МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
МИРЭА – Российский технологический университет

**Е.Н. КАШИРСКАЯ, М.А. МАКАРОВ,
С.Е. ХАРЬКОВСКИЙ**

Программирование алгоритмов сортировки

Учебное пособие

Москва – 2021

УДК 004.62
ББК 16.2

Каширская Е.Н. «Программирование алгоритмов сортировки» [Электронный ресурс]: учебное пособие / Каширская Е.Н., Макаров М.А., Харьковский С.Е. – М.: МИРЭА – Российский технологический университет, 2021. – 10 электрон. опт. диск (CD-ROM)

Разработано в помощь студентам 1 курса, изучающих дисциплины «Информатика», «Процедурное программирование», «Языки программирования», «Линейная алгебра». В пособии рассмотрены основные методы сортировок, различные структуры данных и особенности обработки информации. Также были затронуты дополнительные темы, косвенно связанные с сортировкой, темы алгоритмов, такие как «жадный алгоритм», «алгоритм поиска с возвратом» и т.п. Для большей части теоретического материала в пособии приведен рабочий код, с помощью которого студент может закрепить полученные знания на практике.

Предназначено для освоения учебной программы и отработки практического применения полученных знаний.

Методическое пособие издается в авторской редакции.

Авторский коллектив: Каширская Елизавета Натановна, Макаров Максим Алексеевич, Харьковский Станислав Евгеньевич

Рецензенты:

Долгов Виталий Анатольевич, д.т.н., заместитель директора ООО «ФЦС»

Реганов В.М., к.т.н., заместитель директора АО «НИИВК им. М.А. Карцева»

Минимальные системные требования:
Наличие операционной системы Windows, поддерживаемой производителем.
Наличие свободного места в оперативной памяти не менее 128 Мб.
Наличие свободного места в памяти хранения (на жестком диске) не менее 30 Мб.
Наличие интерфейса ввода информации.
Дополнительные программные средства: программа для чтения pdf-файлов (Adobe Reader).
Подписано к использованию по решению Редакционно-издательского совета
МИРЭА – Российского технологического университета от ___ _____ 2021 г.
Объем ___ Мб
Тираж 10

Содержание

Введение 6
Определение алгоритма 7
Язык программирования *C* 8
Типы данных 12
Структуры данных 14
Массивы 14
Связный список 16
Стеки 21
Очереди 24
Деревья 26
Основные типы деревьев в структуре данных 28
Дерево общего вида 28
Двоичное (бинарное) дерево 28
Дерево двоичного поиска 29
Дерево АВЛ 29
Красно-черное дерево 30
Рекурсия 32
Парадигма «Разделяй и властвуй» 35
Быстрая сортировка 39
Сортировка слиянием 41
Сортировка выбором 45
Сортировка вставками 46
Пузырьковая сортировка 48
Сортировка расческой (Comb Sort) 49
Сортировка Шелла 49
Сортировка подсчетом 52
Двухсторонняя сортировка выбором 54
Бинго сортировка 56
Блинная сортировка 57
Пирамидальная сортировка 60

Плавная сортировка 65
Поразрядная (радиксная) сортировка 67
Блочная сортировка (карманная сортировка) 71
Жадные алгоритмы 76
Поиск с возвратом, бэктрекинг 80
Алгоритм решения задач с множеством решений 89
Основы динамического программирования 92
«Уродливые числа» 96
Заключение 101
СПИСОК ЛИТЕРАТУРЫ 102
Сведения об авторах 103

Введение

Данное методическое пособие было разработано с целью познакомить студентов с широким спектром различных методов сортировки. Из–за большого количества рассматриваемых областей и алгоритмов пособие будет сконцентрировано на «фундаментальных» алгоритмах. Каждый алгоритм рассматривается подробно, с объяснением его характеристик и тонкостей, благодаря чему эти алгоритмы могут быть реализованы студентами в программном коде.

Чтобы хорошо изучить алгоритм, его нужно реализовать в программной среде и запустить на выполнение. Соответственно рекомендуемая стратегия для понимания программ, представленных в настоящем пособии, состоит в том, чтобы реализовать и протестировать их, поэкспериментировать с вариантами исходных данных и опробовать на реальных задачах. Для описания, обсуждения и реализации большинства алгоритмов в программном коде используется язык ***C***. Однако программы могут быть также реализованы и на других языках программирования.

Также в данной работе будут затронуты дополнительные темы, косвенно связанные с сортировкой, тема алгоритмов, таких как «жадный алгоритм», «алгоритм поиска с возвратом» и т.п.

Данное пособие будет наиболее полезно для студентов 1 курса, изучающих дисциплины «Информатика», «Процедурное программирование», «Языки программирования», «Линейная алгебра».

Определение алгоритма

Термин «алгоритм» используется в информатике для описания метода решения проблем, подходящего для реализации в виде компьютерных программ. Алгоритмы – это «материал информатики»: они являются центральными объектами изучения во многих областях науки.

Большинство представляющих интерес алгоритмов включают сложные методы организации данных, используемых в математических вычислениях. Созданные таким образом объекты называются структурами данных, и они также являются центральными объектами изучения информатики. Таким образом, алгоритмы и структуры данных тесно связаны.

При разработке крупного программного решения необходимо понимать решаемые проблемы и декомпозировать их на более мелкие подзадачи, которые легко реализовать. Однако в большинстве случаев есть несколько алгоритмов, выбор которых имеет решающее значение, поскольку большая часть системных ресурсов будет потрачена на выполнение этих алгоритмов. Но также важно помнить, что наиболее эффективное применение алгоритмам можно найти в работе с крупными программами, где правильно подобранный алгоритм сэкономит минуты и даже часы эффективной работы.

Раздел информатики, изучающий подобные вопросы, называется анализом алгоритмов. Многие из алгоритмов, представленные в данном пособии, показали в ходе анализа очень хорошую производительность, в то время как другие просто хорошо работают, исходя из опыта. Но не следует использовать алгоритм, не имея представления о том, какие ресурсы будут задействованы для его реализации.

Язык программирования *C*

C – это язык программирования среднего уровня, разработанный Деннисом Ричи в начале 1970-х годов, когда он работал в AT&T Bell Labs в США. Язык ***C*** был разработан на основе языка ***B***. Целью его разработки было изменение дизайна операционной системы UNIX, позволяющее использовать ее на нескольких компьютерах.

Язык ***B*** ранее использовался для улучшения системы UNIX. Будучи языком высокого уровня, ***B*** позволял создавать код гораздо быстрее, чем на языке ассемблера. Тем не менее, ***B*** страдал недостатками, поскольку он не различал типы данных и не обеспечивал использование составных типов - структур.

Для преодоления этих и других недостатков Ритчи разработал новый язык программирования под названием ***C***. Он сохранил большую часть синтаксиса языка ***B*** и добавил типы данных и многие другие изменения. В результате в 1971-73 гг. появился язык ***C***. В его состав автор включил как высокоуровневые общие функции, так и частные функции, необходимые для разработки операционной системы, благодаря чему многие компоненты UNIX, включая само ядро UNIX, в конечном итоге были переписаны на ***C***.

Примеры в этом пособии написаны на языке программирования ***C***. Все языки имеют свои достоинства и недостатки, хотя многие современные языки программирования схожи, поэтому, используя относительно немного языковых конструкций и избегая решений о реализации, основанных на особенностях ***C***, авторами пособия продемонстрированы программы, которые легко переводятся на другие языки. Основная задача – представить алгоритмы в максимально простой и понятной форме.

Алгоритмы часто описываются в абстрактных формах, при этом теряются важные практические детали, что не позволяет обучающемуся воспользоваться реализацией полученных знаний. Лучший способ понять алгоритм и проверить его полезность – это реализовать его практически. Современные языки достаточно выразительны, поэтому реальные программы, написанные на этих языках, могут быть столь же краткими и элегантными, как и воображаемые. Читателю предлагается познакомиться с локальной средой программирования ***C***, потому что приведенные в пособии примеры представляют собой рабочие программы, которые предназначены для запуска и анализа результатов именно в этой среде.

Для начала рассмотрим программу на языке ***C*** для решения классической элементарной задачи: «Сократить заданную дробь до ее наименьших несократимого вида». Решение этой задачи эквивалентно нахождению наибольшего общего делителя (НОД) числителя и знаменателя - наибольшего целого числа, на которое они оба делятся. Дробь сокращается до несократимого вида путем деления числителя и знаменателя на их наибольший общий делитель. Эффективный метод нахождения наибольшего общего делителя был открыт древними греками более двух тысяч лет назад. Он подробно описан в знаменитом трактате Евклида «Элементы» и получил название алгоритм Евклида.

Метод Евклида основан на том факте, что если ***u*** больше, чем ***v***, то наибольший общий делитель ***u*** и ***v*** совпадает с наибольшим общим делителем ***v*** и ***u–v***. Это наблюдение приводит к следующей реализации в ***C***:

```
#include <stdio.h>
int gcd(int u, int v)
{
	int t;
	while (u > 0)
	{
		if (u < v)
		{
			t = u;
			u = v;
			v = t;
		}
		u = u – v;
	}
	return v;
}

main()
{
	int x, y;
	while (scanf("%d %d", &x, &y) != EOF)
		if (x > 0 && y > 0)
			printf("%d %d %d\n", x, y, gcd(x, y));
	}
```

Давайте рассмотрим свойства языка, представленные этим кодом. Язык ***C*** имеет строгий синтаксис высокого уровня, который позволяет легко идентифицировать основные функции программы. Программа состоит из

списка функций, главная из которых называется **main**. Она содержит тело программы. Любая функция возвращает полученное в ней значение с помощью оператора **return**.

Встроенная функция **scanf** считывает строку из потока ввода и присваивает найденные значения переменным, указанным в качестве аргументов. В круглых скобках «слово» в кавычках – это «формат», указывающий, сколько и каких знаков должно быть прочитано. Функция scanf ссылается на свои аргументы «косвенно»; отсюда и символы **&** - символы ссылки. Встроенный предикат в стандартной библиотеке ввода–вывода**, EOF** (end of file), меняется на **true**, когда ввода больше нет. Директива компилятору **include** разрешает подключить определенную библиотеку.

На протяжении всего пособия используется стандарт ANSI языка ***C***, наиболее важным отличием которого от более ранних версий языка ***C*** является способ объявления функций и их аргументов.

Основная часть приведенной выше программы тривиальна: она считывает пары чисел со входа, а затем, если они оба положительны, записывает их на выходе в переменную **gcd** - наибольший общий делитель. Подумайте, что бы произошло, если бы **gcd** был вызван с отрицательным или нулевым значением ***u*** или ***v***? Функция **gcd** реализует сам алгоритм Евклида: программа представляет собой цикл, который сначала гарантирует, что ***u*** **> =** ***v***, заменяя их, если необходимо, а затем заменяет u пользователя ***u–v***. Наибольший общий делитель переменных ***u*** и ***v*** всегда совпадает с наибольшим общим делителем исходных значений, представленных процедуре: в конечном итоге процесс завершается с ***u***, равным **0**, и ***v***, равным наибольшему общему делителю исходного (и все промежуточные) значения ***u*** и ***v***.

Приведенный выше пример написан как полная программа на ***C***, которую студент должен проанализировать, чтобы познакомиться с системой программирования ***C***. Алгоритм Евклида в этой программе реализован в виде подпрограммы (**gcd**), а основная программа является «драйвером», который выполняет эту подпрограмму.

Преимущества языка С следующие.

- Как язык среднего уровня, ***C*** сочетает в себе функции языков высокого и низкого уровня. Его можно использовать для программирования низкого уровня, такого как создание сценариев для драйверов и ядер, а также

он поддерживает функции языков программирования высокого уровня, такие как создание сценариев для программных приложений и т.п.

- ***C*** – это структурированный язык программирования, который позволяет разбивать сложную программу на более простые программы, называемые функциями. Это также позволяет свободно перемещать данные между этими функциями.
- Различные функции ***C***, включая прямой доступ к аппаратным API-интерфейсам машинного уровня, наличие компиляторов ***C***, детерминированное использование ресурсов и динамическое распределение памяти, делают язык ***C*** оптимально пригодным для приложений-сценариев и драйверов встроенных систем.
- *Я*зык ***C*** чувствителен к регистру, что означает, что строчные и прописные буквы обрабатываются по-разному.
- ***C*** используется для написания сценариев системных приложений, которые составляют основную часть операционных систем Windows, UNIX и Linux.
- ***C*** – это язык программирования общего назначения, который может эффективно работать с корпоративными приложениями, играми, графикой, приложениями, требующими вычислений и т.д.
- *Я*зык ***C*** имеет богатую библиотеку, которая предоставляет ряд встроенных функций. Он также предлагает динамическое распределение памяти.
- ***C*** эффективно по времени реализует алгоритмы и структуры данных, облегчая более быстрые вычисления в программах. Это позволило использовать ***C*** в приложениях, работающих в режиме реального времени и требующих более высоких скоростей вычислений, таких как MATLAB и Mathematica.
- Благодаря этим преимуществам, ***C*** стал доминирующим и быстро распространился за пределы Bell Labs, заменив многие известные языки того времени, такие как ALGOL, B, PL / I, FORTRAN и т.д. *Я*зык ***C*** стал доступен на многих платформах - от встроенных микроконтроллеров до суперкомпьютеров.

*Я*зык ***C*** лег в основу многих языков программирования, включая *C* ++, C–, C #, Objective-C, BitC, C-shell, csh, D, Java, JavaScript, Go, Rust, Julia, Limbo, LPC, PHP, Python, Есть Perl, Seed7, Vala, Verilog и многих других языков.

Типы данных

Большинство алгоритмов в этом пособии работают с простыми типами данных: целыми числами (**int**), действительными числами (**real**), символами (**char**) или строками символов (**string**). Одной из наиболее важных особенностей ***C*** является возможность построения более сложных типов данных из этих элементарных строительных блоков.

Тип данных – это атрибут, связанный с частью данных, который сообщает компьютерной системе, как интерпретировать его значение. Понимание типов данных гарантирует, что данные собираются в предпочтительном формате, а значение каждого свойства соответствует ожидаемому. Кратко рассмотрим основные типы данных.

Integer (**int**) – это наиболее распространенный числовой тип данных, используемый для хранения чисел без дробной части (–654, 0, 404 и т.д.), которые в принципе не могут иметь дробную часть (группа крови, номер квартиры).

Floating point (**float**) – это также числовой тип данных с плавающей точкой, используемый для хранения чисел, которые могут иметь дробный компонент, например, денежные значения (–654.77 или 0.15 или 404.00 и т.д.). Обратите внимание, что любое число – это, как правило, либо **int**, либо, **float**.

Character (**char**) используется для хранения одной буквы, цифры, знака препинания, спецсимвола или пробела (“p”, “3”, “_” и т.д.) – то есть для хранения одного символа.

String (**str**) – строка, представляет собой последовательность символов и является наиболее часто используемым типом данных для хранения текста. Кроме того, строка также может включать цифры и символы, однако всегда обрабатывается как текст (“На талоне было написан номер 792”). Телефонный номер обычно хранится в виде строки (+ 8–(777)–777–99–99), но также может храниться в виде целого числа (7777779999).

Boolean (**bool**) – это логический тип, который может принимать два значение: истина (**true**) и ложь (**false**). При работе с логическим типом данных полезно иметь в виду, что иногда ложное логическое значение представляется как **0**, а истинное – как **1**.

Enumerated type (**enum**) содержит небольшой набор предопределенных уникальных значений (также известных как элементы или перечислители),

которые можно сравнивать между собой и назначать переменной перечислимого типа данных. Значения перечислимого типа могут быть текстовыми или числовыми. Например, логический тип данных представляет собой заранее определенное перечисление значений **true** и **false**. А если перечислителями являются **яблок**» и **груши**, переменной **фрукт** перечислимого типа может быть присвоено одно из двух значений, но не оба сразу. Если вас просят указать свои предпочтения в выборе фруктов и выбрать один из двух фруктов в раскрывающемся меню, то в переменном **фрукт** будет сохранены **яблоки** или **груши**. С помощью перечислимого типа значения можно сохранять и извлекать как числовые индексы (0, 1, 2) или строки.

Array (массив) – это тип данных, который хранит ряд элементов одного типа в определенном порядке. Каждый элемент массива снабжен целочисленным индексом (0, 1, 2, …), а общее количество элементов в массиве является длиной массива. Например, в переменной типа массив **фрукты** может храниться один или несколько элементов – **яблоки**, **груши** и **персики**. Индексы имеют три значения: **0 - яблоки**, **1 - груши** и **2 - персики**, а длина массива равна **3**, поскольку он содержит три элемента.

Если вас попросят выбрать один или несколько из трех фруктов, а вам понравятся все три, переменная **фрукты** сохранит все три элемента (яблоки, груши, персики).

Date обычно хранит дату в формате ГГГГ–ММ–ДД (синтаксис ISO 8601). Пример: 2021–02–21.

Time сохраняет время в формате чч: мм: сс. Помимо текущего времени, его также можно использовать для хранения прошедшего времени или временного интервала между двумя событиями, который может составлять более 24 часов. Например, время, прошедшее с момента события, может составлять 72 часа, 21 минуту и 49 секунд (74:21:49).

Различные языки программирования предлагают и другие типы данных для различных целей, однако мы рассмотрели наиболее часто используемые типы данных, которые нужно знать программисту на ***C***.

Структуры данных

При выборе алгоритма сортировки в компьютерной программе необходимо определиться с организацией данных. В большинстве случаев выбор правильной структуры данных приводит к использованию простых и понятных алгоритмов. Для одинаковых данных, использующих разные структуры, может понадобиться различное количество места, а для операций – разные по эффективности алгоритмы. Эта тема крайне важна, поскольку выбор и алгоритма, и структуры данных тесно связаны, и рациональным шагом будет поиск способа сэкономить время или пространство, сделав правильный выбор.

В данной части пособия будут рассмотрены массивы, связные списки, стеки, очереди, деревья, составляющие основу практически всех алгоритмов, рассматриваемых далее.

Массивы

Наиболее фундаментальной структурой данных является массив, который определяется как примитив в ***C*** и в большинстве других языков программирования. Массив – это фиксированное количество элементов данных одного типа, которые хранятся непрерывно и доступны по индексу.

1–й элемент массива обозначается как **a[i]**. Компилятор должен знать, что хранится позиции массива **a[i]** перед обращением к нему; игнорирование этого – одна из самых распространенных ошибок программирования.

Простой пример использования массива дается следующей программой, которая распечатывает все простые числа, меньшие 100. Используемый метод, восходящий к III веку до нашей эры, называется «Решетом» Эратосфена:

```
#include <stdio.h>
#define N 100
int main()
{
        int i, j, a[N + 1];
        for (a[1] = 0, i = 2; i <= N; i++)
                a[i] = 1;
        for (i = 2; i <= N / 2; i++)
                for (j = 2; j <= N / i; j++) a[i * j] = 0;
        for (i = 1; i <= N; i++)
                if (a[i])
                        printf("%4d", i);
        printf("\n");
        }
```

Эта программа использует массив, состоящий из элементов самого простого типа: логических (0–1) значений. Цель программы – установить для **[i]** значение **1**, если **i** - простое, и **0**, если оно составное.

Программа делает это, устанавливая **0** для каждого **i**-ого элемент массива, соответствующего каждому кратному **i**, поскольку любое число, кратное любому другому числу, является составным.

Затем программа проходит через массив еще раз, распечатывая простые числа. Эту программу можно сделать несколько более эффективной, добавив проверку **if (a [i])** перед циклом **for**, включающую счетчик **j**, поскольку, если **i** не является простым, элементы массива, соответствующие всем кратным ему числам, уже должны быть отмечены. Обратите внимание, что массив сначала инициализируется: алгоритм устанавливает **0** элементов, соответствующих индексам членов натурального ряда, которые заведомо являются составными числами.

«Решето» Эратосфена - типичный пример алгоритма, использующего доступ к элементу массива по его индексу. Алгоритм просматривает элементы массива последовательно, один за другим, и в этом заключается основная особенность массивов: если известен индекс, к любому элементу можно получить доступ в любое время.

Размер массива должен быть известен заранее: чтобы запустить указанную выше программу для другого значения **N**, необходимо изменить константу **N**, затем скомпилировать и выполнить. При использовании динамических типов данных можно объявить размер массива во время выполнения программы, но мы пока используем только статические типы. Фундаментальным же свойством статических массивов является то, что их размеры фиксированы и должны быть известны перед их использованием.

Массивы являются фундаментальными структурами данных в том смысле, что они имеют прямое соответствие с системами памяти практически на всех компьютерах. Например, чтобы получить содержимое слова из памяти на машинном языке, мы должны указать адрес расположения первого символа этого слова. Таким образом, мы можем представить себе всю память компьютера как массив с адресами элементов в памяти, соответствующими индексам массива. Большинство компиляторов языков программирования высокого уровня переводят программы, содержащие обработку массивов, в

довольно эффективные программы на машинном языке, которые обращаются к памяти напрямую.

Связный список

Вторая элементарная структура данных, которая может понадобиться при изучении алгоритмов сортировки, – это связный список, который определен как примитив в некоторых языках программирования.

Связный список – это набор элементов, организованных последовательно, как массив. В массиве последовательная организация обеспечивается неявно (позицией в массиве); в связном списке мы используем явное расположение, в котором каждый элемент является частью узла, который также содержит ссылку на следующий узел. На рисунке 1 показан связный список, в котором элементы представлены буквами, узлы – кружками, а ссылки – линиями, соединяющими узлы.

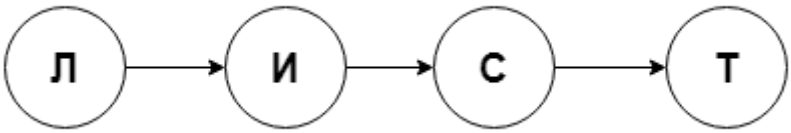

Рисунок 1 – Связный список

Основное преимущество связных списков перед массивами состоит в том, что связные списки могут увеличиваться и уменьшаться в размере в течение срока их службы. В частности, не требуется заранее знать их максимальный размер. В практических приложениях это часто позволяет использовать несколько структур данных в одном и том же пространстве в любое время, не обращая особого внимания на их относительный размер.

Второе преимущество связных списков состоит в том, что они обеспечивают гибкость, позволяя эффективно переупорядочивать элементы. Эта гибкость достигается за счет быстрого доступа к любому произвольному элементу в списке.

Даже простая схема, показанная на рисунке 1, раскрывает два важных свойства связных списков. Во-первых, у каждого узла есть ссылка, поэтому ссылка в последнем узле списка тоже должна указывать на какой–то следующий узел, которого на самом деле нет. Для этой цели используется фиктивный узел, который может для примера быть назван «!»: последний узел списка будет указывать на «!», а «!» укажет на себя. Вдобавок, будет

использоваться фиктивный узел в начале списка. Этот узел, на который нет указателя, называется головным. Он будет указывать на первый узел в списке. Основное назначение фиктивных узлов – сделать определенные манипуляции со ссылками, особенно с теми, которые связаны с первым и последним узлами в списке, более удобными. На рисунке 2 показана структура списка с включенными фиктивными узлами.

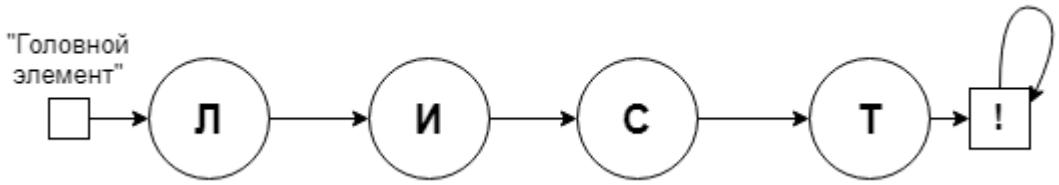

Рисунок 2 – Связный список с фиктивными узлами

Такое явное представление порядка позволяет выполнять операции со списками намного эффективнее, чем это было бы возможно для массивов. Например, предположим, что мы хотим переместить букву «Т» из конца списка в начало. В массиве нам пришлось бы переместить каждый элемент, чтобы освободить место для нового элемента в начале; в связном списке мы просто меняем три ссылки, как показано на рисунке 3.

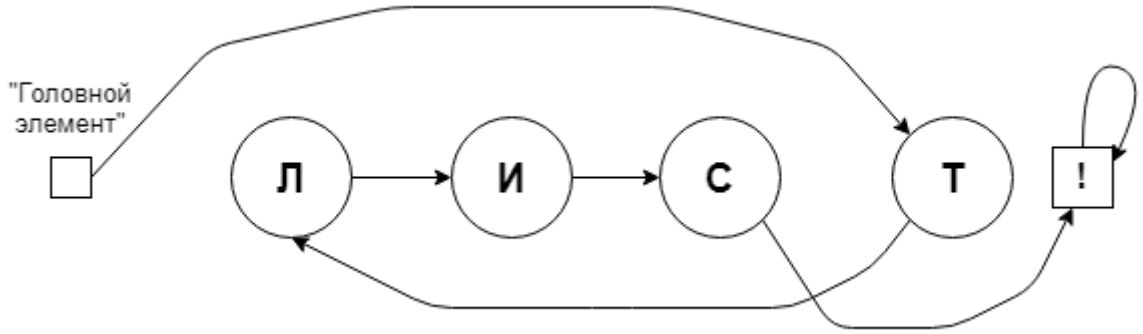

Рисунок 3 – Связный список с изменением порядка

Мы заставляем узел, содержащий «Т», указывать на «Л», узел, содержащий «С», указывать на «!», а «головной элемент» указывать на «Т». Даже если список очень длинный, достаточно изменить всего три ссылки.

Что еще более важно, мы можем говорить о «вставке» элемента в связный список (что заставляет его увеличиваться на единицу в длину) – это операция, которая неестественна и неудобна для массива. На рисунке 4 показано, как вставить «В» в список, поместив «В» в узел, указывающий на «С», а затем заставив узел, содержащий «И», указывать на новый узел.

Для этой операции необходимо изменить только две ссылки, независимо от того, какой длины список.

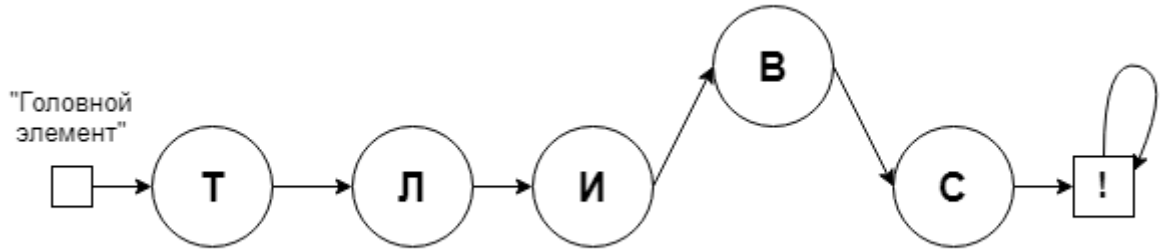

Рисунок 4 – добавление элемента в список

Стоить упомянуть еще о двух видах связных списков: кольцевой связный список и двунаправленный связный список. Преимущество кольцевого списка в том, что используется один «фиктивный» узел вместо двух. Преимущество двунаправленного списка – возможность «перемещения» по списку в обе стороны с целью удобного редактирования списка.

C предоставляет примитивные операции, которые позволяют напрямую реализовать связанные списки.

```
struct node
{ int key;
struct node* next; };
struct node* head, * z, * t;

listinitialize()
{
        head = (struct node*)malloc(sizeof * head);

        z = (struct node*)malloc(sizeof * z);
        head–>next = z; z–>next = z;
}
deletenext(struct node* t)
{
        t–>next = t–>next–>next;
}
struct node* insertafter(int v, struct node* t)
{
        struct node* x;
        x = (struct node*)malloc(sizeof * x);
        x–>key = v; x–>next = t–>next;
        t–>next = x;
```

```
        return x;
}
```

Точный формат списков описан в объявлении **struct**: списки состоят из узлов, каждый узел содержит целое число и указатель на следующий узел в списке. Ключ здесь – это целое число, он является ключом к списку. Переменная **head** – это указатель на первый узел в списке. При использовании списков есть возможность исследовать узлы по порядку, начиная с первого, следуя указателям, пока не достигнем **z** - указателя на фиктивный узел, обозначающий конец списка. Обозначение «стрелка» (–>) используется в языке ***C*** для перехода по указателям в структурах. Необходимо писать ссылку на ссылку, за которой следует этот символ, чтобы указать ссылку на узел, на который указывает эта ссылка. Например, ссылка **head –> next–> key** относится к первому элементу в списке, а ссылка **head–> next–> next–> key** относится ко второму.

Объявление **struct** просто описывает форматы узлов, а сами узлы могут быть созданы только при вызове встроенной процедуры **malloc**. Например, вызов **z = (struct node*) malloc (size of*z**) создает новый узел, помещая указатель на него в **z**. Цель **malloc** – избавить программиста от бремени выделения памяти для узлов по мере роста списка.

Рассмотрим одну из занимательных задач математики – задачу Иосифа Флавия. Иосиф Флавий – знаменитый военачальник, предводитель десятитысячного войска во время Иудейской войны 67-го года и к тому же писатель. При осаде его армии император Веспасиан предложил иудеям сдаться, обещав полную неприкосновенность. Иосиф согласился, но не мог склонить к тому же своих товарищей, покушавшихся убить его за измену. Тогда он предложил по жребию постепенно умерщвлять друг друга. В конце концов остался в живых только Иосиф с товарищем, которого он убедил отдаться римлянам.

Суть задачи Иосифа Флавия заключается в следующем. Есть **N** воинов, стоящих по кругу и убивающих каждого **M**–го. Допустим, что **N** = 10, а **M** = 2. Если производить отсчет, начиная с 1–го в круге солдата, то порядок будет: 2, 4, 6, 8, 10, 3, 7, 1, 9. Остается солдат под номером 5. Для решения задачи нам пригодится связный список и усвоенный материал. Результат показан на рисунке 5.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
struct node
{
	int key;
	struct node *next;
};

 main() {
	int i, N, M;
	struct node *t, *x;
	scanf("%d %d", &N, &M);
	t = (struct node *) malloc(sizeof *t);
	t–>key = 1; x = t;
	for (i = 2; i <= N; i++) {
		t–>next = (struct node *)malloc(sizeof *t);
		t = t–>next;
		t–>key = i;
	}
	t–>next = x;
	while (t != t–>next)
	{
		for (i = 1; i < M; i++)
		t = t–>next;
		printf("%d\t", t–>next–>key);
		x = t–>next; t–>next = t–>next–>next;
		free(x);
	}
	printf("%d\n", t–>key);
	}
```

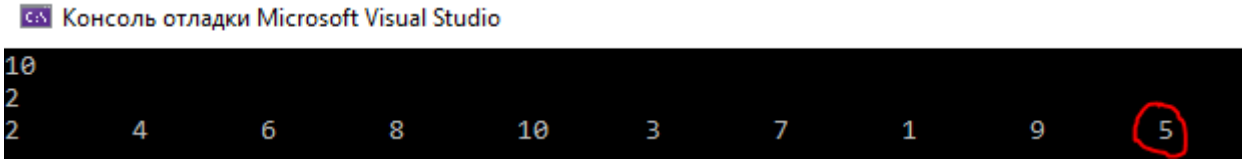

Рисунок 5 – Результат работы программы

Программа работает с круговым связным списком для непосредственного моделирования последовательности выполнения. Сначала список строится с ключами от **1** до **N**: переменная ***x*** сохраняется в начале списка по мере его построения, затем указатель в последнем узле в списке устанавливается на ***x***. Программа перебирает список, считая **M–1** элементов и удаляя следующие, пока не останется только один (который затем указывает на себя).

Стеки

Для многих приложений нет нужды в структурировании данных для произвольных вставок, удалений или доступа к элементам. В случае таких программ достаточно учитывать различные ограничения на доступ к структуре данных. Такие ограничения полезны по двум причинам: во-первых, они могут облегчить потребность в программе, использующей структуру данных, чтобы иметь дело с ее деталями (например, отслеживая ссылки или индексы элементов); во-вторых, они более просты и гибки в реализации, поскольку требуется поддерживать меньшее количество операций.

Самая важная структура данных с ограниченным доступом – это стек. Здесь задействованы только две основные операции: можно поместить элемент в стек (вставить его в начало) и выдвинуть элемент (удалить его из начала, вытолкнуть). Стек работает как стопка бумаг у офисного работника: работа накапливается в стеке, и всякий раз, когда работник готов выполнить некоторую работу, он снимает ее с вершины. Это может означать, что что-то застревает в нижней части стека на какое-то время, но хороший работник, вероятно, сможет периодически очищать стек.

Оказывается, иногда компьютерная программа естественным образом организована так, что одни задачи откладываются, а другие выполняются, и, таким образом, выталкиваемые стеки становятся одной из фундаментальных структур данных для многих алгоритмов.

Давайте рассмотрим использование стеков при вычислении арифметических выражений. Предположим, что кто–то хочет найти значение простого арифметического выражения, содержащего умножение и сложение целых чисел, например: 2* (((5+7)*(3*4))+9).

Стек – идеальный механизм для сохранения промежуточных результатов в таких вычислениях.

Приведенный выше пример можно вычислить с помощью вызовов:

```
push(2);
push(5);
push(7);
push(pop() + pop());
push(3);
push(4);
push(pop()* pop());
push(pop()* pop();
```

```
push(9);
push(pop() + pop());
push(pop()* pop());
printf("%d\n", pop());
```

Порядок, в котором выполняются операции, определяется скобками в выражении и условием, согласно которому мы действуем слева направо. В языке ***C*** порядок, в котором выполняются две операции **pop ()**, не указан, поэтому для некоммутативных операций, таких как вычитание и деление, требуется немного более сложный код. Базовые операции со стеком легко реализовать с помощью связных списков, как в примере ниже:

```
static struct node
{
        int key;
        struct node* next;
};

static struct node* head, * z, * t;

stackinit()
{
        head = (struct node*)malloc(sizeof * head);
        z = (struct node*)malloc(sizeof * z);
        head–>next = z;
        head–>key = 0;
        z–>next = z;
}
push(int v) {
        t = (struct node*)malloc(sizeof * t);
        t–>key = v;
        t–>next = head–>next;
        head–>next = t;
}
int pop()
{
        int x;
        t = head–>next;
        head–>next = t–>next;
        x = t–>key;
        free(t);
        return x;
}
int stackempty()
{
        return head–>next == z;
        }
```

Эта реализация также включает в себя код для инициализации стека и проверки его пустоты. В приложении, в котором используется только один стек, мы можем предположить, что глобальная переменная **head** является ссылкой на стек. В других случаях реализации могут быть изменены, чтобы также передавать ссылку на стек.

Порядок вычислений в приведенном выше арифметическом примере требует, чтобы операнды появлялись перед оператором, чтобы они могли быть в стеке при обнаружении оператора. Перепишем пример следующим образом: 2 5 7 + 3 4 * * 9 + *. Эта форма записи выражения называется постфиксом. Обычный способ написания арифметических выражений называется инфиксным. Одно интересное свойство постфикса состоит в том, что скобки не требуются; в инфиксе они нужны, чтобы отличать, например, 2 * (((5 + 7) * (3 * 4)) + 9) от ((2 * 5) +7) * ((3 * 4) +9) . Следующая программа преобразует инфиксное выражение, заключенное в круглые скобки, в постфиксное выражение (результат выполнения программы представлен на рисунке 6):

```
char c;
for (stackinit(); scanf("%ls", &c) != EOF; )
{
        if (c == ')')
                printf("%lc", (char)pop());

        if (c == '+') push((int)c);
        if (c ==  '*') push((int) c);
                while (c>='0' && c<='9')
                {
                        printf("%lc",c);
                        scanf ("%lc",&c);
                }
                if (c != '(') printf(" ");
}
        printf("\n");
```

```
2 * ( ( ( 5 + 7 ) * ( 3 * 4 ) ) + 9 )
2  5  7 +  3  4 * *  9 + *
```

Рисунок 6 – Результат работы программы

Операторы помещаются в стек, а аргументы просто передаются. Обратите внимание, что аргументы появляются в постфиксном выражении в том же порядке, что и в инфиксном выражении. Затем каждая правая скобка указывает, что оба аргумента для последнего оператора были выведены, поэтому сам оператор может быть извлечен и записан. Для простоты эта программа не

проверяет наличие ошибок во вводе и требует пробелов между операторами, скобками и операндами.

Очереди

Еще одна фундаментальная структура данных с ограниченным доступом называется очередью. Опять же, здесь задействованы только две основные операции: можно вставить элемент в очередь в начале и удалить элемент из конца. Возможно, ящик для входящих сообщений нашего офисного работника должен работать как очередь, поскольку тогда работа, которая приходит первой, будет выполнена в первую очередь. В стопке что-то может зарыться внизу, но в очереди все обрабатывается в порядке поступления.

```
#define max 100

static int queue[max + 1], head, tail;
put(int v)
{
        queue[tail++] = v;
        if (tail > max) tail = 0;
}
int get() {

        int t = queue[head++];
        if (head > max) head = 0;
        return t;
}
queueinit()
{
        head = 1; tail = 10;
}
int queueempty()
{
        return head == tail;
}
```

Необходимо поддерживать два индекса: один - до начала очереди (**head**) и один - в конце (**tail**). Содержимое очереди – это все элементы в массиве между **head** и **tail**, с учетом возврата к **0**, когда встречается конец массива. Если **head** и **tail** равны, то очередь определяется как пустая (рисунок 7).

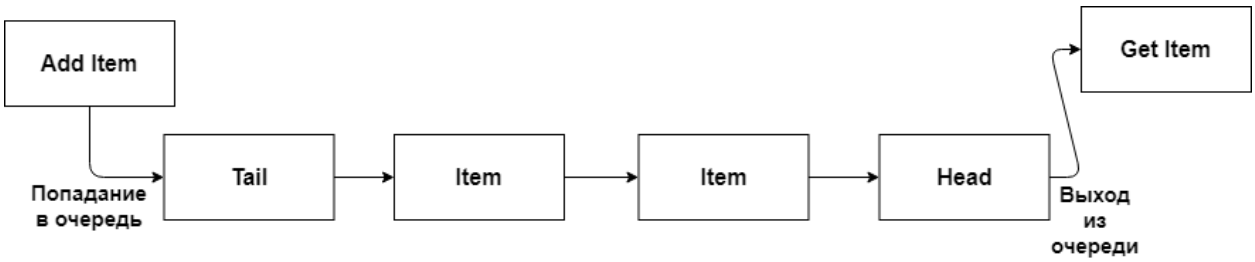

Рисунок 7 – Очередь

Деревья

Структуры данных, о которых говорилось ранее, по своей сути одномерны: один элемент следует за другим. Настало время рассмотреть двумерные связные структуры, называемые **деревьями**, которые лежат в основе многих алгоритмов. **Деревья** часто встречаются в повседневной жизни, например, в семейных альбомах, где люди отслеживают предков с помощью генеалогического древа. Другие примеры – организация спортивных турниров, иерархия сотрудников на работе, библиотечный каталог, файловая система компьютера и т.п.

Дерево в информатике похоже на дерево в реальном мире, с той лишь разницей, что в информатике оно визуализируется перевернутым, с корнем на вершине и ветвями, идущими от корня к листьям дерева. Древовидная структура данных используется для различных реальных приложений, поскольку она может отображать отношения между различными узлами с использованием иерархии «родитель-потомок». Из-за этого она также известна как **иерархическая структура данных**. Такая структура данных широко используется для упрощения и ускорения операций поиска и сортировки.

Дерево – это нелинейная структура данных. Дерево может быть представлено с использованием различных примитивных или определяемых пользователем типов данных. Для реализации дерева мы можем использовать массивы, связные списки, классы или другие типы структур данных. Это набор связанных друг с другом узлов. Чтобы показать связь, узлы соединяются ребрами. На рисунке 8 изображен пример стандартного дерева.

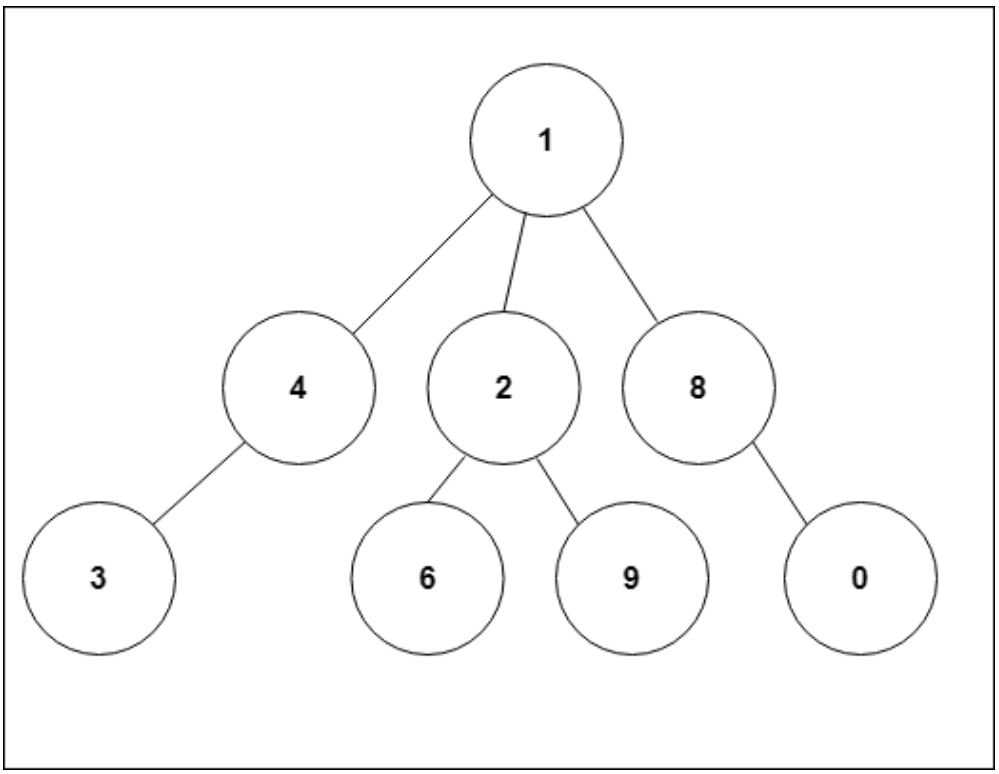

Рисунок 8 – Дерево

Изучив рисунок выше, мы можем сделать несколько выводов:

- «1» – корень дерева, самый верхний узел,
- «1» – «родительский узел» для «2»,«4», «8»,
- «2», «4», «8» – являются потомками «1»,
- ребро – связь, соединение между двумя узлами.

Свойства дерева:

- Каждое дерево имеет специальный узел, называемый корневым узлом. Корневой узел можно использовать для обхода каждого узла дерева. Он называется корневым, потому что дерево происходит только от корня.
- Если дерево имеет N вершин (узлов), то количество ребер всегда на единицу меньше количества узлов (вершин), то есть N-1. Если у него более N-1 ребер, он называется графом, а не деревом.
- У каждого дочернего элемента есть только один родитель, но у родительского элемента может быть несколько дочерних элементов. Узлы без потомков называют листьями.

Основные типы деревьев в структуре данных

Дерево общего вида

Дерево называется деревом общего вида, если на иерархию дерева нет ограничений. В общем дереве каждый узел может иметь бесконечное количество дочерних элементов. Это дерево является “надмножеством” всех остальных типов деревьев. Дерево, показанное на рисунке 8, является деревом общего вида.

Двоичное (бинарное) дерево

Двоичное дерево – это тип дерева, в котором каждый родитель может иметь не более двух дочерних элементов (рисунок 9). Их называют левым и правым потомком. Это одно из наиболее часто используемых деревьев. Когда на двоичное дерево накладываются определенные ограничения и свойства, это приводит к возникновению ряда других широко используемых деревьев, таких как двоичное дерево поиска, дерево АВЛ, красно-черному дереву и т.д. Назначение двоичного дерева – структурировать внутренние узлы (на рисунке 9 обозначены кругами); внешние узлы (на рисунке 9 обозначены квадратами) служат только в качестве заполнителей. Мы включаем их в определение, потому что наиболее часто используемые представления двоичных деревьев должны учитывать каждый внешний узел. Бинарное дерево может быть «пустым», не состоящим из внутренних узлов и одного внешнего узла .

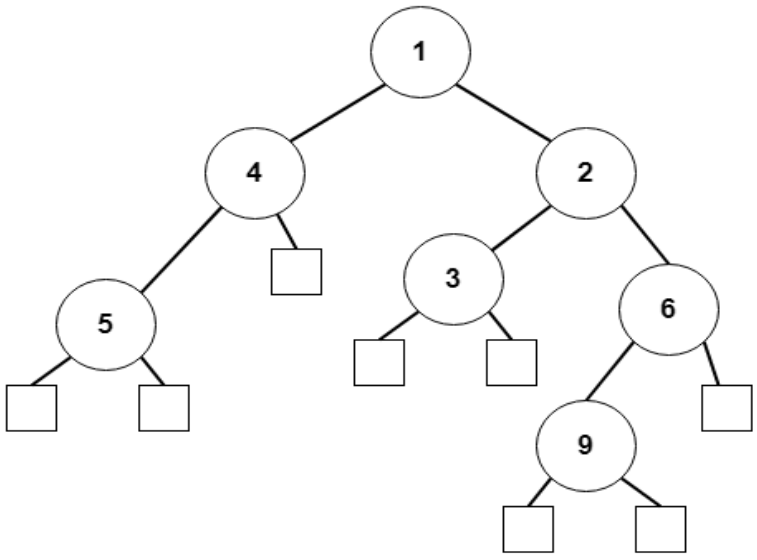

Рисунок 9 – пример двоичного дерева с внутренними и внешними узлами

Дерево двоичного поиска

Дерево двоичного поиска – это подвид двоичного дерева с некоторыми добавленными ограничениями (рисунок 10). В таком дереве значение левого дочернего элемента узла должно быть меньше или равно значению его родительского элемента, а значение правого дочернего элемента всегда больше или равно значению его родителя. Это свойство двоичного дерева поиска делает его подходящим для операций поиска, поскольку на каждом узле мы можем точно решить, будет ли значение находиться в левом поддереве или в правом поддереве (отсюда и название – «дерево поиска»).

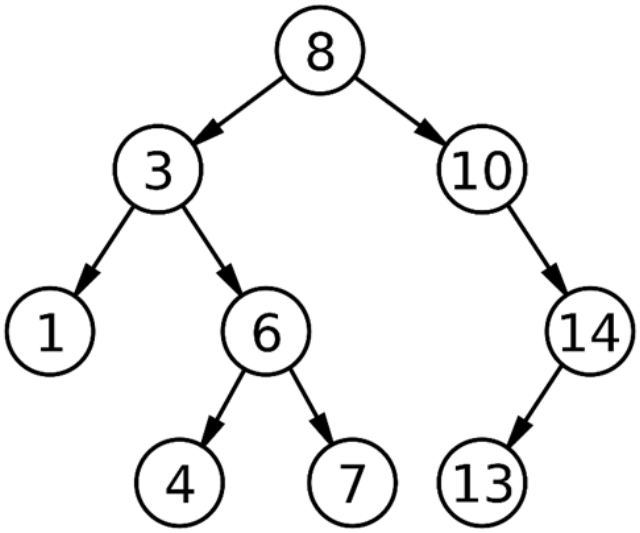

Рисунок 10 – дерево двоичного поиска

Дерево АВЛ

Дерево АВЛ (AVL) – это самобалансирующееся двоичное дерево поиска. Название АВЛ дано по имени его изобретателей – советских ученых Адельсон-Велши и Ландис в 1968 году. Это было первое динамически балансирующееся дерево. В дереве АВЛ каждому узлу назначается коэффициент балансировки (рисунок 11), на основе которого вычисляется, сбалансировано дерево или нет. В дереве АВЛ высота дочерних узлов отличается не более чем на 1. Допустимые коэффициенты балансировки в дереве АВЛ равны 1, 0 и -1. Когда к дереву АВЛ добавляется новый узел и дерево становится несбалансированным, выполняется операция вращения, балансирующая дерево. В дереве АВЛ такие операции как поиск, вставка и удаление, занимают время O(log n).

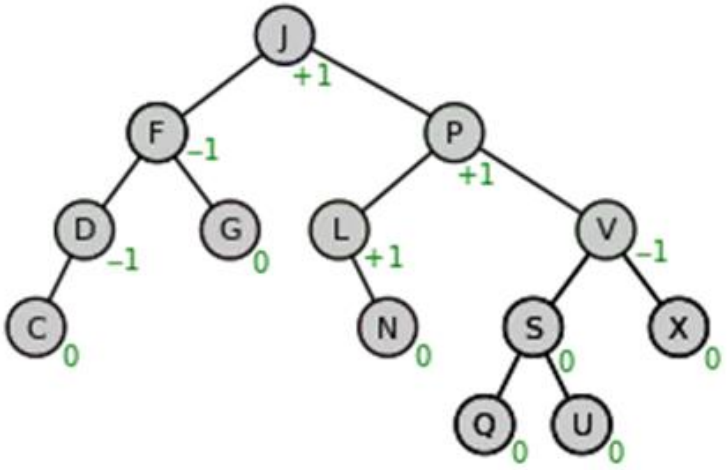

Рисунок 11 – дерево АВЛ

Красно-черное дерево

Красно-черное дерево (RBT) – еще один тип самобалансирующегося дерева (рисунок 12). Такое название оправдывается тем, что каждый узел в данном дереве окрашен в красный или черный цвет, за счет чего и достигается сбалансированность. Красно-черное дерево используется для организации сравнимых данных, таких как числа или фрагменты текста и имеет следующие свойства:

1. Каждый узел красный или черный и имеет два дочерних элемента,
2. каждый лист (NULL) черный,
3. Корневой узел принято считать черным,
4. если узел красный, то оба его дочерних элемента черные,
5. каждый путь от узла к листу-потомку содержит одинаковое количество черных узлов.

Операции вставки, удаления и поиска требуют в худшем случае времени, пропорционального длине дерева, что позволяет красно-чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

Чтобы понять, как это работает, достаточно рассмотреть эффект свойств 4 и 5 вместе. Пусть для красно-чёрного дерева A число чёрных узлов от корня до листа равно X. Тогда кратчайший возможный путь до любого листа содержит X узлов и все они чёрные. Более длинный возможный путь может быть построен путём включения красных узлов. Однако, благодаря свойству 4 в дереве не может быть двух красных узлов подряд, а согласно 2 и 3, путь начнётся и

закончится чёрным узлом. Поэтому самый длинный возможный путь состоит из 2X-1 узлов, попеременно красных и чёрных.

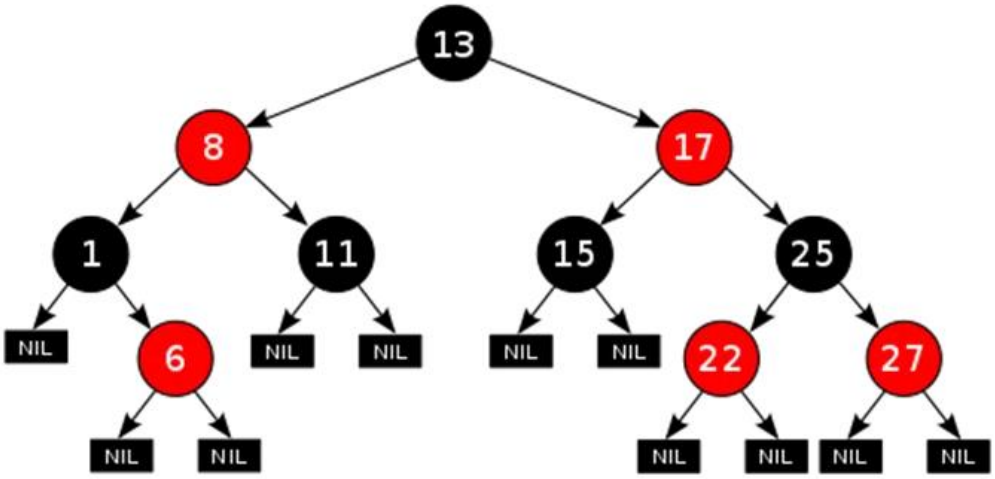

Рисунок 12 – красно-черное дерево

Рекурсия

Рекурсия – одно из фундаментальных понятий в информатике, математике и программировании. Что такое рекурсия? Простыми словами, рекурсивная программа – программа, **вызывающая сама себя** (а рекурсивная функция – это такая функция, которая определяется в терминах самой себя). Однако рекурсивная программа не может вызывать себя всегда (иначе она никогда не остановится), поэтому важным компонентом в таких программах – условие выхода из рекурсии, когда программа может перестать вызывать сама себя. Все практические вычисления могут быть выполнены в рекурсивной структуре. Многие интересные алгоритмы довольно просто выражаются с помощью рекурсивных программ, и многие разработчики алгоритмов предпочитают выражать методы рекурсивно.

Рекурсивные определения функций довольно распространены в математике – простейшие типы, включающие целочисленные аргументы, называются рекуррентными отношениями. Возможно, наиболее известной такой функцией является факториальная функция, определяемая формулой 1:

$$N! = N * (N - 1)!,\ N >= 1, 0! = 1. \quad (1)$$

Это напрямую соответствует следующей простой рекурсивной программе:

```
int factorial(int N)
    {
        if (N == 0) return 1;
        return N * factorial(N – 1);
    }
```

Такая программа иллюстрирует основные особенности рекурсивной программы: она вызывает себя (с меньшим значением своего аргумента) и имеет условие завершения, в котором она вычисляет свой результат. Кроме того, важно помнить, что это программа, а не уравнение: например, ни уравнение, ни программа выше не работают для отрицательного N. Вызов factorial(–1) приводит к бесконечному рекурсивному циклу.

Второе известное рекуррентное соотношение – это то, которое определяет числа Фибоначчи (2):

$$F_N = F_{N-1} + F_{N-2}, N >= 2, F_0 = F_1 = 1. \quad (2)$$

Это определяет последовательность 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, 17711…., что соответствует следующей рекурсивной программе:

```
int fibonacci(int N)
        {
                if (N <= 1) return 1;
                return fibonacci(N – 1) + fibonacci(N – 2);
        }
```

Приведенная выше программа представляет собой алгоритм с экспоненциальным временем для вычисления чисел Фибоначчи.

Для сравнения, можно вычислить F_N за линейное время следующим образом:

```
#define max 25
        int fibonacci(int N)
        {
                int i, F[max];
                F[0] = 1; F[1] = 1;
                for (i = 2; i <= max; i++)
                        F[i] = F[i – 1] + F[i – 2];
                return F[N];
        }
```

Эта программа вычисляет первые максимальные числа Фибоначчи, используя массив размером max (поскольку числа растут экспоненциально, max будет небольшим). Фактически, этот метод использования массива для хранения предыдущих результатов обычно является методом для оценки рекуррентных соотношений, поскольку он позволяет обрабатывать довольно сложные уравнения единообразным и эффективным способом.

Ряд читателей, уже знакомых с таким понятием, как рекурсия в программировании, могли сталкиваться с ошибкой переполнения стека. Если выход из рекурсии не достигнут или не определен – может возникнуть ошибка переполнения стека.

```
int fact(int n)
{

  if (n == 100) // неправильное условие завершения
    return 1;

  else
          return n * fact(n - 1);
```

Если вызывается fact(10), он будет вызывать fact(9), fact(8), fact(7) и так далее, но n никогда не достигнет 100. Таким образом, условие выхода из рекурсии не достигается. Если память будет исчерпана этими функциями, это вызовет ошибку переполнения стека. Ниже приведен пример программы подсчета первых чисел из таблицы факториалов с использованием рекурсии и правильным условием выхода. Результат работы программы представлен на рисунке 13.

```
#include <iostream>

int factorial(int n)
{
   if (n == 0)
      return 1;
   else
      return factorial(n - 1) * n;
}


int main() //считаем первые 10 факториал
{
   for (int count = 0; count < 11; ++count)
      std::cout << factorial(count) << " ";

   return 0;
         }
```

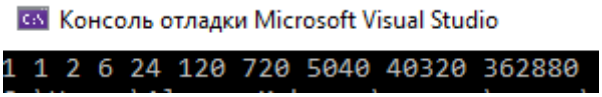

Рисунок 13 – результат работы программы с рекурсивной функцией

Парадигма «Разделяй и властвуй»

Большинство рекурсивных программ используют два рекурсивных вызова, каждый из которых обрабатывает примерно половину входных данных. Это так называемая парадигма «разделяй и властвуй», используемая для разработки алгоритмов, которая часто используется для нахождения оптимального решения.

В качестве примера рассмотрим задачу нахождения максимального и минимального элементов массива. Допустим, что наш массив – а {50, 350, 40, 93, 144, 13}. Решение данной задачи состоит из трех шагов: “разделяй”, “властвуй”, “объединяй”. Для нахождения максимума используется рекурсивный подход, где мы разбиваем массив пополам, а затем все подмассивы до тех пор, пока не останутся отдельные элементы. Шаг 1 (рисунок 14) – выполнен.

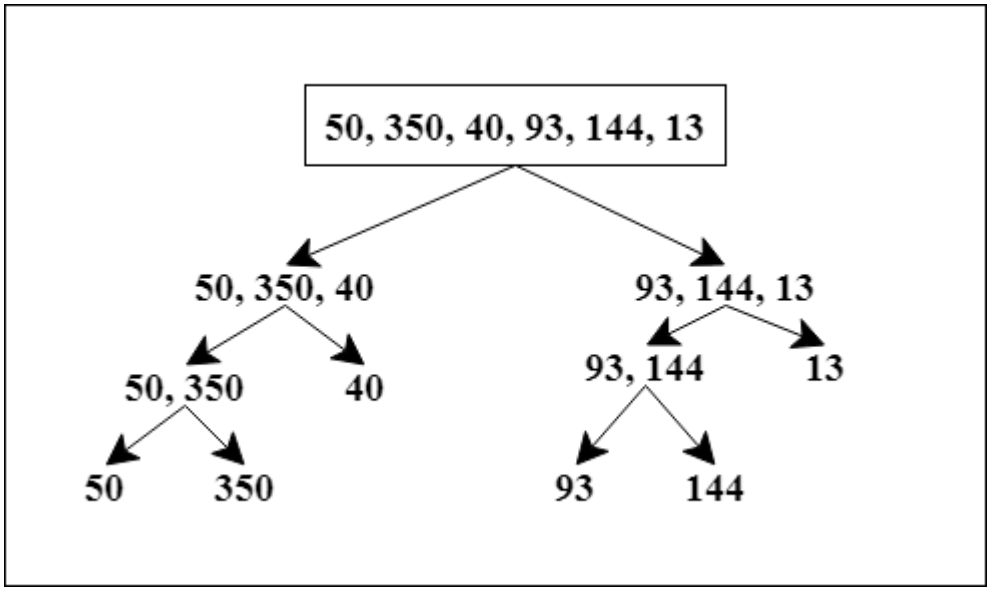

Рисунок 14 – шаг 1 парадигмы “разделяй и властвуй”

Затем, мы поочередно сравниваем элементы между собой в левой и правой части, переставляя их в порядке возрастания. Это шаги 2 и 3 – “властвуй” и “объединяй” (рисунок 15).

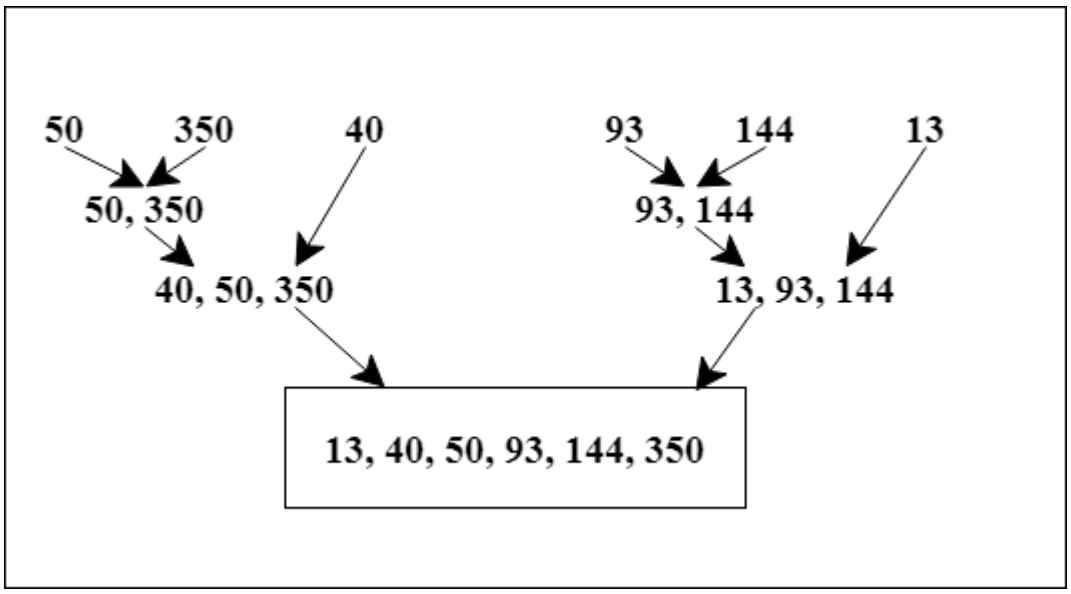

Рисунок 15 – шаги 2,3 парадигмы "разделяй и властвуй"

Разберем пример программы, выполняющих сортировку массива по парадигме "разделяй и властвуй". Для нахождения максимума мы разбиваем массив до состояния сравнения двух элементов, где применяем a[index]>a[index+1].

```
if (index >= l – 2)
{
   if (a[index] > a[index + 1])
   {
      // (a[index]
      // Теперь мы можем сказать, что последний элемент будет максимальным в данном массиве.
   }
   else
   {
      //(a[index+1]
      // Теперь мы можем сказать, что последний элемент будет максимальным в данном массиве.
   }
         }
```

В приведенном выше условии мы проверили условие левой части, чтобы найти максимум. Теперь необходимо условие для нахождения максимума правой стороны. Для этого используем рекурсивную функцию для проверки правой части индекса массива.

```
max = DAC_Max(a, index + 1, l);
// Вызов рекурсии
```

Теперь необходимо сравнить условие и проверить правую сторону в индексе данного массива.

```
// правый элемент – максимальный
if(a[index]>max)
return a[index];
// max – максимальный элемент
else
return max;
```

Для нахождения минимума в заданном массиве мы также собираемся обратиться за помощью к рекурсивному подходу.

```
int DAC_Min(int a[], int index, int l)
//рекурсивный вызов функции для поиска минимального числа в массиве
if (index >= l – 2)
// проверка условия на наличие двух элементов для нахождения минимума
{
   if (a[index] < a[index + 1])
      return a[index];
   else
      return a[index + 1];
         }
```

Теперь проверим условие для правой части массива.

```
// Рекурсивный вызов правой части массива
         min = DAC_Min(a, index + 1, l);

// Правый элемент – минимальный
if (a[index] < min)
   return a[index];
// минимум массива
else
return min;
```

Итоговый листинг программы имеет вид:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
int DAC_Max(int a[], int index, int l);
int DAC_Min(int a[], int index, int l);


int DAC_Max(int a[], int index, int l)
{
   int max;
```

```
    if (index >= l – 2) {
        if (a[index] > a[index + 1])
            return a[index];
        else
            return a[index + 1];
    }

    max = DAC_Max(a, index + 1, l);

    if (a[index] > max)
        return a[index];
    else
        return max;
}


int DAC_Min(int a[], int index, int l)
{
    int min;
    if (index >= l – 2) {
        if (a[index] < a[index + 1])
            return a[index];
        else
            return a[index + 1];
    }

    min = DAC_Min(a, index + 1, l);

    if (a[index] < min)
        return a[index];
    else
        return min;
}

int main()
{

    int min, max, N;


    int a[6] = { 50, 350, 40, 93, 144, 13 };
```

```
    max = DAC_Max(a, 0, 6);
    min = DAC_Min(a, 0, 6);

    printf("The minimum number in a given array is : %d\n", min);
    printf("The maximum number in a given array is : %d", max);
    return 0;
}
```

Результатом должны стать числа 13 и 350. Результат данной программы на рисунке 16.

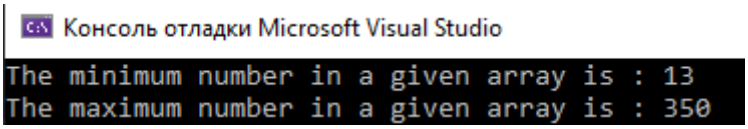

Рисунок 16 – результат компиляции программы

Одна из самых популярных сортировок, использующих данную парадигму является «Быстрая сортировка». Поговорим о ней поподробнее.

Быстрая сортировка

Быстрая сортировка – один из самых распространённых алгоритмов. Базовый алгоритм был изобретен в 1960 году Т. Хоаром. Быстрая сортировка популярна, потому что ее нетрудно реализовать, это хорошая универсальная сортировка (она хорошо работает в различных сценариях) и во многих ситуациях потребляет меньше ресурсов, чем любой другой метод сортировки.

Положительные особенности алгоритма быстрой сортировки заключаются в том, что он требует в среднем всего около NlogN операций для сортировки N элементов и имеет чрезвычайно короткий внутренний цикл. Недостатки алгоритма в том, что он рекурсивный (реализация усложняется, если рекурсия недоступна), в худшем случае требуется около N^2 операций и простая ошибка в реализации может остаться незамеченной, что вызовет ряд проблем при работе с файлами.

Быстрая сортировка – это метод сортировки по принципу «разделяй и властвуй». Он работает путем разделения файла на две части, а затем независимой сортировки частей.

```
quicksort(int a[], int l, int r)
        {
                int i;
                if (r > l)
                {
                        i = partition(l, r);
                        quicksort(a, l, i–1);
                        quicksort(a, i + 1, r);
                }
        }
```

Параметры l и r ограничивают подфайл в исходном файле, который должен быть отсортирован; вызов quicksort (1, N) сортирует весь файл.

Суть метода – процедура разбиения, которая должна переставить массив так, чтобы выполнялись следующие три условия:

- элемент a [i] находится на своем последнем месте в массиве для некоторого i
- все элементы в a [1], ..., a [i–1] меньше или равны a [i],
- все элементы в a [i + 1], ..., a [r] больше или равны a [i].

Это можно просто и легко реализовать с помощью следующей общей стратегии. Во–первых, произвольно выберите [r] как элемент, который войдет в свою конечную позицию.

«Пройдем» массив с левого конца до тех пор, пока не будет найден элемент, больший, чем [r], и просмотрите массив с правого конца, пока не будет найден элемент, меньший, чем [r]. Очевидно, что два элемента, остановивших поиск, неуместны в окончательном секционированном массиве, поэтому поменяйте их местами.

Продолжение этого способа гарантирует, что все элементы массива слева от левого указателя меньше, чем [r], а все элементы массива справа от правого указателя больше, чем [r]. Когда указатели поиска пересекаются, процесс разделения почти завершен: все, что остается, – это заменить [r] крайним левым элементом правой части (элементом, на который указывает левый указатель).

```
quicksort(int a[], int l, int r)
        {
                int v, i, j, t;
                if (r > 1)
```

```
        {
            v = a[r]; i = l – 1; j = r;
            for (;;)
            {
                while (a[++i] < v);
                while (a[—j] > v);
                if (i >= j) break;
                t = a[i]; a[i] = a[j]; a[j] = t;
            }
            t = a[i]; a[i] = a[r]; a[r] = t;
            quicksort(a, l, i – 1);
            quicksort(a, i + 1, r);
        }
}
```

В этой реализации переменная v содержит текущее значение «элемента разделения» a[r], а i и j – левый и правый указатели сканирования файла, соответственно. Цикл разбиения реализован как бесконечный цикл с выходом break, когда указатели пересекаются.

Сортировка слиянием

Как и быстрая сортировка, сортировка слиянием – это алгоритм «разделяй и властвуй». Он делит входной массив на две половины, вызывает себя для двух половин, а затем объединяет две отсортированные половины. Функция merge() используется для объединения двух половинок. Слияние (arr, l, m, r) – это ключевой процесс, который предполагает, что arr [l..m] и arr [m + 1..r] отсортированы, и объединяет два отсортированных подмассива в один. Принцип работы сортировки слиянием можно разделить на 4 пункта.

```
MergeSort(arr[], l, r)
If r > l
// 1. Найдите среднюю точку, чтобы разделить массив на две половины :
middle m = l + (r – l) / 2
// 2. Вызов mergeSort для первой половины :
Call mergeSort(arr, l, m)
// 3. Вызов mergeSort для второй половины :
Call mergeSort(arr, m + 1, r)
// 4. Объедините две половинки, отсортированные на шагах 2 и 3:
Call merge(arr, l, m, r)
```

На рисунке 17 представлен полный процесс сортировки слиянием для примера массива {38, 27, 43, 3, 9, 82, 10}. Если мы внимательно рассмотрим

диаграмму, то увидим, что массив рекурсивно делится на две половины, пока размер не станет равным 1. Как только размер подмассивов станет равным 1, процессы слияния вступят в действие и начнут слияние массивов обратно до тех пор, пока весь массив не станет единым снова.

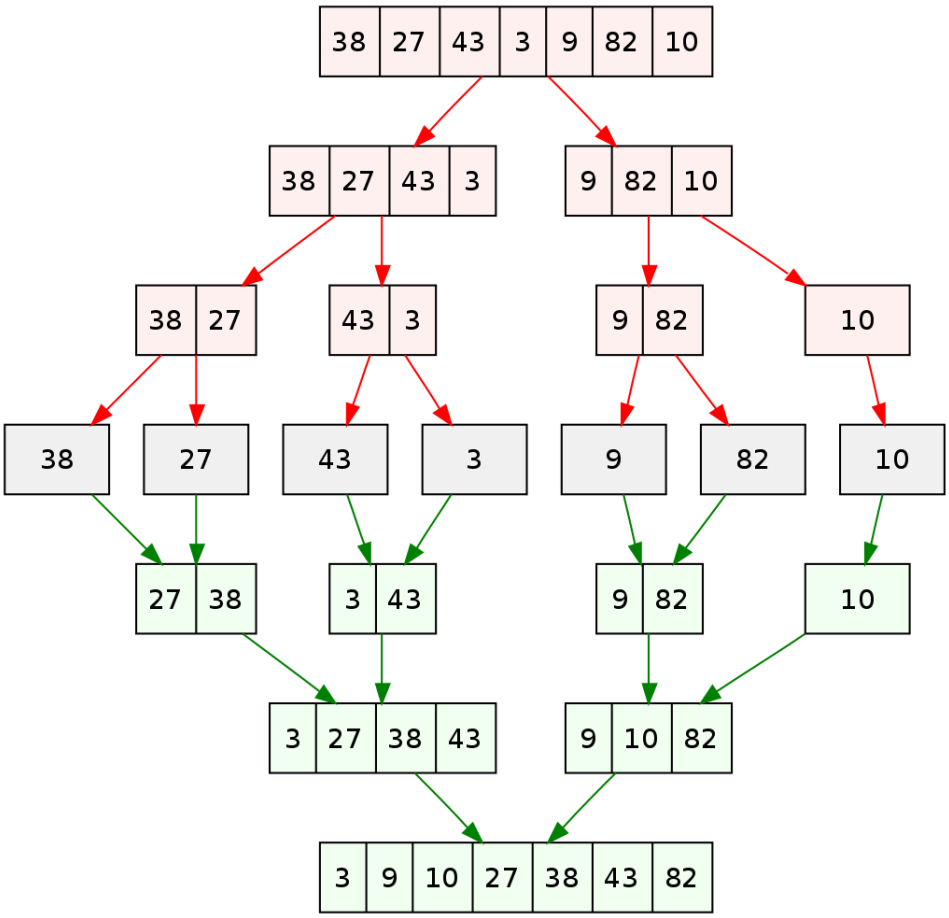

Рисунок 17 – сортировка слиянием

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

// Объединяет два подмассива arr [].
// Первый подмассив – arr [l..m]
// Второй подмассив – arr [m + 1..r]
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
   int n1 = m – l + 1;
    int n2 = r – m;

    // создаем временные массивы
    int L[n1], R[n2];
```

```
    // Копируем данные во временные массивы L[] и R[]
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    // Объединение временных массивов обратно в arr[l..r]
    i = 0; // Начальный индекс первого подмассива
    j = 0; // Начальный индекс второго подмассива
    k = l; // Начальный индекс третьего подмассива
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    /* Скопируйте оставшиеся элементы L [], если есть */
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Скопируйте оставшиеся элементы R [], если есть */
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}

/* l – левый индекс, а r – правый индекс
подмассив arr для сортировки */
void mergeSort(int arr[], int l, int r)
{
    if (l < r) {
        // То же, что (l + r) / 2, но избегает переполнения для больших l и h
        int m = l + (r – l) / 2;

        // Сортировка первой и второй половины
        mergeSort(arr, l, m);
```

```
        mergeSort(arr, m + 1, r);

        merge(arr, l, m, r);
    }
}


void printArray(int A[], int size)
{
    int i;
    for (i = 0; i < size; i++)
        printf("%d ", A[i]);
    printf("\n");
}

int main()
{
    int arr[] = { 12, 11, 13, 5, 6, 7 };
    int arr_size = sizeof(arr) / sizeof(arr[0]);

    printf("Given array is \n");
    printArray(arr, arr_size);

    mergeSort(arr, 0, arr_size – 1);

    printf("\nSorted array is \n");
    printArray(arr, arr_size);
    return 0;
          }
```

Сортировка слиянием полезна для сортировки связных списков за время $O(nLog_n)$. В случае массивов дело обстоит иначе, в основном из–за разницы в распределении памяти для массивов и связных списков. В отличие от массивов, узлы связного списка не могут быть смежными в памяти.

В массивах мы можем выполнять произвольный доступ к любому элементу последовательности, поскольку элементы непрерывны в памяти. Допустим, у нас есть целочисленный (4–байтовый) массив A, и пусть адрес A [0] равен x, тогда для доступа к A [i] мы можем напрямую получить доступ к памяти по адресу (x + i * 4). В отличие от массивов, мы не можем делать произвольный доступ в связном списке. Быстрая сортировка требует большого количества такого типа доступа. В связном списке для доступа к i–му индексу мы должны перемещаться по каждому узлу от «головы» до i–го узла, поскольку у нас нет непрерывного блока памяти. Сортировка слиянием обеспечивает

доступ к данным последовательно, и потребность в произвольном доступе невелика.

Сортировка выбором

Данный алгоритм сортировки работает следующим образом: сначала находится наименьший элемент в массиве и заменяется его элементом в первой позиции, затем второй наименьший элемент и замена его элементом во второй позиции, так до тех пор, пока не будет отсортирован весь массив. Этот метод называется сортировкой по выбору, потому что он работает путем многократного «выбора» наименьшего оставшегося элемента, как показано на рисунке 18. Первый проход показывает, что 1 является наименьшим элементом, поэтому он меняется местами с 8. На втором проходе обнаруживаем, что 4 находится на своем месте. На третьем проходе меняем 5 и 7 местами. На четвертом проходе 7 остается на своем месте, как и 8. Итог – отсортированный массив!

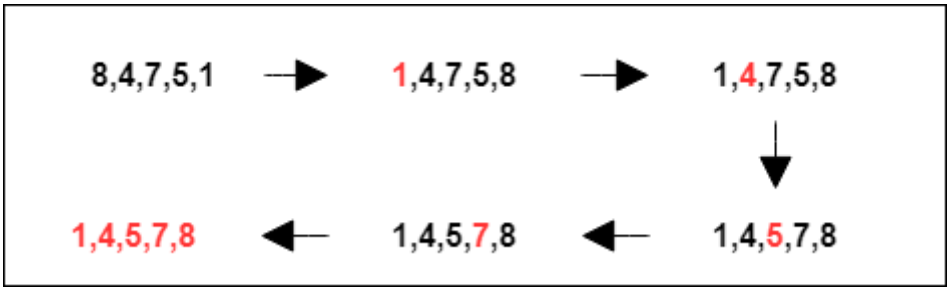

Рисунок 18 – сортировка выбором

Программа, написанная ниже, представляет собой реализацию этого процесса. Для каждого i от 1 до N–l он заменяет a [i] минимальным элементом в a [i] ... A [N]:

```
selection(int a[], int N)
{
        int i, j, min, t; for (i = 1; i < N; i++)
        {
        min = i; for (j = i + 1; j <= N; j++) if (a[j] < a[min]) min – j; t = a[min]; a[min] = a[i]; a[i] =
t;
        }
        }
```

Поскольку индекс i перемещается по файлу слева направо, элементы слева от индекса занимают свои конечные позиции в массиве (и больше не

будут затронуты), так что массив будет полностью отсортирован, когда индекс достигнет правого конца ряда.

Это один из самых простых методов сортировки, и он очень хорошо подходит для небольших файлов. «Внутренний цикл» – это сравнение a [j] <a [min] (плюс код, необходимый для увеличения j и проверки того, что оно не превышает N), что вряд ли может быть проще.

Более того, несмотря на очевидный подход «грубой силы», сортировка по выбору имеет довольно важное применение: поскольку каждый элемент фактически перемещается не более одного раза, сортировка по выбору является методом выбора для сортировки файлов с очень большими записями.

Сортировка вставками

Следующий алгоритм почти такой же простой, как сортировка выбором, но, возможно, более гибкий – сортировка вставкой. Это метод, который люди часто используют для сортировки в карточной игре «Бридж»: рассматривайте элементы по одному, вставляя каждый на свое место среди уже рассмотренных (сохраняя их отсортированными). Рассматриваемый элемент вставляется простым перемещением более крупных элементов на одну позицию вправо, а затем вставкой элемента на освободившееся место (рисунок 19). Этот процесс реализован в следующей программе. Для каждого i от 2 до N элементы a [1] ... a [i] сортируются путем помещения a [i] в позицию среди отсортированного списка элементов в a [1] ... a [i–1]:

```
insertion (int a[], int N)
{
        int i, j, v;
        for(i = 2; i <= N; i++)
        {
                v = a[i]; j = i;
                while(a[j –1] > v)
                {
                        a[j] = a[j – 1]; j—;
                }
                a[j] = v;
        }
        }
```

Как и при сортировке выбором, элементы слева от индекса i находятся в отсортированном порядке во время сортировки, но они не находятся в своей

конечной позиции, так как их, возможно, придется переместить, чтобы освободить место для более мелких элементов, которые встретятся позже. Однако массив полностью отсортирован, когда индекс достигает правого конца.

Следует учитывать еще одну важную деталь: процедура **insertion** не работает для большинства входных данных! **While** будет проходить за левым концом массива, когда v является наименьшим элементом в массиве. Чтобы исправить это, мы помещаем «контрольный ключ» (контрольное значение) в [0], делая его, по крайней мере, таким же маленьким, как наименьший элемент в массиве.

Если по какой–то причине использовать контрольный ключ неудобно (например, возможно, самый маленький ключ нелегко определить), тогда можно использовать тест, while j> l && a[j – 1]> v. Это непредпочтительно, потому что j = l случается крайне редко. Обратите внимание, что, когда j равно 1, в приведенном выше тесте не будет доступа к [j–1] из–за способа вычисления логических выражений в ***C*** – некоторые другие языки могут в таком случае выполнять доступ к массиву вне границ. Другой способ справиться с этой ситуацией в «C» – использовать команду *break* или *go to*.

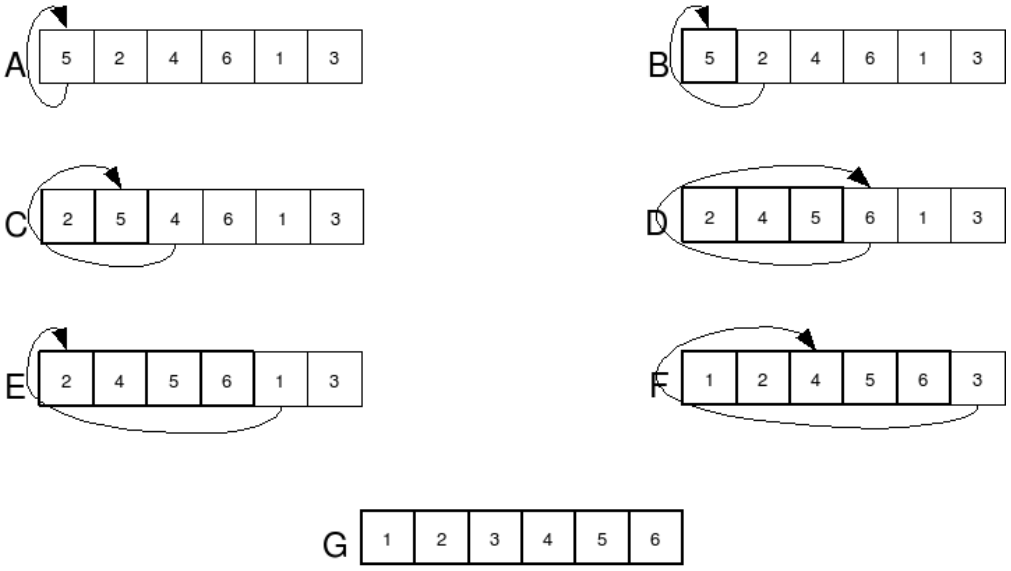

Рисунок 19 – сортировка вставками

Пузырьковая сортировка

Элементарный метод сортировки, который часто преподается на вводных курсах — это пузырьковая сортировка. Смысл сортировки в том, чтобы продолжать просматривать файл, при необходимости меняя местами соседние элементы. Когда на каком–то проходе перестановки не требуются — файл был отсортирован. Ниже представлена реализация этого метода.

```
buble (int a[], int N)
{
        int i, j, t;
        for(i = N; i >= 1; i—)
                for(j = 2; j <= i; j++)
                        if (a[j – 1] > a[j])
                        {
                                t = a[j – 1]; a[j – 1] = a[j]; a[j] = t;
                        }
        }
```

Стоит обратить внимание, что всякий раз, когда во время первого прохода встречается максимальный элемент, он меняется местами с каждым из элементов справа от него, пока не попадет в положение на правом конце массива. Затем на втором проходе второй по величине элемент будет перемещаться на свое место и т. д. Таким образом, пузырьковая сортировка работает как тип сортировки выбором, хотя она выполняет гораздо больше работы, чтобы разместить каждый элемент в нужной позиции. Визуальное представление работы данной сортировки представлены на рисунке 20. Но на рисунке представлен не вся работа сортировки, а лишь первая итерация. Мы можем видеть, как число 3 перемещается по массиву, «всплывая» до позиции, выше которой не может подняться, поскольку 2 «легче» 3.

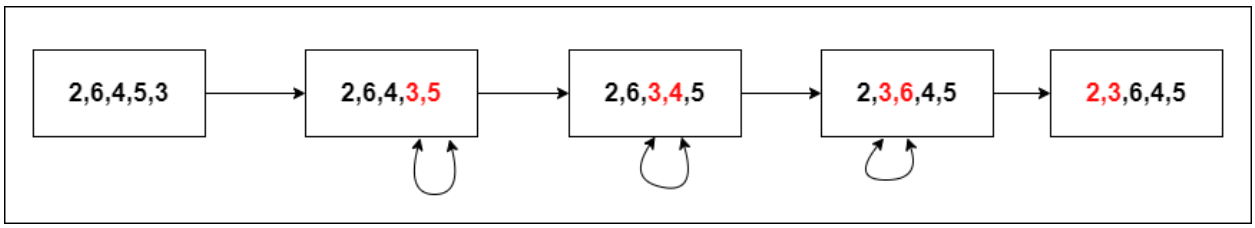

Рисунок 20 – пример работы пузырьковой сортировки

Сортировка расческой (Comb Sort)

Сортировка расческой – это улучшенная версия пузырьковой сортировки (рисунок 21). Сортировка пузырьком всегда сравнивает соседние значения, таким образом, все инверсии удаляются по одной. Сортировка расческой улучшает пузырьковую сортировку за счет использования промежутка размером более 1. Разрыв начинается с большого значения и уменьшается на каждой итерации, пока не достигнет значения 1. Таким образом, сортировка расческой удаляет более одной инверсии с учетом одной перестановки и работает лучше, чем пузырьковая сортировка. Хотя в среднем она работает лучше, чем пузырьковая сортировка, в худшем случае время работы будет равно времени сортировки пузырьком – O(n2).

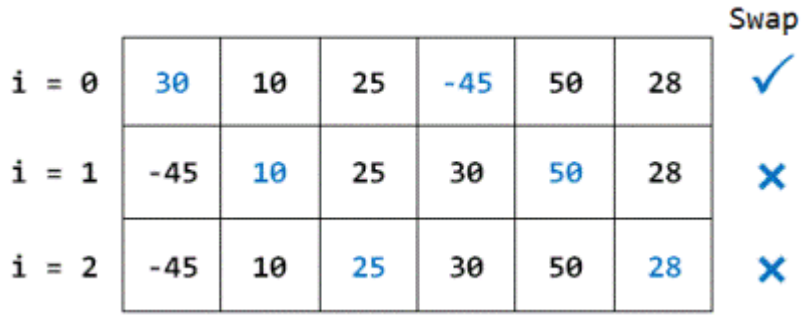

Рисунок 21 – пример сортировки расческой

Сортировка Шелла

Сортировка вставкой выполняется медленно, поскольку меняет местами только соседние элементы. Например, если наименьший элемент оказывается в конце массива, требуется N шагов, чтобы получить его на своем месте. Сортировка Шелла — это простое расширение сортировки вставкой, которое увеличивает скорость, позволяя заменять элементы, которые находятся далеко друг от друга.

Идея состоит в том, чтобы переупорядочить массив, сравнивая значения равноудаленные друг от друга на расстоянии d. После этого, алгоритм повторяется для меньших d до тех пор, пока d не будет равен 1 (аналогично классической сортировке вставками).

На рисунке 22 показана работа сортировки Шелла.

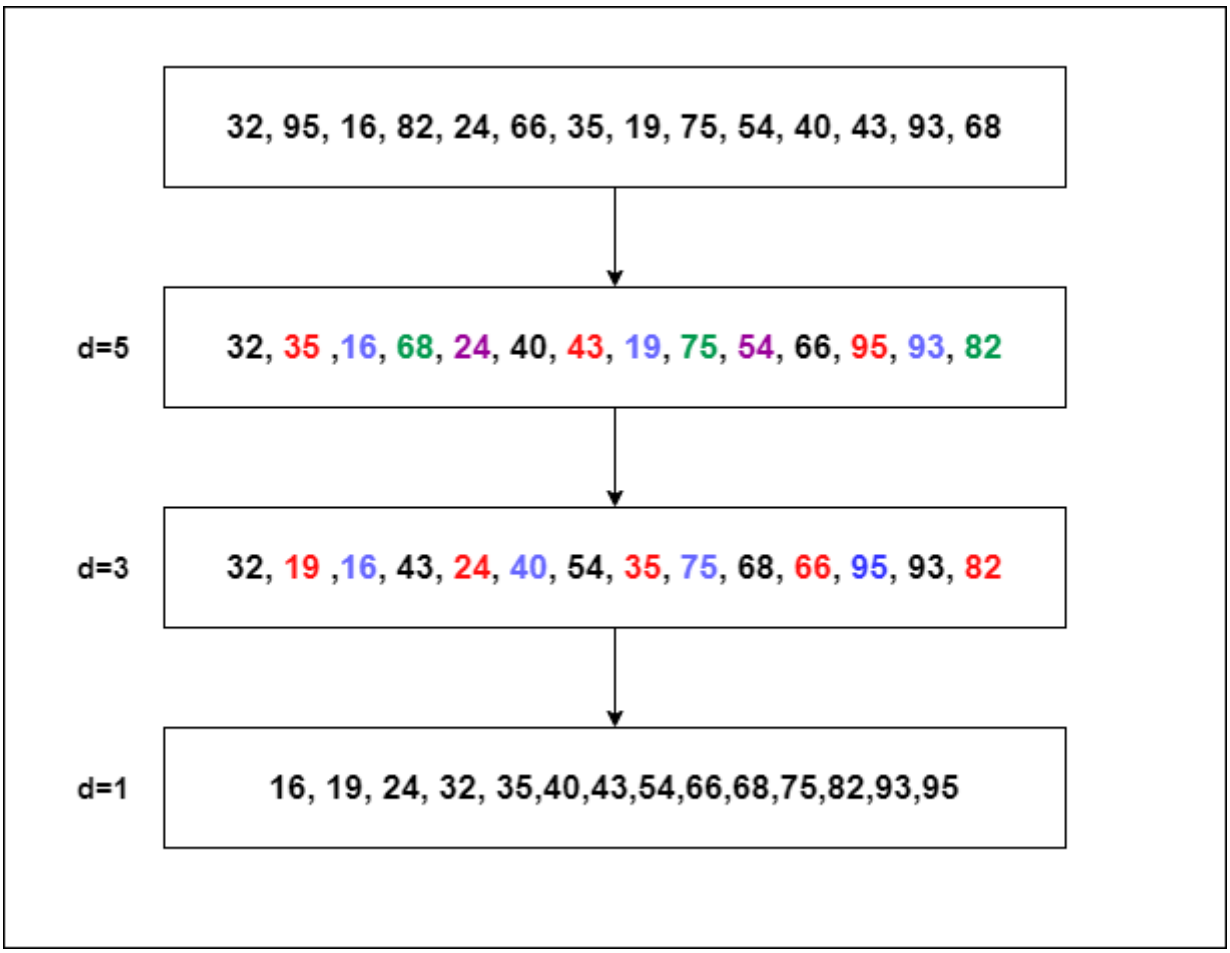

Рисунок 22 – сортировка Шелла

Ниже представлена функция, реализующая работу сортировки Шелла и результат работы данного программного кода (рисунок 23).

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <malloc.h>
#include <conio.h>
//сортировка методом Шелла
void ShellSort(int n, int mass[])
{
   int i, j, step;
   int tmp;
   for (step = n / 2; step > 0; step /= 2)
      for (i = step; i < n; i++)
      {
         tmp = mass[i];
         for (j = i; j >= step; j –= step)
         {
            if (tmp < mass[j – step])
               mass[j] = mass[j – step];
```

```
            else
                break;
        }
        mass[j] = tmp;
    }
}

int main()
{
    //ввод N
    int N;
    printf("Input N: ");
    scanf_s("%d", &N);
    //выделение памяти под массив
    int* mass;
    mass = (int*)malloc(N * sizeof(int));
    //ввод элементов массива
    printf("Input the array elements:\n");
    for (int i = 0; i < N; i++)
        scanf_s("%d", &mass[i]);
    //сортировка методом Шелла
    ShellSort(N, mass);
    //вывод отсортированного массива на экран
    printf("Sorted array:\n");
    for (int i = 0; i < N; i++)
        printf("%d ", mass[i]);
    printf("\n");
    //освобождение памяти
    free(mass);
    _getch();
    return 0;
}
```

```
Input N: 4
Input the array elements:
13 2 1 4
Sorted array:
1 2 4 13
```

Рисунок 23 – результат работы программы

Сортировка подсчетом

Сортировка подсчетом – это алгоритм сортировки, который сохраняет для каждого ключа сортировки количество записей с данным ключом сортировки (таким образом, предполагается, что ключи могут быть не уникальными). С помощью этой информации можно правильно разместить записи в отсортированном файле. Алгоритм полезен, когда ключи попадают в небольшой диапазон и многие из них равны.

Допустим, что есть задача: «отсортировать файл из N записей, ключи которых являются целыми числами от 0 до M–1». Если M не слишком велико, для решения этой проблемы можно использовать алгоритм, называемый сортировкой подсчетом. Идея состоит в том, чтобы подсчитать количество ключей с каждым значением, а затем использовать эти счетчики для перемещения записей в позицию при втором проходе через файл, как в следующем коде:

```
for (j = 0; j < M; j++) count[j] = 0;
        for (i = 1; i <= N; i++) count[a[i]]++;
        for (j = 1; j < M; j++)
                count[j] = count[j – 1] + count[j];
        for (i = N; i >= 1; i – ~)
                b[count[a[i]]—] = a[i];
        for (i = 1; i <= N; i++) a[i] = b[i];
```

Чтобы увидеть, как работает этот код, рассмотрим пример файла целых чисел в верхней строке рисунка 21. Первый цикл for инициализирует счетчики до 0; второй устанавливает count [1] = 4, count [2] = 2, count [3] = 1 и count [4] = 3, потому что есть шесть A, четыре B и т. д. Затем третий цикл for складывает эти числа, чтобы получить count [1] = 4, count [2] = 6, count [3] = 7 и count [4] = 10. То есть четыре ячейки меньше или равны A, шесть ячеек меньше или равны B и т.д. Теперь их можно использовать в качестве адресов для сортировки массива, как показано на рисунке. Исходные массив представлен на рисунке 24. Например, когда встречается A в конце файла, он помещается в ячейку 4, поскольку count [1] говорит, что есть четыре ключа меньших или равных A. Затем count [1] уменьшается, поскольку теперь на один ключ меньше или равно A.

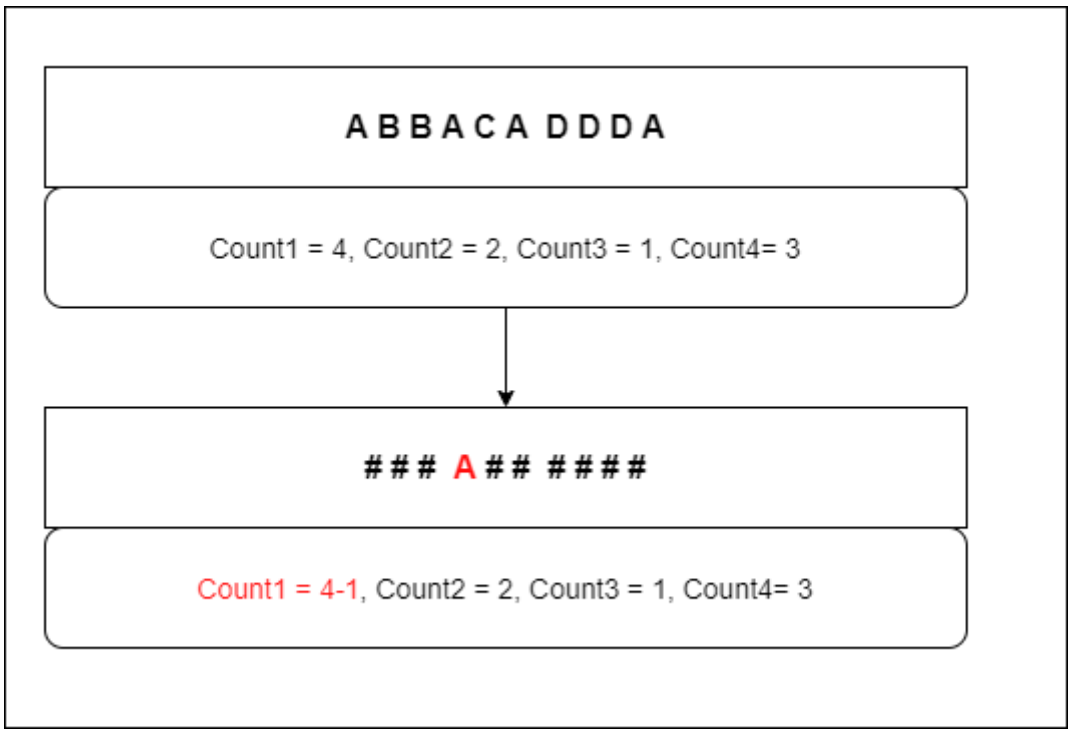

Рисунок 24 – сортировка подсчетом

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <string.h>
#define RANGE 255

// Основная функция, которая сортирует заданную строку arr [] в алфавитном порядке
void countSort(char arr[])
{
    // Выходной массив символов, который будет отсортирован arr
    char output[strlen(arr)];

    // Создауние массива–счетчика для хранения счетчика отдельных символов и инициализации
массива счетчика как 0
    int count[RANGE + 1], i;
    memset(count, 0, sizeof(count));

    // Счетчик каждого символа
    for (i = 0; arr[i]; ++i)
        ++count[arr[i]];

    // Изменение count [i] так, чтобы count [i]  содержал фактическую позицию этого символа в
выходном массиве
    for (i = 1; i <= RANGE; ++i)
        count[i] += count[i – 1];
```

```
    // Создание выходного массива символов
    for (i = 0; arr[i]; ++i) {
        output[count[arr[i]] – 1] = arr[i];
        —count[arr[i]];
    }


    for (i = 0; arr[i]; ++i)
        arr[i] = output[i];
}

int main()
{
    char arr[] = "DoYouStillReadIt"; ;

    countSort(arr);

    printf("Sorted character array is %sn", arr);
    return 0;
}
```

Двухсторонняя сортировка выбором

Двухсторонняя сортировка выбором – это версия классической сортировки выбором, которая была рассмотрена ранее. Суть сортировки в том, что если элемент не является самым маленьким элементом массива, то можно проверить, является ли он самым большим элементом, и, если это так, переместить его в конец массива. Таким образом, мы можем достичь двух целей одновременно и быстрее выполнять работу. Такое дополнение к сортировке пакетным выбором делает его несколько быстрее (но не вдвое быстрее, как можно было бы поспешно подумать, потому что объем проделанной работы на самом деле не снижается до половины, но, тем не менее, несколько падает).

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#define n 10

void swap(int* a, int* b)
{
    int tmp;
```

```
    if (*a == *b)   //на случай одинаковых значений
        return;
    tmp = *a;
    *a = *b;
    *b = tmp;
    printf("%d <–> %d\n", *a, *b);
}

main()
{
    int mas[n] = { 9,5,1,3,6,7,4,8,2,0 };
    int max, min, max_index, min_index, i, j, k;

    k = n – 1;
    for (i = 0; i <= k; i++)
    {
        max = min = mas[i];
        max_index = min_index = i;
        for (j = i + 1; j <= k; j++)
        {
            if (mas[j] > max)
            {
                max = mas[j];
                max_index = j;
            }
            if (mas[j] < min)
            {
                min = mas[j];
                min_index = j;
            }
        }
        if (max_index == i && min_index != k)   //условия для обработки пограничных значений
        {
            swap(&mas[k], &mas[max_index]);
            swap(&mas[i], &mas[min_index]);
        }
        if (min_index == k && max_index != i)
        {
            swap(&mas[i], &mas[min_index]);
            swap(&mas[k], &mas[max_index]);
        }
        if (min_index == k && max_index == i)
            swap(&mas[k], &mas[i]);
```

```
        if (min_index != k && max_index != i)
        {
            swap(&mas[k], &mas[max_index]);
            swap(&mas[i], &mas[min_index]);
        }
        k—; //сокращение границы цикла

    }
    for (i = 0; i < n; i++)
    {
        printf("%d ", mas[i]);
    }
}
```

Итог работы программы представлен на рисунке 25.

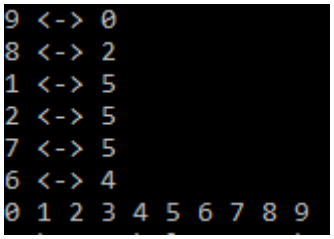

Рисунок 25 – результат работы двухсторонней сортировки выбором

Бинго сортировка

Бинго сортировка – вид сортировки выбором, рассмотренной ранее. Временная сложность данного алгоритма составляет $O(n^2)$, что делает данный тип сортировки неэффективным в сравнении с сортировкой вставками.

Алгоритм упорядочивает элементы, сначала находя наименьшее значение, затем многократно перемещая все элементы с этим значением в их окончательное местоположение и находя наименьшее значение для следующего прохода. Это более эффективно, чем сортировка по выбору, если имеется много повторяющихся значений. Описанный выше метод представлен ниже в виде псевдокода.

```
bingo(array A)

  //Эта процедура выполняет сортировку по возрастанию, многократно перемещая максимальное количество элементов в конец.
begin
last : = length(A) – 1;
```

```
// Первая итерация написана так, чтобы она выглядела очень похожей на последующие, но без перестановок.
nextMax: = A[last];
for i : = last – 1 downto 0 do
if A[i] > nextMax then
nextMax : = A[i];
while (last > 0) and (A[last] = nextMax) do
last : = last – 1;

while last > 0 do begin
prevMax : = nextMax;
nextMax: = A[last];
for i : = last – 1 downto 0 do
if A[i] > nextMax then
if A[i] <> prevMax then
nextMax : = A[i];
else begin
swap(A[i], A[last]);
last: = last – 1;
end
while (last > 0) and (A[last] = nextMax) do
last : = last – 1;
end;
        end;
```

Интересный факт, если в среднем имеется более двух элементов с одинаковым значением, можно ожидать, что сортировка бинго будет быстрее, поскольку она выполняет внутренний цикл меньшее количество раз, чем сортировка выбором.

Блинная сортировка

Вид сортировки на основе разворота, над решением которой бились такие «великие умы», как Билл Гейтс, Христос Пападимитриу. В основе сортировки лежит бытовая задача: перевернуть блины при помощи лопатки наименьшее количество раз. Подобно этому, единственная операция в алгоритме – переворот элементов последовательности до определенного индекса.

Идея состоит в том, чтобы сделать что–то похожее на сортировку по выбору. Мы один за другим помещаем в конец максимальный элемент и уменьшаем размер текущего массива на единицу.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

/* переворачивает arr[0..i] */

void flip(int arr[], int i)
{
	int temp, start = 0;
	while (start < i) {
		temp = arr[start];
		arr[start] = arr[i];
		arr[i] = temp;
		start++;
		i—;
	}
}

// Возвращает индекс максимального элемента в arr [0..n–1]
int findMax(int arr[], int n)
{
	int mi, i;
	for (mi = 0, i = 0; i < n; ++i)
		if (arr[i] > arr[mi])
			mi = i;
	return mi;
}

// Основная функция, которая сортирует заданный массив с помощью операций переворота.
void pancakeSort(int* arr, int n)
{
	// Начинаем с полного массива и один за другим уменьшаем текущий размер на
единицу
	for (int curr_size = n; curr_size > 1;
		—curr_size)
	{
		// Находим индекс максимального элемента в arr [0..curr_size–1]

		int mi = findMax(arr, curr_size);

// Перемещаем максимальный элемент в конец текущего массива, если он еще не в конце

		if (mi != curr_size – 1) {
```

```
// Чтобы перейти в конец, сначала переместите максимальное число в начало
                        flip(arr, mi);

// Теперь переместите максимальное число в конец, изменив текущий массив

                        flip(arr, curr_size – 1);
                }
        }
}

// Специальная функция для печати массива n размером n
void printArray(int arr[], int n)
{
        for (int i = 0; i < n; ++i)
                printf("%d ", arr[i]);
}

//основная функция
int main()
{
        int arr[] = { 23, 10, 20, 11, 12, 6, 7 };
        int n = sizeof(arr) / sizeof(arr[0]);

        pancakeSort(arr, n);

        puts("Sorted Array ");
        printArray(arr, n);

        return 0;
        }
```

Проверьте данный код в программной среде, получив результат, представленный на рисунке 26.

```
Sorted Array
6 7 10 11 12 20 23
```

Рисунок 26 – результат работы блинной сортировки

Пирамидальная сортировка

Пирамидальная сортировка или сортировка кучи – это метод сортировки на основе сравнения, основанный на структуре данных сортирующего дерева (двоичной кучи). Это похоже на сортировку по выбору, где мы сначала находим минимальный элемент и помещаем минимальный элемент в начало. Повторяем тот же процесс для остальных элементов.

Что такое двоичная куча? Давайте сначала дадим определение полному двоичному дереву. Полное двоичное дерево — это двоичное дерево, в котором каждый уровень, кроме, возможно, последнего, полностью заполнен, а все узлы расположены как можно дальше слева.

Двоичная куча – это полное двоичное дерево, в котором элементы хранятся в особом порядке, так что значение в родительском узле больше (или меньше), чем значения в двух его дочерних узлах. Первый называется максимальной кучей, а второй – минимальной кучей. Куча может быть представлена двоичным деревом или массивом (рисунок 27).

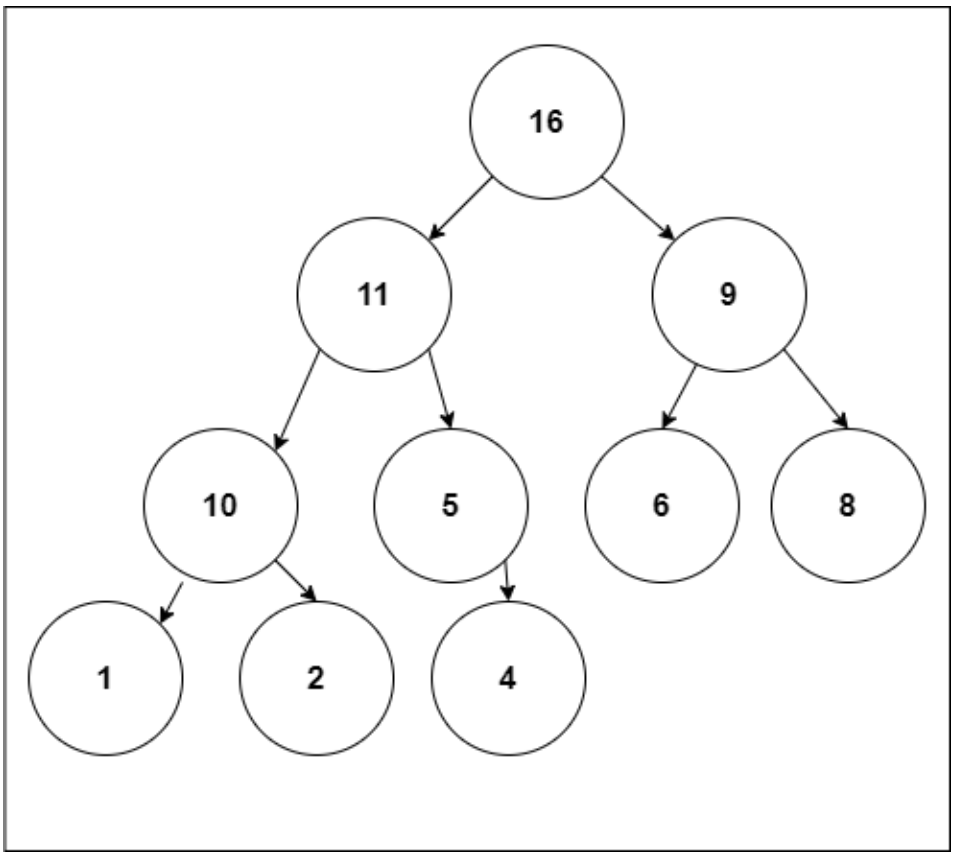

Рисунок 27 – двоичная куча

Поскольку двоичная куча является полным двоичным деревом, ее можно легко представить в виде массива, а представление на основе массива занимает

мало места. Если родительский узел хранится с индексом I, левый дочерний элемент может быть вычислен как 2 * I + 1, а правый дочерний элемент – как 2 * I + 2 (при условии, что индексирование начинается с 0).

Алгоритм сортировки в куче для сортировки в порядке возрастания:

1. Создайте максимальную кучу из входных данных.

2. На этом этапе самый большой элемент хранится в корне кучи. Замените его последним элементом кучи, а затем уменьшите размер кучи на 1. Наконец, скопируйте корень дерева.

3. Повторите шаг 2, пока размер кучи больше 1.

На рисунке 28 дерево уже собрано таким образом, что оно является восходящим.

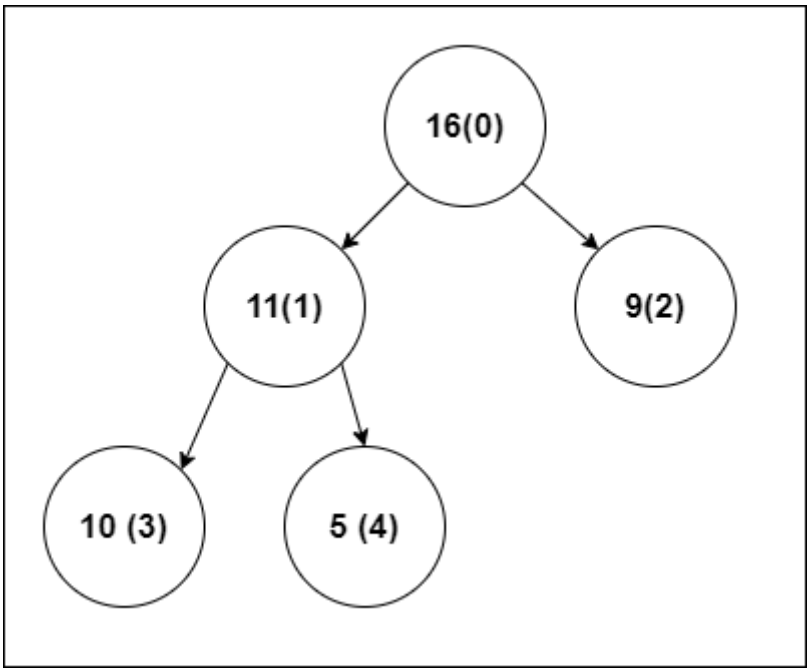

Рисунок 28 – двоичная куча для примера

Как данная концепция работает в коде? Для начала, разберемся с функцией heapify.

```
void heapify(int arr[], int n, int i)
{
   int largest = i;
   int l = 2 * i + 1;
   int r = 2 * i + 2;
```

```
if (l < n && arr[l] > arr[largest])
    largest = l;

if (right < n && arr[r] > arr[largest])
    largest = r;

if (largest != i)
{
    swap(arr[i], arr[largest]);

    // Рекурсия heapify к дереву
    heapify(arr, n, largest);
}
        }
```

Представьте себе дерево, состоящее из трех элементов (рисунок 29).

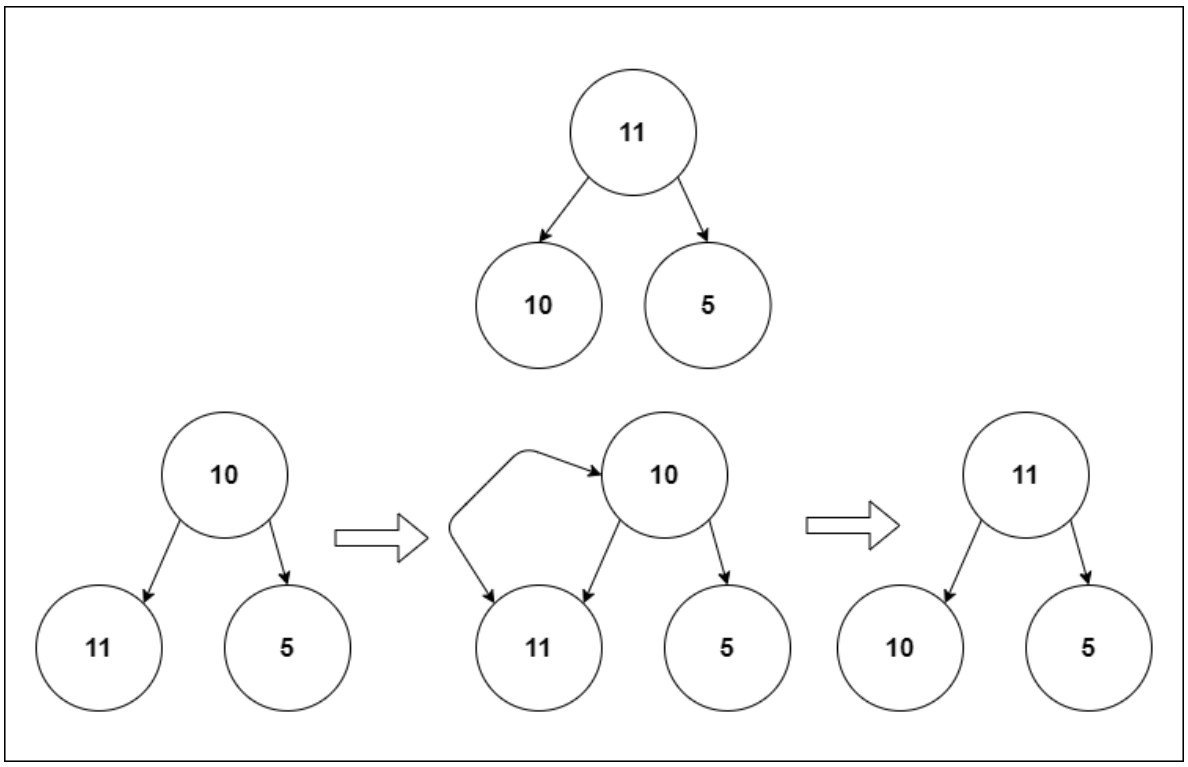

Рисунок 29 – сценарии выстраивания дерева

На рисунке 29 представлено 2 возможных сценария: в первом, родитель (корень) является наибольшим элементом и никаких действий не требуется. Во втором сценарии, чтобы сохранить свойство убывающей кучи нам пришлось поменять элементы местами. Но что, если уровней в дереве больше и корневой элемент не является наибольшим? Тогда функцию heapify придется запускать многократно, пока корневой элемент не станет самым большим или он не станет листовым узлом.

На такой логике и строится пирамидальная сортировка. Код, готовый к изучению студентами представлен ниже.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

// Куча имеет текущий размер и массив элементов
struct MaxHeap
{
   int size;
   int* array;
};


// Служебная функция для переключения на целые числа
void swap(int* a, int* b) { int t = *a; *a = *b;  *b = t; }

// Основная функция для заполнения максимальной кучи

void maxHeapify(struct MaxHeap* maxHeap, int idx)
{
   int largest = idx;  // Initialize largest as root
   int left = (idx << 1) + 1;  // left = 2*idx + 1
   int right = (idx + 1) << 1; // right = 2*idx + 2

   //Существует ли левый дочерний элемент root и сравните с root
   if (left < maxHeap->size &&
      maxHeap->array[left] > maxHeap->array[largest])
      largest = left;

   // Существует ли правый дочерний элемент root и сравните с root
   if (right < maxHeap->size &&
      maxHeap->array[right] > maxHeap->array[largest])
      largest = right;

   // изменить root, если необходимо
   if (largest != idx)
   {
      swap(&maxHeap->array[largest], &maxHeap->array[idx]);
      maxHeapify(maxHeap, largest);
   }
}

// Служебная функция для создания максимальной кучи заданного размера
struct MaxHeap* createAndBuildHeap(int* array, int size)
{
```

```
    int i;
    struct MaxHeap* maxHeap =
        (struct MaxHeap*)malloc(sizeof(struct MaxHeap));
    maxHeap–>size = size;   // инициализировать размер кучи
    maxHeap–>array = array; // Назначить адрес первого элемента массива

    for (i = (maxHeap–>size – 2) / 2; i >= 0; —i)
        maxHeapify(maxHeap, i);
    return maxHeap;
}

// Основная функция для сортировки массива заданного размера
void heapSort(int* array, int size)
{
    // Создание кучи из входных данных.
    struct MaxHeap* maxHeap = createAndBuildHeap(array, size);

    // Повторите следующие шаги, пока размер кучи больше 1.
    // Последний элемент в максимальной куче будет минимальным элементом
    while (maxHeap–>size > 1)
    {
        // Самый большой элемент в куче хранится в корне.Замените его последним элементом кучи,
        // а затем уменьшите размер кучи на 1.
        swap(&maxHeap–>array[0], &maxHeap–>array[maxHeap–>size – 1]);
        —maxHeap–>size;  // Уменьшить размер кучи

        // корень дерева
        maxHeapify(maxHeap, 0);
    }
}

// Служебная функция для печати заданного массива заданного размера
void printArray(int* arr, int size)
{
    int i;
    for (i = 0; i < size; ++i)
        printf("%d ", arr[i]);
}

int main()
{
    int arr[] = { 10, 5, 11, 16, 9 };
    int size = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, size);

    printf("\nSorted array is \n");
```

```
printArray(arr, size);
return 0;
    }
```

Плавная сортировка

Плавная сортировка похожа на сортировку по куче с одним ключевым отличием. Вместо использования двоичной кучи в плавной сортировке используется куча Леонардо. Прежде чем мы погрузимся в работу Леонардо, мы должны поговорить о числах Леонардо. Числа Леонардо – это числа, удовлетворяющие следующей последовательности.

- L(0) = 1
- L(1) = 1
- L(n) = L(n–1) + L(n–2) + 1

Для примера, первые числа Леонардо: 1, 1, 3, 5, 9, 15, 25, 41, 67, 109, 177, 287, 465, 753 и т.д.

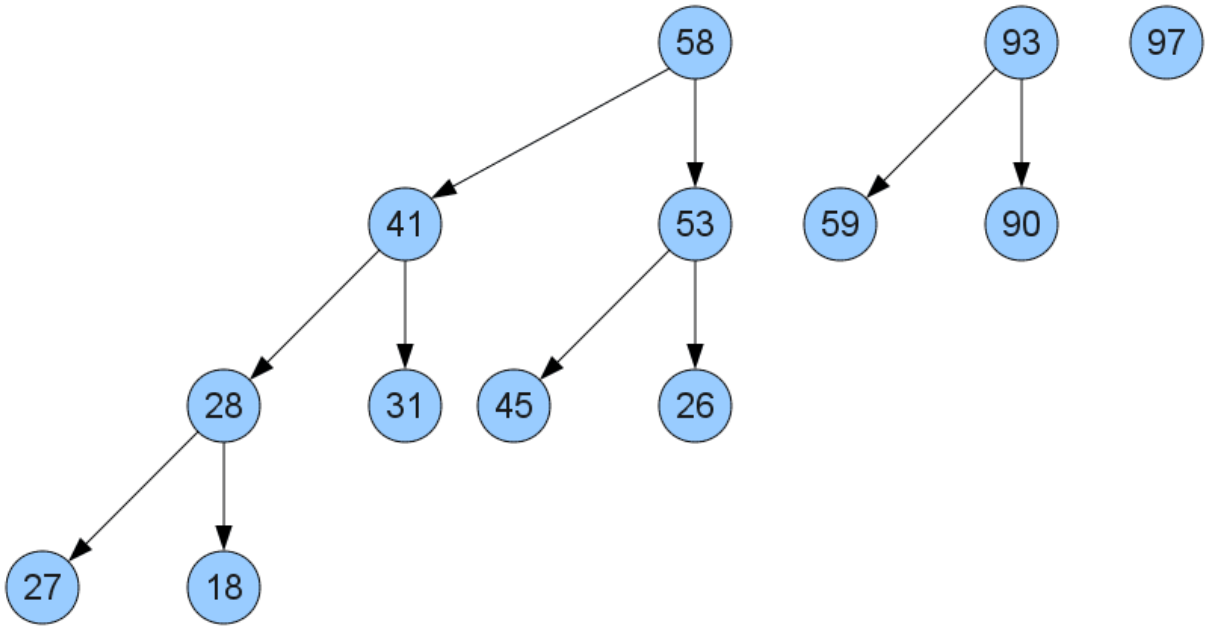

Рисунок 30 – Плавная сортировка

Как видно из рисунка 30, корневой узел в каждом дереве имеет наибольшее значение.

Абсолютно любое целое число можно представить в виде суммы чисел Леонардо, имеющих разные порядковые номера. При добавлении и удалении значений из кучи Леонардо мы должны поддерживать свойства кучи: в

частности, что она должна состоять из пронумерованного Леонардо набора двоичных деревьев.

Когда мы удаляем значения из кучи, корневой узел смещается и становится следующим по величине значением в куче. Поэтому, если мы хотим получить список, отсортированный в порядке убывания, мы удаляем значения из кучи до тех пор, пока не будут удалены все значения. Важно помнить:

1. Каждая Леонардова куча представляет собой несбалансированное бинарное дерево.

2. Корень каждой кучи — это последний (а не первый, как в обычной бинарной куче) элемент соответствующего подмассива.

3. Любой узел со всеми своими потомками также представляет из себя Леонардову кучу меньшего порядка.

Отдельного упоминания достойна рекуррентная форма для чисел Леонардо **Ln = Ln − 1 + Ln − 2 + 1.** Предположим, что в нашем массиве есть два соседних подмассива, соответствующие кучам, построенных на двух соседствующих числах Леонардо. Далее эти кучи можно объединить в одну общую, что будет соответствовать следующему числу Леонардо (рисунок 31).

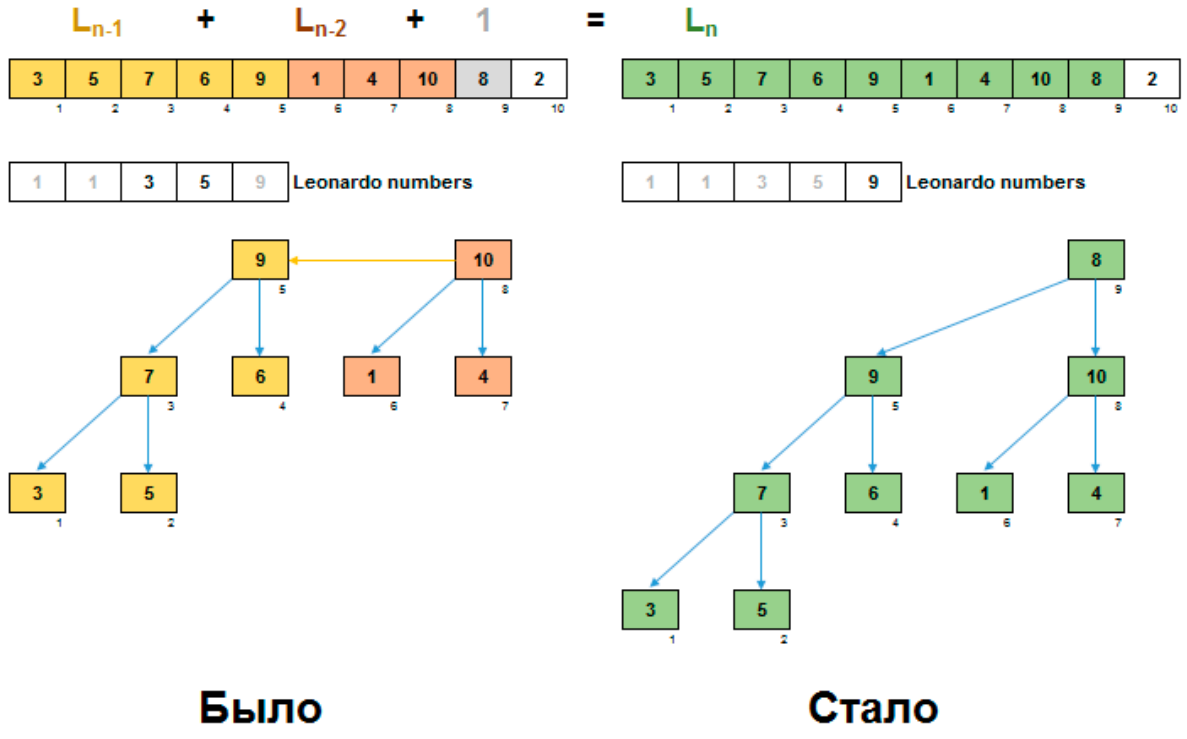

Рисунок 31 – построение кучи

Перебирая элементы массива мы выстраиваем кучу из Леонардовых куч. Если с помощью элемента можно объединить две предыдущие кучи (это возможно только тогда, когда две предыдущие кучи соответствуют двум последовательным числам Леонардо), то они объединяются. Если объединение невозможно (две предыдущие кучи не соответствуют двум последовательным числам Леонардо), то текущий элемент просто образует новую кучу из одного элемента, соответствующую первому (или второму, если первое использовано перед ним) числу Леонардо.

На втором этапе алгоритма происходит обратный процесс — разбор куч. Если в куче удалить корень, то мы получим две кучи поменьше, соответствующие двум предыдущим числам Леонардо. Это можно сделать, поскольку: **Ln = Ln − 1 + Ln − 2 + 1**. Из–за отсутствия такой единички в числах Фибоначчи невозможно использовать кучу Фибоначчи для работы с данным типом сортировки.

Поразрядная (радиксная) сортировка

Поразрядная сортировка – это алгоритм сортировки, который сортирует элементы, сначала группируя отдельные цифры одного и того же разряда, а затем отсортировывает элементы в соответствии с их порядком увеличения или уменьшения.

Допустим, у нас есть массив из 8 элементов. Сначала мы отсортируем элементы в зависимости от значения единиц числа. Затем мы отсортируем элементы по значению десятков. Этот процесс продолжается до последнего значимого разряда.

Допустим, что исходный массив [121, 432, 564, 23, 1, 45, 788]. Он сортируется по основанию системы счисления, как показано на рисунке 32.

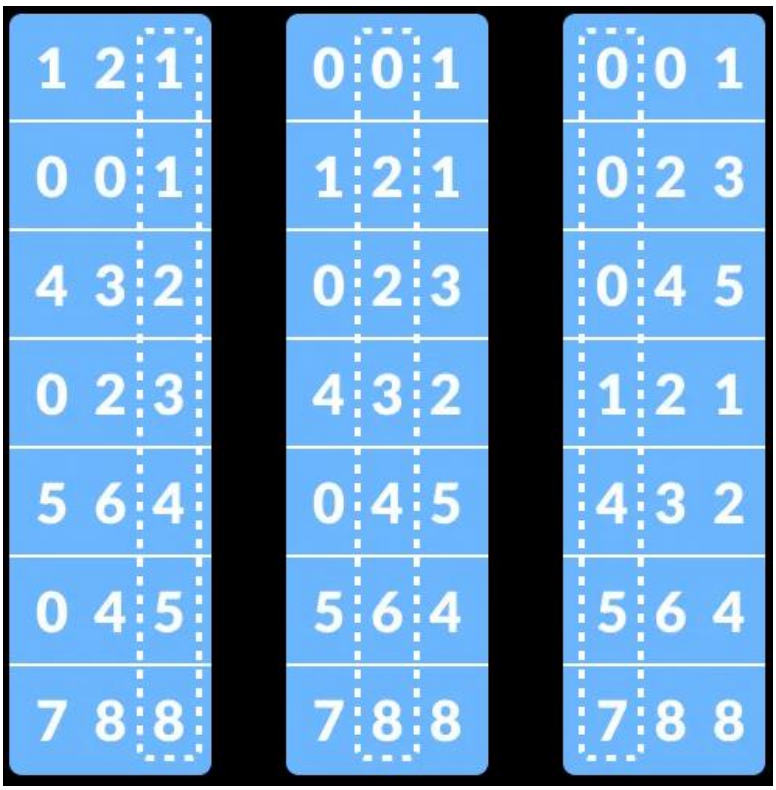

Рисунок 32 – поразрядная сортировка

Как же работает поразрядная сортировка? Найдите самый большой элемент в массиве, т.е. максимум. Пусть X будет количеством цифр в максимуме. В этом массиве [121, 432, 564, 23, 1, 45, 788] у нас есть наибольшее число – 788. Оно состоит из 3 цифр (X=3). Следовательно, петля должна доходить до сотен разряда (3 раза). Далее мы сравниваем поразрядно единицы, десятки и сотни каждого числа, постепенно переставляя их в массиве. Реализация в коде с комментариями представлена ниже.

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>

// Функция для получения самого большого элемента из массива
int getMax(int array[], int n) {
   int max = array[0];
   for (int i = 1; i < n; i++)
      if (array[i] > max)
         max = array[i];
   return max;
}

// Использование счетной сортировки для сортировки элементов по значимым местам
void countingSort(int array[],  int size, int place) {
    int output[size + 1];
   int max = (array[0] / place) % 10;
```

```
    for (int i = 1; i < size; i++) {
        if (((array[i] / place) % 10) > max)
            max = array[i];
    }
    int count[max + 1];

    for (int i = 0; i < max; ++i)
        count[i] = 0;

    // Рассчитать количество элементов
    for (int i = 0; i < size; i++)
        count[(array[i] / place) % 10]++;

    // Рассчитать совокупное количество
    for (int i = 1; i < 10; i++)
        count[i] += count[i – 1];

    // Разместите элементы в отсортированном порядке
    for (int i = size – 1; i >= 0; i—) {
        output[count[(array[i] / place) % 10] – 1] = array[i];
        count[(array[i] / place) % 10]—;
    }

    for (int i = 0; i < size; i++)
        array[i] = output[i];
}


//Основная функция для реализации поразрядной сортировки
void radixsort(int array[], int size) {
    //  максимум
    int max = getMax(array, size);

    // Примените сортировку с подсчетом для сортировки элементов по разряду.
    for (int place = 1; max / place > 0; place *= 10)
        countingSort(array, size, place);
}

// Распечатать массив
void printArray(int array[], int size) {
    for (int i = 0; i < size; ++i) {
        printf("%d  ", array[i]);
    }
    printf("\n");
}
```

```
int main() {
   int array[] = { 121, 432, 564, 23, 1, 45, 788 };
   int n = sizeof(array) / sizeof(array[0]);
   radixsort(array, n);
   printArray(array, n);
         }
```

Поскольку поразрядная сортировка не является сравнительным алгоритмом, она имеет преимущества перед алгоритмами сравнительной сортировки.

Для поразрядной сортировки, которая использует сортировку с подсчетом в качестве промежуточной стабильной сортировки, временная сложность составляет O (d (n + k)). Здесь d – это числовой цикл, а O (n + k) – временная сложность подсчета сортировки.

Таким образом, поразрядная сортировка имеет линейную временную сложность, которая лучше, чем O (nlog n) алгоритмов сравнительной сортировки.

Если мы возьмем очень большие числовые числа или количество других оснований, таких как 32–битные и 64–битные числа, тогда он может работать в линейном времени, однако промежуточная сортировка занимает много места.

Это делает пространство сортировки по системе счисления неэффективным. Это причина, по которой этот вид не используется в программных библиотеках.

Блочная сортировка (карманная сортировка)

Блочная сортировка – это алгоритм сортировки, который разделяет несортированные элементы массива на несколько групп, называемых блоками (в разных источниках можно встретить разные название, в т.ч. «карманы», «корзины», «ведра», «блоки» и т.д.). Затем каждый блок сортируется с использованием любого из подходящих алгоритмов сортировки или рекурсивного применения того же алгоритма сегмента. После отсортированные сегменты объединяются, чтобы сформировать окончательный отсортированный массив.

Допустим, что наш исходный массив {0.42, 0.32, 0.23, 0.52, 0.25, 0.47, 0.51}. Создадим еще один массив размером 10. Каждый слот этого массива используется как корзина для хранения элементов. Вставьте элементы в корзины из массива. Элементы вставляются в соответствии с диапазоном кармана. В нашем примере кода у нас есть сегменты, каждый из которых находится в диапазоне от 0 до 1, от 1 до 2, от 2 до 3, ... (n–1) до n.

Предположим, что на входе взят элемент 0,23. Он умножается на size = 10 (т.е. 0,23 * 10 = 2,3). Затем оно преобразуется в целое число (например, 2,3≈2). Наконец, 0.23 вставляется в блок 2 (рисунок 33).

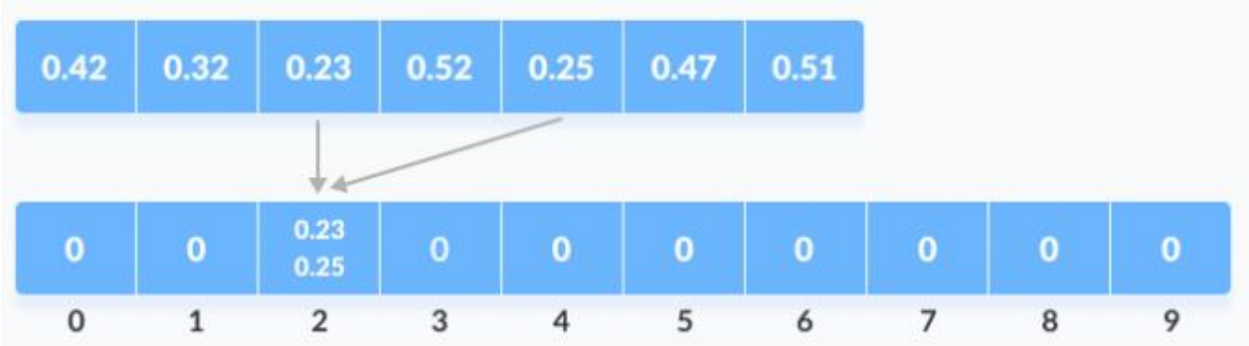

Рисунок 33 – вставка элементов массива в «карманы»

Точно так же в этот блок вставляется и 0,25. Каждый раз берется минимальное значение числа с плавающей запятой. Если мы возьмем на вход целые числа, мы должны разделить их на интервал (здесь 10), чтобы получить минимальное значение. Аналогичным образом другие элементы вставляются в соответствующие корзины.

Элементы каждой корзины сортируются с использованием любого из стабильных алгоритмов сортировки (например, быстрая сортировка). Затем элементы из каждого блока собираются в исходный массив. Это делается путем

итерации по корзине и вставки отдельного элемента в исходный массив в каждом цикле. Элемент из корзины удаляется после копирования в исходный массив (рисунок 34). Ниже также представлен вариант кода данной сортировки и результат работы (рисунок 35).

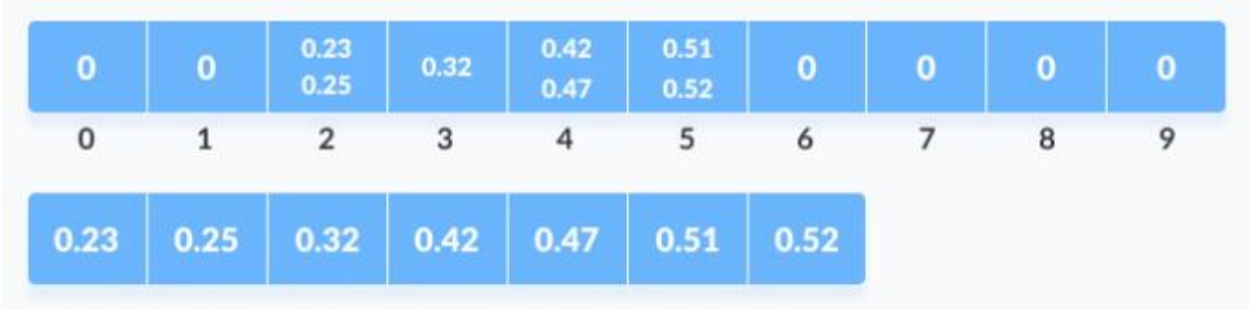

Рисунок 34 – сбор элементов в исходный массив

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>

#define NARRAY 7   // Размер массива
#define NBUCKET 6  // Набор блоков–"карманов"
#define INTERVAL 10 // "емкость" каждого блока

struct Node {
   int data;
   struct Node* next;
};

void BucketSort(int arr[]);
struct Node* InsertionSort(struct Node* list);
void print(int arr[]);
void printBuckets(struct Node* list);
int getBucketIndex(int value);

// функция сортировки
void BucketSort(int arr[]) {
   int i, j;
   struct Node** buckets;

   // Создавание корзин и выделение памяти
   buckets = (struct Node**)malloc(sizeof(struct Node*) * NBUCKET);

   // Инициализация пустых корзин
   for (i = 0; i < NBUCKET; ++i) {
      buckets[i] = NULL;
```

```
    }

    // Заполнение корзин соответствующими элементами
    for (i = 0; i < NARRAY; ++i) {
        struct Node* current;
        int pos = getBucketIndex(arr[i]);
        current = (struct Node*)malloc(sizeof(struct Node));
        current–>data = arr[i];
        current–>next = buckets[pos];
        buckets[pos] = current;
    }

    // Печать корзин с элементами
    for (i = 0; i < NBUCKET; i++) {
        printf("Bucket[%d]: ", i);
        printBuckets(buckets[i]);
        printf("\n");
    }

    // Сортировка элементов в корзинах
    for (i = 0; i < NBUCKET; ++i) {
        buckets[i] = InsertionSort(buckets[i]);
    }

    printf("————————\n");
    printf("Bucktets after sorting\n");
    for (i = 0; i < NBUCKET; i++) {
        printf("Bucket[%d]: ", i);
        printBuckets(buckets[i]);
        printf("\n");
    }

    // Возвращение отсортированных элементов в массив
    for (j = 0, i = 0; i < NBUCKET; ++i) {
        struct Node* node;
        node = buckets[i];
        while (node) {
            arr[j++] = node–>data;
            node = node–>next;
        }
    }

    return;
}

// Функция сортировки в каждой корзине
struct Node* InsertionSort(struct Node* list) {
```

```
    struct Node* k, * nodeList;
    if (list == 0 || list->next == 0) {
        return list;
    }

    nodeList = list;
    k = list->next;
    nodeList->next = 0;
    while (k != 0) {
        struct Node* ptr;
        if (nodeList->data > k->data) {
            struct Node* tmp;
            tmp = k;
            k = k->next;
            tmp->next = nodeList;
            nodeList = tmp;
            continue;
        }

        for (ptr = nodeList; ptr->next != 0; ptr = ptr->next) {
            if (ptr->next->data > k->data)
                break;
        }

        if (ptr->next != 0) {
            struct Node* tmp;
            tmp = k;
            k = k->next;
            tmp->next = ptr->next;
            ptr->next = tmp;
            continue;
        }
        else {
            ptr->next = k;
            k = k->next;
            ptr->next->next = 0;
            continue;
        }
    }
    return nodeList;
}

int getBucketIndex(int value) {
    return value / INTERVAL;
}

void print(int ar[]) {
```

```
    int i;
    for (i = 0; i < NARRAY; ++i) {
        printf("%d ", ar[i]);
    }
    printf("\n");
}

// Печать блоков
void printBuckets(struct Node* list) {
    struct Node* cur = list;
    while (cur) {
        printf("%d ", cur->data);
        cur = cur->next;
    }
}

int main(void) {
    int array[NARRAY] = { 42, 32, 33, 52, 37, 47, 51 };

    printf("Initial array: ");
    print(array);
    printf("————————————\n");

    BucketSort(array);
    printf("————————————\n");
    printf("Sorted array: ");
    print(array);
    return 0;
}
```

```
Initial array: 42 32 33 52 37 47 51
------------
Bucket[0]:
Bucket[1]:
Bucket[2]:
Bucket[3]: 37 33 32
Bucket[4]: 47 42
Bucket[5]: 51 52
------------
Bucktets after sorting
Bucket[0]:
Bucket[1]:
Bucket[2]:
Bucket[3]: 32 33 37
Bucket[4]: 42 47
Bucket[5]: 51 52
------------
Sorted array: 32 33 37 42 47 51 52
```

Рисунок 35 – результат работы корзинной сортировки

В следующих разделах мы поговорим о различных спецефических алгоритмах, позволяющих изящно решать необычные задачи.

Жадные алгоритмы

Жадные алгоритмы – это алгоритмическая парадигма, которая строит решение по частям, всегда выбирая следующую часть, которая предлагает наиболее очевидную и немедленную выгоду. Таким образом, проблемы, в которых выбор локально оптимального решения также приводит к глобальному решению, лучше всего подходят для «жадности».

Например, рассмотрим задачу о дробном ранце (рисунок 36). Оптимальная локальная стратегия состоит в том, чтобы выбрать предмет, который имеет максимальное соотношение цены и веса. Эта стратегия также приводит к глобальному оптимальному решению, потому что мы позволяем брать доли элемента.

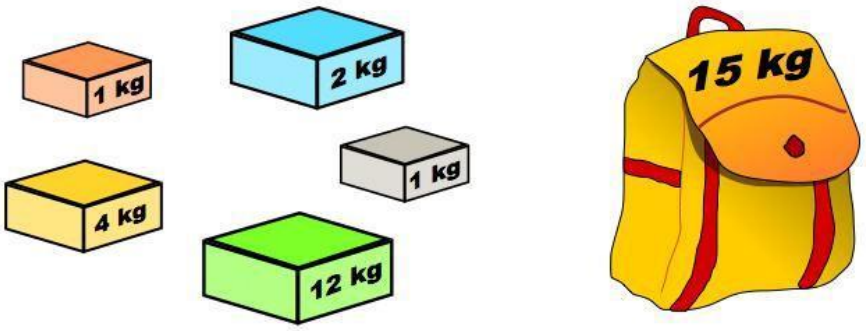

Рисунок 36 – иллюстрация задачи дробного ранца

Если жадный алгоритм может решить проблему, то он обычно становится лучшим методом для решения этой проблемы, поскольку жадные алгоритмы в целом более эффективны, чем другие методы, такие как динамическое программирование. Но жадные алгоритмы не всегда применимы.

Ниже приведены некоторые стандартные алгоритмы, которые являются жадными алгоритмами.

1. **Алгоритм минимального связующего дерева Краскала**

В алгоритме Крускала мы создаем минимальное связующее дерево, выбирая ребра одну за другой. Жадный выбор состоит в том, чтобы выбрать край с наименьшим весом, который не вызывает цикла в построенном на данный момент минимальном связующим деревом. А что такое минимальное связующее дерево? Для связного и неориентированного графа остовное дерево этого графа является подграфом, который является деревом и соединяет все вершины вместе. У одного графа может быть много разных остовных деревьев. Минимальное остовное дерево (MST) или остовное дерево с минимальным

весом для взвешенного связного неориентированного графа – это остовное дерево с весом, меньшим или равным весу любого другого остовного дерева. Вес остовного дерева – это сумма весов, присвоенных каждому краю остовного дерева. Сколько ребер имеет минимальное остовное дерево? Минимальное остовное дерево имеет (V-1) ребер, где V – количество вершин в данном графе.

2. **Минимальное связующее дерево Прима**

В алгоритме Прима мы также создаем минимальное связующее дерево, выбирая ребра одно за другим. Мы поддерживаем два набора: набор вершин, уже включенных в дерево, и набор вершин, еще не включенных. Жадный выбор – выбрать край с наименьшим весом, соединяющий два набора.

3. **Кратчайший путь Дейкстры**

Алгоритм Дейкстры очень похож на алгоритм Прима. Дерево кратчайших путей строится, ребро за ребром. Мы поддерживаем два набора: набор вершин, уже включенных в дерево, и набор вершин, которые еще не включены. Жадный выбор состоит в том, чтобы выбрать ребро, которое соединяет два набора и находится на пути наименьшего веса от источника к набору, который содержит еще не включенные вершины.

4. **Кодирование Хаффмана**

Кодирование Хаффмана – это метод сжатия без потерь. Он назначает битовые коды переменной длины различным символам. Жадный выбор – присвоить код с наименьшей длиной в битах наиболее часто встречающемуся символу.

Жадные алгоритмы иногда также используются, чтобы получить приближение для задач жесткой оптимизации. Например, задача коммивояжера – это NP-сложная задача. Жадный выбор для этой проблемы – выбирать ближайший не посещаемый город из текущего города на каждом этапе. Эти решения не всегда дают наилучшее оптимальное решение, но могут использоваться для получения приблизительно оптимального решения.

Давайте рассмотрим задачу выбора действий как наш первый пример жадных алгоритмов. Ниже приводится постановка проблемы.

Вам дается n действий с указанием времени их начала и окончания. Выберите максимальное количество действий, которое может выполнять один

человек, предполагая, что человек может работать только над одним действием за раз.

Пример 1. Рассмотрим следующие 3 вида деятельности, отсортированные по времени окончания: start[] = {10, 12, 20}; finish[] = {20, 25, 30}; человек может выполнять не более двух видов деятельности. Максимальный набор действий, которые можно выполнить: {0, 2} [это индексы в start [] и finish []].

Пример 2. Рассмотрим следующие 6 видов деятельности, отсортированные по времени окончания: start[] = {1, 3, 0, 5, 8, 5}; finish[] = { 2, 4, 6, 7, 9, 9}; человек может выполнять не более четырех видов деятельности. Максимальный набор действий, которые можно выполнить: { 0, 1, 3, 4} [это индексы в start [] и finish []].

Жадный выбор – всегда выбирать следующее действие, время окончания которого меньше всего среди оставшихся действий, а время начала больше или равно времени окончания ранее выбранного действия. Мы можем отсортировать действия по времени их окончания, чтобы мы всегда считали следующее действие минимальным временем завершения:

1) Отсортируйте активности по времени их окончания.
2) Выберите первое действие из отсортированного массива и напечатайте его.
3) Выполните следующие действия для остальных активностей в отсортированном массиве.

   а) Если время начала этого действия больше или равно времени окончания ранее выбранного действия, выберите это действие и напечатайте его.

В следующей реализации ***C*** предполагается, что действия уже отсортированы по времени их окончания.

```
#define _CRT_SECURE_NO_WARNINGS
#include<stdio.h>

// // Печатает максимальный набор активностей, которые может выполнять один человек
// n --> Общее количество активностей
// s[] --> Массив, содержащий время начала всех активностей
// f[] --> Массив, содержащий время окончания всех активностей
void printMaxActivities(int s[], int f[], int n)
{
	int i, j;
```

```
	printf("Following activities are selected n ");

	// Всегда выбирается первое действие
	i = 0;
	printf("%d ", i);

	// Рассмотрим остальные активностей
	for (j = 1; j < n; j++)
	{
		// Если время начала этой активности больше или равно времени окончания ранее выбранного => выберем его
		if (s[j] >= f[i])
		{
			printf("%d ", j);
			i = j;
		}
	}
}

int main()
{
	int s[] = { 1, 3, 0, 5, 8, 5 };
	int f[] = { 2, 4, 6, 7, 9, 9 };
	int n = sizeof(s) / sizeof(s[0]);
	printMaxActivities(s, f, n);
	return 0;
}
```

В результате, будут выбраны следующие активности: 0,1,3,4.

Поиск с возвратом, бэктрекинг

Поиск с возвратом (на английском «backtracking») – это алгоритмическая парадигма, которая пробует разные решения, пока не находит решение, которое «работает». Проблемы, которые обычно решаются с помощью метода поиска с возвратом, имеют следующее общее свойство. Эти проблемы можно решить, только попробовав **все** возможные конфигурации, и каждая конфигурация будет проверена только один раз. Простое решение этих проблем – попробовать все конфигурации и вывести конфигурацию, которая соответствует заданным ограничениям проблемы. Поиск с возвратом работает поэтапно и является оптимизацией решения «грубого перебора», в котором создаются и опробуются все возможные конфигурации.

Например, рассмотрим следующую задачу «Ход конем». Дана доска N * N с конем, размещенным на первом блоке пустой доски. Двигаясь по правилам шахмат, конь должен посетить каждую клетку ровно один раз. Выведите порядок каждой ячейки, в которой они были посещены (рисунок 37).

*Рисунок 37 – решение задачи «ход конем» для доски 5*5*

Допустим, N =8, а конь стоит в левом верхнем углу. Как коню пройти все клетки? Ответ на рисунке 38. Номера «клеток» – последовательность хода коня.

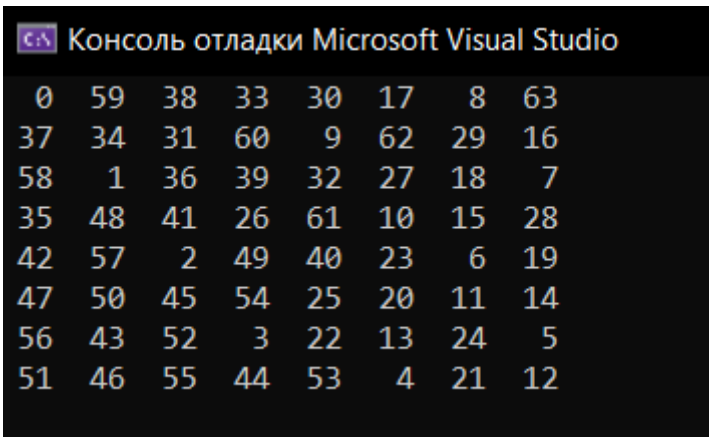

*Рисунок 38 – «ход конем» для поля 8*8*

Давайте сначала обсудим наивный алгоритм для решения этой проблемы, а затем алгоритм поиска с возвратом.

Наивный алгоритм состоит в том, чтобы генерировать все туры один за другим и проверять, удовлетворяет ли созданное решение ограничениям.

```
while есть непроверенные решения
{
   generate следующие решение
   if это решение охватывает все площади
   {
     printf(этот путь);
   }
         }
```

Обратный поиск работает поэтапно для решения проблем. Как правило, мы начинаем с пустого вектора решения и добавляем элементы один за другим (значение элемента варьируется от задачи к проблеме. В контексте задачи «Ход конем» элемент – это ход коня). Когда мы добавляем элемент, мы проверяем, нарушает ли добавление текущего элемента ограничение проблемы, если да, то удаляем элемент и пробуем другие альтернативы. Если ни одна из альтернатив не работает, мы переходим к предыдущему этапу и удаляем элемент, добавленный на предыдущем этапе. Если мы вернемся к начальному этапу, то мы скажем, что решения не существует. Если добавление элемента не нарушает ограничений, мы рекурсивно добавляем элементы один за другим. Если вектор решения становится полным, мы печатаем решение.

Алгоритм поиска с возвратом для данной задачи будет выглядеть следующим образом:

Если все площади посещены -> напечатать решение;

Иначе

a. Добавьте один из следующих ходов к вектору решения и рекурсивно; проверьте, приводит ли этот ход к решению. (Рыцарь может сделать максимум восемь ходов. Мы выбираем один из 8 ходов на этом шаге).

b. Если ход, выбранный на предыдущем шаге, не приводит к решению -> удалите этот ход из вектора решения и попробуйте другие альтернативные ходы.

c. Если ни одна из альтернатив не работает, верните false (возвращение false удалит ранее добавленный элемент в рекурсии, и если false возвращается при первоначальном вызове рекурсии, тогда «решения не существует»).

Реализация решения данной задачи по сути представляет из себя работу с двумерной матрицей 8*8 и сделанной конем шагами.

```
#include <stdio.h>
#define N 8

int solveKTUtil(int x, int y, int movei, int sol[N][N],
   int xMove[], int yMove[]);

/* функция для проверки того, являются ли i,
j допустимыми индексами для шахматной доски N * N */
int isSafe(int x, int y, int sol[N][N])
{
   return (x >= 0 && x < N&& y >= 0 && y < N
      && sol[x][y] == -1);
}

/* функция для печати матрицы решений sol[N][N] */
void printSolution(int sol[N][N])
{
   for (int x = 0; x < N; x++) {
      for (int y = 0; y < N; y++)
         printf(" %2d ", sol[x][y]);
      printf("\n");
   }
}

/* Эта функция решает задачу "ход конем" с помощью поиска с возвратом.
Эта функция в основном использует resolveKTUtil () для решения проблемы.
Она возвращает false, если полный обход невозможен,
в противном случае возвращает true и печатает решение.
Обратите внимание, что может быть несколько решений,
эта функция выводит одно из возможных решений.  */
```

```
int solveKT()
{
   int sol[N][N];

   /* Инициализация матрицы  */
   for (int x = 0; x < N; x++)
      for (int y = 0; y < N; y++)
         sol[x][y] = -1;

   /* xMove[] и yMove[] определяют следующий ход.
     xMove[] для следующей координаты x
     yMove[] для следующей координаты y */
   int xMove[8] = { 2, 1, -1, -2, -2, -1, 1, 2 };
   int yMove[8] = { 1, 2, 2, 1, -1, -2, -2, -1 };

   // позиция коня в первом блоке
   sol[0][0] = 0;

   /* Начнем с 0,0 и исследуем все ходы с помощью resolveKTUtil () */
   if (solveKTUtil(0, 0, 1, sol, xMove, yMove) == 0) {
      printf("Solution does not exist");
      return 0;
   }
   else
      printSolution(sol);

   return 1;
}

/* рекурсия */
int solveKTUtil(int x, int y, int movei, int sol[N][N],
   int xMove[N], int yMove[N])
{
   int k, next_x, next_y;
   if (movei == N * N)
      return 1;

   /* Пробуем все следующие ходы от текущих координат
    */
   for (k = 0; k < 8; k++) {
      next_x = x + xMove[k];
      next_y = y + yMove[k];
      if (isSafe(next_x, next_y, sol)) {
         sol[next_x][next_y] = movei;
         if (solveKTUtil(next_x, next_y, movei + 1, sol,
            xMove, yMove)
            == 1)
```

```
            return 1;
          else
            sol[next_x][next_y] = -1; // Поиск с возвратом
        }
    }

    return 0;
}

int main()
{

    // Вызов функции
    solveKT();
    return 0;
}
```

Давайте обсудим задачу «мыши в лабиринте» как еще один пример проблемы, которую можно решить с помощью обратного поиска.

Лабиринт задается как двоичная матрица блоков размером N * N, где исходный блок – это самый верхний левый блок, т.е. [0] [0], а блок выхода – самый нижний правый блок, т.е. [N-1] [N-1]. Крыса стартует левого верхнего угла и должна добраться до места назначения. Крыса может двигаться только в двух направлениях: вперед и вниз (рисунок 39).

В матрице лабиринта 0 означает, что блок является тупиком, а 1 – что блок может использоваться на пути от источника к месту назначения. Обратите внимание, что это простая версия типичной задачи лабиринта. Например, более сложная версия может заключаться в том, что крыса может двигаться в 4 направлениях или, например, иметь ограниченное количество движений.

Рисунок 39 – условие задачи про мышь в лабиринте

Ниже приводится двоичное матричное представление вышеупомянутого лабиринта.

$$\{1, 0, 0, 0\}$$
$$\{1, 1, 0, 1\}$$
$$\{0, 1, 0, 0\}$$
$$\{1, 1, 1, 1\}$$

Далее приведен лабиринт с выделенным путем решения (рисунок 40).

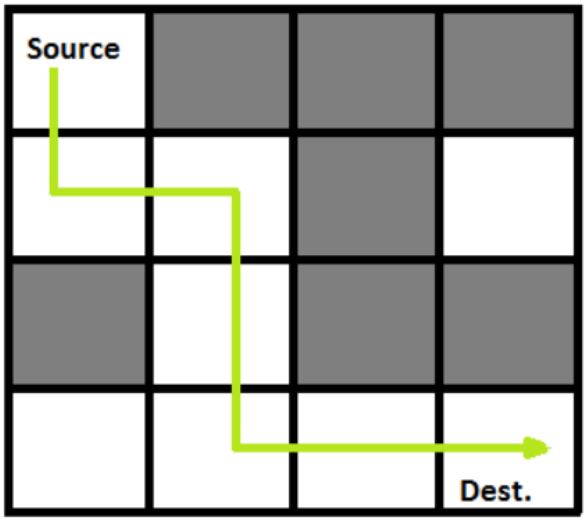

Рисунок 40 – графическое решение задачи про мышь в лабиринте

Также, данное решение представим в матричном представлении.

{1, 0, 0, 0}
{1, 1, 0, 0}
{0, 1, 0, 0}
{0, 1, 1, 1}

// Все клетки в пути решения отмечены как "1"

Как подойти к решению данной задачи? Сформируем рекурсивную функцию, которая будет следовать по пути и проверять, достигает ли путь места назначения или нет. Если путь не достигает места назначения, нам нужно вернутся и попробовать другие пути.

Алгоритм:

1. Создайте матрицу, изначально заполненную нулями.
2. Создайте рекурсивную функцию, которая принимает исходную матрицу, выходную матрицу и положение грызуна (i, j).
3. Если позиция находится вне матрицы или позиция недействительна, тогда вернитесь.
4. Отметьте выход положения [i] [j] как 1 и проверьте, является ли текущая позиция местом назначения. Если место назначения достигнуто, напечатайте матрицу вывода и вернитесь.
5. Рекурсивно вызывайте позиции (i + 1, j) и (i, j + 1).
6. Снимите отметку с позиции (i, j), т.е. output [i] [j] = 0.

Ниже приведен код решения данной задачи.

```
#include <stdio.h>

// Размер лабиринта
#define N 4

bool solveMazeUtil(
        int maze[N][N], int x,
        int y, int sol[N][N]);

/* функция для печати матрицы  sol [N][N] */
void printSolution(int sol[N][N])
{
        for (int i = 0; i < N; i++) {
                for (int j = 0; j < N; j++)
                        printf(" %d ", sol[i][j]);
                printf("\n");
        }
}
```

```
/* Служебная функция для проверки того, что x,
y - допустимый индекс для лабиринта N * N */
bool isSafe(int maze[N][N], int x, int y)
{
	// if (x, y вне лабиринта) вернуть false
	if (
		x >= 0 && x < N && y >= 0
		&& y < N && maze[x][y] == 1)
		return true;

	return false;
}

/* Эта функция решает проблему лабиринта с помощью поиска с возвратом.
В основном для решения проблемы используется resolveMazeUtil().
Он возвращает false, если путь невозможен, в противном случае возвращает true и печатает путь в виде единиц.
Обратите внимание, что может быть несколько решений, эта функция выводит одно из возможных решений..*/
bool solveMaze(int maze[N][N])
{
	int sol[N][N] = { { 0, 0, 0, 0 },
					{ 0, 0, 0, 0 },
					{ 0, 0, 0, 0 },
					{ 0, 0, 0, 0 } };

	if (solveMazeUtil(
		maze, 0, 0, sol)
		== false) {
		printf("Solution doesn't exist");
		return false;
	}

	printSolution(sol);
	return true;
}


bool solveMazeUtil(
	int maze[N][N], int x,
	int y, int sol[N][N])
{
	// если (x, y - цель) вернуть true
	if (
		x == N - 1 && y == N - 1
		&& maze[x][y] == 1) {
```

```
                sol[x][y] = 1;
                return true;
        }

        // Проверить, действителен ли лабиринт [x] [y]
        if (isSafe(maze, x, y) == true) {
                // Проверить, является ли текущий блок уже частью пути решения.
                if (sol[x][y] == 1)
                        return false;

                // отметить x, y как часть пути решения
                sol[x][y] = 1;

                /* Движение вперед в направлении x */
                if (solveMazeUtil(
                        maze, x + 1, y, sol)
                        == true)
                        return true;

                /* Если движение в направлении x не дает решения, двигаемся
                вниз в направлении y. */
                if (solveMazeUtil(
                        maze, x, y + 1, sol)
                        == true)
                        return true;

                /* Если ни одно из вышеперечисленных
                движений не работает, то снимем отметку x, y как часть пути решения. */
                sol[x][y] = 0;
                return false;
        }

        return false;
}


int main()
{
        int maze[N][N] = { { 1, 0, 0, 0 },
                                        { 1, 1, 0, 1 },
                                        { 0, 1, 0, 0 },
                                        { 1, 1, 1, 1 } };

        solveMaze(maze);
        return 0;
}
```

Результат работы в консоли отладки представлен на рисунке 41.

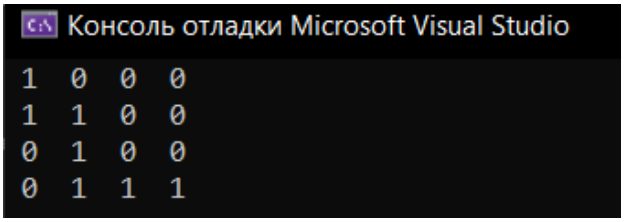

Рисунок 41 – решение задачи с лабиринтом

Но что, если путей решения больше одного?

Алгоритм решения задач с множеством решений

Допустим, что наша мышь из последней задачи попала в лабиринт посложнее. Новая задача звучит следующим образом: учитывая лабиринт с препятствиями, подсчитайте количество путей, чтобы добраться до самой правой нижней ячейки от самой верхней левой ячейки. Ячейка в данном лабиринте имеет значение **-1**, если это завал или тупик, в противном случае – **0** (рисунок 42).

Из данной ячейки нам разрешено перемещаться только в ячейки (i + 1, j) и (i, j + 1).

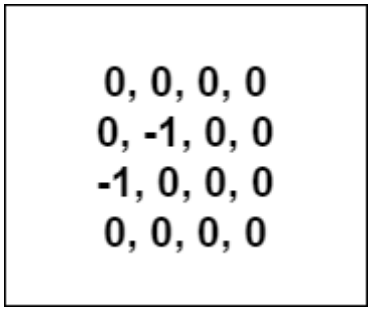

Рисунок 42 – условие задачи с множеством решений

В матричном представлении:

{ 0, 0, 0, 0 },
{ 0, -1, 0, 0 },
{ -1, 0, 0, 0 },
{ 0, 0, 0, 0 };

Сколько же путей существует чтобы из верхнего левого нуля добраться до правого нижнего? Авторы насчитали – 4. Давайте докажем, что путей именно столько. Идея состоит в том, чтобы изменить данную матрицу [] [] так,

чтобы матрица [i] [j] содержала количество путей для достижения (i, j) из (0, 0), если (i, j) не является блокировкой, иначе grid [i] [j] остается -1.

```
#include <iostream>
using namespace std;
#define R 4
#define C 4

// Возвращает количество возможных путей
// в лабиринте [R] [C] от (0,0) до (R-1, C-1).
int countPaths(int maze[][C])
{
	// Если исходная ячейка заблокирована,
	// нет возможности никуда двигаться
	if (maze[0][0] == -1)
		return 0;

	// Инициализация крайнего левого столбца
	for (int i = 0; i < R; i++)
	{
		if (maze[i][0] == 0)
			maze[i][0] = 1;

	// Если мы обнаруживаем заблокированную ячейку в крайнем левом ряду,
	// то у нас нет возможности посетить любую ячейку непосредственно под ней.
		else
			break;
	}

	// Аналогичным образом инициализируем самую верхнюю строку
	for (int i = 1; i < C; i++)
	{
		if (maze[0][i] == 0)
			maze[0][i] = 1;

	// Если мы обнаруживаем заблокированную ячейку в самом нижнем ряду,
	// то у нас нет возможности посетить любую ячейку непосредственно под ней.
		else
			break;
	}

	// Единственное отличие состоит в том, что если ячейка равна -1,
	// просто игнорируйте ее, иначе рекурсивно вычисляйте значения счетчика [i] [j]
	for (int i = 1; i < R; i++)
	{
		for (int j = 1; j < C; j++)
		{
```

```
                        // игнорирование ячейки при найденом завале
                        if (maze[i][j] == -1)
                                continue;

                        // Если мы можем добраться до клетки [i] [j]
                        //из клетки [i-1] [j], тогда увеличиваем счетчик.
                        if (maze[i - 1][j] > 0)
                                maze[i][j] = (maze[i][j] + maze[i - 1][j]);

                        // Если мы можем добраться до клетки [i] [j] из
                        // клетки [i] [j-1], тогда увеличиваем счетчик.
                        if (maze[i][j - 1] > 0)
                                maze[i][j] = (maze[i][j] + maze[i][j - 1]);
                }
        }

        // Если последняя ячейка заблокирована вывести 0, иначе вывести ответ
        return (maze[R - 1][C - 1] > 0) ? maze[R - 1][C - 1] : 0;
}

int main()
{
        int maze[R][C] = {      {0, 0, 0, 0},
                                {0, -1, 0, 0},
                                {-1, 0, 0, 0},
                                {0, 0, 0, 0}      };
        cout << countPaths(maze);
        return 0;
}
```

Ответ, спрогнозированный нами до написания кода, совпадает с программой (рисунок 43)!

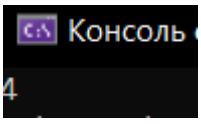

Рисунок 43 – ответ на задачу с множеством решений

Основы динамического программирования

Динамическое программирование – это алгоритмическая парадигма, которая решает сложную проблему, разбивая ее на подзадачи и сохраняя результаты подзадач, чтобы избежать повторного вычисления тех же результатов. Ниже приведены два основных свойства проблем, которые предполагают, что данные проблемы могут быть решены с помощью динамического программирования:

- Перекрывающиеся подзадачи,
- Оптимальная подконструкция.

1) Перекрывающиеся подзадачи:

Как и «разделяй и властвуй», динамическое программирование объединяет решения подзадач. Динамическое программирование в основном используется, когда решения одних и тех же подзадач требуются снова и снова. В динамическом программировании вычисленные решения подзадач хранятся в таблице, поэтому их не нужно вычислять заново. Таким образом, динамическое программирование бесполезно, когда нет общих (перекрывающихся) подзадач, потому что нет смысла хранить решения, если они больше не нужны. Например, у двоичного поиска нет общих подзадач. Если мы возьмем пример следующей рекурсивной программы для чисел Фибоначчи, то найдем много подзадач, которые решаются снова и снова.

```
/* простая рекурсивная программа для чисел Фибоначчи */
int fib(int n)
{
   if (n <= 1)
      return n;

   return fib(n - 1) + fib(n - 2);
         }
```

Дерево рекурсии для выполнения функции fib (5) выглядит следующим образом (рисунок 44).

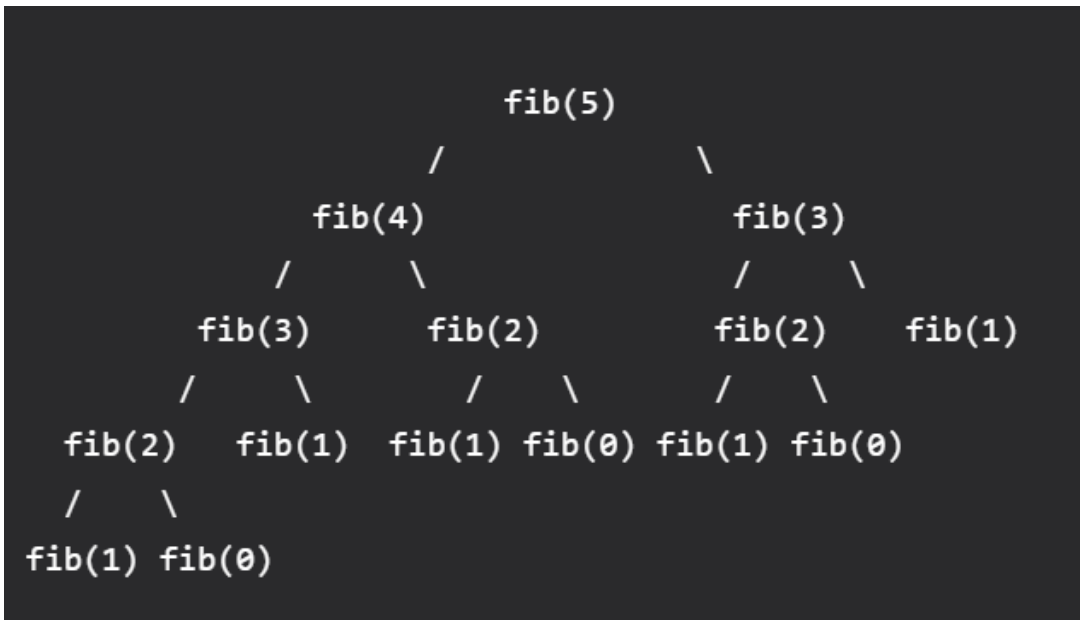

Рисунок 44 – дерево рекурсии для функции Фибоначчи

Мы видим, что функция fib (3) вызывается 2 раза. Если бы мы сохранили значение fib (3), то вместо того, чтобы вычислять его снова, мы могли бы повторно использовать старое сохраненное значение. Есть два разных способа сохранить значения, чтобы их можно было использовать повторно:

1) Запоминание (сверху вниз)
2) Табулирование (снизу вверх)

1) Запоминание (сверху вниз): запомненная программа для проблемы похожа на рекурсивную версию с небольшой модификацией, которая просматривает справочную таблицу перед вычислением решений. Мы инициализируем поисковый массив со всеми начальными значениями как ноль. Всякий раз, когда нам нужно решение подзадачи, мы сначала заглядываем в справочную таблицу. Если предварительно вычисленное значение есть, мы возвращаем это значение, в противном случае мы вычисляем значение и помещаем результат в таблицу поиска, чтобы его можно было повторно использовать позже. Программная реализация представлена ниже:

```
#include<stdio.h>
#define NIL -1
#define MAX 100

int lookup[MAX];

/* Функция для инициализации значений в таблице поиска */
void _initialize()
```

```
{
   int i;
   for (i = 0; i < MAX; i++)
      lookup[i] = NIL;
}

/* функция для n-го числа Фибоначчи */
int fib(int n)
{
   if (lookup[n] == NIL)
   {
      if (n <= 1)
         lookup[n] = n;
      else
         lookup[n] = fib(n - 1) + fib(n - 2);
   }

   return lookup[n];
}

int main()
{
   int n = 40;
   _initialize();
   printf("Fibonacci number is %d ", fib(n));
   return 0;
}
```

В результате, 40-вым числом Фибоначчи является 102334155 (рисунок 45).

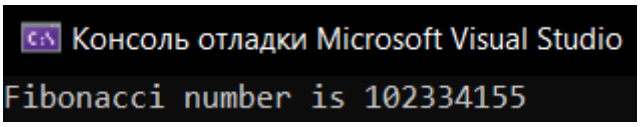

Рисунок 45 – 40ое число Фибоначчи

2) Табулирование (снизу вверх): программа с таблицами для данной проблемы строит таблицу снизу вверх и возвращает последнюю запись из таблицы. Например, для того же числа Фибоначчи мы сначала вычисляем fib (0), затем fib (1), затем fib (2), затем fib (3) и так далее. Итак, буквально, мы строим решения подзадач снизу-вверх. Программная реализация представлена ниже.

```
#include<stdio.h>
int fib(int n)
{
```

```
int f[n + 1];
int i;
f[0] = 0; f[1] = 1;
for (i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];

return f[n];
}

int main()
{
int n = 9;
printf("Fibonacci number is %d ", fib(n));
return 0;
}
```

И в табулировании, и в запоминании хранятся решения подзадач. В запомненной версии таблица заполняется по запросу, в то время как в табулированной версии, начиная с первой записи, все записи заполняются одна за другой. В отличие от табличной версии, все записи справочной таблицы не обязательно заполняются в запомненной версии.

Чтобы увидеть оптимизацию, достигаемую запоминаемыми и табличными решениями по сравнению с базовым рекурсивным решением, просмотрите время, затраченное на следующие прогоны для вычисления 40-го числа Фибоначчи:

- Рекурсивное решение
- Мемоизированное решение
- Табличное решение

Время, затрачиваемое на метод рекурсии, намного превышает два упомянутых выше метода динамического программирования - запоминание и табуляцию! В качестве еще одного проблема, рассмотрим метод «уродливых чисел».

«Уродливые числа»

Уродливые числа - это числа, единственные простые множители которых равны 2, 3 или 5. Последовательность 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15,… показывает первые 11 злых чисел. По умолчанию – 1 включен в этот список.

Для данного числа n задача состоит в том, чтобы найти n-е уродливое число. Например:

- Ввод: n = 7
  Вывод : 8

- Ввод : n = 10
  Вывод : 12

- Ввод : n = 15
  Вывод : 24

- Ввод : n = 150
  Вывод : 5832

Чтобы проверить, «уродливо» ли число, разделите число на наибольшие делимые степени 2, 3 и 5. Если число становится 1, то это некрасивое число, в противном случае – нет.

Например, давайте посмотрим, как проверить 300 – уродливо или нет. Наибольшая делимая степень 2 равна 4, после деления 300 на 4 мы получаем 75. Наибольшая делимая степень 3 равна 3, после деления 75 на 3 получаем 25. Наибольшая делимая степень 5 равна 25, после деления 25 на 25 получаем 1 Поскольку в итоге мы получаем 1, 300 – уродливое число.

Ниже представлена реализация описанного выше подхода:

```
#include <stdio.h>
#include <stdlib.h>

// Эта функция делит a на наибольшую делимую степень b
int maxDivide(int a, int b)
{
        while (a % b == 0)
                a = a / b;
        return a;
}
```

```
// Функция проверки "уродливого" числа
int isUgly(int no)
{
	no = maxDivide(no, 2);
	no = maxDivide(no, 3);
	no = maxDivide(no, 5);

	return (no == 1) ? 1 : 0;
}

// Функция для получения n-го уродливого числа
int getNthUglyNo(int n)
{

	int i = 1;

	// количество уродливых чисел
	int count = 1;

	//Проверка всех целых чисел, пока счетчик "уродства" не станет n
	while (n > count) {
		i++;
		if (isUgly(i))
			count++;
	}
	return i;
}

int main()
{

	unsigned no = getNthUglyNo(150);
	printf("150th ugly no. is %d ", no);
	getchar();
	return 0;
}
```

В итоге, 150-ым «уродливым» числом является число 5832 (рисунок 46).

```
150th ugly no. is 5832
```

Рисунок 46 – результат нахождения 150ого «уродливого» числа

Этот метод неэффективен по времени, поскольку он проверяет все целые числа до тех пор, пока количество уродливых чисел не станет n, а сложность этого метода составляет O(1).

А что, если мы будем использовать динамическое программирование для решения задач, связанных с уродливыми числами? Последовательность уродливых чисел: 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15,… поскольку каждое число можно разделить только на 2, 3, 5, один из способов взглянуть на последовательность - разделить ее на три группы, как показано ниже:

(1) 1 × 2, 2 × 2, 3 × 2, 4 × 2, 5 × 2,…

(2) 1 × 3, 2 × 3, 3 × 3, 4 × 3, 5 × 3,…

(3) 1 × 5, 2 × 5, 3 × 5, 4 × 5, 5 × 5,…

Мы можем обнаружить, что каждая подпоследовательность является самой уродливой последовательностью (1, 2, 3, 4, 5,…), умноженной на 2, 3, 5. Затем мы используем такой же метод слияния, что и сортировка слиянием, чтобы получить каждое уродливое число из трех подпоследовательности. На каждом шаге мы выбираем самый маленький и продвигаемся на один шаг после.

1) Объявите массив некрасивых чисел: ugly [n];
2) Инициализировать первое уродливое число: ugly [0] = 1;
3) Инициализируйте три индексные переменные массива i2, i3, i5, чтобы они указывали на 1-й элемент уродливого массива: i2 = i3 = i5 = 0;
4) Инициализируйте 3 варианта следующего уродливого числа:

next_mulitple_of_2 = ugly [i2] * 2;
next_mulitple_of_3 = ugly [i3] * 3
next_mulitple_of_5 = ugly [i5] * 5;

5) Теперь выполним цикл, чтобы заполнить все уродливые числа до 150:

```
For(i = 1; i < 150; i++)
{
next_ugly_no = Min(next_mulitple_of_2,
   next_mulitple_of_3,
   next_mulitple_of_5);

ugly[i] = next_ugly_no

if (next_ugly_no == next_mulitple_of_2)
{
   i2 = i2 + 1;
   next_mulitple_of_2 = ugly[i2] * 2;
}
if (next_ugly_no == next_mulitple_of_3)
{
```

```
    i3 = i3 + 1;
    next_mulitple_of_3 = ugly[i3] * 3;
}
if (next_ugly_no == next_mulitple_of_5)
{
    i5 = i5 + 1;
    next_mulitple_of_5 = ugly[i5] * 5;
}

        }/* end of for loop */
```

6) Получить уродливое число.

Посмотрим, как это сработает в нашем коде.

```
#include <iostream>
using namespace std;

// Функция для получения n-го уродливого числа
unsigned getNthUglyNo(unsigned n)
{
    // Для хранения уродливых чисел
    unsigned ugly[n];
    unsigned i2 = 0, i3 = 0, i5 = 0;
    unsigned next_multiple_of_2 = 2;
    unsigned next_multiple_of_3 = 3;
    unsigned next_multiple_of_5 = 5;
    unsigned next_ugly_no = 1;

    ugly[0] = 1;
    for (int i = 1; i < n; i++) {
        next_ugly_no = min(
            next_multiple_of_2,
            min(next_multiple_of_3, next_multiple_of_5));
        ugly[i] = next_ugly_no;
        if (next_ugly_no == next_multiple_of_2) {
            i2 = i2 + 1;
            next_multiple_of_2 = ugly[i2] * 2;
        }
        if (next_ugly_no == next_multiple_of_3) {
            i3 = i3 + 1;
            next_multiple_of_3 = ugly[i3] * 3;
        }
        if (next_ugly_no == next_multiple_of_5) {
            i5 = i5 + 1;
            next_multiple_of_5 = ugly[i5] * 5;
        }
    }
```

```
    // конец петли (i=1; i<n; i++)
    return next_ugly_no;
}

int main()
{
    int n = 150;
    cout << getNthUglyNo(n);
    return 0;
}
```

В результате работы программы получаем ответ, равный 5832 (рисунок 47).

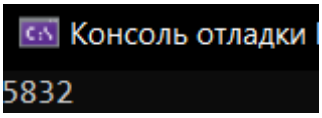

Рисунок 47 – решение задачи использованием динамического программирования

Заключение

В данном пособии мы постарались рассмотреть основные методы сортировок, познакомились с различными структурами данных и особенностями работы с информацией. Также в данной работе были затронуты дополнительные темы, косвенно связанные с сортировкой, тема алгоритмов, такие как «жадный алгоритм», «алгоритм поиска с возвратом» и т.п. Данное пособие не может вместить в себя абсолютно все типы сортировок и тонкости работы с данными, поскольку эти темы динамически обновляются и дополняются. Однако авторы надеются, что изучив данный источник, студенты получат основные представления, связанные с работой с данными и смогут правильно выбрать направления, необходимые им лично, для дальнейшей работы.

СПИСОК ЛИТЕРАТУРЫ

1. Дж, Макконелл Анализ алгоритмов. Активный обучающий подход. / Макконелл Дж. — 3-е дополненное издание. — : М: Техносфера, 2009. — 416 с.

2. Миллер, Р. Последовательные и параллельные алгоритмы: Общий подход / Р. Миллер, Л. Боксер. — М. : БИНОМ. Лаборатория знаний, 2006. — 406 с.

3. Скиена, С. Алгоритмы. Руководство по разработке. / С. Скиена. — 2-е изд. — Санкт-Петербург : БХВ-Петербург., 2011. — 720 с.

4. Алгоритмы. Построение и анализ. / Т. Х. Кормен, Ч. И. Лейзерсон, Р. Л. Ривест, Клиффорд Штайн. — 3-е изд. — : Издательство «Вильямс», 2019. — 1328 с.

Сведения об авторах

1. Каширская Елизавета Натановна – доцент кафедры промышленной информатики института кибернетики МИРЭА – Российский технологический университет.

2. Макаров Максим Алексеевич – ассистент кафедры промышленной информатики института кибернетики МИРЭА – Российский технологический университет.

3. Харьковский Станислав Евгеньевич – старший преподаватель кафедры промышленной информатики института кибернетики МИРЭА – Российский технологический университет.