Федеральное государственное бюджетное

образовательное учреждение

высшего образования

«МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ»

(национальный исследовательский университет)

Факультет № 8 «Компьютерные науки и прикладная математика»

Кафедра 806 «Вычислительная математика и программирование»

Задание № 9
«Сортировка и поиск»

по дисциплине «Практикум на ЭВМ»

2 семестр

Студент	Мезенин О.А.
Группа	М8О-106Б-21
Преподаватель	Дубинин А.В.
Оценка	

Москва, 2022

Содержание

1 Введение **1**

2 Алгоритмы поиска **2**

2.1 Линейный поиск 2

2.2 Двоичный поиск 2

3 Алгоритмы сортировки **3**

3.1 Линейный выбор с обменом 3

3.2 Линейный выбор с подсчётом 4

3.3 Сортировка пузырьком 5

3.4 Шейкер-сортировка 5

3.5 Метод простой вставки 6

3.6 Метод двоичной вставки 7

3.7 Метод Шелла 8

3.8 Турнирная сортировка 9

3.9 Пирамидальная сортировка с просеиванием 10

3.9.1 Построение пирамиды 10

3.9.2 Сортировка 12

3.10 Простое двухпоточное слияние 13

3.11 Быстрая сортировка Хоара 15

4 Практика **18**

4.1 Описание функций 18

4.2 Тесты 21

5 Выводы **23**

6 Список источников **24**

1 Введение

Задачи сортировки и поиска являются одними из наиболее часто встречающихся задач в программировании. К задачам поиска относят не только отыскание слов в тексте или полей в базе данных, но и, например, поиск в интернете. Сортировка данных необходима, как правило, для облегчения поиска.

Для решения задач сортировки и поиска существует множество алгоритмов, имеющих разную эффективность, преимущества и недостатки. Целью работы является ознакомление с некоторыми из этих алгоритмов.

2 Алгоритмы поиска

Для рассмотрения алгоритмов предположим, что имеется массив данных, допускающий прямой доступ к каждому элементу. Требуется найти элемент key.

2.1 Линейный поиск

Линейный поиск заключается в последовательном просмотре элементов массива, начиная с начала, до тех пор, пока не встретится элемент key. Такой метод является самым простым и может применяться для неупорядоченной последовательности, но временная сложность такого алгоритма составляет $O(n)$.

2.2 Двоичный поиск

Такой поиск может применяться только для упорядоченного множества. Алгоритм заключается в последовательном половинном делении массива.

На каждой итерации рассматривается промежуточный элемент m, стоящий в середине массива. Если элемент m равен элементу key, то поиск успешно заканчивается. Если m больше key, то отбрасывается левая часть массива, т.е. все элементы $i \leq m$ больше не рассматриваются в дальнейшем поиске (левая граница поиска сдвигается к m). Если m меньше key, то отбрасывается правая половина рассматриваемого массива (правая граница поиска сдвигается к m).

Поиск идёт до тех пор, пока не найдётся элемент key или пока левая и правая границы поиска не совпадут. Если левая и правая границы совпали, значит, элемент key отсутствует в массиве.

Сложность такого алгоритма составляет $O(\log_2 n)$, но метод требует отсортированный массив.

3 Алгоритмы сортировки

Сортировка - это процесс перестановки объектов в множестве в некотором порядке. Различают *внутренние* и *внешние* сортировки. Внутренние сортировки работают с массивами, хранящимся в оперативной, внутренней памяти машины с произвольным доступом. Внешние сортировки работают с файлами, которые размещаются в медленной, но более емкой и долговременной внешней памяти.

Метод сортировки называется *устойчивым*, если в процессе сортировки относительное местоположение элементов с равными ключами не изменяется. Такие сортировки полезны, если предполагается последующая упорядочивание по вторичным ключам среди равнозначных элементов.

Для рассмотрения алгоритмов будем считать, что необходимо сортировать массив с размером n по возрастанию.

3.1 Линейный выбор с обменом

Будем считать, что массив, который нужно отсортировать, разбивается на сортированную и неотсортированную часть. В начале алгоритма весь массив считается неотсортированным.

Алгоритм линейным выбором с обменом будет следующий:

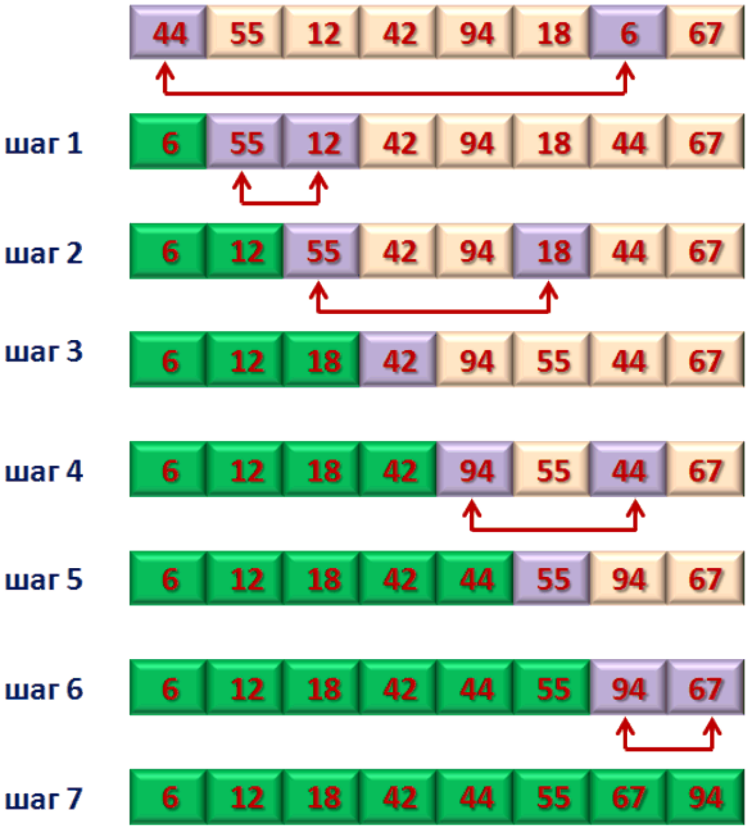
1. В неотсортированной части выбирается элемент с наименьшим ключом a_{min}.

2. Элемент a_{min} меняется местами с первым элементом a_0 в неотсортированной части.

3. Затем элемент a_{min} добавляется в отсортированную часть.

4. Шаги (1)–(3) повторяются, пока в неотсортированной части не останется один элемент, являющийся самым большим.

Иллюстрация на примере:



Асимптотика алгоритма равна $O(n^2)$. Сортировка не является устойчивой.

3.2 Линейный выбор с подсчётом

Создадим массив размера n и назовём его индексным. Индексный массив изначально заполнен нулями.

Алгоритм линейного выбора с подсчётом будет следующий:

1. Выбирается первый элемент a_i в рассматриваемой части массива и сравнивается с каждым элементом a_j, $j > i$.

2. Если элемент a_i больше a_j, то значение индексного массива с индексом i увеличивается на 1. Если a_j больше a_i, то увеличивается значение с индексом j.

3. Шаги (1)–(2) повторяются для массива от 1 до $n - 1$, затем от 2 до $n - 1$ и так далее, пока в рассматриваемой части не останется один элемент.

4. Каждый элемент a_i соответствует элементу d_i из получившегося в результате сравнений индексного массива. Элемент d_i является позицией, на которой должен стоять элемент a_i в сортированном массиве. Остаётся поставить элементы a_i на место d_i.

Временная сложность алгоритма будет равна $O(n^2)$. Метод также требует $O(n)$ памяти.

3.3 Сортировка пузырьком

Алгоритм основывается на сравнении и перестановке пар соседних элементов до тех пор, пока не будут упорядочены все элементы.

Алгоритм сортировкой пузырьком можно описать следующим образом:

1. Идём по массиву слева направо от 0 до $n - 2$.
2. Если элемент $a_i > a_{i+1}$, то меняем их местами.
3. Повторяем шаги (1)–(2), пока массив не будет отсортирован.

Метод можно немного оптимизировать.

Во-первых, можно завести ключ $changed$, который будет менять значение на $true$, если за итерацию выполнилась перестановка элементов. Таким образом, если ключ не менял своего значения, то сортировку можно считать успешно законченной.

Во-вторых, можно заметить, что после первой итерации в конце массива будет находиться самый большой элемент, после двух итераций в конце массива будут находиться два самых больших элемента и т.д. Значит можно идти после каждой итерации не до $n - 2$, а до $n - 2 - r$, где r – количество пройденных итераций.

Временная сложность алгоритма равна $O(n^2)$.

3.4 Шейкер-сортировка

Шейкерная сортировка является модифицированной сортировкой пузырьком. Заметим, что сортировка пузырьком работает медленно на тестах, в которых маленькие элементы стоят в конце. Такие элементы на каждом шаге алгоритма будут сдвигаться всего на одну позицию влево.

Теперь за каждую итерацию проход будет осуществляться в двух направлениях. Для описания алгоритма заведём две переменные l и r - левая и правая границы неотсортированной части. Алгоритм шейкер-сортировки можно описать следующим образом:

1. Идём по массиву слева направо от l до r.

2. Если элемент $a_i > a_{i+1}$, то меняем их местами.

3. Если перестановок не было, то сортировка завершена.

4. Уменьшаем значение r на один.

5. Идём по массиву слева направо от r до l.

6. Если элемент $a_{i-1} > a_i$, то меняем их местами.

7. Если перестановок не было, то сортировка завершена.

8. Увеличиваем значение l на один.

9. Повторяем шаги (1)–(8).

Иллюстрация на примере:

Хотя реальное время работы этого алгоритма лучше, асимптотика остаётся прежней – $O(n^2)$.

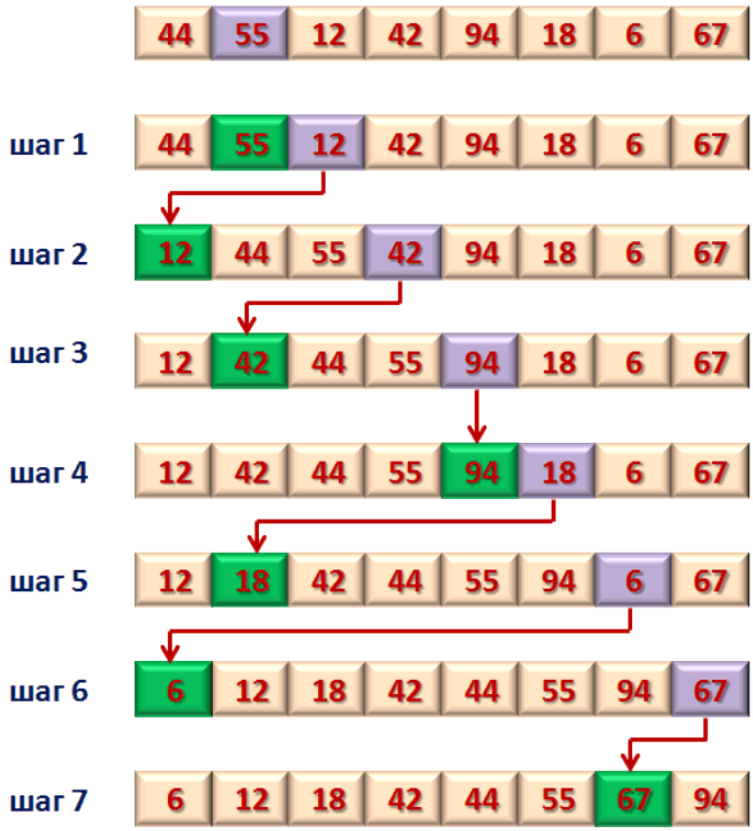
3.5 Метод простой вставки

Условно делим массив на отсортированную и неотсортированную части. Алгоритм заключается в последовательном взятии элемента из неотсортированной части и его вставкой в нужное место в сортированной части.

Алгоритм простой вставки можно описать следующим образом:

1. Выбираем первый элемент a_0 в неотсортированной части.
2. Проходим с конца до начала отсортированной части, смещая элементы v_i на позицию $i + 1$ до тех пор, пока элемент a_0 не станет больше или равен элементу v_i, либо пока i не станет меньше нуля.
3. Вставляем элемент a_0 на позицию $i + 1$.
4. Повторяем шаги (1)–(3) для неотсортированной части, пока в ней есть элементы.

Иллюстрация на примере:



Временная сложность алгоритма равна $O(n^2)$.

3.6 Метод двоичной вставки

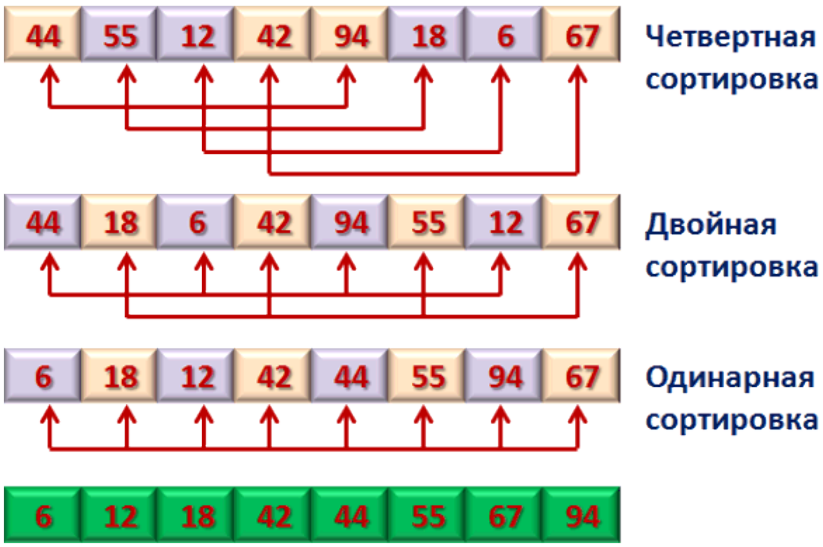
Метод двоичной вставкой является модифицированной версией метода простой вставкой. Модификация заключается в двоичном поиске позиции, на которое нужно вставить элемент a_0. Двоичный поиск можно применить, так как он применяется для отсортированной части. Однако асимптотика от этого не меняется: $O(n(\log_2 n + n)) = O(n \log_2 n + n^2) = O(n^2)$.

3.7 Метод Шелла

Простые методы сортировки продвигали элемент на одну позицию за шаг, поэтому они квадратичны и неудовлетворительны. Серьёзного улучшения в скорости сортировки можно достичь только перемещая сортируемые элементы на большие расстояние на каждой итерации.

В 1959 г. Д. Шеллом было предложено усовершенствование сортировки вставкой. Идея метода заключается в выделении в сортируемой последовательности периодических подпоследовательностей регулярного шага, в каждой из которых отдельно выполняется обычная (или двоичная) сортировка вставкой. После каждого прохода шаг подпоследовательностей уменьшается и сортировка повторяется с новыми прыжками. Последней выполняется сортировка с шагом 1.

Приведём пример с шагами 4 2 1:

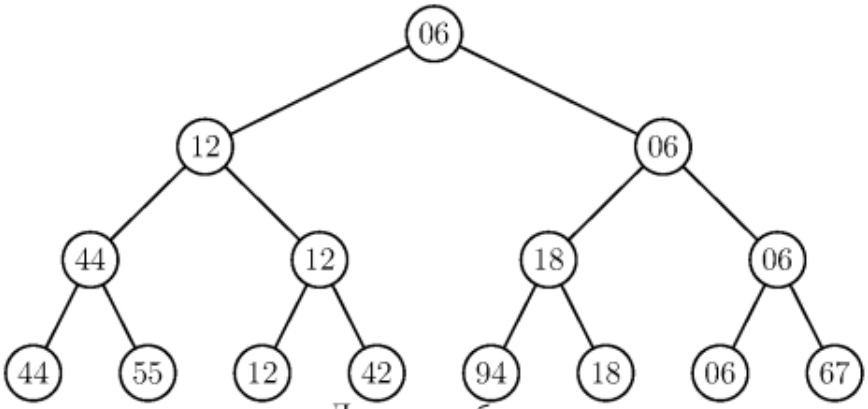


Выигрыш сортировки Шелла происходит за счёт того, что на каждом этапе либо сортируется относительно мало элементов, либо эти элементы уже довольно хорошо упорядочены и происходит сравнительно немного перестановок.

Недостатком сортировки Шелла является его неустойчивость. Математический анализ сортировки Шелла исключительно сложен и зависит от оптимальности выбранной последовательности уменьшающегося шага. Общая степенная сложностная оценка сортировок Шелла равна $O(n^{1+\delta})$, где $0 < \delta < 1$.

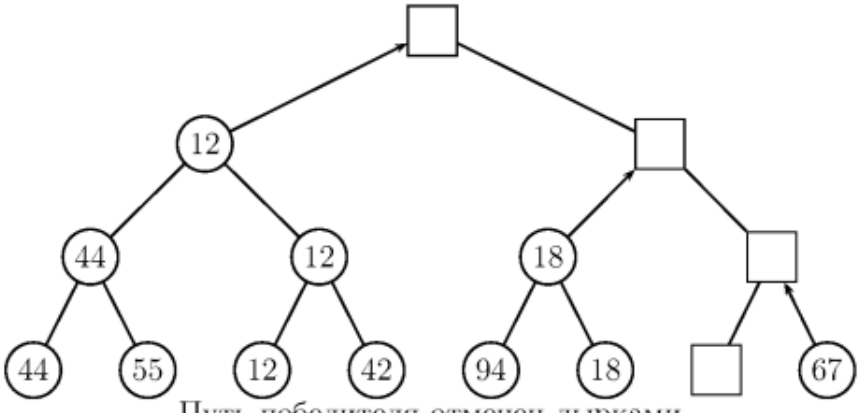
3.8 Турнирная сортировка

Существует модификация сортировки выборкой, оставляющая после каждого прохода гораздо больше информации, чем просто минимальное значение. В сортируемом множестве можно сравнить пары соседних элементов. В результате $n/2$ сравнений мы получим такое же число победителей – элементов с меньшими ключами. Если среди победителей провести такое же сравнение, то получим $n/4$ меньших элементов. Продолжая процесс, мы получим дерево выбора минимального элемента. Например:

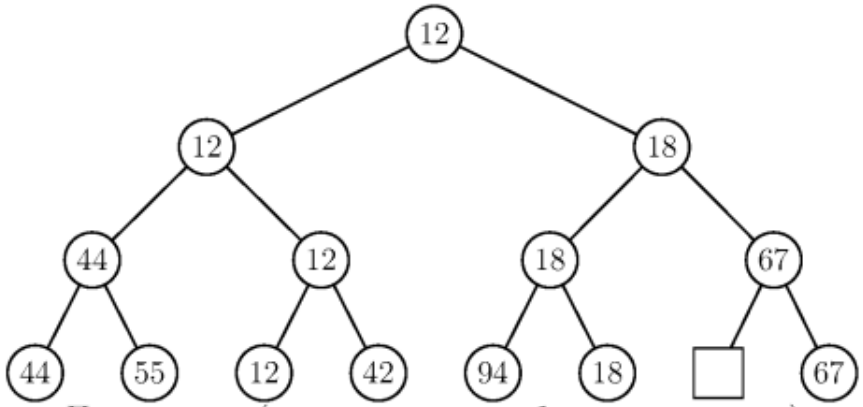


Дерево выбора

Чтобы воспользоваться этой структурой, надо от победителя отправиться к призерам, осуществив спуск вдоль пути, отмеченного наименьшим элементом. Все вершины его пути надо исключить из дерева, заменив на пустой элемент. Далее, поднимаясь назад по пустым элементам, надо выполнить переигровку турнира для определения нового победителя. При этом элемент, соревнующийся с пустым местом, автоматически проходит в следующий тур. Повторяя этот процесс до полной выемки непустых элементов из дерева, получим отсортированную последовательность.



Путь победителя отмечен дырками



Переигровка (дырка на листе победителя остается)

Временная сложность алгоритма линеарифмическая, $O(nlog_2 n)$.

3.9 Пирамидальная сортировка с просеиванием

Пирамида (куча) - это бинарное дерево, удовлетворяющее условиям:

1. корень больше, чем его дети;
2. поддеревья являются кучами;
3. дерево должно быть полным.

Идея пирамидальной сортировки заключается в следующем:

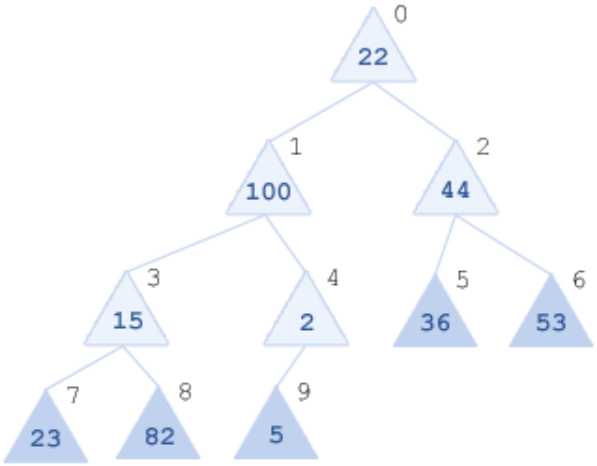
1. интерпретируем массив как кучу;
2. делаем некоторые изменения, чтобы исправить нарушения свойства кучи (сын больше отца);
3. разбираем полученную кучу, начиная с максимального элемента.

3.9.1 Построение пирамиды

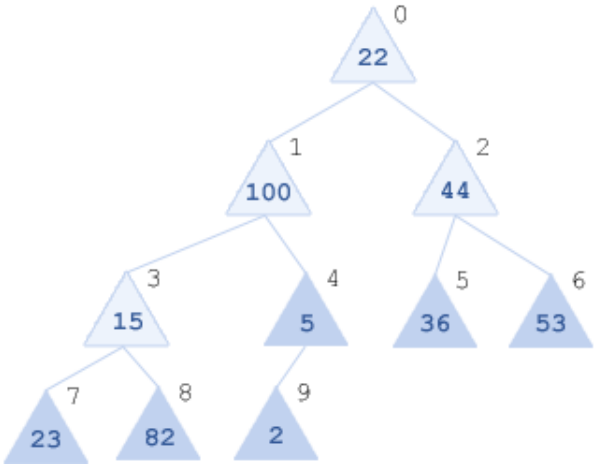
Исправлять нарушения свойств кучи будем через алгоритм просеивания:

1. Для текущего элемента a_i найти максимального из его сыновей: $maxChild = max(a_{2*i+1}, a_{2*i+2})$.
2. Если $a_i \geq maxChild$, то завершить алгоритм, т.к. элемент занимает положенное ему место, иначе – перейти к шагу (3).
3. Если $maxChild > a_i$, то поменять их местами. Перейти к шагу (1), учитывая новое положение добавляемого элемента.

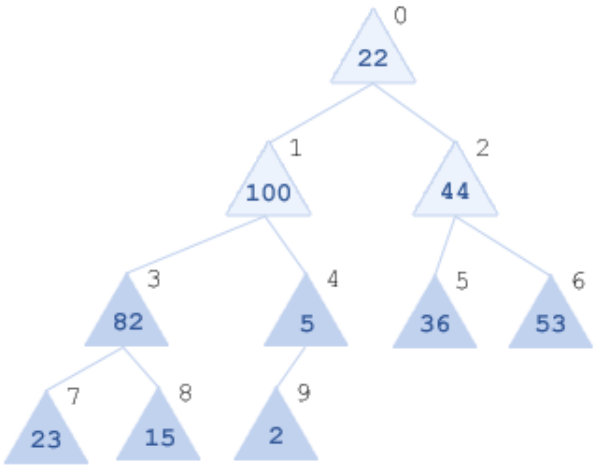
Начнём применять просеивания, начиная с середины массива $n/2$ (т.к. элементы a_j, $j > n/2$ сыновей не имеют) до его начала. Например, построим пирамиду для последовательности a = [22, 100, 44, 15, 2, 36, 53, 23, 82, 5].
Делим пополам: 22, 100, 44, 15, 2 | 36, 53, 23, 82, 5. То, что справа, уже пирамида, слева - то, что будет просеиваться через нее. Таким образом получим (темные элементы - пирамида, светлые - что предстоит просеивать):



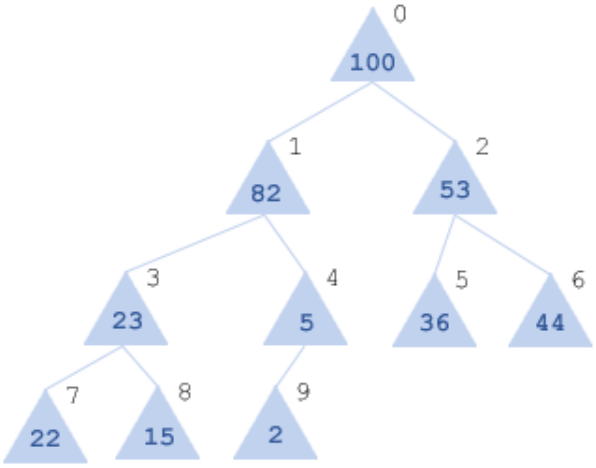
1. Шаг. Просеиваем $a[4] = 2$ через пирамиду [36, 53, 23, 82, 5].
   У этого элемента только один ребёнок: $a[9] = 5$. $5 > 2 =>$ меняем местами $a[4]$, $a[9]$. У элемента $a[9]$ нет детей $=>$ итерация завершена, новая пирамида: [5, 36, 53, 23, 82, 2]



2. Шаг. Просеиваем $a[3] = 15$ через [5, 36, 53, 23, 82, 2]. Дети: $a[7] = 23$, $a[8] = 82$. $a[8]$ - наибольший. Т.к. $a[8] = 82 > a[3] = 15$ => Обмениваем $a[3]$ и $a[8]$. У $a[8]$ детей уже нет $=>$ итерация завершена, новая пирамида: [82, 5, 36, 53, 23, 15, 2]



Действуя таким образом, получим следующую пирамиду: [100, 82, 53, 23, 5, 36, 44, 22, 15, 2].



3.9.2 Сортировка

После построения пирамиды запускается процедура сортировки.
Чтобы не заводить ещё один массив для разбора пирамиды, будем на каждом шагу вершину не изымать, а просто обменивать с конечным элементом пирамиды. Алгоритм будет следующий:

1. Устанавливаем границы пирамиды с 0 по n-1, т.е. на данном этапе вся пирамида a[0..n-1]. Вершина $h = 0$, конец $t = n - 1$.

2. Меняем a_h и a_t.

3. $t = t - 1$.

4. Просеиваем новую вершину a_0 через пирамиду a[1..t].

5. Если $t = 1$, то конец сортировки, иначе - шаг (2).

Например, применяем алгоритм для полученной ранее пирамиды a = [100, 82, 53, 23, 5, 36, 44, 22, 15, 2]:

1. Обмен a[0]=100, a[9]=2; просеивание a[0]=2 через [82 53 23 5 36 44 22 15] Получим: [82 23 53 22 5 36 44 2 15 100].

2. Обмен a[0]=82 и a[8]=15, просеивание a[0]=15 через [23 53 22 5 36 44 2] Получим: [53 23 44 22 5 36 15 2 82 100].

3. Обмен a[0]=53 и a[7]=2, просеивание a[0]=53 через [23 44 22 5 36 15] Получим: [44 23 36 22 5 2 15 53 82 100].

4. Обмен a[0]=44 и a[6]=15, просеивание a[0]=44 через [23 36 22 5 2] Получим: [36 23 15 22 5 2 44 53 82 100].

5. Обмен a[0]=36 и a[5]=2, просеивание a[0]=2 через [23 15 22 5] Получим: [23 22 15 2 5 36 44 53 82 100].

6. Обмен a[0]=23 и a[4]=5, просеивание a[0]=5 через [23 15 2] Получим: [22 5 15 2 23 36 44 53 82 100].

7. Обмен a[0]=22 и a[3]=2, просеивание a[0]=2 через [5 15] Получим: [15 5 2 22 23 36 44 53 82 100].

8. Обмен a[0]=15 и a[2]=2, просеивание a[0]=2 через [5]. В данном случае все просеивание сведется к обмену a[0]=2 и a[1]=5. Получим: [5 2 15 22 23 36 44 53 82 100]

9. Обмен a[0]=2 и a[1]=5. Просеивание делать уже не имеет смысла.

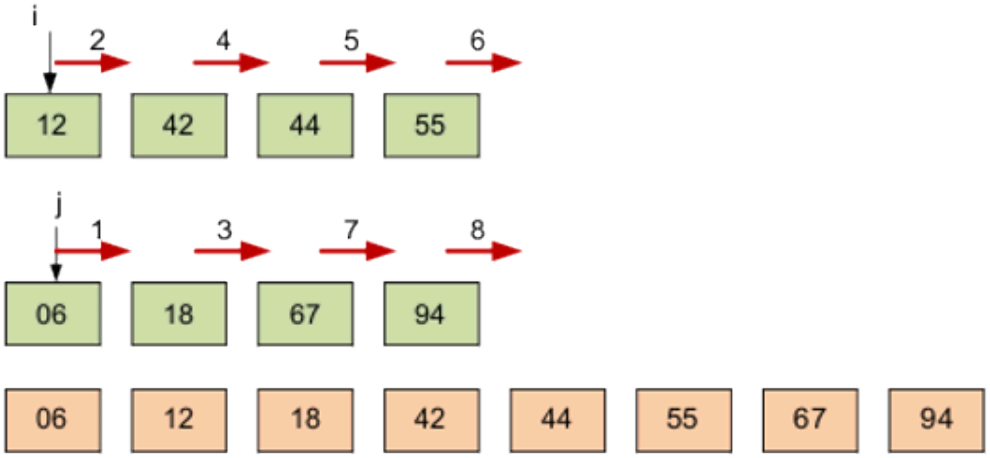
Получили сортированный массив a = [2 5 15 22 23 36 44 53 82 100].
Сложность сортировки в целом составляет $O(n \log n)$. Сортировка не является устойчивой.

3.10 Простое двухпоточное слияние

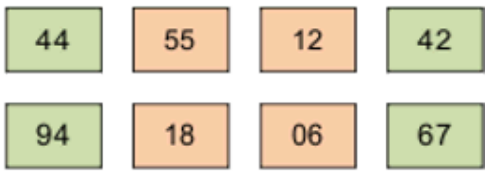
Основной применяемый метод для сортировки файлов — сортировка слиянием.
Сортировка слиянием — алгоритм сортировки, который упорядочивает списки (или другие структуры данных, доступ к элементам которых можно получать только последовательно, например — потоки) в определённом порядке.

Слияние означает объединение двух (или более) последовательностей в одну упорядоченную последовательность при помощи циклического выбора элементов, доступных в данный момент.

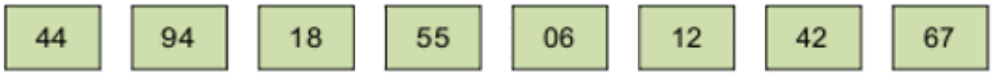
Начальные элементы предварительно упорядоченных последовательностей сравниваются между собой, и из них выбирается наименьший. Соответствующий указатель перемещается на следующий элемент. Процедура повторяется до тех пор, пока не достигнут конец одной из подпоследовательностей. Оставшиеся элементы другой подпоследовательности при этом передаются в результирующую последовательность в неизменном виде.



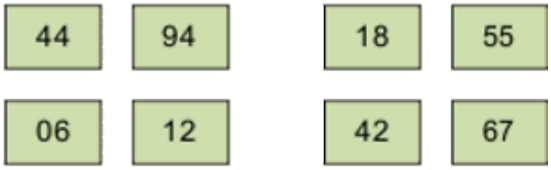
Пример. Дана последовательность [44, 55, 12, 42, 94, 18, 06, 67]. Первым шагом она разбивается на две подпоследовательности:



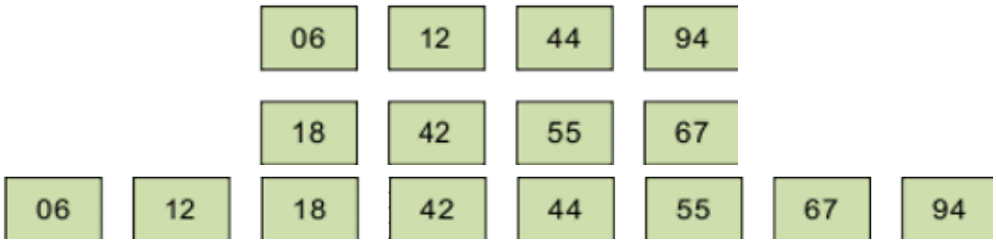
Эти две подпоследовательности объединяются в одну, содержащую упорядоченные пары.



Полученная последовательность снова разбивается на две, и пары объединяются в упорядоченные четверки:



Полученная последовательность снова разбивается на две и собирается в упорядоченные восьмерки.



Данная операция повторяется до тех пор, пока полученная упорядоченная последовательность не будет иметь такой же размер, как у сортируемой. Основной операцией является слияние. При слиянии требуется дополнительная память для размещения файла, образующегося при слиянии.

Временная сложность алгоритма равна $O(n \log n)$, метод требует $O(n)$ памяти.

3.11 Быстрая сортировка Хоара

Быстрая сортировка представляет собой усовершенствованный метод сортировки, основанный на принципе обмена. Опишем этот алгоритм:

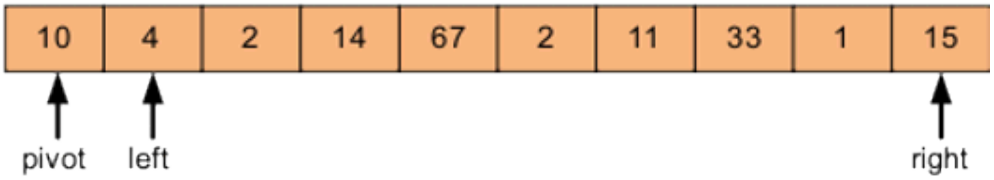
1. в рассматриваемом массиве выбирается случайный элемент, называемый опорным;

2. массив разбивается на две части: слева – элементы, меньшие опорного, справа - элементы, большие опорного;

3. чтобы отсортировать эти два меньших подмассива, алгоритм вызывается рекурсивно для каждой из этих частей.

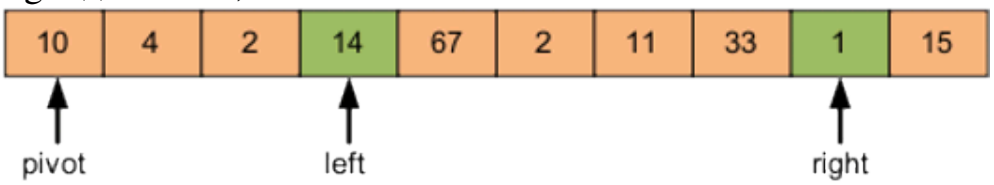
Рассмотрим сортировку на примере массива: 10, 4, 2, 14, 67, 2, 11, 33, 1, 15.
Для реализации алгоритма переупорядочения используем указатель left на крайний левый элемент массива. Указатель движется вправо, пока элементы, на которые он показывает, остаются меньше опорного. Указатель right поставим на крайний правый элемент массива, и он движется влево, пока элементы, на которые он показывает, остаются больше опорного.

Пусть крайний левый элемент — опорный pivot. Установим указатель left на следующий за ним элемент; right — на последний. Алгоритм должен определить правильное положение элемента 10 и по ходу дела поменять местами неправильно расположенные элементы.

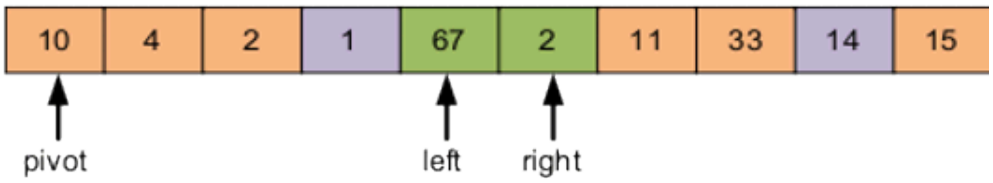


Движение указателей останавливается, как только встречаются элементы, порядок расположения которых относительно опорного элемента неправильный.

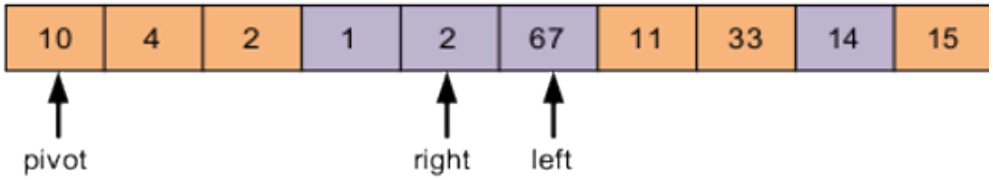
Указатель left перемещается до тех пор, пока не покажет элемент больше 10; right движется, пока не покажет элемент меньше 10.



Эти элементы меняются местами и движение указателей возобновляется.

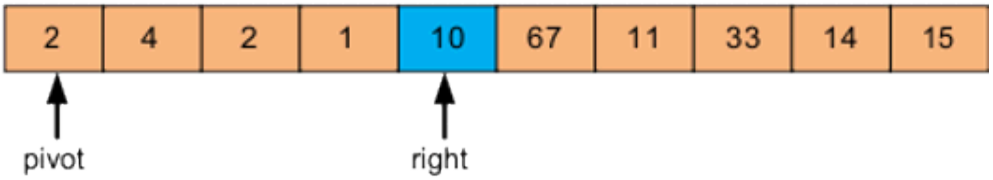


Процесс продолжается до тех пор, пока right не окажется слева от left.



Тем самым будет определено правильное место опорного элемента.

Осуществляется перестановка опорного элемента с элементом, на который указывает right.



Опорный элемент находится в нужном месте: элементы слева от него имеют меньшие значения; справа — большие. Алгоритм рекурсивно вызывается для сортировки подмассивов слева от опорного и справа от него.

Временная сложность алгоритма равна $O(n \log n)$. Сортировка не является

устойчивой.

4 Практика

Требуется составить программу на языке Си с использованием процедур и функций для сортировки таблицы методом быстрой сортировкой Хоара, реализованной нерекурсивно, и двоичного поиска по ключу в таблице. Структура таблицы следующая: тип ключа – комплексный 32-х байтный, данные являются строками; хранение данных и ключей организовать вместе.
Программа должна:

1. вводить значения элементов неупорядоченной таблицы;
2. печатать таблицу;
3. сортировать таблицу по ключу;
4. печатать результат сортировки;
5. после выполнения сортировки программа должна вводить ключи и для каждого выполнять поиск в упорядоченной таблице с помощью процедуры двоичного поиска и печатать найденные элементы, если они присутствуют в таблице.

4.1 Описание функций

Опишем структуру элемента, который будет хранится в таблице.

```
#define MAX_LEN_VALUE 1024
#include <quadmath.h>

typedef struct {
    __complex128 key;
    char val[MAX_LEN_VALUE];
} item;
```

Библиотека $quadmath.h$ содержит типы $\_\_float128$ и $\_\_complex128$, а также функции для работы с элементами этих типов.
Функция сравнения ключей будет проводиться по действительной части комплексных чисел. Если действительная часть равна, то функция сравнивает числа по мнимой части. Функция возвращает число больше нуля, если первый элемент больше второго; число меньше нуля, если первый элемент меньше второго; ноль, если числа равны.

```
int key_cmp(__complex128 key1, __complex128 key2) {
    return (crealq(key1) != crealq(key2))?
           (int) (crealq(key1) - crealq(key2)) :
           (int) (cimagq(key1) - cimagq(key2));
}
```

Функция быстрой сортировки Хоара реализована по нерекурсивному варианту:

```
void sort_hoar(vector_item *v) {
    pair stack[sizeof(size_t)*8];
    stack[0].l = 0;
    stack[0].r = vitem_get_size(v)-1;
    int stack_count = 1;
    while (stack_count > 0) {
        pair p = stack[stack_count-1];
        stack_count--;
        int l = p.l;
        int r = p.r;
        while (l < r) {
            item pivot = peek_pivot(v, l, r);
            int d = partition(v, l, r, pivot);
            if (r-l-d > d) {
                stack[stack_count].l = d;
                stack[stack_count].r = r;
                r = d-1;
            } else {
                stack[stack_count].l = l;
                stack[stack_count].r = d-1;
                l = d;
            }
            stack_count++;
        }
    }
}
```

Чтобы промоделировать рекурсию, используется стек пар чисел (границ сортировки). Стек ограничиваем таким образом, что он теперь не будет больше $\log n$. А чтоб он мог вместить логарифм от максимального объёма адресуемой памяти, сделаем его статическим на $sizeof(size\_t) * 8$ элемента - такой стек сможет обработать до 2^{64} элементов в 64-битной системе.
Функция $peek\_pivot$ берёт опорный элемент. В данном случае берём элемент, стоящий почти по середине:

```
static item peek_pivot(vector_item *v, int l, int r) {
    return vitem_get(v, (l+r)/2);
}
```

Функция *partition* делит массив от l до r на две части: элементы, меньшие *pivot*, будут находится слева; элементы, большие *pivot*, будут находиться справа. Функция возвращает индекс границы между частями.

```
static int partition(vector_item *v, int l, int r, item pivot) {
    int i = l, j = r;
    while (i <= j) {
        while (key_cmp(vitem_get(v, i).key, pivot.key) < 0)
            i++;
        while (key_cmp(vitem_get(v, j).key, pivot.key) > 0)
            j--;
        if (i <= j) {
            vitem_swap(v, i, j);
            i++;
            j--;
        }
    }
    return i;
}
```

Функция бинарного поиска будет возвращать индекс первого вхождения элемента с заданным ключом в массив. Если элемент не найден, то функция возвращает -1.

```
int vitem_bin_search_key(vector_item *v, __complex128 key) {
    int l = 0, r = vitem_get_size(v);
    while (l < r) {
        int m = (l + r) / 2;
        __complex128 k = vitem_get(v, m).key;
        int res_cmp = key_cmp(key, k);
        if (res_cmp == 0) {
            if (m == 0 || key_cmp(vitem_get(v, m - 1).key, key))
                return m;
            r = m;
        }
        else if (res_cmp > 0)
            l = m + 1;
        else
            r = m;
```

```
    }
    if (r != vitem_get_size(v) && !key_cmp(vitem_get(v, r).key, key))
        return r;
    return -1;
}
```

4.2 Тесты

```
Input items (key format: A+Bi) (to finish entering items, type "end"):
3+4i Hellow World!
1+2i Hello world
4 + 3i Hell world
1 + 2i world
3 + 2i Wello horld
0+100.909i Hello C!
4 + 3i H! H!
1+2i !!!
-1 + 9i CCC
end
Items:
3.000000+4.000000i Hellow World!
1.000000+2.000000i Hello world
4.000000+3.000000i Hell world
1.000000+2.000000i world
3.000000+2.000000i Wello horld
0.000000+100.909000i Hello C!
4.000000+3.000000i H! H!
1.000000+2.000000i !!!
-1.000000+9.000000i CCC
Sorted items:
-1.000000+9.000000i CCC
0.000000+100.909000i Hello C!
1.000000+2.000000i world
1.000000+2.000000i Hello world
1.000000+2.000000i !!!
3.000000+2.000000i Wello horld
3.000000+4.000000i Hellow World!
4.000000+3.000000i Hell world
4.000000+3.000000i H! H!
Input keys (key format: A+Bi) (to finish entering keys, type "end"):
3+4i
```

```
3.000000+4.000000i Hellow World!
4+3i
4.000000+3.000000i Hell world
4.000000+3.000000i H! H!
1+ 2i
1.000000+2.000000i world
1.000000+2.000000i Hello world
1.000000+2.000000i !!!
777+777i
the key is not in the table
0+100.909i
0.000000+100.909000i Hello C!
end
```

5 Выводы

В ходе работы были рассмотрены некоторые алгоритмы поиска и сортировок. Простыми сортировками с квадратичной сложностью являются: линейный выбор с обменом и подсчётом, сортировка пузырьком, шейкерная сортировка, методы простой и двоичной вставки. Анализ сортировки Шелла сложен, но в среднем он равен $O(n^{1+\delta})$. Самые быстрые из рассмотренных сортировок являются пирамидальная, быстрая и сортировка слиянием. Сортировка слиянием в отличии от пирамидальной и быстрой, является устойчивой и может применяться для структур данных, доступ к элементам которых можно получать только последовательно (например, списки), но она требует $O(n)$ дополнительной памяти.

Реальное время работы алгоритмов зависит не только от количества элементов, но и от других факторов, например, аппаратных особенностей или местонахождения ключей.

6 Список источников

1. Алгоритмы сортировки и поиска - https://prog-cpp.ru/algorithm-sort/
2. Описание алгоритмов сортировки и сравнение их производительности - https://habr.com/ru/post/335920/
3. Турнирная сортировка - https://habr.com/ru/company/edison/blog/508646/
4. Пирамидальная сортировка - https://codelab.ru/task/pyramid_sort/
5. GCC libquadmath - https://gcc.gnu.org/onlinedocs/libquadmath/