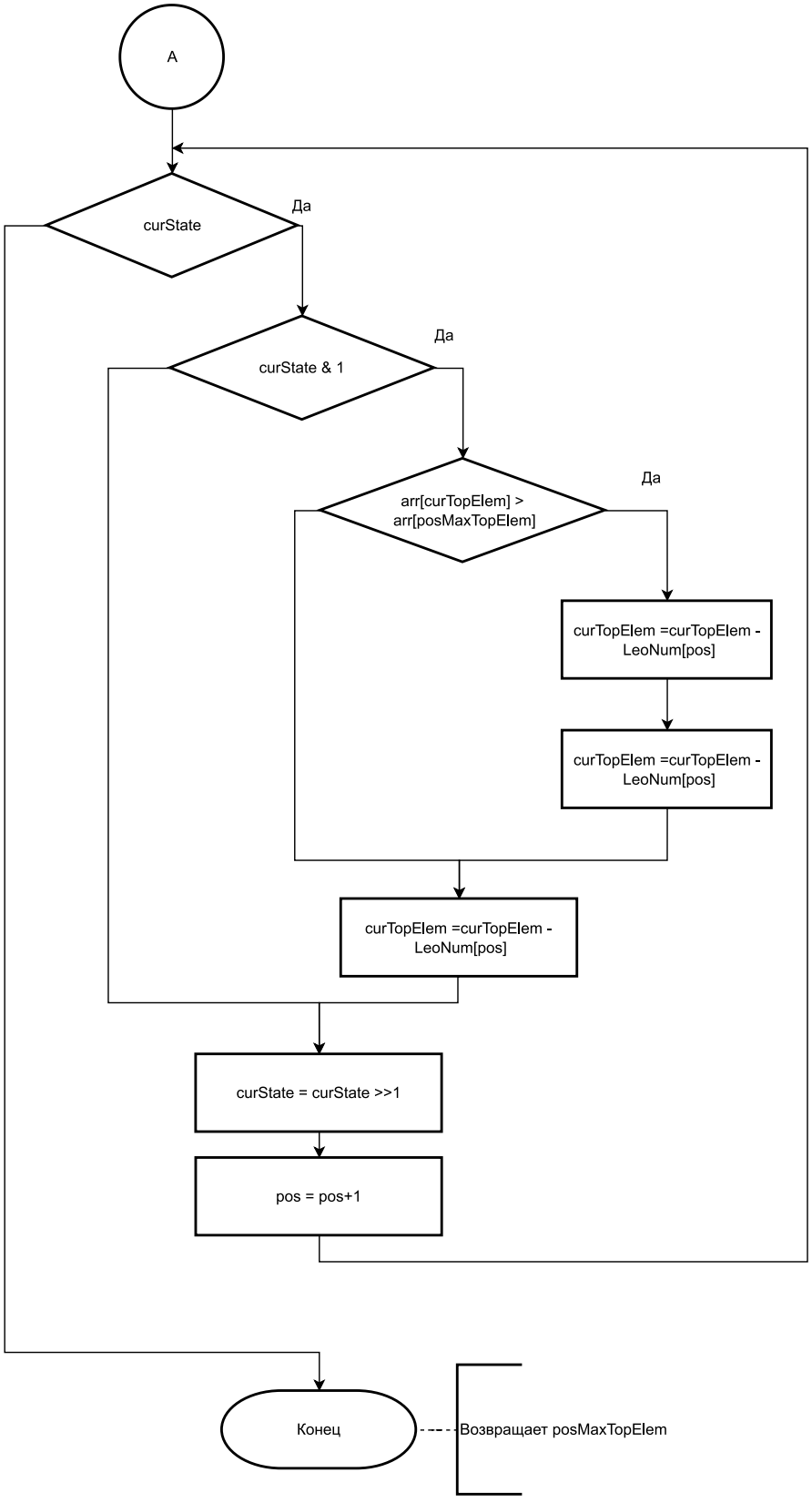

Рис. 2.3 – Схема функции findPosMaxElem, часть 2

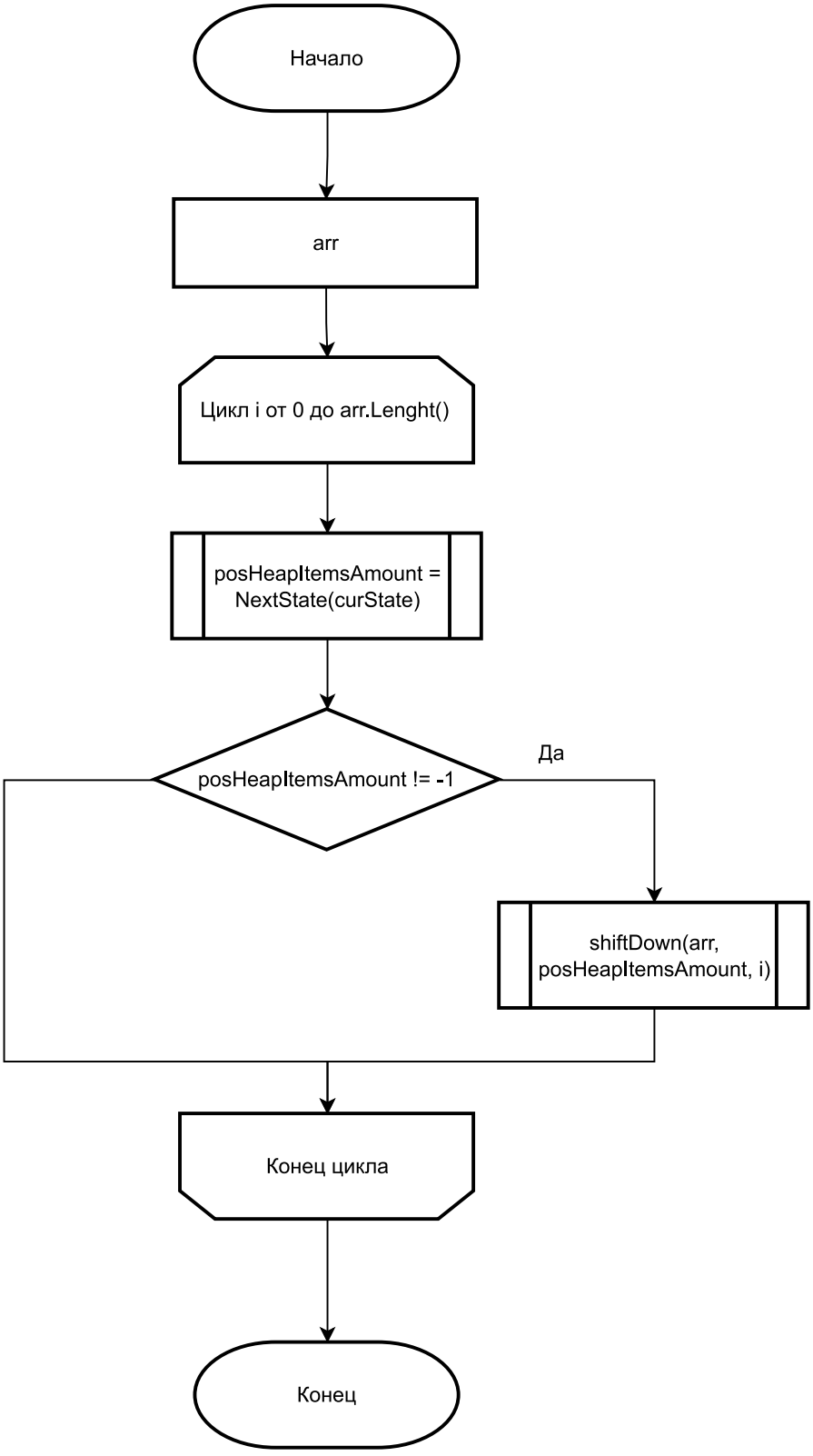

Рис. 2.4 – Схема функции makeHeapPool

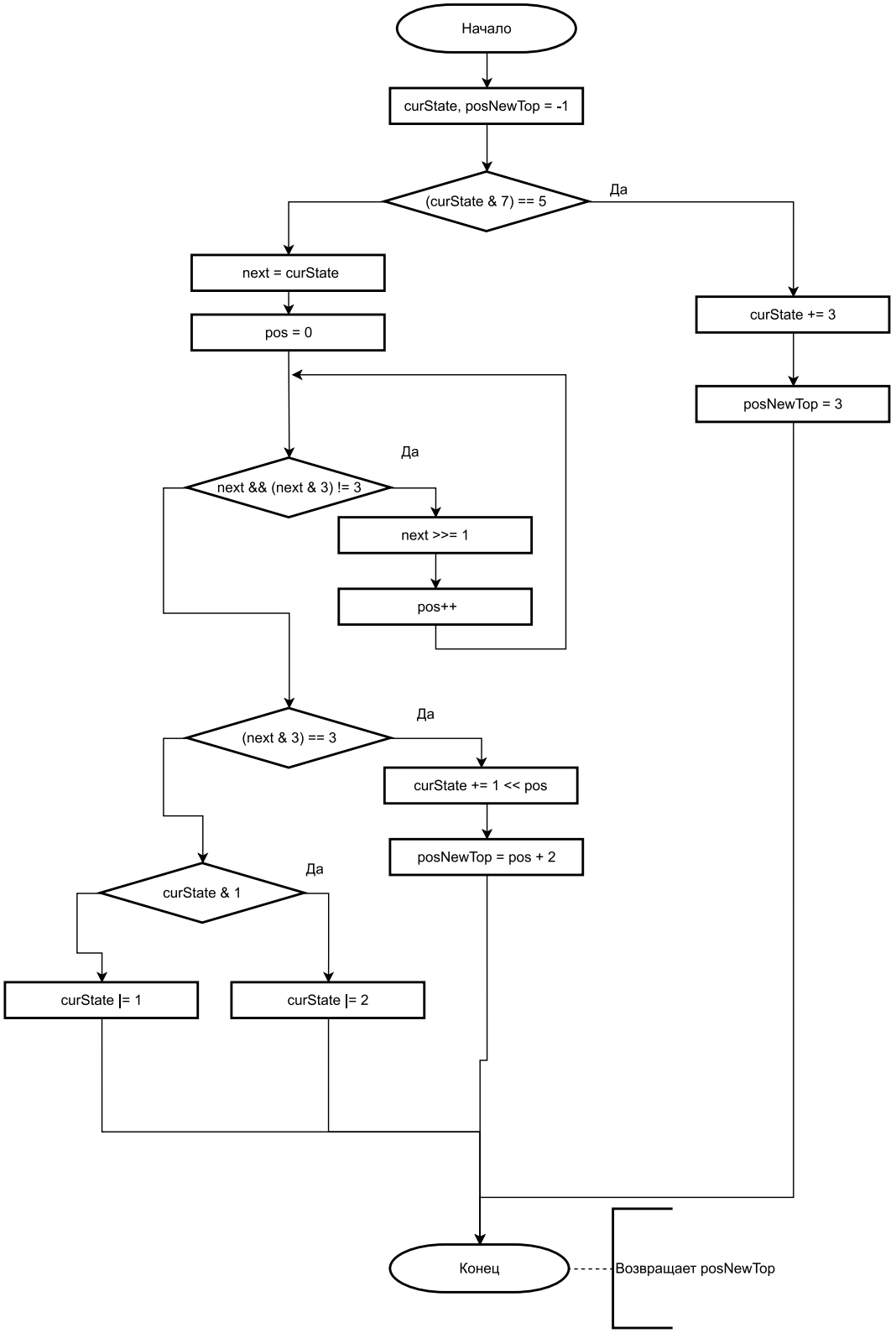

Рис. 2.5 – Схема функции nextState

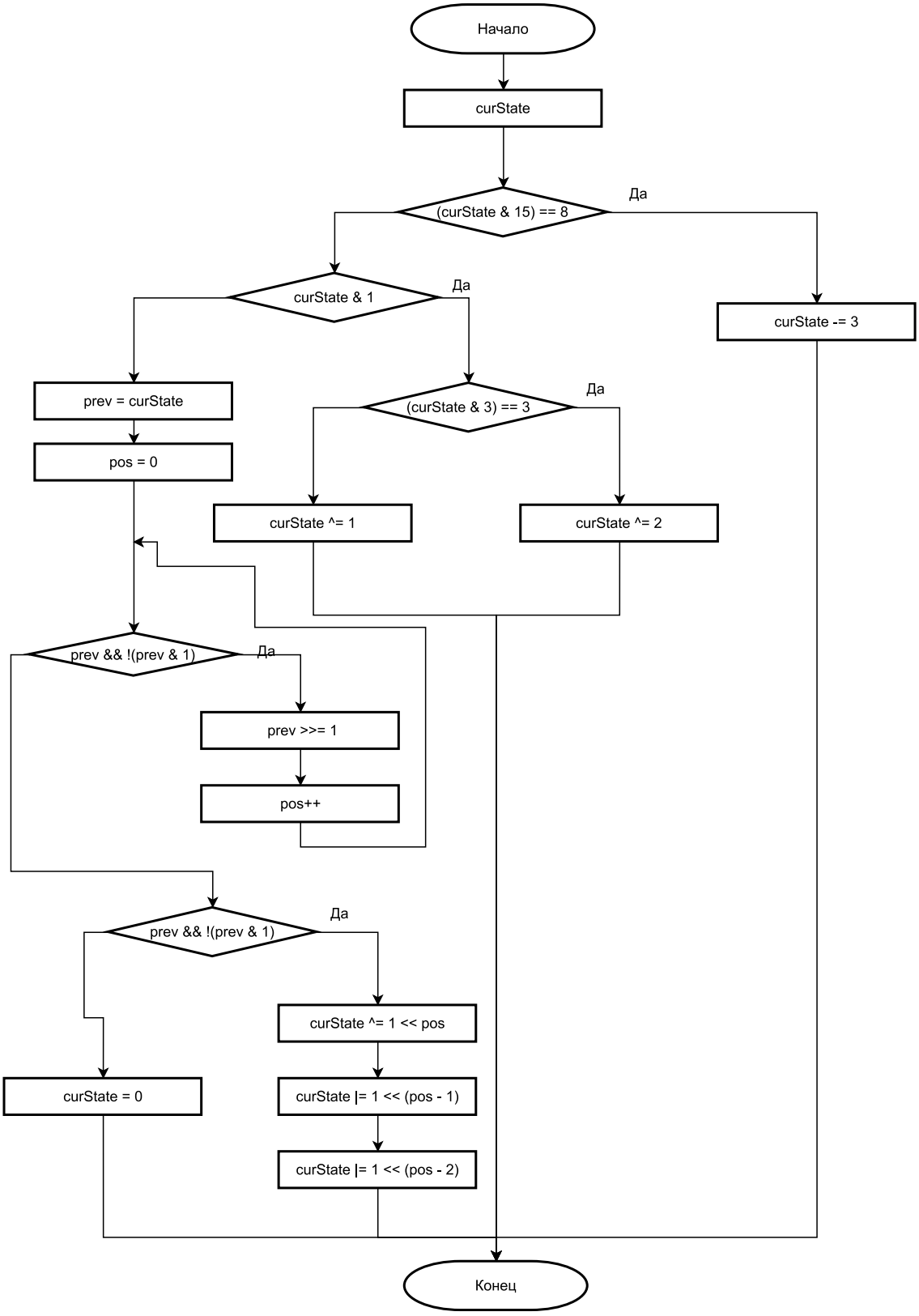

Рис. 2.6 – Схема функции prevState

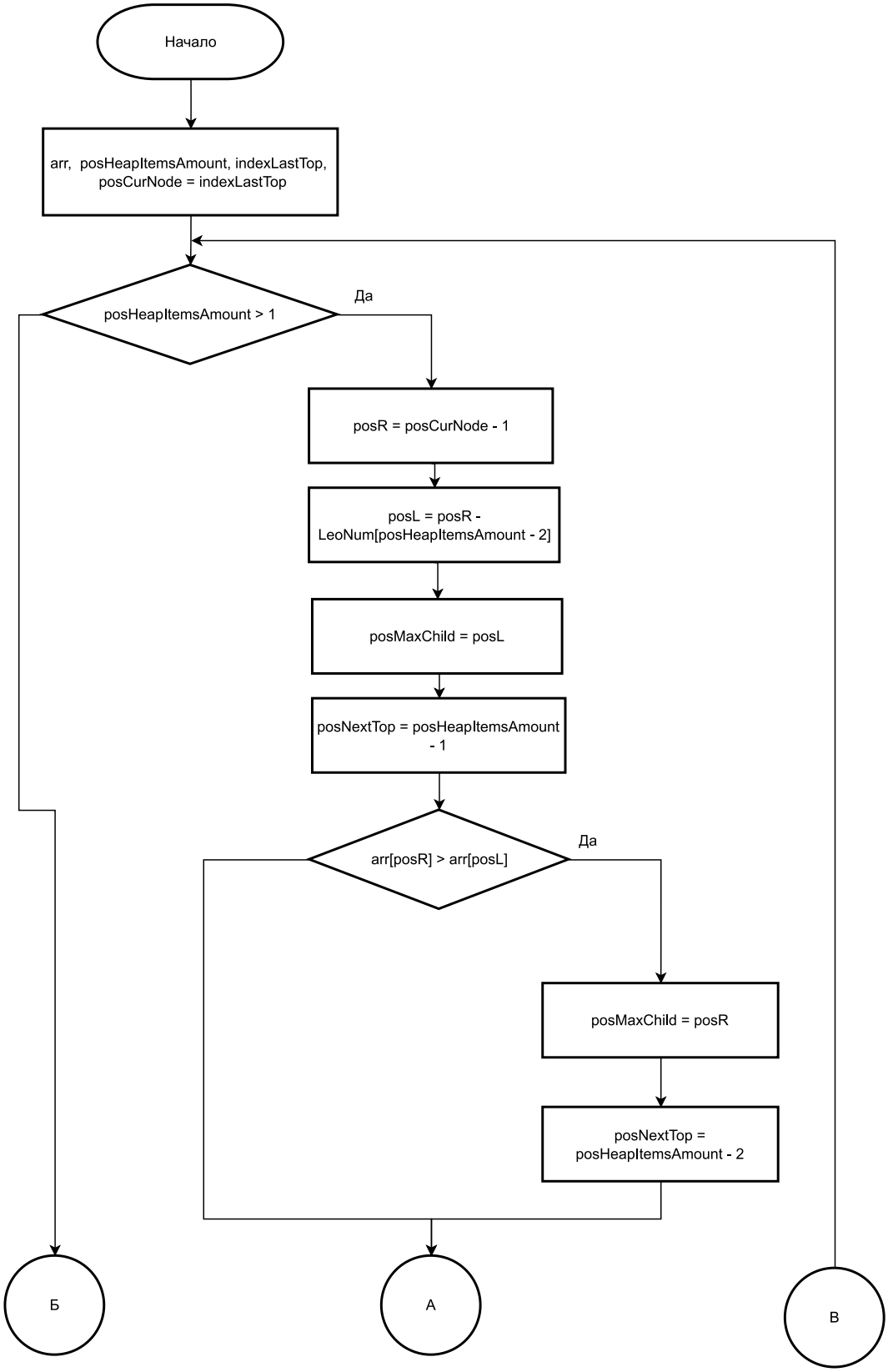

Рис. 2.7 – Схема функции shiftDown часть 1

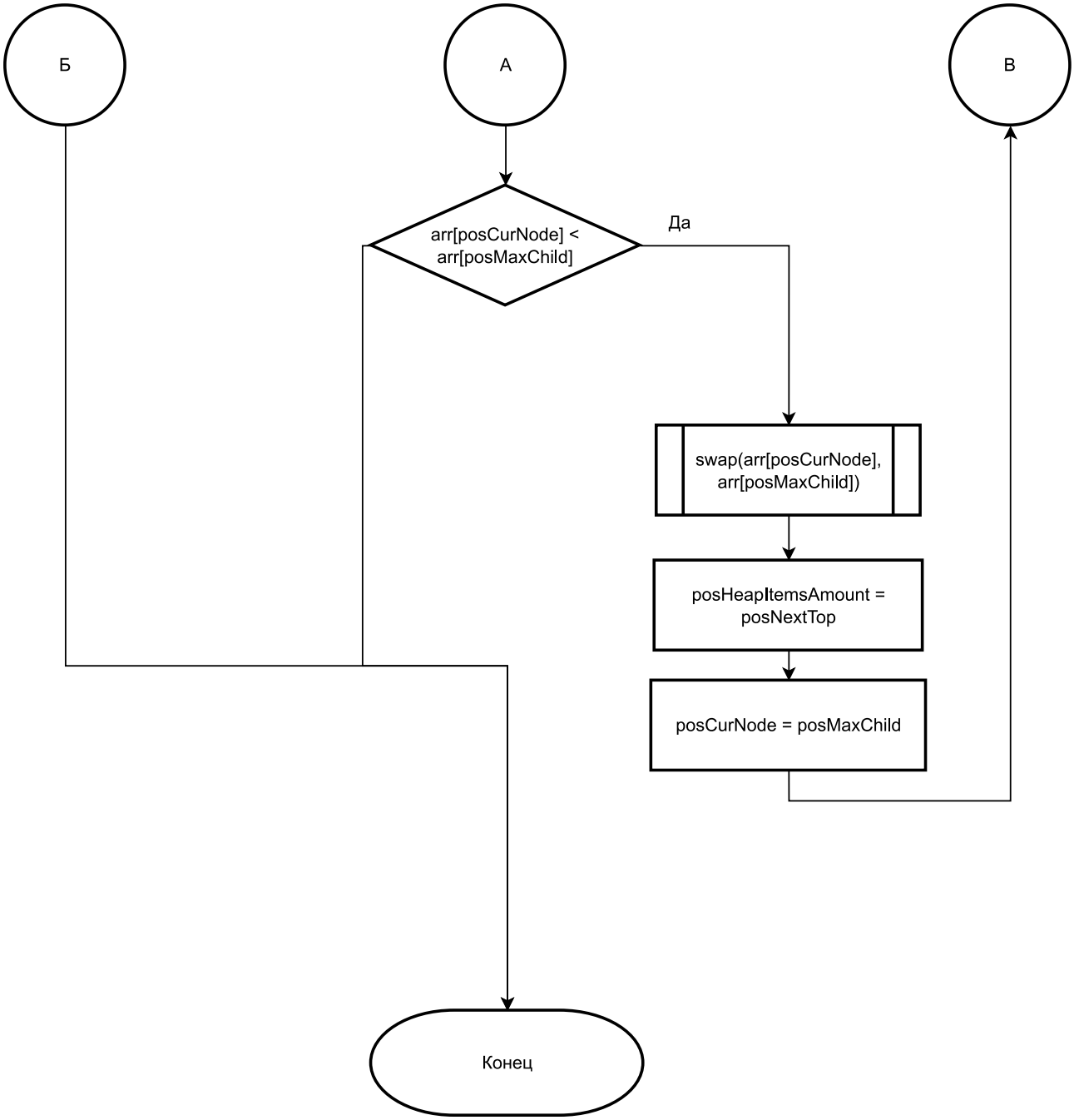

Рис. 2.8 – Схема функции shiftDown часть 2

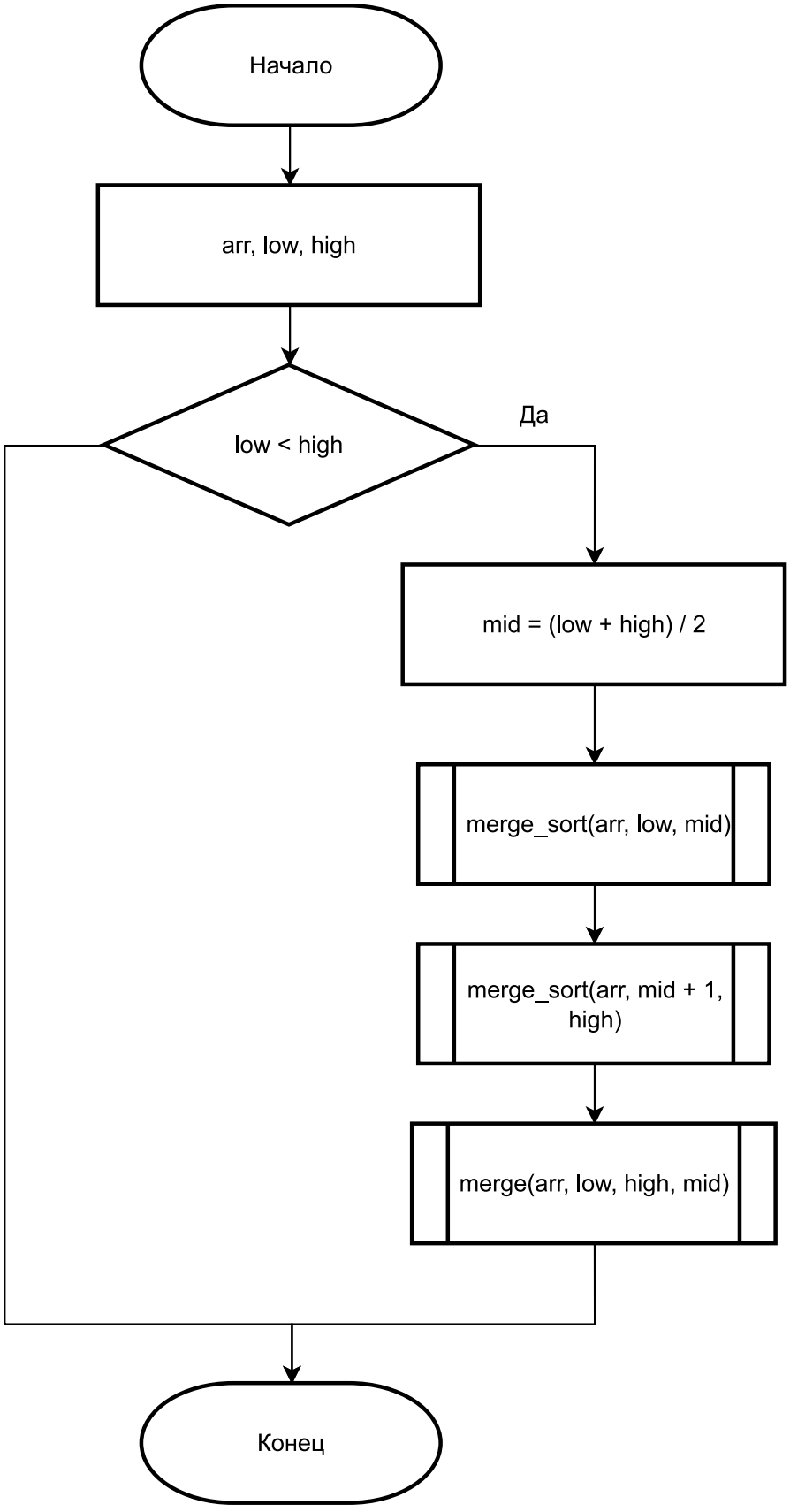

Рис. 2.9 – Схема алгоритма сортировки слиянием

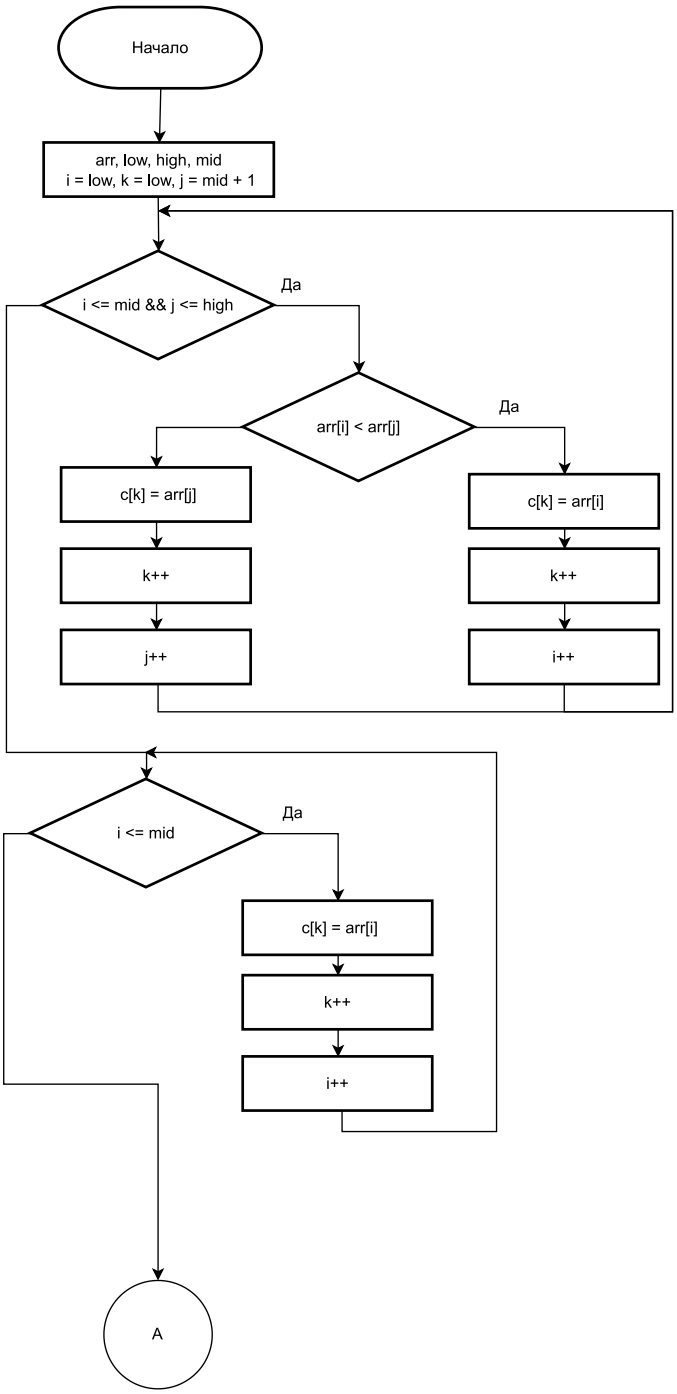

Рис. 2.10 – Схема функции merge часть 1

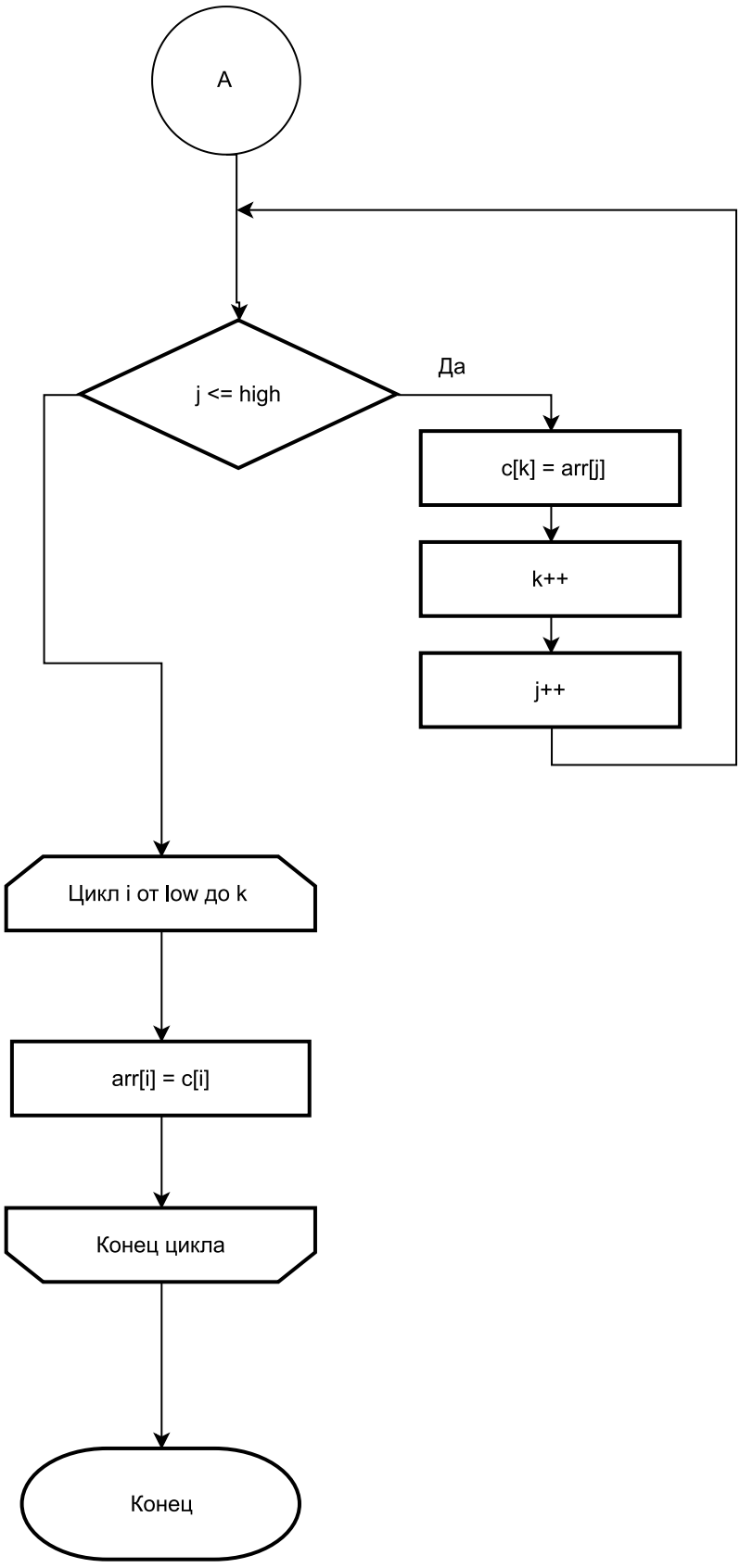

Рис. 2.11 – Схема функции merge часть 2

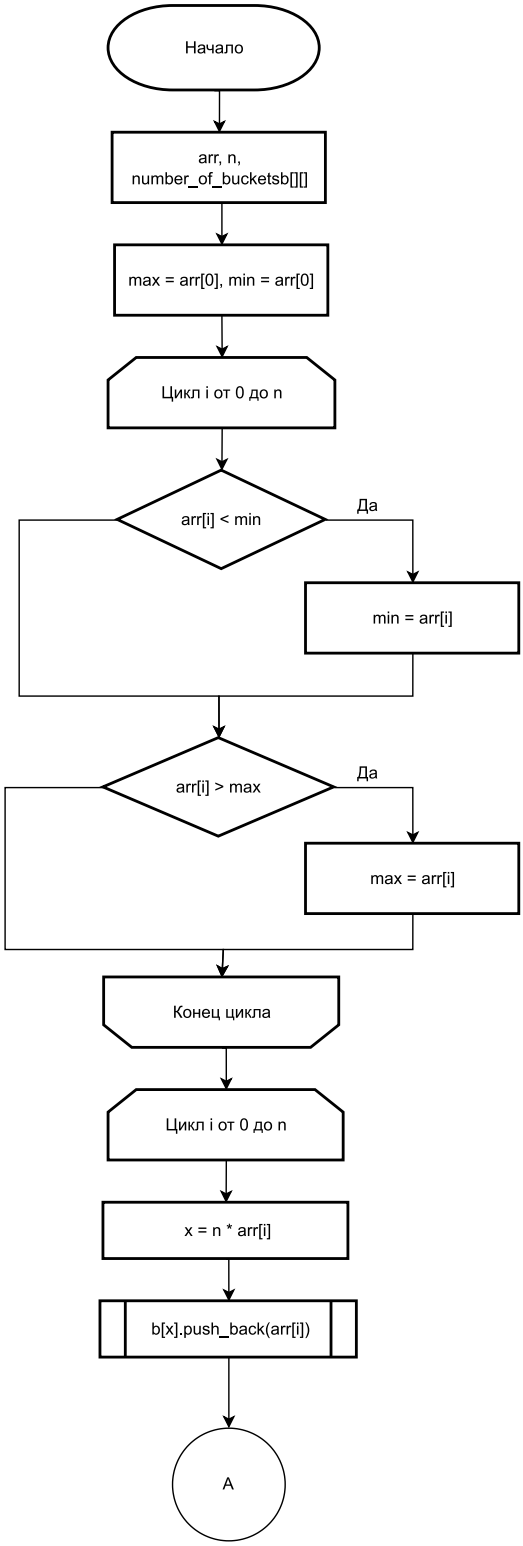

Рис. 2.12 – Схема алгоритма блочной сортировки часть 1

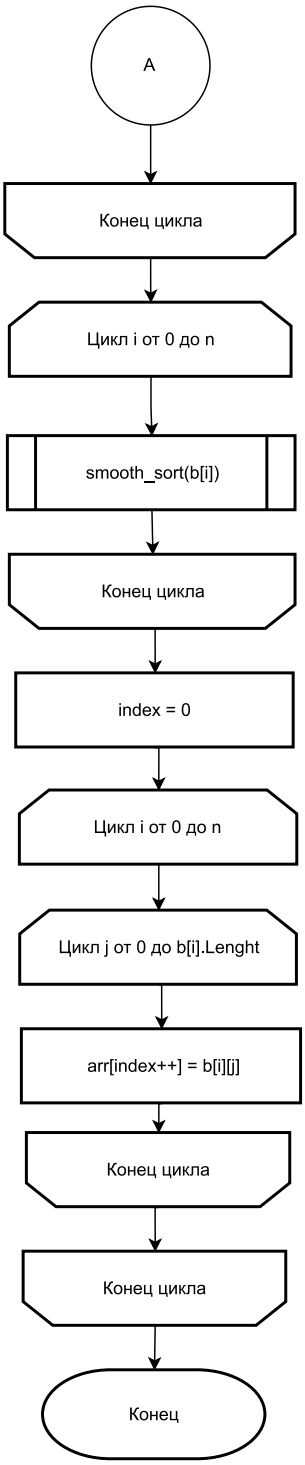

Рис. 2.13 – Схема алгоритма блочной сортировки часть 2

Вывод

В данном разделе были проанализированны три алгоритма сортировки. Рассчитаны трудоёмкости алгоритмов в лучшем случае (л.с.) и в худшем случае (х.с.).

Сортировка слиянием: л.с. – $O(nlog(n))$, х.с. – $O(nlog(n))$.

Плавная сортировка: л.с. – $O(n)$, х.с. – $O(nlog(n))$.

Блочная сортировка: л.с. – $O(n)$, х.с. – $O(nlog(n))$.

3. Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги сортировок.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования $c++$. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы, а также построить графики. Для визуализации результатов использовался язык $Python$.

Время работы было замерено с помощью функции *GetProcessTimes(...)* [3] из библиотеки $Windows.h$.

3.2 Сведения о модулях программы

Программа состоит из следующих модулей:

- $ConsoleApplication1.cpp$ - файл, содержащий весь служебный код;
- $sorst.cpp$ - файл, содержащий код всех сортировок;
- $time\_compare.cpp$ - файл, производящий замеры времени;
- $getCPUTime.cpp$ - файл, определяющий функцию замера времени;
- $arr\_gen.cpp$ - файл, генерирующий входной массив;
- $sorst.hpp$ - заголовочный файл модуля $sorst.cpp$;

- $time\_compare.hpp$ - заголовочный файл модуля $time\_compare.cpp$;
- $getCPUTime.hpp$ - заголовочный файл модуля $getCPUTime.cpp$;
- $arr\_gen.hpp$ - заголовочный файл модуля $arr\_gen.cpp$.

3.3 Реализация алгоритмов

В листингах 3.1, 3.3, 3.9, представлены реализации алгоритмов сортировок (слиянием, плавной, блочной). В листнингах 3.2, 3.4, 3.5, 3.6, 3.8, 3.7 представлены реализации вспомогательных функций.

Листинг 3.1 — Алгоритм сортировки слиянием

```
1  void merge_sort(vector<int>& arr, int low, int high)
2  {
3    int mid;
4    if (low < high) {
5      mid = (low + high) / 2;
6      merge_sort(arr, low, mid);
7      merge_sort(arr, mid + 1, high);
8      merge(arr, low, high, mid);
9    }
10 }
```

Листинг 3.2 — Алгоритм функции merge

```
void merge(vector<int>& arr, int low, int high, int mid)
{
  int i, j, k, c[50];
  i = low;
  k = low;
  j = mid + 1;
  while (i <= mid && j <= high) {
    if (arr[i] < arr[j]) {
      c[k] = arr[i];
      k++;
      i++;
    }
    else {
      c[k] = arr[j];
      k++;
      j++;
    }
  }
  while (i <= mid) {
    c[k] = arr[i];
    k++;
    i++;
  }
  while (j <= high) {
    c[k] = arr[j];
    k++;
    j++;
  }
  for (i = low; i < k; i++) {
    arr[i] = c[i];
  }
}
```

Листинг 3.3 — Алгоритм плавной сортировки

```
typedef int INT;

L(N) = L(N-1) + L(N-2) + 1
int LeoNum[] = { 1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465,
    753, 1219, 1973, 3193, 5167, 8361, 13529, 21891, 35421, 57313,
    92735, 150049, 242785, 392835, 635621, 1028457, 1664079, 2692537,
    4356617, 7049155, 11405773, 18454929, 29860703, 48315633, 78176337,
    126491971, 204668309, 331160281, 535828591, 866988873, 1402817465 };
INT curState;

void smooth_sort(vector<int>& mas)
{
  make_heap_pool(mas);

  for (int i = mas.size() - 1; i >= 0; i--)
  {
    int nextPosHeapItemsAmount;
    int posMaxTopElem = findPosMaxElem(mas, curState, i,
       nextPosHeapItemsAmount);
    if (posMaxTopElem != i)
    {
      swap(mas[i], mas[posMaxTopElem]);
      shiftDown(mas, nextPosHeapItemsAmount, posMaxTopElem);
    }
    PrevState(curState);
  }
}
```

Листинг 3.4 — Алгоритм функции findPosMaxElem

```
int findPosMaxElem(vector<int>& mas, INT curState, int indexLastTop,
    int& nextPosHeapItemsAmount)
{
  int pos = 0;
  while (!(curState & 1))
  {
    curState >>= 1;
    pos++;
  }

  int posMaxTopElem = indexLastTop;
  nextPosHeapItemsAmount = pos;

  int curTopElem = indexLastTop - LeoNum[pos];

  curState >>= 1;
  pos++;
  while (curState)
  {
    if (curState & 1)
    {
      if (mas[curTopElem] > mas[posMaxTopElem])
      {
        posMaxTopElem = curTopElem;
        nextPosHeapItemsAmount = pos;
      }
      curTopElem -= LeoNum[pos];
    }
    curState >>= 1;
    pos++;
  }

  return posMaxTopElem;
}
```

Листинг 3.5 — Алгоритм функции make_heap_pool

```
void make_heap_pool(vector<int>& mas)
{
  for (size_t i = 0; i < mas.size(); i++)
  {
    int posHeapItemsAmount = NextState(curState);
    if (posHeapItemsAmount != -1)
    shiftDown(mas, posHeapItemsAmount, i);
  }
}
```

Листинг 3.6 — Алгоритм функции shiftDown

```
void shiftDown(vector<int>& mas, int posHeapItemsAmount, int
    indexLastTop)
{
  int posCurNode = indexLastTop;
  while (posHeapItemsAmount > 1)
  {
    int posR = posCurNode - 1;
    int posL = posR - LeoNum[posHeapItemsAmount - 2];

    int posMaxChild = posL;
    int posNextTop = posHeapItemsAmount - 1;
    if (mas[posR] > mas[posL])
    {
      posMaxChild = posR;
      posNextTop = posHeapItemsAmount - 2;
    }
    if (mas[posCurNode] < mas[posMaxChild])
    {
      swap(mas[posCurNode], mas[posMaxChild]);
      posHeapItemsAmount = posNextTop;
      posCurNode = posMaxChild;
    }
    else
    break;
  }
}
```

Листинг 3.7 — Алгоритм функции PrevState

```
void PrevState(INT& curState)
{
  if ((curState & 15) == 8)
  {
    curState -= 3;
  }
  else
  if (curState & 1)
  {
    if ((curState & 3) == 3)
    curState ^= 2;
    else
    curState ^= 1;
  }
  else
  {
    INT prev = curState;
    int pos = 0;
    while (prev && !(prev & 1))
    {
      prev >>= 1;
      pos++;
    }
    if (prev)
    {
      curState ^= 1 << pos;
      curState |= 1 << (pos - 1);
      curState |= 1 << (pos - 2);
    }
    else
    curState = 0;
  }
}
```

Листинг 3.8 — Алгоритм функции NextState

```
int NextState(INT& curState)
{
  int posNewTop = -1;

  if ((curState & 7) == 5)
  {
    curState += 3;
    posNewTop = 3;
  }
  else
  {
    INT next = curState;
    int pos = 0;
    while (next && (next & 3) != 3)
    {
      next >>= 1;
      pos++;
    }
    if ((next & 3) == 3)
    {
      curState += 1 << pos;
      posNewTop = pos + 2;
    }
    else
    if (curState & 1)
    curState |= 2;
    else
    curState |= 1;
  }
  return posNewTop;
}
```

Листинг 3.9 — Алгоритм блочной сортировки

```
void bucketSort(vector<int>& arr, int n, int number_of_buckets)
{
  vector<vector<int>> b;
  int max = arr[0], min = arr[0];

  for (int i = 0; i < n; i++)
  {
    if (arr[i] < min)
    min = arr[i];
    if (arr[i] > max)
    max = arr[i];
  }

  for (int i = 0; i < number_of_buckets; i++)
  {
    vector<int> k;
    b.push_back(k);
  }

  for (int i = 0; i < n; i++)
  {
    int x = floor(number_of_buckets * (arr[i] - min) / (max + 1 - min))
        ;
    b[x].push_back(arr[i]);
  }
  for (int i = 0; i < number_of_buckets; i++)
  smooth_sort(b[i]);
  int index = 0;
  for (int i = 0; i < number_of_buckets; i++)
  for (int j = 0; j < b[i].size(); j++)
  arr[index++] = b[i][j];
}
```

3.4 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы сортировки. Тесты для всех сортировок пройдены успешно.

Таблица 3.1 — Функциональные тесты

Входной массив	Ожидаемый результат	Результат
$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[9, 7, 5, 1, 4]$	$[1, 4, 5, 7, 9]$	$[1, 4, 5, 7, 9]$
$[5, 4, 3, 2, 1]$	$[1, 2, 3, 4, 5]$	$[1, 2, 3, 4, 5]$
$[5]$	$[5]$	$[5]$
[]	[]	[]

Вывод

В данном разделе были представлены реализации алгоритмов плавной сортировки, сортировки слиянием и блочной сортировки.

4. Исследовательская часть

В данном разделе будут приведены примеры работы программа, а также проведен сравнительный анализ алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры времени представлены далее:

- операционная система Windows 11 Pro Версия 22H2 (22621.674) [4];
- память 16 ГБ;
- процессор 11th Gen Intel(R) Core(TM) i5-11400 @ 2.60GГц 2.59 Ггц [5].

При тестировании компьютер был включен в сеть электропитания. Во время тестирования устройство было нагружено только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы. На экран выводятся результаты заемров времени для разных длин массива и разных видов сортировок в мс.

```
Results for the best case:
                        Time in ms
Len          Smooth           Merge          Bucket
 10    0.000390625      0.00195312       0.0159063
 20    0.000796875      0.00432813       0.0206562
 30     0.00123437      0.00689062        0.026625
 40     0.00164062       0.0096875          0.0295
 50      0.0021875       0.0122344        0.031625
 60     0.00270312       0.0149688       0.0338906
 70     0.00321875       0.0178594       0.0355938
 80      0.0038125       0.0206563       0.0381563
 90     0.00432813       0.0237969        0.041875
100     0.00515625       0.0278125       0.0439063
200       0.010625        0.056875       0.0617188
300      0.0170312       0.0876563       0.0792188
400      0.0257812        0.118438       0.0959375
500      0.0346875        0.149688        0.112031

Results for normal case:
                        Time in ms
Len          Smooth           Merge          Bucket
 10    0.000796875      0.00201563           0.016
 20     0.00235937       0.0046875       0.0205625
 30       0.003875      0.00735937       0.0247344
 40     0.00645312       0.0102969       0.0304219
 50     0.00984375       0.0132656       0.0351406
 60      0.0121875       0.0165312       0.0367969
 70        0.01425       0.0194531       0.0414219
 80        0.01925       0.0229219       0.0434531
 90      0.0217812       0.0259531       0.0489062
100      0.0257813       0.0295312       0.0534375
200      0.0596875       0.0642188       0.0882813
300       0.101562             0.1        0.129219
400       0.149219        0.137813         0.16625
```

Рис. 4.1 – Пример работы программы

4.3 Время выполнения реализаций алгоритмов

Как было сказано выше, используется функция замера процессорного времени GetProcessTimes(...) из библиотеки Windows.h. Функция возвращает пользовательское процессорное временя типа float.

Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

Входные данные: целые числа от 0 до длины массива.

Результаты замеров времени работы реализаций алгоритмов сортировки на различных входных данных (в мс) приведены в таблицах 4.1, 4.2 и 4.3.

Таблица 4.1 — Процессорное время работы реализаций алгоритмов на отсортированных данных

Размер	Плавная	Слиянием	Блочная
100	0.00515625	0.0278125	0.0439063
200	0.010625	0.056875	0.0617188
300	0.0170312	0.0876563	0.0792188
400	0.0257812	0.118438	0.09593751
500	0.0346875	0.149688	0.112031

Таблица 4.2 — Процессорное время работы реализаций алгоритмов на случайных данных

Размер	Плавная	Слиянием	Блочная
100	0.0257813	0.0295312	0.0534375
200	0.0596875	0.0642188	0.0882813
300	0.101562	0.1	0.129219
400	0.149219	0.137813	0.16625
500	0.191562	0.177813	0.217812

Таблица 4.3 — Процессорное время работы реализаций алгоритмов на отсортированных в обратном порядке данных

Размер	Плавная	Слиянием	Блочная
100	0.0259375	0.0267188	0.0546875
200	0.060625	0.0559375	0.0909375
300	0.0960938	0.0876562	0.134844
400	0.135313	0.117031	0.173281
500	0.176563	0.147813	0.212031

Также на рисунках 4.2, 4.4, 4.3 приведены графические результаты замеров времени работы сортировок в зависимости от размера входного массива.

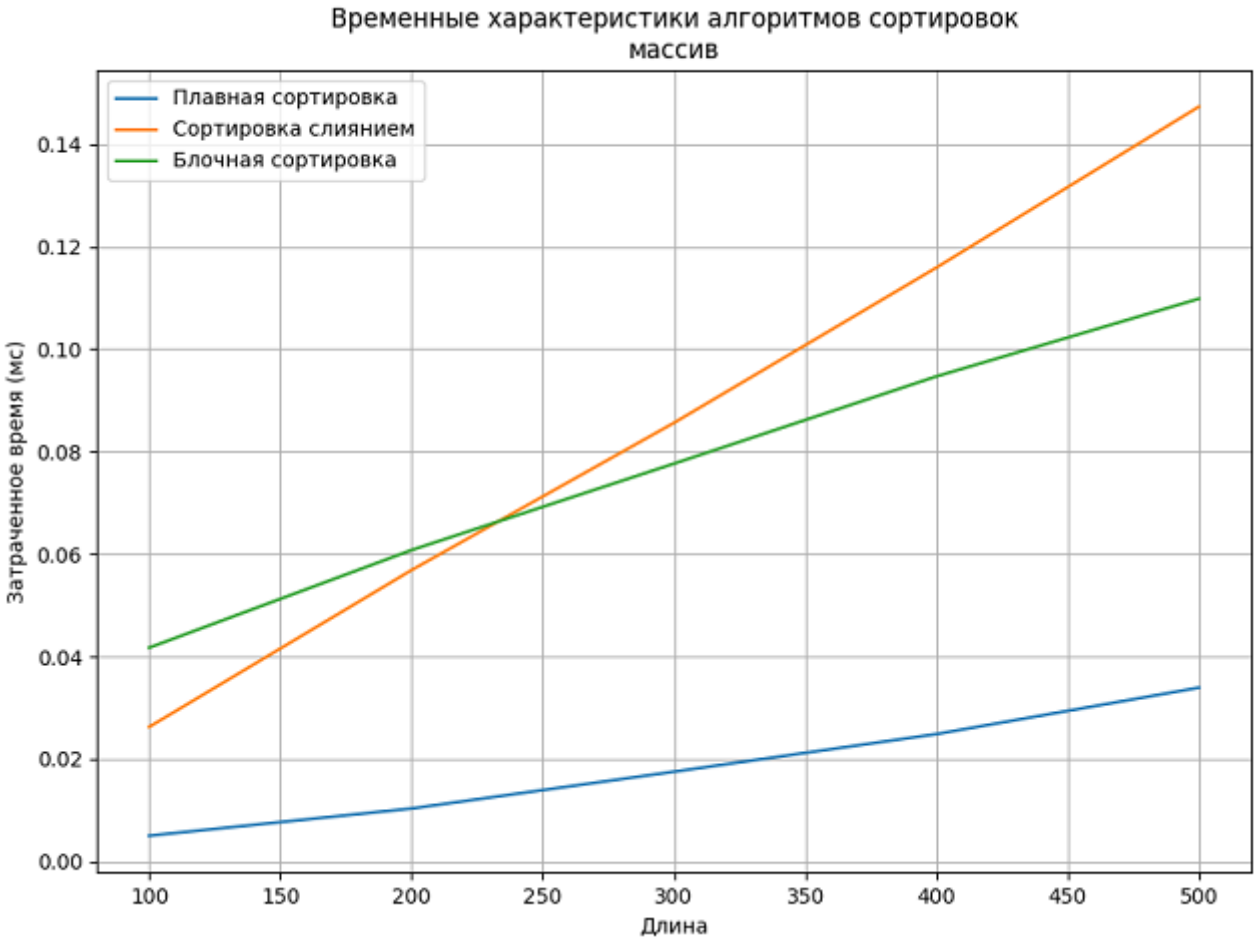

Рис. 4.2 – Процессорное время вычислений: упорядоченные массивы

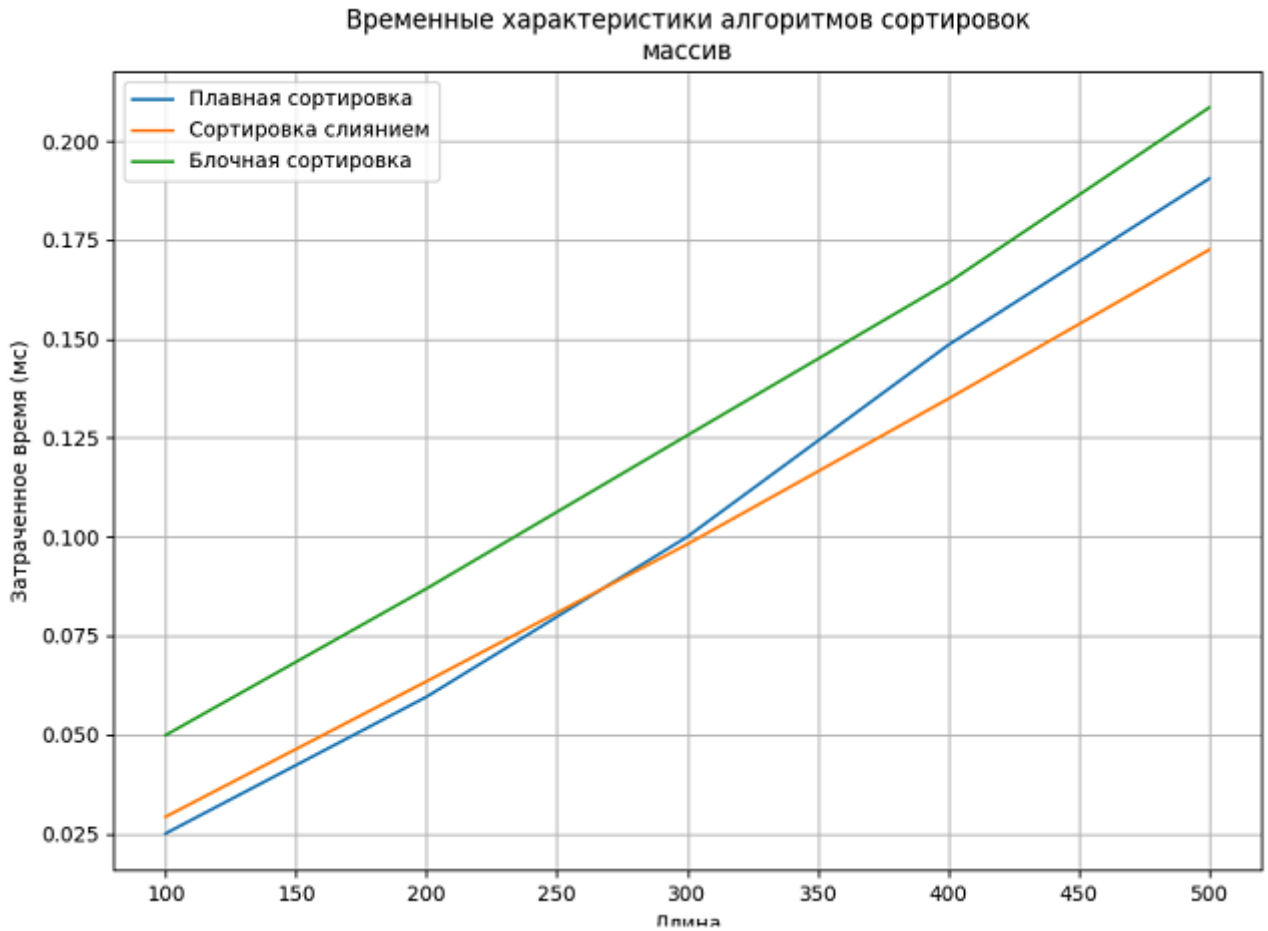

Рис. 4.3 – Процессорное время вычислений: массив случайных данных

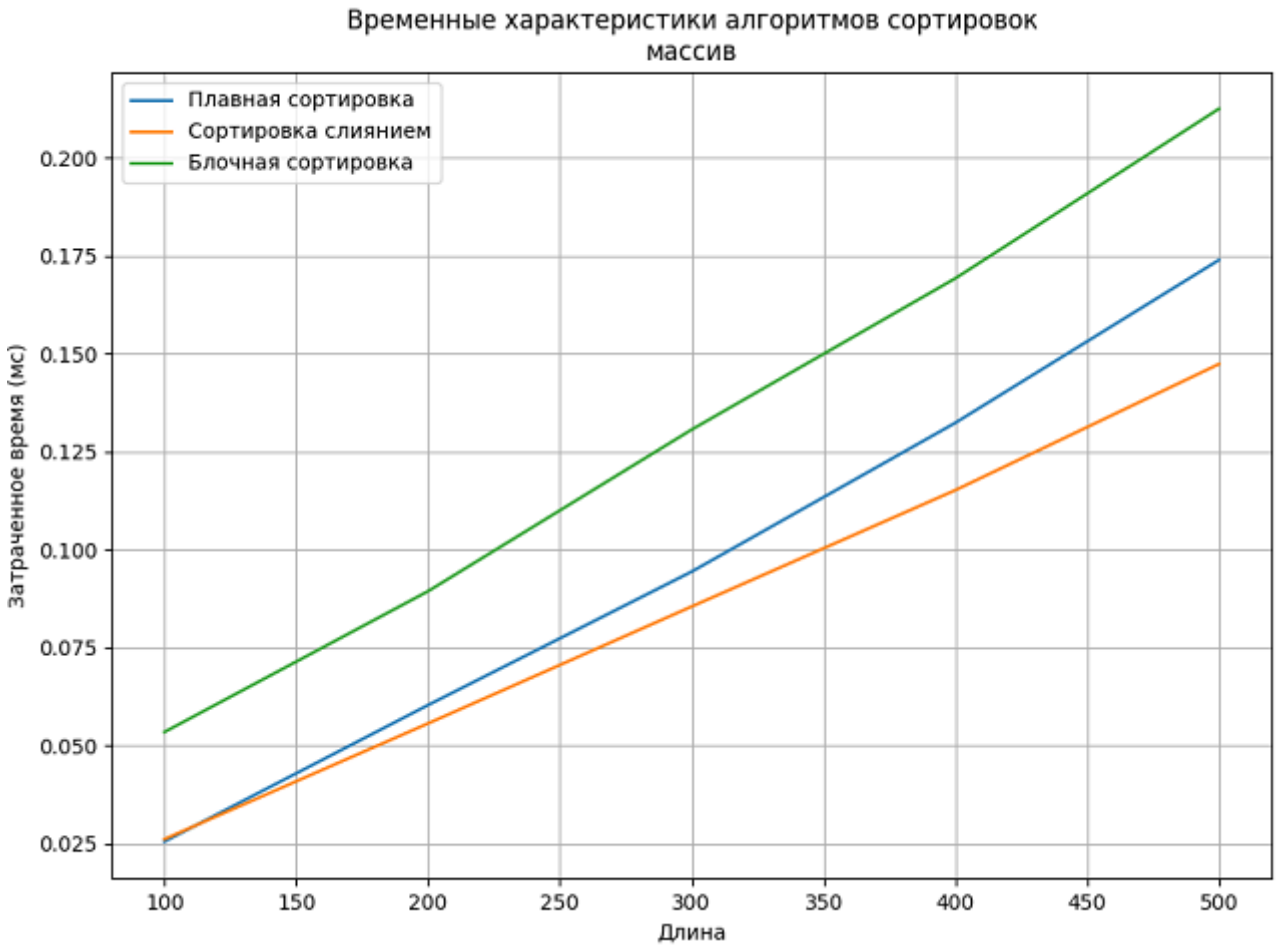

Рис. 4.4 – Процессорное время вычислений: обратно упорядоченные массивы

Вывод

Исходя из полученных результатов, на отсортированных данных плавная сортировка оказалась быстрее двух остальных. Также заметно, что зависимость процессорного времени выполнения реализации алгоритма от размера массива для данного вида упорядочивания принимает логарифмический вид при ухудшении упорядоченности входных данных. Для всех трех случаев сложность блочной сортировки подчиняется линейному закону, что говорит о том, что количество ”корзин” было выбрано оптимально.

Теоретические результаты замеров и полученные практически результаты совпадают.

Заключение

Цель, которая была поставлена в начале лабораторной работы была достигнута: выполнен анализ трудоемкости реализаций алгоритмов сортировки.

В ходе выполнения лабораторной работы были решены все задачи.

1. Были изучены и реализованы алгоритмы сортировки: слиянием, плавная, блочная.
2. Была выбрана модель вычисления и проведен сравнительный анализ трудоёмкостей выбранных алгоритмов сортировки.
3. На основе экспериментальных данных проведено сравнение выбранных алгоритмов сортировки.
4. Подготовлен отчет о лабораторной работе.

В ходе проделанной работы было выявлено, что блочная сортировка является самой трудоемкой для всех входных видов данных. В то время как сортировка слиянием, значительно опережает блочную. Так же на уже отсортированных входных данных лучше использовать плавную сортировку из-за ее скорости.

Список использованных источников

[1] Dijkstra E. Smoothsort, an alternative for sorting in situ // Science of Computer Programming. 1982. Vol. 1, no. 3. P. 223—233.

[2] Cormen T. H. Introduction to Algorithms, 3rd Edition. MIT Press, 2009. P. 200 — 204.

[3] GetProcessTimes function [Эл. ресурс]. Режим доступа: https://clck.ru/32NCYi (дата обращения: 13.10.2022).

[4] Windows 11, version 22H2 [Эл. ресурс]. Режим доступа: https://clck.ru/32NCXx (дата обращения: 14.10.2022).

[5] Процессор Intel® Core™ i7 [Эл. ресурс]. Режим доступа: https://clck.ru/yeQa8 (дата обращения: 14.10.2022).