МИНОБРНАУКИ РОССИИ

Федеральное государственное бюджетное образовательное учреждение
высшего образования
«МИРЭА – Российский технологический университет»
РТУ МИРЭА

ЛЕКЦИОННЫЕ МАТЕРИАЛЫ

Структуры и алгоритмы обработки данных (часть 1)

(наименование дисциплины (модуля) в соответствии с учебным планом)

Уровень **бакалавриат**
(бакалавриат, магистратура, специалитет)

Форма обучения **очная**
(очная, очно-заочная, заочная)

Направление(-я) подготовки **09.03.04 «Программная инженерия»**
(код(-ы) и наименование(-я))

Институт **информационных технологи (ИТ)**
(полное и краткое наименование)

Кафедра **Математического обеспечение и стандартизации информационных технологий (МОСИТ)**
(полное и краткое наименование кафедры, реализующей дисциплину(модуль))

Лектор **Доцент Рысин Михаил Леонидович, к.п.н., доцент**
(сокращенно – ученая степень, ученое звание; полностью – ФИО)

Используются в данной редакции с учебного года 2022/23
(учебный год цифрами)

Проверено и согласовано «31» августа 2022 г. Зуев А.С
(подпись директора Института/Филиала с расшифровкой)

Москва – 2022

ОГЛАВЛЕНИЕ

ЛЕКЦИЯ 1
ВВОДНЫЕ ПОНЯТИЯ АЛГОРИТМИЗАЦИИ

1.1. Понятие алгоритма и исполнителя

Неформальное определение **алгоритма** (лат. algorithmi) – это набор инструкций, описывающих порядок действий исполнителя, для достижения определённого результата.

Алгоритм – это одно из базисных понятий в математике, т.к. собственно **вычисления** (решение вычислительной задачи) – это обработка числовой информации по определённому алгоритму.

Алгоритм решения вычислительной задачи – это корректно определённая вычислительная процедура, на вход которой подаётся значение (набор значений), и результатом выполнения которой является выходное значение (набор значений).

Обсуждая понятие алгоритма, нельзя не назвать связанное с ним понятие – **исполнитель**. Исполнитель – это абстрактная или реальная (техническая или биологическая) система, способная выполнить действия, предписываемые алгоритмом (т.е. решить задачу).

Исполнители: ***неформальный*** (знает конечную цель алгоритма, например, человек) и ***формальный*** (например, процессор компьютера).

Характеристики исполнителя:

- среда (обстановка) – место действия исполнителя;
- система допустимых команд (должны быть заданы условия применимости – необходимое состояние среды и описаны результаты выполнения);
- набор возможных действий;
- отказы (недопустимое для выполнения команды состояние среды).

Теория алгоритмов – это наука на стыке математики и информатики об общих свойствах и закономерностях алгоритмов и разнообразных формальных моделях их представления. Это теоретическая основа вычислительных наук, в рамках которой решаются задачи:

- формализации задач;
- формализация алгоритма (модели вычислений);
- алгоритмическая неразрешимость;
- уровни сложности (классификация, анализ, критерии качества).

Способы формализации алгоритма:

- теория автоматов:
    - машина Тьюринга (модель процедурного программирования);
    - машина Поста;
- рекурсивные функции Гёделя — Эрбрана — Клини;
- нормальный алгоритм Маркова;
- λ-исчисление Чёрча.

Рассмотрение этих подходов выходит за рамки предмета нашего пособия, подробную информацию можно найти в специальной литературе.

Виды алгоритмов:

- ***детерминированные*** (они же жёсткие, механические) – единственная и достоверная последовательность инструкций, приводящая к однозначному результату;
- ***гибкие***:
    - ***вероятностные*** (стохастические) – используют случайные величины (ГСЧ), допускают несколько путей решения, приводящие к высоко вероятному достижению результата;
    - ***эвристические*** – используют различные разумные соображения без строгих обоснований.

Свойства алгоритма:

- дискретность – может быть разбит на конечное количество отдельных элементарных шагов (операций, команд);
- понятность – включает только команды из набора допустимых команд исполнителя;
- детерминированность (определённость) – каждый следующий шаг однозначно определяется состоянием системы, и, как следствие, всегда получается один и тот же ответ для одних и тех же исходных данных;
- результативность – алгоритм всегда приводит к получению определённого результата;
- массовость – применимость к множеству наборов начальных данных;
- завершаемость (конечность) – результат должен быть получен за конечное время (число шагов).

Способы записи алгоритма:

- словесный (на естественном языке);
- формульный (алгоритм решения математических задач);
- табличный (для реляционных задач, например, в Excel);

– графический (блок-схемы, UML-диаграммы, ДРАКОН-схемы);
– операторный – из конечного набора допустимых команд исполнителя (языки программирования, псевдокод).

1.2. Компьютерная программа

Компьютерная программа – это алгоритм решения вычислительной задачи компьютером (вычислительной машиной).

Компьютер как исполнитель[1] способен выполнять программы только на языке двоичных машинных команд (**машинный код**, он же двоичный или нативный). При этом машинная команда структурно состоит из двух частей: КОп (код операции, обязательная часть) и адресной части. Общая длина машинной команды определяется разрядностью шины управления конкретной аппаратной платформы ЭВМ (обычно 8, 16, 32, 64 бита).

На практике написание компьютерных программ в машинном коде слишком трудоёмко и неэффективно по времени (и, как следствие, слишком дорого). Реальные программы создаются на языках программирования более высокого уровня – ассемблерах и ЯВУ. Одним из многочисленных примеров ЯВУ является язык C++, используемый в данном пособии.

Язык программирования – это набор допустимых операторов, синтаксических и семантических правил их использования для создания компьютерных программ. В составе любого языка выделяют алфавит (набор допустимых символов), лексемы, комментарии, директивы препроцессора, синтаксис (выражения и конструкции) и семантику (смысл элементов, выражений и конструкций).

Компьютерная программа на языке ассемблера или ЯВУ – это текст, который не может быть непосредственно исполнен процессором ЭВМ. Поэтому необходим предварительный этап – **трансляция** кода – это преобразование текста программы в нативный код.

Трансляция кода возможна в двух вариантах – **компиляция** или/и **интерпретация**.

Компьютерной программой часто называют **скрипт** – это набор команд, предназначенный для исполнения не процессором, а программной средой (исполнительной средой). Например, SQL-скрипт – это программа для её исполнения средой СУБД (Microsoft SQL Server, MySQL, PostgreSQL и пр.).

[1] Строго говоря, исполнителем машинного кода является устройство управления в составе процессорного ядра.

2. РАЗРАБОТКА КОРРЕКТНЫХ АЛГОРИТМОВ

2.1. Разработка алгоритмов

Этапы создания эффективных алгоритмов:

1. Разработать алгоритм и записать его, используя один из способов:
   - формализованный: словесный, графический, псевдокод;
   - формальный: язык программирования.
2. Доказать ***корректность алгоритма***.
3. Проанализировать алгоритм для определения его вычислительной сложности.

Псевдокод – это компактный способ представления алгоритмов человеку без подробностей и специфического синтаксиса языка программирования (табл. 1). Впоследствии, псевдокод реализуется на определённом ЯВУ.

Таблица 1. Форматы представления действий в псевдокоде

Действие	Формат представления в псевдокоде	Пример реализации в алгоритме
Заголовок алгоритма	Имя_алгоритма (<список_параметров>)	InsertSort (x,n)
Операция присваивания	Имя_переменной←выражение	t←25

Условный оператор	**If** условие **then**
	Действие 1
	Else
	Действие2
	EndIf
		If a>b **then**
		max←a
		Else
		max←b
		EndIf

Цикл for	**For** иниц_счётчика **to** конечн_знач_счётчика **do** Тело цикла **od**	**For** i←1 **to** n **do** Вывод A[i] **od**
Цикл while	**While** (условие) **do** Тело цикла **od**	**While** (i<=n) **do** Ввод A[i] **od**
Объявление переменных	Имя и семантика (вкл. тип)	a,b – целые числа min – минимум из a и b – целое число

К настоящему моменту сформировались два подхода к проектированию алгоритмов:

1) нисходящее проектирование;

2) восходящее проектирование.

Нисходящее проектирование – это метод «сверху вниз» (рис. 1). Основные его принципы:

- Исходная задача разбивается на конечное число вспомогательных подзадач, формулируемых и решаемых в терминах более простых и элементарных операций (процедур) – метод **декомпозиции**;

– Записывается обобщенный алгоритм задачи с применением выделенных на предыдущем шаге подзадач (процедур-заглушек);
– Выделенные подзадачи вновь разбиваются на более простые и элементарные вплоть до отдельных команд исполнителя (инструкций).

Рисунок 1. Нисходящее проектирование алгоритма

Восходящее проектирование предполагает создание алгоритма из уже готового корректного набора подалгоритмов:

1) строятся функционально завершенные низкоуровневые подзадачи;

2) от них переходят к более общим,

3) и так далее, до уровня, на котором можно записать решение исходной задачи.

За историю развития современного программирования сложились на практике и развивались в теории несколько основных методологий **(парадигм) программирования** (рис. 2).

Рисунок 2. Современные парадигмы программирования

Предмет нашего изучения подразумевает разработку алгоритмов в рамках методологии **структурного программирования**. В его основе – работы Э. Дейкстры и Ч. Хоара, а также теорема Бёма-Якопини.

Структурированный алгоритм:

– Реализует три **базовые управляющие структуры** – простое следование, ветвление, цикл. Базовые конструкции могут быть

вложены друг в друга произвольным образом.

- Использует понятия **блок** и **подпрограмма**.
- Применим как для нисходящего, так и восходящего проектирования.

В рамках методологии структурного программирования могут быть реализованы следующие основные **методы разработки** алгоритмов:

Метод **грубой силы** – прямолинейное решение – полный перебор всех входных значений.

Поиск с возвратом – возврат к точке принятия неверного решения с переходом в другое подмножество дерева решений.

Метод **ветвей и границ** – отсев подмножеств допустимых решений, заведомо не содержащих оптимальных решений.

Уменьшение размера задачи – основан на связи решений для данного экземпляра и меньшего.

Метод **преобразования** – к другой форме или в более простой случай (предварительная сортировка).

Жадный метод – подход к решению некоторых задач оптимизации, когда на каждом этапе принимаются локально оптимальные решения, допуская, что конечное решение также окажется оптимальным.

Разделяй и властвуй (декомпозиция) – рекурсивное деление задачи на отдельные подзадачи.

Динамическое программирование – оптимизация решения сложных задач за счёт однократного выполнения одинаковых подзадач, что делает возможным многократное использование расчётных данных без необходимости их перерасчёта.

2.2. Инвариант цикла

Корректность является одним из основных свойств алгоритма. Алгоритм ***корректен***, если для каждого ввода результатом его работы является корректный вывод.

Верификация здесь возможна в виде тестирования (практический метод) и формальная (аналитический метод).

Один из возможных подходов к аналитической оценке корректности алгоритма основан на **методе математической индукции**, который, в свою очередь, предполагает цепочку рассуждений. При этом используется понятие инварианта цикла (для алгоритмов, основанных на циклах).

Инвариант – это свойство некоторого класса (множества) математических

объектов, остающееся неизменным при определённого вида преобразованиях над элементами этого класса.

В цикле происходит преобразование данных, при этом можно найти свойство такое, что несмотря на эти преобразования, оно останется неизменным.

Инвариант цикла – то самое свойство, сохраняемое циклом – это логическое выражение (предикат), истинное непосредственно перед и сразу после каждой итерации цикла, а также сразу по выходу из цикла, зависящее от переменных, изменяющихся в теле цикла.

Инвариант цикла отражает в себе ***желаемый результат*** – то соотношение переменных, которое, в конечном счёте, останется истинным при выходе из цикла, что мы и интерпретируем как ***корректный*** результат работы циклического алгоритма.

Инвариант может быть использован для доказательства корректности циклического алгоритма без необходимости его непосредственного выполнения.

Инвариантом цикла не может быть условие цикла, т.к. условие при выходе из цикла становится ложным, а инвариант должен остаться истинным.

Циклический алгоритм корректен, если найден инвариант цикла и доказана его конечность (условие завершения цикла достижимо).

Пусть дан простой алгоритм:

```
j←9
for i←0 to 9 do
  j--;
od
```

Примеры инвариантов для цикла в этом алгоритме:

а) $i + j \geq 9$;

б) $i \geq 0$ и $i \leq 10$.

Первый инвариант более «сильный», т.к. в нём используется больше переменных из анализируемого цикла.

2.3. Корректность циклического алгоритма

Доказательство корректности цикла на основе инварианта включает в себя 4 пункта:

1. Доказывается, что выражение инварианта истинно перед началом цикла (**инициализация**).

2. Доказывается, что выражение инварианта сохраняет свою истинность после первого выполнения тела цикла и после произвольного выполнения тела (**сохранение**). Так по индукции доказывается, что по завершении любой

итерации инвариант будет выполняться.

3. Доказывается, что выражение инварианта (желаемое соотношение переменных) сохраняет свою истинность по выходу из цикла (**завершение**).

4. Доказывается (возможно, без применения инварианта), что цикл ***завершится***, то есть условие завершения рано или поздно будет выполнено.

Истинность утверждений на этих этапах однозначно свидетельствует о том, что цикл выполнится за конечное время и даст желаемый результат.

Пример – алгоритм поиска минимума в массиве [2] [3]:

```
Min←A[1]
for i←2 to n do
  if A[i] < Min then
   Min←A[i]
  endif
od
```

Формулировка инварианта цикла здесь (желаемый результат): «В переменной Min записан минимум из первых i элементов [1,i) массива».

Область изменения параметров этой задачи [1,n) можно разделить на две части:

- **исследованную область**, для которой найден Min в [1,i);
- **область неопределенности** [i+1,n).

Необходимо составлять цикл так, чтобы ***на каждой итерации*** область неопределенности ***сокращалась***, это, собственно, и будет доказывать, что условие завершения рано или поздно будет выполнено, т.е. цикл конечный:

- в начале первой итерации исследованная область представляет собой единственную точку 1, а область неопределенности составляет [2,n);
- на втором шаге область неопределенности сокращается до [3,n);
- на третьем – до [4,n) и т.д., пока не превратится в пустое множество.

Пример – алгоритм суммирования элементов массива:

```
Sum←0
for i←1 to n do
  Sum←Sum+A[i]
od
```

После каждого шага цикла при любом i к переменной Sum добавляется элемент массива A[i].

Тогда инвариант цикла: «В Sum накоплена сумма первых i элементов [1,i) массива».

[2] В псевдокоде нумерация элементов массива обычно начинается с 1.

[3] Подробнее о массивах и средствах языка C++ для работы с ними см. в п.7.

Пример – сортировка массива пузырьком:

```
for i←1 to n-1 do
for j←n-1 downto i do
  if A[j] > A[j+1] then
   c←A[i]; A[j]←A[j+1]; A[j+1]←c;
  endif
od
```

На каждом шаге внешнего цикла на свое место «всплывает» один элемент массива, поэтому инвариант внешнего цикла: «После выполнения i-го шага цикла первые i элементов массива [1,i) отсортированы и установлены на свои места».

Во внутреннем цикле очередной «лёгкий» элемент поднимается вверх к началу массива. Перед первым шагом внутреннего цикла элемент, который будет стоять на i-м месте в отсортированном массиве, может находиться в любой ячейке от A[i] до A[n].

После каждого шага его «зона нахождения» (область неопределённости) сокращается на одну ячейку.

Тогда инвариант внутреннего цикла: «Элемент на i-м месте в отсортированном массиве может находиться в любой ячейке от A[i] до A[j]».

Когда в конце этого цикла j = i, элемент A[i] встаёт на своё место.

ЛЕКЦИЯ 2
3. ВЫЧИСЛИТЕЛЬНАЯ СЛОЖНОСТЬ АЛГОРИТМА

3.1. Эффективность алгоритма

Любой алгоритм, в т.ч и вычислительный, нуждается в определённого рода ресурсах для своего выполнения.

Тогда **анализ алгоритма** позволит предсказать требуемые для его выполнения ресурсы (для алгоритмов решения вычислительных задач[4] – время работы процессора, память и пр.).

Кроме того, на основе анализа нескольких алгоритмов можно выбрать наиболее **эффективный**.

Критериями эффективности алгоритма являются **скорость** (время выполнения или, то же, время работы исполнителя – процессора ЭВМ) и расход **памяти** (внутренней в первую очередь) и/или других ресурсов.

Так, алгоритм A_1 эффективнее алгоритма A_2, если алгоритм A_1 выполняется за меньшее время и (или) требует меньше вычислительных ресурсов.

Иными словами, составляющие эффективности алгоритма:

- время – это мера системной эффективности;
- расход памяти – мера пространственной эффективности.

Количество команд, выполняющихся в алгоритме, относительно количества обрабатываемых данных (т.е., по сути, совокупность системной и пространственной эффективности) используется как мера вычислительной эффективности алгоритма.

Сложность алгоритма как характеристика непосредственно связана с его эффективностью, т.к. эффективный алгоритм требует приемлемое время исполнения и разумную ресурсоемкость.

Сложность возрастает при увеличении времени исполнения алгоритма и (или) задействованных ресурсов. Т.о., для одной и той же задачи более сложный алгоритм из нескольких характеризуется меньшей эффективностью.

Вычислительная сложность, составляющие:

- **временная сложность** – отражает временные затраты на реализацию алгоритма (время работы процессора ЭВМ);
- **ёмкостная сложность** – отражает объём требующейся алгоритму внутренней памяти ЭВМ.

[4] Здесь и далее будем обсуждать только алгоритмы решения вычислительных задач в форме компьютерных программ.

3.2. Эмпирический анализ вычислительной сложности

Можно обозначить два основных подхода к оценке вычислительной сложности алгоритма:

- **Эмпирический анализ** (он же экспериментальный или практический):
  - **практический метод** – прямой подсчёт времени работы алгоритма или количества выполненных команд;
  - **теоретический метод** – формальная оценка – вывод на основе анализа кода алгоритма формулы функциональной зависимости времени его работы (или количества выполненных команд) от количества входных значений – **функции роста**.
- **Асимптотический анализ** – оценка поведения функции роста при увеличении количества входных значений (объёма входа) в пределе до $+\infty$.

Практический метод характеризуется измеримыми параметрами:

- время исполнения (временная сложность) – во временных единицах (микро-, милли-, секунды), в количестве тактов процессора или в количестве выполненных команд;
- объем требуемой памяти (ёмкостная сложность) – в битах (байтах и производных единицах), минимальных аппаратных требованиях и пр.

Факторы, влияющие на результат в рамках практического метода оценки сложности алгоритма:

- особенности аппаратно-программной платформы:
  - характеристики оборудования (разрядность системы, тактовая частота процессора, частота шины, объём оперативной и сверхоперативной памяти, размер файла подкачки и пр.);
  - архитектура программной среды (вытесняющая многозадачность, алгоритм работы планировщика задач и архитектурные особенности операционных систем).
- язык программирования (уровень языка и транслятор – объём нативного кода на выходе);
- квалификация (опыт) программиста (***метод решения***).

В целом, практическая оценка не является абсолютным показателем эффективности (сложности) алгоритма.

Для определения временной сложности в программах на языке C++ можно

использовать возможности библиотек ctime или chrono (примеры их использования см. в листингах 1.1 и 1.2).

Листинг 1.1. Использование функции clock() модуля ctime (реальное время)

```
#include <ctime>
time_t begin = clock();
//вызов функции с алгоритмом…
time_t end = clock();
double time_spent = (double) (end - begin) / CLOCKS_PER_SEC;
```

Листинг 1.2. Использование библиотеки chrono (стабильное время)

```
#include <chrono>
auto begin = chrono::steady_clock::now();
//вызов функции с алгоритмом…
auto end = chrono::steady_clock::now();
auto elapsed_ms = chrono::duration_cast <chrono::milliseconds>(end - begin);
```

3.3. Теоретический подход к анализу сложности

Теоретический подход характеризует алгоритм без привязки к конкретному оборудованию, ПО и средствам реализации, т.о. устраняются факторы, которые влияют на оценку в практическом подходе.

Уход от влияния аппаратно-программных особенностей реальной вычислительной системы возможен на основе ***идеализированной модели*** вычислительной машины, в которой:

- однопроцессорная машина с памятью с произвольным доступом (англ. Random Access Memory, RAM);
- типы команд – арифметические, перемещения данных, управляющие;
- каждая команда выполняется за определённое фиксированное время (константное, часто для упрощения – единичное).

Время работы алгоритма складывается из элементарных операций (шагов), которые необходимо выполнить. Время выполнения различных строк псевдокода может отличаться, но пусть в нашей идеализированной модели ЭВМ одна и та же i-я строка всегда выполняется за константное время c_i.

Временную сложность при этом можно оценивать не в единицах времени, а в количестве операций (или тактах работы машины Тьюринга и прочих типах) с ***константным*** (или даже единичным) ***временем выполнения***.

Ёмкостная сложность (потребность в памяти) определяется объёмом данных – входных, промежуточных и выходных (или числом задействованных ячеек на

ленте машины Тьюринга и пр.).

Тогда из факторов, влияющих на оценку в целом вычислительной эффективности (сложности) в рамках теоретической оценки, останется только ***метод решения*** – например, тот или иной алгоритм сортировки.

Вычислительная и пространственная сложность – это внешняя мера эффективности. Но можно выделить и внутреннее качество – интеллектуальная сложность – определяет сложность разработки и понятность алгоритма.

3.4. Функция роста. Критические операции

Время работы алгоритма зависит от количества входных значений, например, чем больше размер файла для архивирования, тем дольше его будет сжимать архиватор.

Пусть n – количество входных данных (**объём входа**) для некоторого алгоритма (безотносительно к размеру в памяти одного входного значения).

Тогда T(n) – **функция роста**, показывающая рост времени работы алгоритма при увеличении объёма входных данных. **Скорость роста** (порядок роста) – функция наиболее высокого порядка, главный член полинома T(n).

Пусть рассматривается алгоритм проверки наличия числа в некотором массиве (например, числа 1). Если этот массив упорядочен по возрастанию, то проверяем до первого элемента, который равен или больше искомого. В этом случае T(n) < n и это лучший (рис. 3.б) и средний случаи для данного алгоритма.

Рисунок 3. Задача поиска числа в отсортированном массиве
а) в худшем случае; б) в лучшем случае

Однако в худшем случае (когда искомый элемент – последний в массиве) нужно просмотреть все элементы (рис. 3.а), тогда T(n) = n.

Для любого алгоритма **лучшим случаем** для него будет ситуация, при которой будет выполнено наименьшее количество команд (операций), в **худшем случае** – наибольшее.

Критическими (основными) называют **операции**, которые выполняются наиболее часто и время выполнения которых составляет основную часть общего времени выполнения алгоритма.

В ходе анализа эффективности алгоритма следует предварительно выявить эти критические операции (группы операций).

В наших примерах критическими будут операции ***перемещения данных***

(присваивания) и ***сравнения данных*** (в логических выражениях, условиях ветвлений и циклов).

Правила определения количества операторов в алгоритме:

1. Если в строке алгоритма расположена одна простая команда, то количество операторов равно 1.
2. Учитывается каждая команда в блоке, т.е. операторы в блоке суммируются.
3. В цикле с предусловием, в котором количество итераций n, количество сравнений в условии цикла равно n+1.
4. С учётом п.2, если тело цикла выполняется n раз, тогда количество операций в теле цикла после выполнения всех итераций равно количеству операторов тела цикла, умноженному на n[5].

3.5. Примеры вычислений функции роста

Пример 1. Найти среднее арифметическое всех положительных чисел в массиве A[n] (табл. 2).

Таблица 2. Псевдокод и анализ алгоритма нахождения среднего арифметического всех положительных чисел в массиве

Операторы кода	Количество выполнений оператора
sum←0	1
count←0	1
For i←1 to n do	n+1
If A[i]>0 then	n
sum←sum+A[i]	n
count←count+1 EndIf od	n
If count≠0 then return sum/count	1 1

Else
return -1
EndIf	

Здесь функция роста, выраженная в количестве команд, равна

T(n) = 1+1+(n+1)+n+n+n+1+1 = 4*n+5.

В этой формуле 4 и 5 – константы, рост будет определяться значением переменной n (константы при определении порядка роста в выражении игнорируются). Т.о. получаем ***линейную*** зависимость количества операций от n.

Пример 2. Алгоритм с вложенным циклом, параметр которого (значение,

[5] Более сложным случаем является анализ вложенного цикла, он будет рассмотрен позже.

определяющее число собственных итераций) не зависит от внешнего цикла.

Дана матрица размером n*m. Найти максимальный элемент (табл. 3).

Здесь худший случай, когда каждое следующее число больше текущего максимума: T(n) = 1+(n+1)+3n*m+n+1 = 3*n*m+2*n+3, т.е. сложность линейно зависит от количества элементов в матрице n*m.

Лучший случай, когда A[1,1] и есть наибольшее значение в матрице, тогда операторы в ветке if не выполняются, и сложность T(n) = 1+(n+1)+2n*m+n+1 = 2*n*m+2*n+3, т.е. сложность также линейно зависит от произведения n*m.

Таблица 3. Псевдокод и анализ алгоритма нахождения максимального значения в прямоугольной матрице

Операторы кода	Количество выполнений оператора
max←A[1,1]	1
For i←1 to n do	n+1
For j←1 to m do	n*(m+1)
If A[i,j]>max then	n*m
max←A[i,j]	n*m
EndIf od od	
return max	1

Средний случай: 2nm+2n+3 ≤ T(n) ≤ 3nm+2n+3, или 2nm ≤ T(n) ≤ 3nm.

Пример 3. Алгоритм с вложенным циклом с заранее неизвестным числом повторов (число его собственных итераций изменяется при каждой новой итерации внешнего цикла).

Пусть дан массив A из n чисел: 5 3 7 2 8 1 9. Все числа, меньшие A[1], разместить перед ним в возрастающем порядке, т.е. в результате массив станет следующим: 1 2 3 5 7 8 9. Псевдокод этого алгоритма представлен в табл. 4.

Таблица 4. Псевдокод и анализ алгоритма перестановки элементов в линейном массиве

Операторы кода	Время выполнения оператора	Количество выполнений оператора
tmp←A[1]	C_1	1
For i←2 to n do	C_2	n
If A[i]<=tmp then	C_3	n-1
j←i;	C_4	n-1
w←A[i]	C_5	n-1
while j>1 do	C_6	$\sum_{j=2}^{n} t_j$

A[j]←A[j-1]	C_7	$\sum_{j=2}^{n}(t_j-1)$
j←j-1 od	C_8	$\sum_{j=2}^{n}(t_j-1)$
A[1]=w	C_9	n-1
EndIf od		

Здесь t_j – это количество проверок условия в цикле while.

С учётом времени выполнения каждой команды (C_1, C_2, …, C_9) время работы алгоритма: $T(n) = C_1*1 + C_2*n + C_3*(n-1) + C_4*(n-1) + C_5*(n-1) + C_6*\sum_{j=2}^{n} t_j + C_7*\sum_{j=2}^{n}(t_j-1) + C_8*\sum_{j=2}^{n}(t_j-1) + C_9*(n-1)$. (1)

Тогда в лучшем случае (за первым элементом следуют все, большие его) не будет выполняться ветка if с циклом while:

$T(n) = C_1*1 + C_2*n + C_3*(n-1) = (C_2+C_3)*n + C_1 - C_3 = A*n + B$, где A и B – константные временные коэффициенты. Порядок роста времени в лучшем случае линейно зависит от количества элементов массива.

Приравняв временные коэффициенты к 1, можно выразить функцию роста через общее количество критических операций в алгоритме. Так, для лучшего случая: $T(n) = 2*n$. (2)

Худший случай для этого алгоритма, когда все элементы после первого меньше его. Тогда количество проверок условия в цикле while на каждой итерации внешнего цикла будет определяться счётчиком j: $t_j = j$, образуя арифметическую прогрессию {2, 3, …, n}.

Отсюда по формуле суммы арифметической прогрессии общее количество проверок условия будет: $\sum_{j=2}^{n} t_j = 1+2+3+\ldots+n = (1+2+3+\ldots+n) - 1 = \frac{(1+n)}{2}n - 1 =$
$= (n^2 + n)/2 - 1 = 0{,}5*n^2 + 0{,}5*n - 1$.

Подставив это в формулу (1), получим: $T(n) = C_1*1 + C_2*n + C_3*(n-1) +$
$+ C_4*(n-1) + C_5*(n-1) + C_6(n^2+n-2)/2 + C_7*(n^2-n)/2 + C_8*(n^2-n)/2 + C_9*(n-1) =$
$= A*n^2 + B*n + C$, т.е. скорость роста квадратичная.

Или в количестве операций: $T(n) = 2*n + 2*(n-1) + 0{,}5*n^2 + 0.5*n - 1 + (n^2 - n) + (n-1) = 1{,}5*n^2 + 4{,}5*n - 4$.

Средний случай в этом примере, когда числа после первого случайные. Зависимость времени от n вычислить, применяя индукцию, сложно.

Так как не требуется дополнительного массива для решения задачи, то требуется только один массив длиной n (ёмкостная сложность n).

ЛЕКЦИЯ 3
4. АСИМПТОТИЧЕСКИЙ АНАЛИЗ СЛОЖНОСТИ

4.1. Понятие асимптотической сложности

Как было сказано выше, T(n) является некоторой функцией от объёма входных данных n (см. рис. 4). Часто T(n) выражается полиномиальной, экспоненциальной или логарифмической функцией от n.

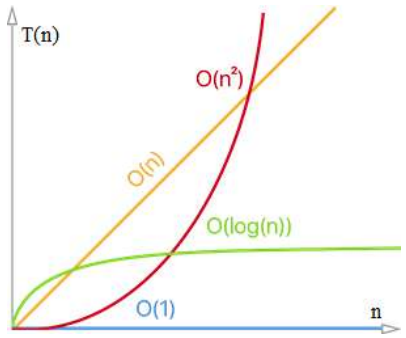


Рисунок 4. Поведение различных функций T(n)

Поведение T(n) в зависимости от увеличения n в пределе называют **асимптотической сложностью** алгоритма.

Алгоритм, более эффективный в асимптотическом смысле (т.е. рост в пределе меньше), будет более производительным для всех наборов входных данных, за исключением ***очень малых***. Шкалу классов сложности см. на рис. 5.

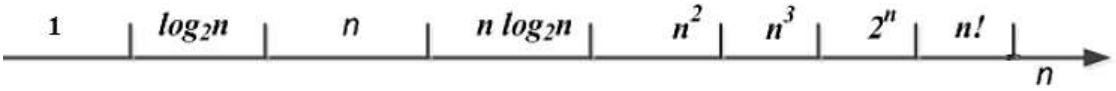


Рисунок 5. Шкала увеличения сложности алгоритмов с различными функциями

При анализе вычислительной сложности алгоритма не важно точное знание количества операций или времени работы (как это было в эмпирическом подходе). Более важным показателем является **скорость роста** количества выполняемых им операций при возрастании объёма входных данных (т.е. общий характер поведения алгоритма, а не его подробности).

Например, если для первого алгоритма $T_1(n)=12*n^2+500*n+148$, а для второго $T_2(n)=n^3$, то, начиная с некоторого n_0, при дальнейшем увеличении объёма входа скорость роста T_2 будет выше T_1, т.е. второй алгоритм признаётся менее эффективным.

Сложность алгоритма описывается порядком роста функции роста T(n), который чаще всего представляют в нотации O (о-большое)[6].

К примеру, сложность (время работы) алгоритма сортировки слиянием равна

[6] Подробнее об асимптотических нотациях см. ниже.

$O(n \cdot \log_2(n))$. То, что под знаком O – это часть полинома T(n), вносящая наибольший вклад в скорость роста.

Сложность алгоритма, время которого константно (не зависит от n) обозначается как O(1) (читается «О один»).

Типовые функции описания сложности (**классы сложности**) алгоритмов (рис. 5 и табл. 5).

Таблица 5. Сравнительные значения количества операций в алгоритмах разных классов сложности

	1	**$\log_2 n$**	**n**	**$n^* \log_2 n$**	**n^2**	**n^3**	**2^n**
1	1	0,0	1	0,0	1	1	2
2	1	1,0	2	2,0	4	8	4
5	1	2,3	5	11,6	25	125	32
10	1	3,3	10	33,2	100	1000	1024
15	1	3,9	15	58,6	225	3375	32768
20	1	4,3	20	86,4	400	8000	1048576
30	1	4,9	30	147,2	900	27000	1073741824
40	1	5,3	40	212,9	1600	64000	1099511627776
50	1	5,6	50	282,2	2500	125000	1125899906842620
60	1	5,9	60	354,4	3600	216000	1152921504606850000
70	1	6,1	70	429,0	4900	343000	1180591620717410000000
80	1	6,3	80	505,8	6400	512000	1208925819614630000000000
90	1	6,5	90	584,3	8100	729000	1237940039285380000000000000
100	1	6,6	100	664,4	10^4	10^6	1267650600228230000000000000000
10^3	1	10	10^3	$1{,}0*10^4$	10^6	10^9	2^{1000}
10^4	1	13	10^4	$1{,}3*10^5$	10^8	10^{12}	2^{10000}
10^5	1	17	10^5	$1{,}7*10^6$	10^{10}	10^{15}	2^{100000}
10^6	1	20	10^6	$2{,}0*10^7$	10^{12}	10^{18}	$2^{1000000}$

Самый малый порядок роста – константный (программы будут выполняться практически мгновенно вне зависимости от объёма входных данных.

С помощью алгоритмов, в которых количество выполняемых операций растет по экспоненциальному закону (2^n и n!), можно решать задачи лишь очень малой размерности.

Две функции f(x) и g(x) вещественного аргумента x **асимптотически равны** (f(x) ~ g(x)) при стремлении аргумента x к ∞ ($x \to \infty$), если $\lim\limits_{n\to\infty} \frac{f(x)}{g(x)} = 1$.

Функция f(x) = O(g(x)), т.е. f(x) есть O(g(x)) (O-большое от g от x) при стремлении аргумента x к ∞ ($x \to \infty$), если существует константа C такая, что $\lim\limits_{n\to\infty} |\frac{f(x)}{g(x)}| = C$.

Функция f(x) = o(g(x)), т.е. f(x) есть o(g(x)) (o-малое от g от x) при стремлении аргумента x к ∞ ($x \to \infty$), если $\lim\limits_{n\to\infty} |\frac{f(x)}{g(x)}| = 0$.

Виды асимптотических обозначений: θ (тета), O (о-большое), Ω (омега-большое), o (о-малое), ω (омега-малое)[7].

4.2. θ-обозначение асимптотической оценки

Для некоторой функции g(n) запись f(n)=θ(g(n)) обозначает множество функций {f(n), для которых: существуют положительные константы c_1, c_2, такие что $0 \le c_1 g(n) \le f(n) \le c_2\, g(n)$ для всех $n > n_o$}.

Функция f(n) принадлежит множеству θ(g(n)), если существуют константы $c_1>0$ и $c_2>0$, позволяющие заключить эту функцию в рамки между $c_1 g(n)$ и $c_2 g(n)$ для достаточно больших $n>n_o$.

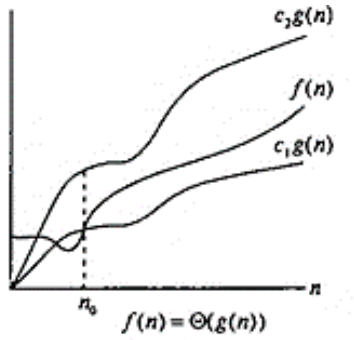


Рисунок 6. Асимптотически точная оценка функции f(n)

Т.о. в θ-обозначениях функция f(n) асимптотически ограничивается ***сверху и снизу***: для всех $n>n_o$ f(n) = g(n) с точностью до постоянного множителя (рис. 6). Т.е. g(n) – это аппроксимация для функции роста T(n), являющаяся:

1) асимптотически точной оценкой функции T(n) и

2) используется для оценки сложности алгоритма с функцией роста T(n) в худшем, лучшем и среднем случае.

Пример. Доказать, что асимптотическая оценка функции $T(n)=0{,}5*n^2 – 3*n$ есть $\theta(n^2)$ или $T(n)= \theta(n^2)$.

Для этого определим константы c_1, c_2 и n_0, для которых справедливо неравенство: $c_1*n^2 \le 0{,}5*n^2 – 3*n \le c_2*n^2$ для всех $n \ge n_0$.

Разделив неравенство на n^2, получим: $c_1 \le 0{,}5 – 3/n \le c_2$.

Правая часть $0{,}5 – 3/n \le c_2$ выполнится для всех $n \ge 1$, если выбрать $c_2 \ge 0.5$ (действительно, при $n\to\infty$ $3/n\to 0$).

Аналогично левая часть $c_1 \le 0{,}5 – 3/n$ выполнится для всех $n \ge 7$, если выбрать $c_1 \le 1/14$ (т.к. 0,5 – 3/7 = 1/14).

Тогда найдены $c_1 = 1/14$, $c_2 = 0.5$ и $n_0=7$, а, значит, по определению, $T(n)=\theta(n^2)$, ч.т.д.

[7] На практике по причинам, описанным ниже, используется чаще всего только O-нотация.

Константы можно выбрать и по-другому, но важны не сами константы, а то, что они существуют.

4.3. O-обозначение асимптотической оценки

Запись f(n)=O(g(n)) означает множество функций {f(n) таких, что: существуют положительная константа c и n_0 такие, что $0 \le f(n) \le c*g(n)$ для всех $n \ge n_0$}.

O-обозначения применяются, когда нужно указать ***верхнюю границу*** функции роста с точностью до постоянного множителя c. Т.е. для всех $n>n_o$ функция f(n) не превышает значения функции g(n) с точностью до постоянного множителя (рис. 7).

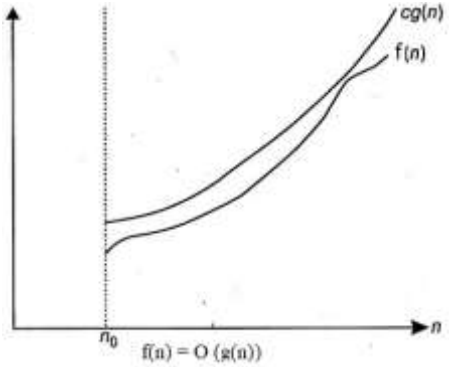


Рисунок 7. Асимптотическая оценка функции f(n) сверху

Из $f(n) = \theta(g(n))$ следует $f(n) = O(g(n))$, т.к. θ-обозначения более сильные, чем O-обозначения: $\theta(g(n)) \subset O(g(n))$.

Т.к. T(n) – это функция, то она принадлежит некоторому множеству функций. Для аппроксимации (приближения) T(n) подбирается простая g(n) и константа C, так, что $C \cdot g(n)$ превышает T(n), по мере того как n значительно растёт, т.е. для больших n поведение T(n) ограничивается $C \cdot g(n)$.

T(n) имеет порядок роста O(g(n)), если имеется константа c и счетчик n_0, такие что $0 < T(n) \le c \cdot g(n)$, для $n>=n_0$.

На практике границы сверху достаточно для оценки сложности алгоритма в худшем случае.

4.4. Ω-обозначение асимптотической оценки

Запись f(n)=Ω(g(n)) обозначает множество функций {f(n): таких, что существуют константа c>0 и n_0, для которых $0 < c \cdot g(n) \le f(n)$ для всех $n \ge n_0$}.

Т.е. для всех n, лежащих справа от n_0, значения функции $f(n) \ge c \cdot g(n)$.

Суть Ω-нотации: существует константа c такая, что для бесконечного числа значений $n>n_0$ выполняется неравенство $T(n) \ge c \cdot g(n)$.

В Ω-обозначениях функции роста даётся её асимптотическая ***нижняя граница*** с точностью до постоянного множителя (рис. 8).

Нижней границы достаточно для оценки времени работы алгоритма в наилучшем случае.

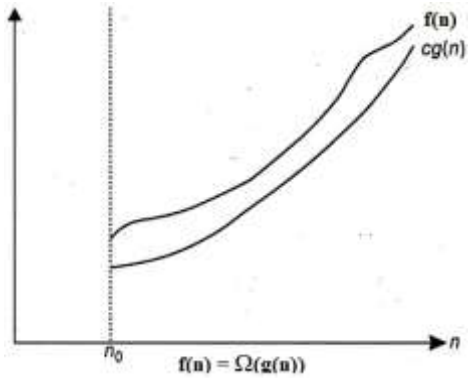


Рисунок 8. Асимптотическая оценка функции f(n) снизу

Например, время работы алгоритма сортировки вставкой находится в пределах между $\Omega(n)$ и $O(n^2)$, асимптотически точную оценку для этого алгоритма получить сложно.

Теорема [4]. Для двух функций f(n) и g(n) соотношение $f(n) = \theta(g(n))$ выполняется тогда и только тогда, когда $f(n) = O(g(n))$ и $f(n) = \Omega(g(n))$.

Доказательство непосредственно следует из вышеприведенных определений. На практике теорема применяется для определения асимптотически точной оценки с помощью асимптотических верхней и нижней границ.

Пример: если a>0, b и c – константы, то из соотношений $an^2 + bn + c = O(n^2)$ и $an^2 + bn + c = \Omega(n^2)$ непосредственно следует, что $an^2 + bn + c = \theta(n^2)$.

4.5. Асимптотические нотации в уравнениях и неравенствах

Как интерпретируются формулы типа $T(n)=O(n^3)$ и $5n^2+n+1=5n^2+\theta(n)$?

Если в правой части формулы находится только асимптотическое обозначение ($T(n)=O(n^3)$), то знак равенства используется для указания ***принадлежности множеству***: $T(n)\in O(n^3)$.

В другой ситуации они рассматриваются как подставляемые взамен некоторой функции, имя которой ***не имеет значения***.

Пример: $5n^2+n+1=5n^2+\theta(n)$ означает, что $5n^2+n+1=5n^2+f(n)$, где f(n) — функция из множества $\theta(n)$. Здесь $f(n) = n +1$, и она принадлежит множеству $\theta(n)$.

Например, время работы алгоритма сортировки ***методом слияния*** в наихудшем случае было выражено в виде рекуррентного уравнения: $T(n) = 2*T(n/2) + \theta(n)$.

Если интересует только асимптотическое поведение T(n), то нет смысла точно выписывать все слагаемые низших порядков, подразумевается, что все они включены в безымянную функцию, обозначаемую θ(n).

Если асимптотические обозначения появляются в левой части уравнения, например, $2n^2+\theta(n)=\theta(n^2)$, то уравнения интерпретируются в соответствии с правилом: при любом выборе безымянных функций, подставляемых вместо асимптотических обозначений в левую часть уравнения, можно выбрать и подставить в правую часть такие безымянные функции, что уравнение будет правильным.

В примере: для любой функции $f(n)\in\theta(n)$ существует некоторая функция $g(n)\in\theta(n^2)$, такая что $2n^2+f(n)=g(n^2)$ для всех n. Т.е. правая часть уравнения предоставляет меньший уровень детализации, чем левая.

4.6. o-обозначение

Верхняя O-граница может описывать асимптотическое поведение функции с разной точностью. Так, граница $5n^2 = O(n^2)$ дает точное представление об асимптотическом поведении функции, а граница $5n^2=o(n^3)$ гораздо ***менее точна***.

Если верхняя граница не является асимптотически точной оценкой функции, то применяются o-обозначения.

Формальное определение множества o(g(n)):

o(g(n)) = {f(n) : для любой константы $c>0$ существует $n_0>0$ такое, что $0 \le f(n) < c(g(n))$ для всех $n\ge n_0$}.

Определения O-обозначений и o-обозначений похожи, отличие: определение $f(n)=O(g(n))$ ограничивает функцию f(n) неравенством $0 \le f(n) \le c*g(n)$ лишь для некоторой константы $c>0$, а определение $f(n) = o(g(n))$ ограничивает её неравенством $0 \le f(n) < c*g(n)$ для любой константы $c>0$.

Интуитивно понятно, что в o-обозначениях, если n стремится к бесконечности, то f(n) пренебрежимо мала по сравнению с функцией g(n), т.е.

$$\lim_{n\to\infty} \left| \frac{f(x)}{g(x)} \right| = 0.$$

Например, $2n=o(n^2)$, но $2n^2\neq o(n)$.

4.7. ω-обозначение

По аналогии, ω-обозначения соотносятся с Ω-обозначениями так же, как o-обозначения с O-обозначениями.

ω-обозначениями указывается нижний предел, не являющийся

асимптотически точной оценкой.

Один из возможных способов определения для ω-обозначения:

f(n) = ω(g(n)) тогда и только тогда, когда g(n) = o(f(n)).

Формально, ω(g(n)) («омега-малое от g от n») определяется как множество ω (g (n)) = {f (n): для любой положительной константы c существует n_0>0, такое что $0 \leq c*g(n) < f(n)$ для всех $n > n_0$}.

Например, $n^2/2 = \omega(n)$, но $n^2/2 \neq \omega(n^2)$.

Соотношение f(n) = ω(g(n)) подразумевает, что

$$\lim_{n\to\infty} \left| \frac{f(x)}{g(x)} \right| = \infty,$$

если этот предел существует. Функция f(n) становится сколь угодно большой по сравнению с функцией g(n), если n стремится к бесконечности.

4.8. Применение асимптотических оценок для оценки сложности

Использование асимптотических оценок отражено в табл. 6.

Таблица 6. Использование асимптотических оценок

Обозначение	Граница	Рост
θ	Нижняя и верхняя границы, точная оценка	Равно
O	Верхняя граница	Меньше или равно
o	Верхняя граница, неточная оценка	Меньше
Ω	Нижняя граница	Больше или равно
ω	Нижняя граница, неточная оценка	Больше

Классы сложности алгоритмов:

- f(n) = O(1) константа
- f(n) = O(log(n)) логарифмический рост
- f(n) = O(n) линейный рост
- f(n) = O(n·log(n)) квазилинейный рост
- f(n) = O(n·m) полиномиальный рост
- $f(n) = O(2^n)$ экспоненциальный рост.

Особенности применения типовых функций для оценки сложности алгоритмов см. табл. 7.

Таблица 7. Применение типовых функций для оценки сложности алгоритмов

Порядок	Виды алгоритмов	Пример алгоритмов
O(1)	Нет зависимости от n, алгоритм выполняется за конечное время	Обращение к значению в массиве по индексу

O(n)	Линейное время	Линейный поиск
$O(n^2)$	Квадратичная сложность, используется для обработки небольших наборов данных	Простые сортировки
$O(n^3)$	Высокая сложность	Алгоритм Уоршелла
$O(\log_2 n)$	Логарифмическая сложность	Бинарный поиск
$O(n*\log_2 n)$	Квазилинейное время	Быстрая сортировка (схема Хоара)
$O(2^n)$	Экспоненциальная сложность, только для малых n	Рекурсивный поиск n-го числа Фибоначчи

4.9. Свойства асимптотических сравнений

1. Транзитивность.

   Из $f(n) = \beta(g(n))$ и $g(n) = \beta(h(n)) \Rightarrow f(n) = \beta(h(n))$, где $\beta \in \{\theta, O, \Omega, o, \omega\}$

2. Рефлексивность

   $f(n) = \gamma(f(n))$, где $\gamma \in \{\theta, O, \Omega\}$

3. Симметричность.

   $f(n) = \theta(g(n)) \Leftrightarrow g(n) = \theta(f(n))$.

4. Перестановочная симметрия.

   $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$,

   $f(n) = o(g(n)) \Leftrightarrow g(n) = \omega(f(n))$.

4.10. Правила вычисления асимптотических оценок

Пусть f, g – это функции, k – константа. Тогда:

1. O(k*f) = O(f) – постоянные множители не имеют значения при определения порядка сложности.

2. O(f*g) = O(f)*O(g) или O(f/g) = O(f)/O(g) – порядок сложности произведения двух функций равен произведению их сложностей.

3. O(f+g) = max(O(f), O(g)) – порядок сложности суммы двух функций определяется как порядок доминанты первого и второго слагаемых (из суммы двух функций выбирается лишь одна с наибольшим порядком сложности).

Примеры:

1. O(5n) = O(n).

2. O(8*n*n) = O(8*n)*O(n) = O(n)*O(n) = O(n*n) = $O(n^2)$.

3. $O(n^5+n^2) = O(n^5)$.

Основание логарифма при оценке сложности можно не указывать.

Доказательство:

Пусть есть $\log_2 n$, тогда $\log_2 n = \log_3 n/\log_3 2$, т.е. $O(\log_2 n) = O(\log_3 n)$, что

справедливо и для любого другого основания ($\log_3 2$ как константа не учитывается).

Асимптотическое время выполнения алгоритма аналогична эмпирической оценке сложности:

1. Время вычисления любого выражения (в т.ч. проверка условия цикла или ветвления), оператора присваивания – O(1).

2. Время выполнения блока определяется по правилу суммы.

3. Время условных операторов состоит из:

- времени выполнения логического выражения – O(1);
- максимального времени веток then или else, например:

Пусть время блока then имеет порядок O(n), а блок else $O(n^2)$, тогда оператор if имеет порядок $O(1) + \max(O(n), O(n^2)) = O(n^2)$.

4. Время выполнения цикла равно сумме времени всех итераций цикла, причём каждая итерация равна сумме времени выполнения тела цикла и времени вычисления условия цикла, которое всегда на 1 больше.

Пример 1.

```
x=0;
for (int i=0;i< 500;i++)
  x=x+i;
```

T(n) = 1+501+501+500 = O(1), т.к. не зависит от n – количества входных данных.

Пример 2.

```
while (n>1)
  {  n=n/2; }
```

T(n) равно количеству делений.

Если n – это степень 2, и n становится равным 1 на k-ом шаге, то найти k (сколько раз отработал цикл) можно из соотношения $n = 2^k$, из которого $k = \log_2 n$.

Тогда T(n) = O(log n).

ЛЕКЦИЯ 4
5. ПРОСТЫЕ АЛГОРИТМЫ ВНУТРЕННИХ СОРТИРОВОК

5.1. Сортировки: понятие, задача, виды

Сортировка (sorting problem) – это упорядочение значений в некоторой последовательности данных (например, в массиве) в соответствии с заданным критерием – убывание, возрастание, неубывание, невозрастание (рис. 9).

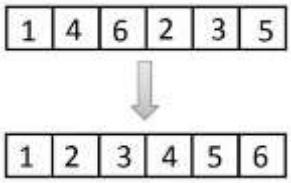


Рисунок 9. Входной и выходной массив в задаче сортировки

Поле элемента последовательности, служащее критерием порядка, называется **ключом**.

Цель сортировки – повышение эффективности дальнейшей обработки данных (например, в задачах поиска). Это одна из наиболее важных и востребованных задач при изучении алгоритмов[8].

Методы в различных алгоритмах сортировок во многом зависят от объёма и структуры обрабатываемых данных (т.е. от предметной области применения).

Алгоритмам **внутренней сортировки** достаточно для решения задачи ресурса внутренней (оперативной) памяти.

Внешние сортировки применяются для больших объёмов данных во внешней памяти (например, в файлах), когда внутренней памяти заведомо недостаточно.

Внутренние сортировки допускают более гибкие возможности для структурирования и манипуляции данными за счёт многократного обращения к значениям в последовательности (быстродействие внутренней памяти это позволяет).

Важная характеристика алгоритма сортировки – **устойчивость** алгоритма – когда не меняется взаимное расположение значений с одинаковыми ключами.

Критерии оценки алгоритма:

- время – вычислительная сложность – быстродействие алгоритма;
- память – ёмкостная сложность – необходима ли дополнительная память сверх той, которая занята исходным массивом.

[8] Задача сортировки была одной из первых задач, решённых на ЭВМ ENIAC (Джон фон Нейман, алгоритм слиянием).

Формализация задачи сортировки.

Пусть имеется множество элементов a_1, a_2, …, a_n, в котором каждый элемент имеет непустое поле, называемое ключом.

Задача сортировки – выполнить перестановку элементов a_i в соответствии со значением ключей a_{k1}, a_{k2}, …, a_{kn} в таком порядке, что при заданной функции упорядочения f справедливо отношение $f(a_{k1}) \le f(a_{k2}) \le \ldots \le f(a_{kn})$.

Внутренние сортировки можно разделить на две группы – сортировки на месте (on place) и основанные на цифровых свойствах ключей (рис. 10).

Рисунок 10. Классификация алгоритмов внутренних сортировок [4]

Класс сортировок на цифровых свойствах ключей, имеют сложность O(n), но используют дополнительную память для вспомогательного массива, списка или очереди.

Сортировки «на месте» (on place) могут быть для линейных и нелинейных структур данных. Выделяют простые, усовершенствованные и быстрые алгоритмы.

Простые алгоритмы используются при малом n, оцениваются асимптотически как $O(n^2)$ и являются устойчивыми. ***Улучшенные*** и ***быстрые*** варианты могут показать сложность до $O(n \cdot \log n)$ и являются неустойчивыми.

Простые алгоритмы для малых n могут работать даже быстрее усовершенствованных, но используются, в основном, только в учебных целях.

Есть два типа основных (критических) операций с сортируемыми данными – ***сравнение*** ключей двух элементов и ***перемещение*** элемента.

Поэтому естественной мерой практической оценки временной сложности алгоритма сортировки массива a[n] являются количество сравнений C(n) и количество перемещений элементов M(n), т.е. T(n) = C(n) + M(n).

Оценка проводится по количеству основных операций в лучшем, среднем и

худшем случаях.

5.2. Сортировка простым обменом (Exchange Sort)

Сортировка простым обменом (Exchange Sort, пузырьковая) – очень известный алгоритм, который изучают ещё в рамках школьного курса информатики.

В цикле, с начала массива до границы рассматриваемой области массива (в первом проходе до n-1, уменьшаясь с каждым разом на 1) соседние значения попарно сравниваются между собой и в случае, если предшествующий элемент больше последующего, меняются местами (для случая сортировки по возрастанию).

После первого прохода всего массива элемент с максимальным значением займет место последнего элемента – свою окончательную позицию, потому исключается из дальнейшего рассмотрения.

Алгоритм завершится, когда: а) рассматриваемая область станет пустой, или б) когда на очередном проходе по массиву не будет произведено ни одной перестановки (условие Айверсона).

Пример работы сортировки в худшем случае:

Вход: 5 4 3 2 1 0.

После соответствующих проходов по массиву:

1) 4 3 2 1 0 5
2) 3 2 1 0 4 5
3) 2 1 0 3 4 5
4) 1 0 2 3 4 5
5) 0 1 2 3 4 5 – выход.

Псевдокод и анализ алгоритма простого обмена представлен в табл. 8[9]:

Таблица 8. Псевдокод и анализ алгоритма пузырьковой сортировки

function ExchangeSort(a):	Количество команд
for i ← 0 to (n - 2) do	n
for j ← 1 to (n - 1 - i) do	$\sum_{j=1}^{n-1} t_j = n + n - 1 + \cdots + 2 = (n-2) \cdot \frac{n-1}{2} =$ $= (n^2 - n - 2n + 2)/2 = 0.5n^2 - 1.5n + 1$
if (a[j] > a[j + 1]) then	$\sum_{j=1}^{n-1} t_j - 1 = \sum_{j=1}^{n-1} t_j - (n-1) = 0.5n^2 - 2.5n + 2$
swap(a[j], a[j + 1]) endif od	$3 \cdot \sum_{j=1}^{n-1} (t_j - 1) = 1.5\, n^2 - 7.5n + 6$

[9] Напомним, что в псевдокоде нумерация элементов начинается с 1.

od	

Вне зависимости от исходной упорядоченности исходного массива всегда выполняются операции сравнения C(n) (внешний, внутренний циклы и условие ветвления).

Операций обмена M(n) нет в лучшем случае.

По мере ухудшения упорядоченности исходного массива число перемещений элементов приближается к числу сравнений (каждый неупорядоченный элемент требует 3 операции обмена).

Сложность алгоритма $O(n^2)$ операций сравнения и $O(n^2)$ операций перемещения, т.о. в целом сложность алгоритма $O(n^2)$, что не является хорошим результатом. На больших n алгоритму потребуется очень много времени (квадратичная сложность).

5.3. Условие Айверсона

Массив может оказаться отсортированным задолго до последнего прохождения по нему.

Проверка отсортированности массива добавит операций сравнения, но в целом позволит уменьшить временную сложность. Способ проверки – переменная-флаг, отслеживающая факт обмена (в примере ниже – t).

Условие Айверсона: если в очередном проходе сортировки при сравнении элементов не было сделано ни одной перестановки, то множество считается упорядоченным.

Псевдокод пузырьковой сортировки с условием Айверсона может выглядеть следующим образом:

```
function ExchangeSort(a):
 i ← 0
 t ← true
 while t do
  t ← false
  for j ← 1 to n - 1 - i do
   if (a[j] > a[j + 1]) then
    swap(a[j], a[j + 1])
    t ← true
    endif
   od
 i ← i + 1
 od
```

5.4. Сортировка методом простого выбора (Selection Sort)

На первом шаге неупорядоченная часть – весь массив длиной n. В

неупорядоченной части выбирается элемент с минимальным значением, который обменивается местами с первым элементом этой части массива и исключается из дальнейшей сортировки (включается в уже упорядоченную часть массива).

Затем выбирается минимальный элемент среди оставшихся n-1 элементов исходного массива, обменивается местами начальным элементом неупорядоченной части и также исключается из сортировки.

Процесс продолжается, пока в неотсортированной части массива не окажется единственный элемент, который считается упорядоченным.

Пример работы сортировки в худшем случае:

Вход: 5 4 3 2 1 0.

После соответствующих проходов по массиву:

1) 0 4 3 2 1 5
2) 0 1 3 2 4 5
3) 0 1 2 3 4 5
4) 0 1 2 3 4 5
5) 0 1 2 3 4 5 – выход.

На каждом шаге алгоритма текущий минимальный элемент записывается на i-е место исходного массива (i = 1, 2, …, n), а элемент с i-ого места записывается на его место. В приведённом примере на 4 и 5 проходе внешнего цикла i-й элемент является минимальным, поэтому перемещений не происходит.

Это простейший алгоритм, реализующий стратегию «грубой силы», т.е. полный перебор элементов.

Алгоритм можно записать на псевдокоде в виде процедуры, единственным параметром которой является целочисленный массив a[n] (табл. 9).

Таблица 9. Псевдокод и анализ алгоритма сортировки выбором

Операторы кода	Количество выполнений оператора
SelectionSort(a,n)	
For i←1 to n-1 do	n
imax←i	n-1
For j←i+1 to n-1 do	$(n^2-n)/2$
If a[j]<=a[imax] then	$(n^2-n)/2-(n-1)$
imax←j	$(n^2-n)/2-(n-1)$
EndIf od	
swap(a[i],a[imax])	3*(n-1)
od	

В худшем случае каждый раз выполняются сравнение, перемещение (внутри

вложенного цикла) и swap во внешнем цикле, причём количество обменов M(n) = 3(n - 1), т.к. на каждый обмен требуется 3 операции перемещения.

В лучшем случае (исходный массив уже упорядочен) потребуется поменять местами n-1 элементов.

В среднем число операций обмена равно $n \cdot (\log n + y)$, где y = 0,577216 (константа Эйлера).

Количество операций сравнения здесь совпадает с методом простого обмена, но операций перемещения в среднем случае будет значительно меньше.

Метод выбора эффективней сортировки простым обменом по критерию M(n), т.к. при сортировке выбором каждый элемент передвигается в массиве не более чем 1 раз.

Сложность алгоритма $O(n^2)$ для операций сравнения и O(n) для операций перемещения, т.о. сложность алгоритма $O(n^2)$, что не является хорошим результатом. В целом можно констатировать, что для большого n сортировка выбором будет работать слишком медленно, она подходит лишь для небольших массивов.

5.5. Сортировка методом простой вставки (Insertion Sort)

Идея: последовательное добавление элементов из неотсортированной к уже отсортированной части массива с сохранением в ней упорядоченности.

Эта сортировка обладает естественным поведением, т.е. алгоритм работает быстрее для частично упорядоченного массива. Алгоритм устойчив – элементы с одинаковыми ключами не переставляются.

Отсортированной частью массива по умолчанию считаем начальный элемент. На первом шаге следующий второй элемент (он же первый элемент неотсортированной части) сравнивается с первым (крайним справа элементом отсортированной части).

Если он больше первого, то остаётся на своём месте и просто включается в отсортированную часть, а остальная (неотсортированная) часть массива остаётся без изменений.

Если же он меньше первого, то последовательно сравнивается с элементами в упорядоченной части (начиная с наибольшего – крайнего справа), пока не встретится элемент меньший. После чего происходит вставка на место первого, большего его в отсортированной части, со сдвигом всех элементов правее в отсортированной части на одну позицию вправо.

Этот процесс повторяется аналогично для всех последующих элементов

неотсортированной части исходного массива.

Пример работы сортировки в худшем случае:

Вход: 5 4 3 2 1 0.

После соответствующих проходов по массиву:

1) 4 5 3 2 1 0
2) 3 4 5 2 1 0
3) 2 3 4 5 1 0
4) 1 2 3 4 5 0
5) 0 1 2 3 4 5 – выход.

Псевдокод сортировки вставками представлен в табл. 10.

Очевидно, что в худшем случае время будет $O(n^2)$, в лучшем $O(n)$.

Таблица 10. Псевдокод и анализ алгоритма сортировки вставками.

Операторы кода	Количество выполнений оператора
InsertionSort(a,n)	
For i←2 to length(a) do	n
key←a[j]	n-1
i←j-1	n-1
While i>0 и a[i]>key do	$\sum_{j=2}^{n} t_j$
a[i+1]←a[i]	$\sum_{j=2}^{n} (t_j - 1)$
i←i-1	$\sum_{j=2}^{n} (t_j - 1)$
od	
a[i+1]←key	n-1
od	

В качестве упражнения выведите точную функцию роста T(n) в худшем и лучшем случаях.

6. УЛУЧШЕННЫЕ ВАРИАНТЫ СОРТИРОВОК

6.1. Шейкерная сортировка (Cocktail Sort)

Улучшить сортировку – т.е. уменьшить количество сравнений C и/или перемещений M значений в массиве. Рассмотрим примеры подобных сортировок.

Шейкерная сортировка (также называется коктейльная, перемешиванием, двунаправленная пузырьковая – Shaker Sort, Shuffle Sort, Shuttle Sort) – это улучшение ***обменной*** сортировки (пузырька), в том числе за счёт:

- ***условия Айверсона***: все элементы с индексами, превышающими k (индекс элемента, участвовавшего в последнем обмене), уже упорядочены;

- ***чередования направлений*** просмотра массива, что обеспечивает более быстрое перемещение элементов на свои места.

Листинг 6.1. Псевдокод шейкерной сортировки

```
SheikerSort(a)
l←0; //индекс левого элемента
r ← n-1 ; //индекс правого элемента
k ← 0;
while(l<r)// проходы
do
//слева направо
i ← l;
  while(i<r)
  do
    if(a[i]>a[i+1])
          swap(a[i],a[i+1]);
          k ← i;
        endIf
    i ←i+1;  // увеличение i
  od

//справа налево
l ← k+1;
  while(i>r)
      do
     if(a[i]<a[i-1])
           swap(a[i],a[i+1]) ;
           k ← i;
       endIf
  od
  r ← r-1;//уменьшение r
od
```

Сложность остаётся квадратичной. Количество операций сравнения не изменяется (минимум n-1), количество перемещений уменьшится не значительно.

По Кнуту [3], среднее число проходов получается $n-k_1$, среднее число сравнений – $\frac{1}{2}(n^2-n(k_2+\ln(n)))$.

Метод во всех случаях хуже, чем вставка и выбор, его имеет смысл использовать только для почти упорядоченных массивов.

Принципиально иной метод основан на выборе наименьшего ключа среди n значений, затем среди n-1 и т.д. Улучшением станет снижение C и M за счёт извлечения большей информации за каждый проход.

Например, n/2 сравнений позволят выбрать наименьший ключ из каждой пары элементов, следующие n/4 сравнения позволят выбрать наименьшие из пар наименьших и т.д. Тогда за n-1 сравнение можно построить дерево выбора.

Реализация усовершенствования - турнирная (***пирамидальная***) сортировка.

6.2. Сортировка Шелла (Shell sort)

Это улучшенный вариант сортировки ***вставками***.

Сортируются подгруппы (подмножества) элементов массива, причём в подгруппе элементы идут не последовательно, а равномерно выбираются с некоторой дельтой (d-смещением) по индексу. Смещение методично уменьшается, пока не составит 1.

Первоначальные проходы – грубые, позволяют сгруппировать малые значения слева, большие – справа. Т.о. достигается неполная упорядоченность.

При смещении 1 метод Шелла естественным образом трансформируется в метод вставки, который в условиях неполной упорядоченности показывает свою наилучшую эффективность (окупая временные затраты на предсортировку).

Описание сортировки Шелла (листинг 6.2):

На каждом шаге сравниваются элементы массива, отстоящие на d-смещение d =n/2, на каждом следующем шаге уменьшается вдвое, т.е. $d=n/2^k$, где k (количество смещений) от наибольшего $\log_2 n$ до 1. Если первое значение в паре больше, происходит обмен.

Например, для массива из 8 элементов на первом шаге будут сравниваться элементы: 1 и 5, 2 и 6, 3 и 7, 4 и 8.

Дополнительная проверка $j \geq 0$ нужна для предотвращения выхода за пределы массива и в некоторой степени ухудшает эффективность алгоритма.

В целом, вычислительная сложность в худшем случае $O(n^2)$.

Листинг 6.2. Псевдокод сортировки Шелла

```
d← n / 2; //инициализация смещения половиной размера массива
while (d >1) do   // пока смещение >1
  for i ← 0 to d  do
    j ← i;
    //как в алгоритме вставки, только сравниваем i-ый и j+d
     while(j>=0 and  a[j] > a[j + d])  do
        h ← a[j];
        a[j] ← a[j + d];
        a[j + d] ← h;
        j ← j-1;
     od
  od
  d ← d/ 2;
od
```

Среднее время работы зависит от длин d-смещений. Возможны следующие подходы к выбору d:

- смещение **Седжвика** – $O(n^{4/3})$ в худшем и $O(n^{7/6})$ в среднем случаях:

  Значение смещения, записываемого в элемент массива d, вычисляется по формуле:

$$\mathbf{d}[i] = \begin{cases} 9*2^{i} - 9*2^{i/2} + 1 \text{ при i - четном} \\ 8*2^{i} - 6*2^{(i+1)/2} + 1 \text{ при i нечетном} \end{cases}$$

  Остановить создание и заполнение массива d необходимо на значении d[i-1], если 3*d[i] > n (больше размера массива).

- смещение **Кнута** – в среднем случае до O(n):

  1) Вариант 1:

  1, 4, 13, 40, 121,… где $d_{i-1} = 3 \cdot d_i + 1$, $d_k=1$ и $k=\log_3 n - 1$.

  2) Вариант 2:

  1, 3, 7, 15, 31,… где $d_{i-1}=2 \cdot d_i+1$, $d_k=1$ и $k=\log_2 n - 1$.

  Следует избегать последовательностей степеней двойки – исследования показывают снижение эффективности сортировки [3].

- смещение Хиббарда – $O(n^{3/2})$;

- смещение Пратта – $O(n \cdot \log_2 n)$.

ЛЕКЦИЯ 5
7. БЫСТРЫЕ (УСКОРЕННЫЕ) СОРТИРОВКИ

7.1. Простое слияние

Ускоренные варианты сортировок:

Слиянием: **простое слияние** (временная сложность O(n log(n)), ёмкостная сложность 2n) и **естественное слияние**.

Обменом: **быстрая сортировка** (Хоара).

Рассмотрим подробнее сортировку ***простым слиянием***.

На вход подаётся массив A [1..n], содержащий последовательность из n сортируемых чисел. Сортируемый массив разбивается на две части примерно одинакового размера. Каждая из получившихся частей сортируется отдельно, например, тем же самым алгоритмом. Два упорядоченных подмассива половинного размера соединяются в один.

Т.о. задача разбивается на несколько более простых, подобных исходной, а сами простые задачи решаются ***рекурсивно***. Полученные решения комбинируются для получения окончательного решения. Этапы решения:

А. ***Разделение***:

Сортируемая последовательность, состоящая из n элементов, разбивается на две меньшие последовательности, каждая из которых содержит n/2 элементов.

Б. ***Рекурсивное решение***:

Сортировка обеих вспомогательных последовательностей методом слияния.

В. ***Комбинирование*** (слияние):

Слияние двух отсортированных последовательностей для получения окончательного результата.

Рекурсия достигает своего нижнего предела, когда длина сортируемой подпоследовательности становится равной 1 (массив единичной длины считается упорядоченным).

Рекурсивное решение:

Итак, исходный массив A: 8 7 6 5 4 3 2 1 нужно сортировать по возрастанию

Разбиение:

B: (8 7 6 5) - (8 7) (6 5) - (8) (7) (6) (5)

C: (4 3 2 1) - (4 3) (2 1) - (4) (3) (2) (1)

A:(1 5) (2 6) (3 7) (4 8) – слияние ***по одному элементу*** из B и C в упорядоченные пары;

Разбиение:

B: (1 5) (2 6)

C: (3 7) (4 8)

Сливаем в A упорядоченные пары, выбирая ***по два элемента*** из B и C;

A: (1 3 5 7) (2 4 6 8) – упорядоченные четверки

Разбиение:

B: (1 3 5 7)

C: (2 4 6 8)

Сливаем ***четверки*** и получаем A: 1 2 3 4 5 6 7 8 – упорядоченную восьмерку – окончательный результат.

Вспомогательный алгоритм разбиения и слияния:

```
Merge(A,p,q,r) разделение и слияние
1  n1 ← q - p + 1
2  n2 ← r - q
3  Создаем массивы L[1. .n1 +1] и R[1.. n2 +1] и разделяем на две части
4   for i ← 1 to n1
5       do L[i] ← A[p+i-1] od
6   for j ← 1 to n2
7       do R[j] ← A[q+j] od
8   L[n1 +1] ← ∞
9   R[n2 +1] ← ∞
10   i ← 1
11   j ← 1
// сливаем упорядочивая
12  for k ← p to r
13  do  if  L[i] ≤ R[j]
14          then A[k] ← L[i]
15                  i ← i + 1
16           else   A[k] ← R[j]
17                  j ← j + 1
        endif
  od
```

Здесь A - массив, p,q,r – индексы элементов массива, такие, что $p \le q < r$.

В строке 1 вычисляется длина n_1 подмассива A[p..q], в стр. 2 — длина n_2 подмассива A[q+1..r]. Предполагается, что элементы подмассивов A[p..q] и A[q+1..r] упорядочены.

В строке 3 создаются массивы L ("левый") и R ("правый"), длины которых соответственно n_1+1 и n_2+1.

В цикле в строках 4 и 5 подмассив A[p..q] копируется в массив $L[1..n_1]$.

В цикле в строках 6 и 7 подмассив A[q+1..r] копируется в массив $R[1..n_2]$.

В строках 8 и 9 последним элементам массивов L и R присваиваются

барьерные значения.

Эти два подмассива сливаются в один отсортированный (с заменой текущих элементов подмассива A[p..r]).

Доказательство ***корректности*** алгоритма простого слияния:

Инвариант цикла: перед каждой итерацией цикла (строки 12-17), подмассив A[p..k-1] содержит k-p наименьших элементов массивов $L[1..n_1+1]$ и $R[1..n_2+1]$ в отсортированном порядке. И, элементы L[i] и R[j] являются наименьшими элементами массивов L и R, которые еще не скопированы в массив A.

Инициализация:

Перед первой итерацией цикла k = p, поэтому подмассив A [p..k-1] пуст. Он содержит k-p = 0 наименьших элементов массивов L и R. Поскольку
i = j = 1, элементы L[i] и R[j] – наименьшие элементы массивов L и R, не скопированные обратно в массив A.

Сохранение:

1) Если $L[i] \leq R[j]$, тогда L[i] - наименьший элемент, не скопированный в A. Т.к. в подмассиве A[p..k-1] содержится k-p наименьших элементов, то после выполнения стр. 14 (значение L[i] присваивается в A[k]) в подмассиве A[p..k] будет содержаться k-p+1 наименьший элемент.

При увеличении параметра цикла k и значения i (строка 15) инвариант восстанавливается перед следующей итерацией.

2) При L[i]>R[j] в строках 16 и 17 также сохраняется инвариант цикла.

Завершение:

Алгоритм завершается, когда k = r+1. В соответствии с инвариантом цикла, подмассив A[p..k-1] (т.е. подмассив A[p..r]) содержит k-p = r-p+1 наименьших элементов массивов L[1..n1+1] и R[1..n2+1] в отсортированном порядке.

Суммарное количество элементов в массивах L и R равно n1+n2+2 = r-p+3

Все они, кроме двух самых больших, скопированы обратно в массив A, а два оставшихся элемента являются барьерными.

Анализ кода:

Каждая из строк 1-3 и 8-11 выполняется в течение фиксированного времени. Длительность циклов for в строках 4-7 кратна n_1+n_2, а значит, n. В цикле for в строках 12-17 выполняется n итераций, на каждую из которых затрачивается фиксированное время.

Т.о. время работы процедуры Merge линейно зависит от n, где n = r-p+1.

Основная программа Merge_Sort(A,p,r) выполняет сортировку элементов в подмассиве A[p..r] с использованием подпрограммы Merge:

```
Merge_Sort(A,p, r)
1  if p < r
2     then q ← ⌊ (p + r)/2 ⌋
3          Merge_s ort(A,p, q)
4          Merge_s ort(A,q+1, r)
5          Merge (A,p, q, r) fi
```

Если справедливо p≥r, то в этом подмассиве содержится не более одного элемента, и он отсортирован.

Иначе – разбиение: вычисляется q, разделяющий массив A[p..r] на два подмассива: A[p..q] с [n/2] элементами и A[q+1..r] с [n /2] элементами[10].

Происходит попарное слияние одноэлементных последовательностей в отсортированные последовательности длины 2, затем попарное объединение 2-элементных последовательностей в отсортированные последовательности длины 4 и т.д., пока не будут получены две последовательности, состоящие из n/2 элементов, которые объединяются в конечную отсортированную последовательность длины n.

Методика оценки сложности рекурсивных алгоритмов – это нахождение решения рекуррентного соотношения. Приёмы: метод подстановок, метод дерева рекурсии, основной метод[11].

В рекуррентном соотношении полное время, требуемое для решения всей задачи с объемом ввода n, выражается через время решения вспомогательных подзадач, полученных в ходе декомпозиции алгоритма.

Получение рекуррентного соотношения [4]:

Пусть T(n) – время решения вычислительной задачи, размер которой равен n – в нашем случае это длина массива (рис. 11).

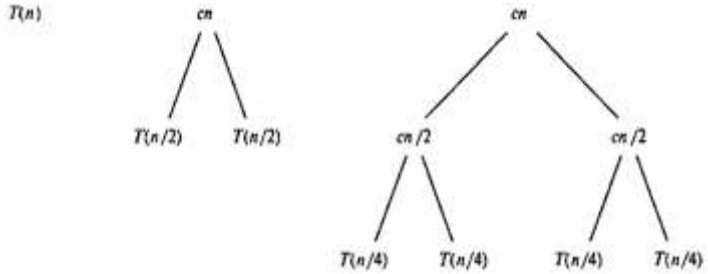


Рисунок 11. Дерево рекурсии алгоритма слиянием

Если n ≤ c, где c – известная константа (т.е. размер задачи мал), то задача решается за определенное фиксированное время θ(1).

Пусть задача делится на a подзадач, объём каждой 1/b от объёма исходной

[10] В угловых скобках – наибольшее целое число, которое меньше или равно x.
[11] Подробно рекурсивный метод рассмотрен в п. 9.

задачи (в сортировке слиянием a=b=2, в общем случае a≠b).

С учётом временных затрат на разбиение задачи на подзадачи D(n) и на объединение решений подзадач в решение исходной задачи C(n), рекуррентное соотношение примет вид:

$$T(n) = \begin{cases} \theta(1) \text{ при } n \leq c \\ aT\left(n/b\right) + D(n) + C(n) \end{cases}$$

Для простоты пусть n – это степень 2, тогда деление массива на каждом шаге рекурсии будет уменьшать количество элементов в подмассивах в 2 раза.

Тогда для алгоритма сортировки слиянием справедливо рекуррентное соотношение: T (n) = 2T(n/2) + cn, где cn – время выполнения на одном уровне рекурсии. Отсюда получим $\log_2(n) + 1$ уровня раскрытия рекурсии (далее lg(n)).

Общее время работы алгоритма на всех уровнях рекурсии: cn·(lg(n) + 1) = cn·lg(n) + cn. Пренебрегая членами более низких порядков, получаем cn·lg(n).

Порядок роста времени сортировки в среднем случае определяется как θ (n · lg (n)).

На нижнем k-ом шаге размер задачи T(n)=θ(1)=1, т.е. n/2k =1 или n= 2k , значит k=lg(n). Общее время k·cn=cn·lg(n), значит O(n·lg(n)).

7.2. Естественное слияние

В алгоритме простого слияния алгоритм разбивает массив на подмассивы фиксированной длины, соответствующей степени двойки (нет учёта первичной упорядоченности).

Пусть **серия** – это уже упорядоченная часть массива наибольшей возможной длины. Тогда в отличие от простого слияния можно сразу сливать длинные упорядоченные серии.

Как и простое слияние каждый проход имеет 2 фазы:

- разделение – обеспечивает ***распределение серий поровну*** из исходного массива f в массивы f1 и f2;

- слияние – обеспечивает ***слияние серий*** из f1 и f2 в f.

Пример работы алгоритма естественного слияния.

Исходный массив f:17 31 5 59 13 41 43 67 11 23 29 47 3 7 71 2 19 57 37 61.

Выделим апострофами серии, чтобы было нагляднее:

17 31’ 5 59’ 13 41 43 67’ 11 23 29 47’ 3 7 71’ 2 19 57’ 37 61

Получилось 7 серий. Разделим массив на два подмассива f1 и f2, переписывая в них поочередно по серии:

f1: 17 31’ 13 41 43 67’ 3 7 71’ 37 61

f2: 5 59’ 11 23 29 47’ 2 19 57

Сольем серии, используя упорядоченность:
f:5 17 31 59’ 11 13 23 29 41 43 47 67’2 3 7 19 57 71’37 61
Опять разобьем в f1 и f2, поочередно переписывая серии:
f1: 5 17 31 59’ 2 3 7 19 57 71
f2: 11 13 23 29 41 43 47 67’37 61
Сливаем в f по сериям:
f: 5 11 13 17 23 29 31 41 43 47 59 67’ 2 3 7 19 37 57 61 71
И снова продолжаем до тех пор, пока в массив A не будет переписана серия длины n. Потребовалось три прохода.

Анализ алгоритма естественного слияния: на каждом проходе общее число серий уменьшается вдвое и число необходимых пересылок элементов в худшем случае n*log(n), в среднем и лучшем случае меньше.

Количество сравнений намного больше, т.к. помимо сравнений элементов надо проверять конец серии (тоже через сравнение соседних элементов).

Алгоритму требуется дополнительная память.

7.3. Быстрая сортировка

Это улучшенный вариант сортировки ***пузырьком***. Основные шаги:

1. Выбрать опорный элемент.
2. Разбиение: элементы меньше опорного помещаются перед ним, а не меньшие – после.
3. Рекурсивно применить шаги 1 и 2 к двум подмассивам слева и справа от опорного элемента (в подмассиве должно быть >1 элемента).
4. Комбинирование не нужно (подмассивы сортируются на месте) – в результате весь массив оказывается отсортирован.

Листинг 7.1. Основной алгоритм быстрой сортировки

```
Quicksort (A, p, r)
1    if p < r
2    then q ← Partition (A, p, r)
3    Quicksort (A, p, q - 1)
4    Quicksort (A, q + 1, r)
```

Листинг 7.2. Алгоритм этапа разбиения

```
Partition (A, p, r)
1.    x ← A[r]
2.    i ← p - 1
3.    for j ← p to r- 1
4.       do if A[j] ≤ x
5.             then i ← i+ 1
6.    Обменять A[i] ↔ A[j] fi  od
7.    Обменять A[i + 1] ↔ A[r]
8.    return i + 1
```

Здесь основная процедура сортировки всего массива – Quicksort (A, 1, length[A]). Ключевая часть сортировки – процедура Partition, изменяющая порядок элементов подмассива A[p..r] без дополнительной памяти.

В этом варианте опорным всегда выбирается последний элемент в подмассиве A[r]. Разбиение подмассива A[p..r] будет выполняться относительно этого элемента (рис. 12).

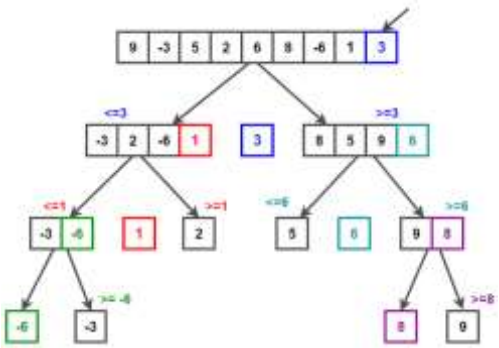


Рисунок 12. Пример работы быстрой сортировки

Время выполнения алгоритма зависит от сбалансированности разбиения. Худшая оценка достигается при неудачном выборе опорного элемента, когда получаются подмассивы длиной 1 и n-1 элемент.

В худшем случае $O(n^2)$, в среднем и лучшем – $n \cdot \log n$. Т.е. при максимальной несбалансированности производительность быстрой сортировки не превышает сортировку обменом или вставкой. Отсортированность массива при этом на производительность не повлияет. Эффективность падает для небольших массивов.

Варианты выбора опорного элемента:

- крайний (несбалансированность, использовался в ранних реализациях);
- срединный;
- случайный;
- медиана первого, среднего и последнего;

- медиана всей последовательности (среднее в упорядоченном ряду) – лучший опорный элемент, но вычисление слишком трудоёмко.

Разбиение (схема) **Хоара**:

Используется 2 индекса (в начале массива и в конце), они приближаются друг к другу, пока не найдётся пара элементов, где один больше опорного (расположен перед ним), а второй меньше (расположен после) и эти элементы меняются местами.

Проверка идёт, пока индексы не пересекутся.

8. СТРУКТУРЫ ДАННЫХ

8.1. Понятие структуры данных

Структура данных (data structure) – это именованная совокупность логически связанных значений в памяти ЭВМ в соответствии с определённым макетом.

Макет задаёт:

- логические (причинно-следственные) связи между элементами;
- способ доступа к значениям (прямой или последовательный).

Один и тот же макет может быть эффективен для одних операций и неэффективен для других (т.е. нет универсальной структуры данных для всех вычислительных задач).

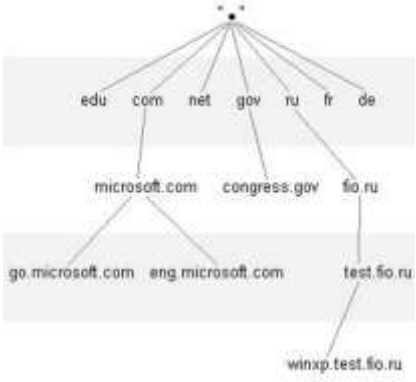


Рисунок 13. Пример древовидной структуры представления данных некоторой предметной области

Структуры часто изображают в виде графа.

Граф G [R,A] – это совокупность двух множеств – вершин R и рёбер A (в неориентированном) или дуг (в ориентированном).

Граф моделирует отношения (связи) между вершинами, это эффективное средство моделирования в реальных задачах.

Виды структур данных по ***характеру связи***:

- **линейные** – элементы образуют последовательность или список (обход узлов линеен) – цепь, стек, очередь, дек (например – список игроков команды);
- **нелинейные** – совокупность элементов без позиционного упорядочения:
  - **графовые** – дерево, лес, граф, сеть (например, структура доменов в сетевой службе DNS);
  - **теоретико-множественные** – реляционные.

8.2. Структуры данных в памяти ЭВМ

Логическая структура данных располагается в адресуемых ячейках ОЗУ. Адрес ячейки – это номер байта, начиная с которого ячейка размещена в памяти

Структура хранения данных – это представление логической структуры данных в памяти ЭВМ (машинный образ абстрактной структуры).

Одну и ту же логическую структуру данных можно представить разными структурами хранения. Отражение структуры данных в структуру хранения обеспечивается средствами ЯВУ.

Структуры хранения данных:

- **статические** (определены в коде на этапе компиляции);
- **динамические** (создаются на этапе исполнения кода).

Структуры хранения данных:

1. **Векторные** – вектор, массив.
2. **Списочные** – связанные линейные списки: однонаправленный м двунаправленный список, стек, очередь, дек.

Вектор – это линейная структура хранения для многоэлементных структур данных, состоит из нескольких последовательно расположенных ячеек в памяти ЭВМ (рис. 14).

Ячейка хранит только значение без ссылочных связей с другими ячейками.

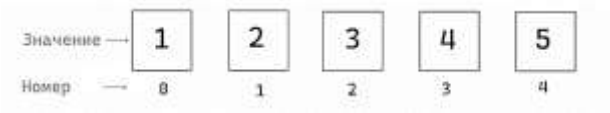


Рисунок 14. Пример векторной структуры

Если все значения (элементы данных) одного типа, то все ячейки одинакового размера.

Дескриптор вектора – это управляющая информация: адрес первого байта вектора A_0, размер ячейки l, индекс первого элемента m, индекс последнего элемента n.

Тогда в однородном векторе A доступ к элементу с индексом i по адресу A_i: $A_i=A_0+(i-m)*l$.

Реализация в ЯВУ – пользовательские типы данных – массивы и классы-вектора (в ООП).

ЛЕКЦИЯ 6

8.3. Массивы в языке C++

Объявление массива требует задание имени, типа входящих в массив элементов и его размера (количества элементов). Размер массива вместе с типом его элементов определяет необходимый для его хранения объём ОЗУ.

Инициализация массива – это присвоение начальных значений его элементам.

Массивы могут быть **одно-** и **многомерными**, когда положение элемента в массиве задаётся не одной, а несколькими индексами (двумя в 2-мерном массиве – номером строки и номером столбца).

Статические массивы – с фиксированной (константной) размерностью, тогда как у **динамических** массивов размерность при исполнении программы может быть изменена.

Константная размерность статического массива единожды определяется на этапе компиляции и фиксируется в машинном коде (не может изменяться в ходе выполнения программы). Размерность динамического массива может определяться (и переопределяться) в ходе работы (на этапе исполнения) программы.

Объявление статического массива в языке C++ требует указания типа входящих в массив элементов, имени массива и его размера, например:

```
int a[25]; //1 – массив из 25 целых чисел
float b[7]; //массив из 7 чисел с плавающей точкой
```

Квадратные скобки – это элемент синтаксиса языка, а не признак необязательности.

При описании массива используются те же модификаторы (например, const или static), что и при описании простых переменных.

Возможен вариант объявления массива одновременно с его инициализацией (т.е. присвоением начальных значений его элементам):

```
int m[7] = { 0, 2, 0, 4, 77 }; //2 – использование инициализатора
```

Если инициализирующих значений меньше, чем элементов в массиве, остаток массива обнуляется, если больше – лишние значения не используются.

Элементы массива ***нумеруются с 0***, поэтому максимальный **индекс** (номер) элемента всегда на 1 меньше размера. В примере выше в массиве m номера его элементов от 0 до 6. Номер вызываемого элемента массива задаётся после его имени в квадратных скобках: m[0], m[5].

Автоматический контроль выхода индекса за границы массива в C++ не

выполняется, программист должен следить за этим сам.

В C++ размер статического массива может быть задан только единожды и только целой положительной константой (или константным выражением), не переменной.

Если при объявлении массива не указан размер, должен присутствовать инициализатор, в этом случае компилятор выделит память по количеству инициализирующих значений.

Статические массивы размещаются в очень ограниченной области выделяемой программе памяти – в **стеке вызовов**. Переполнение стека (что весьма вероятно в случае больших массивов) приведет к аварийному завершению работы программы. Поэтому массивы целесообразно размещать в динамической (динамически распределяемой) памяти – «**куче**».

Размещение массива в «куче» позволит также уйти от константной определенности размера массива. Размер массива в «куче» можно задать с помощью переменной или выражения.

В языке C++ используют два типа массивов, размещаемых в «куче»:

а) массивы **переменной длины** (new/delete);

б) **malloc-массивы**, унаследованные из языка C (malloc/free).

С помощью команд new/delete можно соответственно создавать/уничтожать массивы переменной длины в динамически распределяемой памяти:

```
int n; int* q = new int [n];
//...
delete [] q;
```

Размер созданного массива new/delete изменить не удастся, но его можно многократно пересоздавать заново (с потерей хранящихся в нём значений), каждый раз задавая новый размер (например, посредством ввода с клавиатуры).

Массивы new/delete в языке C++ часто называют динамическими массивами, что, с одной стороны, отражает место расположения массива – в динамической памяти («куче»), но, с другой стороны, вводит в заблуждение – изменить размер уже созданного массива динамически нельзя.

В полном смысле динамический массив (размер которого при необходимости может изменяться на этапе выполнения программы) в C++ реализуют с помощью функции malloc из стандартной библиотеки языка C:

```
int * u = (int *) malloc(5*sizeof(int)); //массив 5 целых чисел
//...
free (u); //освобождение памяти
```

Здесь функция free освобождает «кучу» после окончания работы с массивом.

Динамически изменить размер такого массива позволит функция realloc:

*u = (int *) realloc (u, 25 * sizeof(int));*

При этом размер массива u увеличится на 25 ячеек для хранения целых чисел.

Многомерные ***статические массивы*** в C++ требуют указания каждого измерения в квадратных скобках, например, оператор

int a[6][8];

задает описание двумерного статического массива из 6 строк и 8 столбцов.

В памяти такой массив располагается в последовательных ячейках построчно (т.е. двумерный массив можно рассматривать как линейный массив линейных массивов). Строки в памяти ничем не разделены, т.е. прямоугольной матрицей этот массив является только в нашем воображении.

Для доступа к элементу многомерного массива последовательно указываются все его индексы: a[i][j]. Можно обратиться к элементу и другими способами: *(a[i]+j) или *(*(a+i)+j).

В C++ элементы статического массива задаются инициализатором в порядке их расположения в памяти. Например, команда:

int b[2][5] = { 0, 2, 1, 7, 8, 2, 3, 4, 5, 7 };

определяет матрицу со следующими значениями элементов:

0 2 1 7 8

2 3 4 5 7

Если количество значений в фигурных скобках превышает количество элементов в массиве, возникает ошибка компиляции. Если значений меньше, оставшиеся элементы массива инициализируются значением по умолчанию (для основных типов это 0).

При инициализации многомерного массива он представляется либо как массив из массивов, при этом каждый массив заключается в свои фигурные скобки (в этом случае левую размерность при описании можно не указывать), либо задается общий список элементов в порядке расположения элементов в памяти:

int mass2[][2] = {{1,1},{0,2},{1,0}};

int mass2[3][2] = {1,1,0,2,1,0};

N-мерный массив ***в динамически распределяемой памяти*** в языке C++ представляется как линейный массив указателей на (N-1)-мерные массивы (так, 3-мерный массив – это линейный массив указателей на 2-мерные массивы).

Пример создания в «куче» двухмерной прямоугольной матрицы (рис. 15):

int nrow, ncol; //две размерности – строки и столбцы

```
cout << "Введите количество строк и столбцов: "; cin >> nrow >> ncol;
int ** a = new int *[nrow]; //1
for (int i=0; i < nrow; i++) //2
   a[i] = new int[ncol];    //3
```

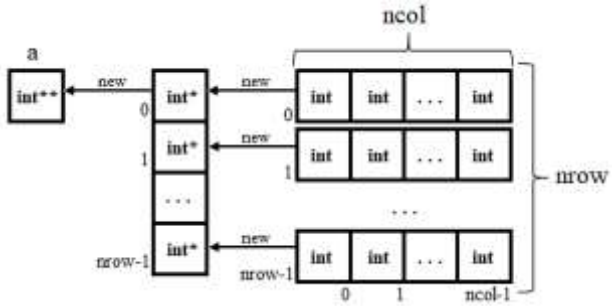


Рисунок 15. Выделение памяти под двумерный целочисленный массив переменной длины

Оператор new: а) выделяет область в «куче» под заданное количество элементов определённого типа; б) возвращает адрес начала массива (0-го элемента) в выделенной области «кучи», который присваивается указателю соответствующего типа.

В строке #1 вышеприведённого кода:

- слева от оператора присваивания объявляется переменная a типа «указатель на указатель на значение типа int»;

- new выделяет память в «куче» под пустой линейный массив из nrow указателей (по количеству строк в матрице) на начальные элементы nrow целочисленных строк;

- переменной a присваивается возвращённый оператором new адрес начала этого служебного массива указателей на строки. По сути, a будет именем всей матрицы.

2: организуется цикл для выделения памяти под каждую из nrow строк матрицы.

3: в цикле в каждый элемент линейного массива a указателей на строки возвращается адрес начала участка памяти, выделенного под соответствующую строку. Каждая строка состоит из ncol элементов типа int (рис. 15).

Обращение к элементам динамических массивов в программе производится так же, как к элементам статических – с помощью конструкции вида a[i][j].

Освобождение памяти в «куче» из-под многомерного массива выполняется с помощью операций delete [] в порядке, обратном порядку создания этого массива. Так, массив a из предыдущего примера будет удалён следующим образом:

```
for (int i=0; i < nrow; i++)   // в цикле по количеству строк
```

```
    delete a[i];   // удаляем из кучи строки
delete []a;  // удаляем служебный массив указателей
```

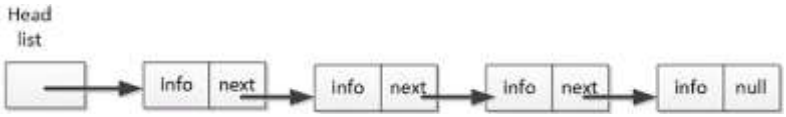
Если удалить только массив указателей, то массивы-строки с данными продолжат занимать «кучу», что, в свою очередь, приведёт к эффекту «утечки» памяти.

8.4. Связные линейные списки

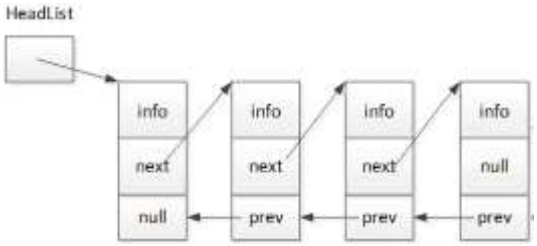
Связные линейные списки – предназначены для хранения многоэлементных последовательностей. Ячейка (элемент списка, узел) хранит значение (информационная часть) и связи (ссылки) с другими ячейками списка.

Не существует понятия позиции (индекса) узла. Для организации таких структур в ЯВУ C++ используется механизм указателей.

В зависимости от количества связей в элементе различают: однонаправленные (или односвязные) и двунаправленные (двухсвязные) списки (соответственно, рис. 16.а и 16.б).



а) Односвязный список



б) Двусвязный список

Рисунок 16. Линейные динамические списки

Линейные структуры данных: списки, стеки (метод LIFO), очереди (метод FIFO), деки.

Яркий пример использования линейного односвязного списка (англ. chain – цепь) – это технология ***блокчейн***.

Реестр транзакций – это совокупность записей о транзакциях в БД, записи можно только добавлять, изменять их нельзя. Т.е. реестр – это БД об операциях над элементами другой (возможно, распределённой) БД.

Блокчейн – это технология реализации децентрализованной сетевой службы на основе данных из БД по определенной предметной области, используется также и реестр транзакций – это реплицированная распределенная БД записей о транзакциях над предметной БД.

Реестр транзакций – ***цепь*** (линейный связный список) блоков записей о транзакциях над предметной БД. Блоки упорядочены в цепь по хешам предыдущих блоков в своих заголовках. Подробнее об архитектуре технологии блокчейн можно узнать из специальной литературы.

Способы представления структур: пользовательские типы (массивы и классы на основе указателей).

Базовые операции над структурами данных:

- найти нужный элемент в структуре;
- обратиться к нужному элементу структуры;
- добавить элемент в структуру;
- удалить элемент из структуры.

8.5. Однонаправленный список средствами языка C++

В языке C++ имеется удобный и мощный механизм реализации динамических структур – указатели и структурный тип (или абстрактные типы данных).

Начать лучше с описания типа данных для узла списка (листинг 8.1).

Листинг 8.1. Реализация узла однонаправленного списка

```
struct Node {
  string val;
  Node* next;
  Node(string _val) : val(_val), next(nullptr){}
};
```

Здесь поле string – информационное (полезная строка данных), next – связь, Node – конструктор экземпляра узла списка.

Далее реализуем сам односвязный список. В списке будет реализовано:

1) указатель на первый узел;
2) указатель на последний узел;
3) конструктор списка;
4) функция проверки наличия узлов в списке;
5) функция добавления элемента в конец списка;
6) функция печати списка;
7) функция поиска узла в списке по ключу;
8) функция удаления узла по ключу.

В следующем коде реализуем пункты 1-3 (листинг 8.2).

Листинг 8.2. Описание типа данных для односвязного списка

```
struct list {
    Node* first;
    Node* last;
    list() : first(nullptr), last(nullptr) {}
};
```

В код главной функции main необходимо добавить создание экземпляра списка, например:

list l;

Добавим в структуру list вспомогательную функцию для проверки наличия узлов в списке – однострочная, в ней необходимо проверить не пуст ли первый узел (листинг 8.3).

Листинг 8.3. Проверка наличия узлов в односвязном списке

```
bool is_empty() {
    return first == nullptr;
}
```

Реализуем в структуре list функцию добавления элемента в конец списка. Здесь возможны два случая: а) список пустой и б) список не пустой.

В обоих случаях необходимо создать сам узел с заданным (переданным в функцию) значением.

Для первого случая понадобится определённая ранее функция проверки наличия узлов. Если список пуст, указатели на первый и последний узел направляем на единственный новый узел.

Для второго случая нужно указать, что новый узел стоит после последнего узла, после чего меняем значения указателя на последний узел last (листинг 8.4).

Листинг 8.4. Добавление узла в конец списка.

```
void push_back(string _val) {
    Node* p = new Node(_val);
    if (is_empty()) {
      first = p;
      last = p;
      return;
    }
    last->next = p;
    last = p;
}
```

Т.о. в список можно добавлять и другие узлы.

В структуре list в функции печати всего списка (если список не пуст) направляем указатель p на первый узел списка и выводим значения узлов, пока указатель p не пустой. При каждой итерации перенаправляем p на следующий

узел (листинг 8.5).

Листинг 8.5. Вывод списка в консоль.

```
void print() {
    if (is_empty()) return;
    Node* p = first;
    while (p) {
        cout << p->val << " ";
        p = p->next;
    }
    cout << endl;
}
```

Для поиска узла в списке по ключевому значению в структуре list добавим функцию обхода списка, пока указатель p не пустой и пока значение узла p не равно ключу, после чего возвращаем найденный узел, если он есть (листинг 8.6).

Листинг 8.6. Поиск узла в списке по ключу.

```
Node* find(string _val) {
    Node* p = first;
    while (p && p->val != _val) p = p->next;
        return (p && p->val == _val) ? p : nullptr;
}
```

Листинг 8.7. Удаление узла из списка по ключу.

```
void remove(string _val) {
    if (is_empty()) return;
    if (first->val == _val) {
        remove_first();
        return;
    }
    else if (last->val == _val) {
            remove_last();
        return;
    }
    Node* slow = first;
    Node* fast = first->next;
    while (fast && fast->val != _val) {
        fast = fast->next;
        slow = slow->next;
    }
    if (!fast) {
        cout << "This element does not exist" << endl;
        return;
    }
    slow->next = fast->next;
    delete fast;
}
```

Добавим в структуру list функцию удаления узла из списка по заданному ключу.

Если список не пуст, то возможны три случая:

1) узел с искомым значением равен первому;
2) узел с искомым значением равен последнему;
3) не первый и не второй случаи.

Первый случай: сравниваем значение первого узла с ключом, если значения совпадают, тогда вызываем функцию удаления первого узла.

Второй случай: сравниваем значение последнего узла с ключом, если значения совпадают, тогда вызываем функцию удаления последнего узла.

Третий случай:

Создаются указатели slow на первый узел, и fast – на следующий после первого. Затем, пока fast не пустой и пока значение текущего узла fast не равно ключу, при каждой итерации перенаправляем slow и fast на следующий после них узел. Если указатель fast пустой, то сообщаем об ошибке, иначе просто удаляем узел fast.

Описанный код вместе с вспомогательными функциями удаления первого и последнего узлов в списке, приведён в листинге 8.7.

Здесь приведён один из возможных способов реализации базовых операций с односвязным списком. Возможны и другие реализации, в т.ч на основе абстрактных типов данных в рамках ООП.

Базовые операции с другими типами линейных структур (двунаправленный список, стек, очередь, дек) реализуются аналогичным образом с поправкой на их специфику.

ЛЕКЦИЯ 7
9. РЕКУРСИВНЫЕ АЛГОРИТМЫ

9.1. Рекурсия и рекуррентное соотношение

Рекурсивным называется объект, частично состоящий из самого себя или определяемый с помощью себя. Например:

1. Определение натурального числа: 1 – есть натуральное число; целое число, следующее за натуральным, есть натуральное число.

2. Алгоритм вычисления n!:

$$n! = \begin{cases} n * (n-1)! \text{ при } n > 1 \\ 1 \text{ при } n = 1 \end{cases}.$$

3. Вычисление НОД двух натуральных чисел по алгоритму Евклида: если a≠0 и b≠0 то НОД(a,b) = НОД(b,r) где r = a % b.

Например, НОД(18,4)=НОД(4,2)=НОД(2,0).

4. Геометрические фракталы (например, треугольник Серпинского).

5. Функция Аккермана.

6. Узел однонаправленного списка:

```
struct node {
    Titem info;
    node *next; }
```

Рекурсия часто используются в задачах, решаемых **методом декомпозиции** – разделяй и властвуй. Парадигма, лежащая в основе метода:

1. Разделение задачи на несколько подзадач

2. Рекурсивное решение этих подзадач, когда объем подзадачи достаточно мал, выделенные подзадачи решаются непосредственно (получаем простейшее решение)

3. Комбинирование решения исходной задачи из решений вспомогательных подзадач.

Рекуррентное соотношение (recurrence) – это уравнение или неравенство, описывающее функцию с использованием её самой. Например, $a_{n+1} = q \cdot a_n$.

Рекуррентное соотношение определяет ***природу*** рекурсивного алгоритма. Действительно, рекурсивный алгоритм для вычисления значения реализуемой задачи использует на каждом шаге ранее вычисленные значения, начиная с наименьшего.

Примеры представления рекурсивного процесса через рекуррентное соотношение:

1. Задача вычисления x^n сводится к умножению n раз или $x^n = x*x^{n-1}$:

$$x^n = \begin{cases} 1, n = 0 \\ x \cdot x^{n-1}, n > 0. \end{cases}$$

2. Задача вычисления суммы n первых натуральных чисел:

$$s(n) = \begin{cases} 1, n = 1 \\ s(n-1) + n, n > 1. \end{cases}$$

3. Задача целочисленного деления a на b (b≠0):

$$div(a,b) = \begin{cases} 0, a < b \\ div(a-b,b), \text{иначе.} \end{cases}$$

4. Задача вычисления n-го числа Фибоначчи: $f_1=1$, $f_2=1$, $f_n=f_{n-1}+f_{n-2}$ или

$$f(n) = \begin{cases} 1, n < 3 \\ f(n-1) + f(n-2), \text{иначе.} \end{cases}$$

9.2. Рекурсивный алгоритм. Дерево рекурсии

Итак, **рекурсивный алгоритм** – это алгоритм, в описании которого прямо или косвенно содержится обращение к самому себе. Косвенная рекурсия (рис. 17), если алгоритм A вызывает алгоритм B, и алгоритм B вновь вызывает алгоритм A.

Рисунок 17. Схема вызовов при косвенной рекурсии

И прямая рекурсия (рис. 18), если решение задачи сводится к разделению ее на меньшие подзадачи, выполняемые с помощью того же алгоритма. Процесс разбиения завершается, когда достигается простейшее возможное решение.

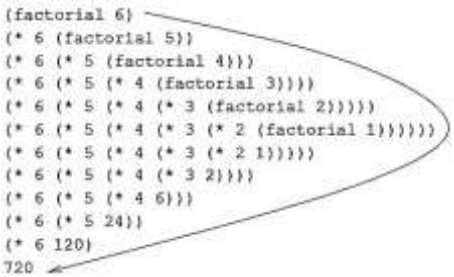


Рисунок 18. Схема вызовов при прямой рекурсии на примере факториала

Условие выхода из рекурсии определяет завершение рекурсии и формирование конкретного выходного значения вычислительного процесса.

Шаг рекурсии – это выполнение одного действия.

Глубина рекурсивных вызовов – это наибольшее одновременное количество рекурсивных вызовов функции, определяющее максимальное количество слоёв рекурсивного стека, в котором осуществляется хранение отложенных вычислений. Глубину рекурсии можно изобразить в виде графа – дерева рекурсии – это ***полное дерево***, т.е. на каждом уровне, кроме последнего, располагается максимально допустимое количество узлов (рис. 19).

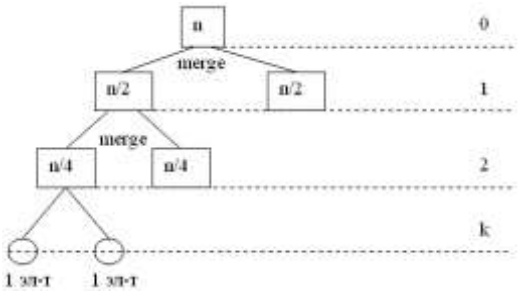


Рисунок 19. Фрагмент дерева рекурсии алгоритма сортировки слиянием

Количество узлов на уровне зависит от степени дерева: если степень 2 (бинарное), то количество узлов на i-ом уровне 2^i.

На последнем уровне размещаются вершины – листья, показывающие вызовы, приводящие к завершению вычислений (нумеруются с 0).

Обозначения характеристик дерева рекурсии для конкретного входного параметра x: R(x) – общее число вершин дерева рекурсии, $R_V(x)$ – объем рекурсии без листьев (только внутренние вершины), $R_L(x)$ – количество листьев дерева рекурсии, $H_R(x)$ – глубина рекурсии, определяется как количество незавершенных входов в рекурсию, т.е. это количество внутренних вершин (без учета листьев).

Пример дерева рекурсии алгоритма вычисления 5-ого числа Фибоначчи (рис. 20) и определение характеристик этого дерева рекурсии.

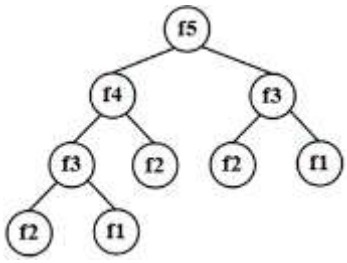


Рисунок 20. Дерево рекурсивных вызовов для вычисления 5-го числа Фибоначчи

Здесь R(5)=9, $R_V(x)$=4, $R_L(x)$=5, $H_R(x)$=4.

2 правила создания рекурсивных алгоритмов:

1. Определить рекурсивную зависимость и оформить рекуррентное соотношение: P=W[S_i,P].

2. Определить условие завершения рекурсии.

Варианты схем организации рекурсивного алгоритма:

1. Без указания явного выхода из рекурсии (выход не указан но выполниться по завершении входов):

P ≡ if (B) then W[Si,P] //шаг в рекурсию

2. С указанием выхода из рекурсии при достижении простейшего решения:

```
P≡ if (B) then W[Si,P]          //шаг в рекурсию
else return начальное значение       //явный выход из рекурсии
```

3. Явный выход из рекурсии при достижении определенного значения параметра x (с явным уменьшением значения параметра в каждом вызове):

```
P (x)≡ if (x=нач.значение)
return x //завершение рекурсии
else return P(x-1)] //шаг в рекурсию
```

9.3. Рекурсивные функции

Повторное выполнение алгоритма при новых значениях входных переменных можно реализовать в виде подпрограммы (процедуры или функции), которую можно вызвать с новыми значениями параметров.

Общий формат рекурсивной подпрограммы P можно представить как композицию W некоторых базовых операторов и операторов вызова подпрограммы: $P=W[S_i,P]$.

Подпрограмму (функцию) называют рекурсивной, если хотя бы одна конструкция содержит имя этой подпрограммы [2].

Наиболее надежный способ обеспечить завершение конечной рекурсии – это связать с подпрограммой P некоторое значение и рекурсивно вызвать P с параметром, измененным на это значение, в конечном счёте приводящим к начальному значению (простейшему решению).

Например, задача вычисления n!. Здесь параметр – это число, которое при каждом вызове уменьшается на 1, приводя к выполнению условия выхода из рекурсии:

$$n! = \begin{cases} n * (n-1)!, & n > 1 \\ 1, & n = 1. \end{cases}$$

Соответствующий код на языке C++:

```
int fact (int n) {
  if (n==1) return 1;
  return n*fact(n-1);
}
```

Рекурсия по механизму выполнения похожа на цикл: цикл имеет условие

завершения и рекурсивный алгоритм тоже. Если условие цикла записано неверно, то цикл может стать бесконечным, это же возможно и с рекурсивным процессом.

Для очередного рекурсивного вызова функции создается новое множество локальных переменных, фактических параметров, служебной информации для возврата. Всему этому выделяется в области стековой памяти процесса кадр (стековый кадр). При выходе из вызова стек будет освобождаться.

Виды рекурсий и графы рекурсивных вызовов

1. Линейная рекурсия: каждый вызов порождает ровно 1 новый вызов (на примере графа вызовов n! – рис. 21):

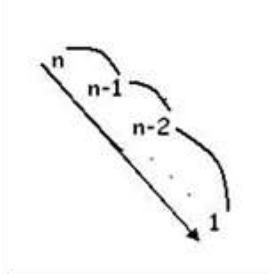


Рисунок 21. Линейная рекурсия

2. Каскадная рекурсия: каждый вызов порождает несколько новых вызовов (на примере вычисления 5-го числа Фибоначчи – рис. 22):

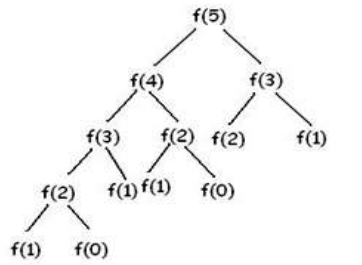


Рисунок 22. Каскадная рекурсия

3. Удалённая рекурсия, когда в определении рекурсивной функции в списке фактических параметров вызов содержит саму эту функцию.

Например, сумма цифр натурального числа:

```
int sum(int n) {
  if (n<9) return n;
  else return sum(sum(n / 10)) + n % 10;
}
```

9.4. Рекурсивный алгоритм бинарного поиска

Алгоритм бинарного поиска (метод деления пополам, дихотомия) – это классический алгоритм поиска элемента в отсортированном массиве на основе дробления массива на половины.

Алгоритм бинарного поиска основан на методе «разделяй и властвуй», т.е. он рекурсивен.

Пусть задан отсортированный массив x и ключ поиска key.

Основные этапы решения:

1. Определение значения элемента в середине массива (по индексам l и r левого и правого края). Полученное значение сравнивается с ключом.

2. Если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй (переопределение индекса l или r).

3. Пункт 1.

4. Процесс продолжается до тех пор, пока не будет найден элемент со значением ключа или не станет пустым интервал для поиска.

Этот алгоритм может быть реализован как итерационным, так и рекурсивным методом.

Код рекурсивного решения представлен в листинге 9.1.

Листинг 9.1. Алгоритм бинарного поиска

```
int binfind(int l, int r, int *x, int key){
  if (l>r) return -1;    //неудача
  int i=(l+r)/2;
  if(x[i]==key) return i;   //успех
  if(x[i]>key)
     return binfind(l, i-1, x, key);
  else
    return binfind(i+1, r, x, key);
}
```

Проверка: пусть дан исходный массив ключей: 2 7 15 139 347.

1) Найти: ключ, равный 15. Результат 0.

2) Найти: ключ, равный 10. Результат -1.

Рекуррентное соотношение: $T(n)=a\cdot T(n/b)+O(1)$.

Для бинарного поиска выполняется только одна из двух подзадач, т.е. a=1 и b=2:

$$T(n) = T(n/2) + O(1) \le T(n/2)+c.$$

Для получения решения можно применить метод дерева рекурсии, который даст сложность $O(\log_2 n)$ или просто $O(\log n)$.

10. ПОИСК В МАССИВАХ (ТАБЛИЦАХ)

10.1. Задача алгоритмов поиска

Пусть дано:

1) Множество из N элементов:

- **элементы** (записи) R_1 R_2 R_3 … R_N, которые содержат специальное поле – ключ; ключи в записях уникальны, т.е. ключ однозначно определяет запись;

- **таблица** – это массив записей или небольшой по объёму файл с записями.

2) Аргумент поиска – К (**ключ поиска**).

Результатом будет **индекс** записи, имеющей К своим ключом.

Пример создания подобной таблицы средствами С++:

```
struct student {
    char num[8]; // ключ записи – номер
    char fio[50];
    unsigned int date;
    char group;
};
student R[30];// таблица записей
```

Два случая завершения поиска: ***удачный*** – запись с ключом К найдена (позволяет определить положение – индекс записи, содержащей аргумент К); ***неудачный*** – показывает, что элемент с аргументом К не может быть найден ни в одной из записей.

Поиск со вставкой: некоторые алгоритмы при неудачном поиске предусматривают вставку новой записи, содержащей ключ К, в таблицу. Такие алгоритмы называют алгоритмами «поиск со вставкой».

Рисунок 23. Классификация методов поиска

Критерий оценки сложности алгоритмов поиска – количество операций сравнения (перемещений практически не происходит).

10.2. Линейный (последовательный) поиск

Задача известна – поиск записи с ключом К в таблице (массиве или файле) записей. При поиске допустим один из двух исходов:

- для всех i $K \neq K_i$, т.е. нет записи с ключом К;

- $K = K_i$, запись с ключом К найдена в позиции i

Результат: индекс записи (при $K = K_i$) – ключ найден (удачно) или код завершения -1 (при $K \neq K_i$) – признак неудачного поиск.

Алгоритм линейного (последовательного) поиска:

Задача. Имеется таблица записей R_1 R_2 $R_3 \ldots R_N$, имеющих, соответственно, ключи K_1 K_2 $K_3 \ldots K_N$. Требуется найти запись с ключом поиска К ($N \geq 1$).

Шаг 1. Начальная установка: К и i←1.

Шаг 2. Сравнение ключей, результат $K = K_i$ – вернуть i // удача

Шаг 3. Продвижение: i=i+1, если $i \leq N$ перейти к шагу 2,

иначе вернуть -1 // не удача

Листинг 10.1. Псевдокод алгоритма линейного поиска

```
LINEAR_SEARCH
    i ← 1     // инициализация параметра цикла
        While (i<n) And (a[i]<>x)
            Do i ← i + 1
        If (i = n) Then <Элемент найден>
                Else <Элемент не найден>
```

Линейный поиск иллюстрирует стратегию «грубой силы» - полный перебор (простота и низкая эффективность).

Для неотсортированных (неподготовленных) данных линейный поиск – единственно возможный алгоритм.

Анализ алгоритма:

Порядок следования частей в условии важен: первая часть условия – проверка принадлежности к массиву; вторая часть – продолжение поиска.

В среднем случае будет проверяться n/2 значений, в худшем – n , в лучшем 1, т.е. асимптотическая сложность O(n), Ω(1).

Быстрый линейный поиск с использованием барьерного элемента.

Цель – упростить логическое выражение в условии цикла путём упразднения проверки границ массива.

Барьер (sentinel) – это вспомогательный элемент R_{N+1}, добавленный в конец таблицы и содержащий значение аргумента поиска К. Например, есть массив ключей 1 2 4 3 5, и ищем 4, тогда массив с барьером: 1 2 4 3 5 **4** (результат i=3).

Листинг 10.2. Псевдокод алгоритма быстрого линейного поиска

```
LINEAR_SEARCH_SENTINEL (a,n)
    a[n+1] ←= x
    i ←1     // инициализация параметра цикла
        While a[i]<>x
            Do i ←i + 1
        If (i = n) Then < элемент найден >
                Else < элемент не найден >
```

Алгоритм работает быстрее линейного поиска (~30%), но сложность в худшем случае всё равно будет O(n).

10.3. Поиск в упорядоченном массиве

Если значение нужно найти 1 раз, то лучше использовать линейный поиск, но если поиск в таблице выполняется часто, то эффективнее её упорядочить:

$$K_1 < K_2 < K_3 < \ldots < K_N.$$

При этом из поиска могут быть исключены заведомо неподходящие записи.

После сравнения аргумента К с ключом K_i, поиск продолжается одним из способов (в зависимости от того, какое из сравнений верно): $K<K_i$, $K=K_i$, $K>K_i$.

Двоичный поиск (бинарный, деления пополам, дихотомии)[12].

Алгоритм соответствует методу разработки «разделяй и властвуй».

Постановка задачи: имеется таблица записей R_1 R_2 $R_3 \ldots R_N$, причем ключи удовлетворяют условию $K_1 < K_2 < K_3 < \ldots < K_N$. Требуется найти запись с ключом К.

Схема алгоритма бинарного поиска:

1. Выбирается элемент в середине массива $K_{ср}$.
2. Его значение сравнивается с аргументом К.
3. Если $K = K_{ср}$ то элемент найден, возвращается индекс.
4. Если $K<K_{ср}$, поиск продолжается в части таблицы R_1 $R_2 \ldots R_{ср-1}$ тем же методом.
5. Если $K > K_{ср}$, поиск продолжается в части таблицы $R_{ср+1} \ldots R_N$ тем же методом.

Листинг 10.3. Псевдокод алгоритма двоичного поиска

[12] Рекурсивное решение было приведено в п. 9.4.

```
BINARY_SEARCH_ MODIF (a,n)
left ←1   right ← n        // инициализация переменных
      While  left < right
          Do
             d ←Div ((left+right) /2)
             If x > a[d] Then left←d +1
                         Else right ←d
      If a[l] = x Then < элемент найден >
                  Else < элемент не найден >
```

Анализ итерационного алгоритма бинарного поиска:

В цикле упразднена проверка на соответствие, при завершении цикла будет выполняться двойное равенство left = right = d.

Алгоритм на каждом шаге отбрасывает половину данных в неисследованной части массива. Тогда вычислительная сложность алгоритма $O(\log_2 n)$ или просто O(log n)

Поиск в линейных связных списках подобен поиску в массиве. Кроме того, алгоритм может быть эффективно распараллелен.

Дерево бинарного поиска.

Работу алгоритма бинарного поиска можно изобразить в виде дерева[13] (рис. 24).

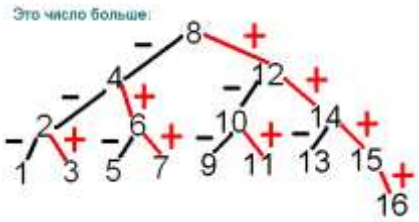


Рисунок 24. Дерево бинарного поиска

Пусть N=16, значения границ: l = 1, r = 16.

Корень дерева – это первое деление диапазона поиска на две части:
mid = (l + r) / 2 = 8.

Из корня дерева выходят два ребра – левое соответствует false, правое – true условия if (a[mid] > K), где K – аргумент поиска.

Две дочерние вершины корня – это следующие переходы по границе mid.

10.4. Однородный (равномерный) бинарный поиск

Является модификацией алгоритма бинарного поиска. В отличии от бинарного поиска, этот алгоритм определяет не границы, а ***величину интервала***

[13] Нелинейные структуры, в т.ч. деревья, будут подробно рассмотрены во второй части пособия.

для поиска K.

Идея: использовать вместо 3-х переменных (l, r, mid) две: i (вместо mid) – текущее положение элемента, delta – величина изменения i (интервал).

При выполнении текущего сравнения, не приведшего к удачному исходу:

а) i ← i± delta (смещаемся влево или вправо) и

б) delta ← delta/2 (устанавливаем новый интервал).

Практическая реализация этого алгоритма требует от разработчика исключительного внимания к деталям (например, куда округлять результат деления).

Алгоритм однородного бинарного поиска:

Шаг 1. i← ⌈N/2⌉, delta ← ⌊N/2⌋.

Шаг 2. Если K < K_i то шаг 3 (влево, уменьшить i).

Если K > K_i то шаг 4 (вправо, увеличить i).

Если K = K_i то вернуть i (успех).

Шаг 3. (Уменьшение i) (интервал, где нужно продолжать поиск, содержит delta или delta–1 записей; i указывает на первый элемент справа от интервала).

Если delta = 0, то алгоритм завершается неудачно.

Иначе устанавливаем: i ← i−⌈delta/2⌉ и delta ← ⌊delta/2⌋ и шаг 2.

Шаг 4. (Увеличение i) (i указывает на первый элемент слева от интервала).

Если delta = 0, то алгоритм завершается неудачно.

Иначе устанавливаем: i ← i+⌈delta/2⌉ и delta ← ⌊delta /2⌋ и шаг 2.

Бинарное дерево для однородного бинарного поиска см. на рис. 25.

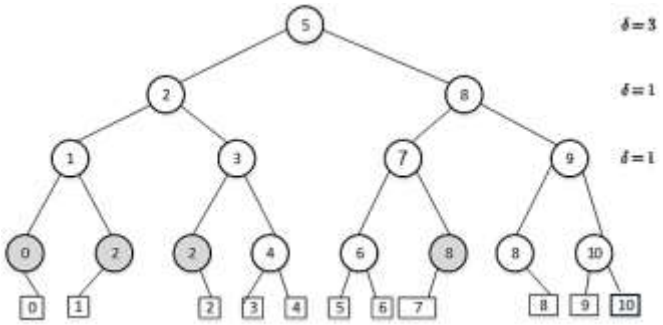


Рисунок 25. Бинарное дерево для однородного бинарного поиска

Пример – пусть есть массив из 10 чисел: 10 12 15 18 20 32 45 60 70 81. Найти индекс числа K=70.

1) i=5, delta=5, K_5 = 20.

2) K > K_i, тогда

3) i ← i+⌈delta/2⌉ и delta ← ⌊delta/2⌋, получаем: i=8 и delta=2 и на шаг 2.

4) K_8 =60, K > K_8, тогда

5) i ← i+⌈delta/2⌉ и delta ← ⌊delta/2⌋, получаем: i=9 и delta =1 и на шаг 2.

Успешное завершение: i=9.

Поиск однородный, т.к. разность между числами в узлах уровня i и числом в узле-родителе (i–1 уровень) – постоянная величина delta для всех узлов уровня i.

На этом основана следующая модификация алгоритма – ***однородный бинарный поиск с таблицей смещений***.

Идея – исключить все операции деления, связанные с delta.

Для этого используется вспомогательная таблица (если нет ограничений по объему памяти) с вычисленными заранее значениями delta:

$$\text{delta}[j]=\left\lfloor \frac{N+2^{j-1}}{2^j} \right\rfloor.$$

Для чётного n необходимо имя-сторож $x_0 = -\infty$.

Шаг 1. i←delta[1] и j← 2

Шаг 2. Если K < K_i, то шаг 3.

Если K > K_i, то шаг 4.

Если K= K_i, то вернуть i (удача).

Шаг 3. (Уменьшение i)

Если delta[j] =0, то неудача.

Иначе устанавливаем: i ← i - delta[j] и j ← j+1 и к шагу 2.

Шаг 4. (Увеличение i)

Если delta[j]=0, то алгоритм завершается неудачно.

Иначе устанавливаем: i ← i+delta[j] и j ← j+1 и переходим к шагу 2.

Подобная модификация позволяет осуществлять поиск быстрее обычного бинарного поиска.

Сложность алгоритма O(log n), т.е. соответствует дереву бинарного поиска с той же длиной внутреннего пути.

ЛЕКЦИЯ 8

10.5. Поиск Фибоначчи

Алгоритм поиска Фибоначчи – альтернатива бинарному поиску. Алгоритм применим для массивов (таблиц), записи в которых упорядочены в порядке возрастания ключей.

Числа Фибоначчи можно вычислять, используя сложение, которое выполняется быстрее, чем деление.

Тогда если разделять исходное множество ключей не пополам, а на подмножества, начальные и конечные индексы которых являются числами Фибоначчи, то при компьютерном исполнении можно избежать операций

деления.

В ряде случаев этот алгоритм предпочтительнее.

Пример массива: {3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}

Пусть K=42.

Последовательность чисел Фибоначчи определяется рекуррентным соотношением: $F_0=0$, $F_1=1$, $F_m=F_{m-1}+F_{m-2}$ при $m \geq 2$, т.е. каждое число равно сумме двух предыдущих: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

Тогда в массиве можно выделить подмножества элементов, начальный и конечный индексы которых составляют два последовательных числа Фибоначчи:

{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}
{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}
{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}
{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}
{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}
{3, 5, 8, 9, 11, 14, 15, 19, 21, 22, 28, 33, 35, 37, 42, 45, 48, 52}

Процесс выделения подмножеств продолжается, пока не будет найдено совпадение ключей (или пока невозможно будет найти такие числа F_{m-1} и F_m).

Подмножество, содержащее искомый ключ: {35, 37, 42, 45, 48, 52}, что соответствует m= 8, для которого $F_{m-2} = 8$, $F_{m-1} = 13$ и $F_m = 21$ (последнее значение 21 превышает количество ключей n = 18).

Алгоритм:

1. Предполагается, что количество элементов массива $N+1=F_{k+1}$ (k+1-е число Фибоначчи), но можно скорректировать алгоритм и для массива, для которого это условие не выполняется.

2. Определяется такое минимальное k, что для искомого ключа K выполняется соотношение $K_{Fk-1} < K < K_{Fk}$, т.е. K расположен между ключами, ***индексы*** которых являются двумя последовательными числами Фибоначчи.

3. Затем этот же алгоритм поиска Фибоначчи применяется ко множеству от K_{Fk-1} до K_{Fk}.

Реализация алгоритма:

1) ***Инициализация*** переменных:

Вычислить k такое, что $F(k+1) \geq N+1$.

M=F(k+1)-(N+1) – смещение-корректировка размера массива.

Определить начальные параметры:

i=F(k)-M // индекс элемента массива для сравнения с K

p=F(k-1) // k-1–ое число Фибоначчи

q=F(k-2) // k-2–ое число Фибоначчи

2) ***Поиск***:

While flag<>true do

Шаг 1. Если K < K_i то шаг 2

Если K > K_i то шаг 3

Если K = K_i, то flag = false и вернуть i

Шаг 2. (Уменьшение i)

Если q=0, неудачный поиск flag=true

Иначе i ← i-q, p ← q, q ← q-p и на шаг 1

Шаг 3. (Увеличение i).

Если p=1, неудачный поиск flag=true.

Иначе i ← $i + q$, p←p-q, q←q-p и на шаг 1

od

Анализ алгоритма Фибоначчиева поиска: среднее количество операций $8.6 \cdot log_2 n$, что медленнее чем в однородном бинарном поиске. Но сами операции – сложение и вычитание – в разы легче для процессора, чем деление. Поэтому поиск Фибоначчи по времени эффективнее.

Временная сложность алгортма O(log n).

10.6. Интерполяционный поиск

Интерполяция – нахождение промежуточных значений между двумя и более известными величинами.

Идея:

1. На каждом шаге выбирается некоторая позиция в пространстве поиска в упорядоченном массиве, основываясь на граничных значениях и искомом ключе (обычно с помощью **линейной интерполяции**).

2. Ключ в найденной позиции сравнивается с искомым и, если они не равны, то (в зависимости от результата сравнения) пространство поиска сводится к части до или после ключа (как в двоичном поиске).

Но, в отличие от двоичного поиска, когда область поиска делится на пополам, здесь оценивается размер новой области по ***расстоянию*** между искомым K и текущим значением ключа K_i.

Листинг 10.4. Псевдокод алгоритма интерполяционного поиска

```
INTERPOLAR_SEARCH (a, n, x)
    left ←1
    right←n
    While  left<>right
        Do
            d ←left + Div ((right–left)*(x–a[left])/(a[right]–a[left]))
              If x > a[d] Then left ← d + 1
                        Else right ←d
    If a[left] = x Then < элемент найден >
                  Else < элемент не найден >
```

Исходный массив (таблица) должен быть упорядочен по возрастанию ключей. Эффективность зависит от ***равномерности распределения*** значений в массиве.

Если ключ К между ключами K_l и K_r (l и r – номера граничных элементов области поиска), то делаем пробу в позиции d:

$$d = l + div\ ((l - r) \cdot (K - K_l), (K_r\text{-}K_l)).$$

Здесь операция деления строго целочисленная (дробная часть отбрасывается).

Искомое значение сравнивается с элементом массива в граничной позиции при l==r.

Пример работы интерполяционного поиска:

Пусть дано множество ключей А:

{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95, 98, 102}.

Требуется найти ключ К = 90.

Шаг 1. Определяется шаг d для исходного массива (l=1; r= 17):

d = [(17-1)(90-4)/(102-4)] = 14.

Тогда l+d = 15.

A[15]=95, 95>90, сужается область поиска до

{4, 5, 10, 23, 24, 30, 47, 50, 59, 60, 64, 65, 77, 90, 95}.

Шаг 2. Определяется шаг d для поля поиска (l=1; r=15):

d = [(15-1)(90-4)/(95-4)] = 13.

l+d = 1+13 = 14.

A[15]=90, 90 = 90, ключ найден.

Анализ интерполяционного поиска:

Интерполяционный поиск предпочтительнее бинарного: один шаг бинарного поиска уменьшает количество записей с n до ½ n, один шаг интерполяционного (при равномерном распределении ключей) – с n до $\sqrt{n}$.

Требуется в среднем около $log_2(log_2 n)$ шагов.

Сложность алгоритма поиска O(log(log n)).

10.7. Поиск в статических и динамических таблицах

В предыдущих алгоритмах ***неявное*** бинарное дерево использовалось только для понимания алгоритмов бинарного и Фибоначчиева поиска.

Эти алгоритмы предназначены для поиска в линейных упорядоченных структурах фиксированного размера (операции вставки и удаления достаточно трудоемкие).

На практике бывают нужны динамические таблицы, в которых часто требуется удалять записи по ключу и вставлять записи в определенном порядке.

Для этих целей используют ***нелинейные динамические структуры***, для которых предыдущие алгоритмы сортировки и поиска не работают.

Алгоритм поиска по бинарному дереву:

В 1946 – Джон Мочни предложил ***явное*** использование структуры бинарного дерева для поиска[14].

По предварительно построенному из исходного потока данных сбалансированному **бинарному дереву поиска** работа алгоритма поиска проста: поисковый элемент сравнивается с корневым ключом, затем, в зависимости от результата, с корнем левого или правого поддерева. Процесс сравнения рекурсивно повторяется на следующих уровнях.

Вычислительная сложность поиска O(log n).

Симметричный обход бинарного дерева поиска, в свою очередь, позволяет получить отсортированную последовательность значений.

10.8. Поиск в хеш-таблицах

Хеширование как преобразование исходных данных в выходную битовую строку находит применение в таких сферах, как контроль целостности при передаче данных (контрольные суммы), информационная безопасность (защита паролей, ЭЦП) и некоторые другие.

В том числе хеширование может быть использовано и для организации эффективного (с ***константным временем*** O(1)) поиска (также вставки и удаления) элементов данных в ***динамическом множестве*** значений.

Хеш-функция при этом создаёт отображение множества ключевых значений во множество индексов соответствующих записей данных в массиве – хеш-таблице (рис. 26).

[14] Подробно алгоритм будет рассмотрен во второй части учебного пособия.

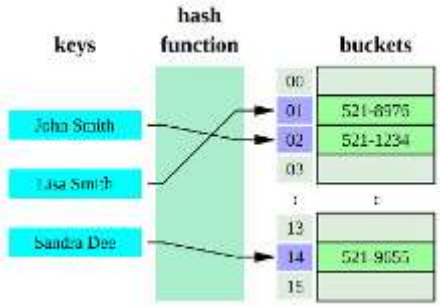


Рисунок 26. Индексы элементов динамического множества данных как результат хеширования значения ключевых полей элементов полезных данных

В этом случае при вводе ключа поиска программа вычислит хеш и затем в хеш-таблице обратится по индексу к искомой записи.

Алгоритм хеш-функции может быть основан на делении (модальная арифметика, полиномиальный хеш), умножении (хеширование Фибоначчи), на подходе под названием «универсальное хеширование», а также некоторых других.

Например, для алгоритма, основанного на делении, хеш-функция может быть реализована на основе модальной арифметики:

$$h = K \bmod Q, \qquad (1)$$

где K – ключевое значение, Q – наибольшее необходимое количество различных значений хеш-функции (и, как следствие, допустимое количество записей в динамическом множестве).

Если K – составное значение (например, строка символов), то его можно представить в виде полинома.

Одним из свойств хеш-функции является ***необязательность уникальности*** значений хеша для различных входных наборов данных. Это объясняет ненулевую вероятность возникновения **коллизии** – ситуации, когда по разным ключевым значениям может быть вычислено одинаковое хеш-значение. Таким образом, двум или более наборам данных может быть сопоставлен одинаковый индекс в массиве – а это недопустимо.

Для устранения (разрешения, преодоления) коллизии можно использовать методы **цепного хеширования** и хеш с **открытой адресацией**.

Цепным хешированием называется способ разрешения коллизий, когда динамическое множество полезных данных организуется в виде ***массива линейных списков***, состоящих из элементов с одинаковыми хеш-значениями, т.е. индексами в массиве (рис. 27).

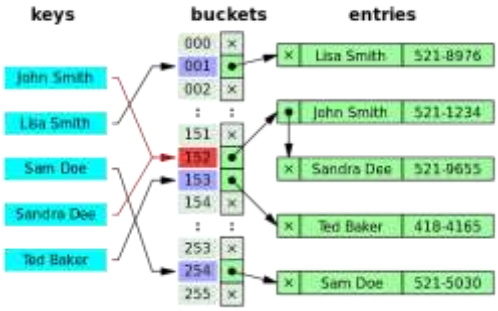


Рисунок 27. Схема организации цепного хеширования

При этом в хеш-таблице ключам сопоставляются индексы головных элементов этих списков в массиве.

Массив списков может стать на некотором этапе работы программы неоднородным – несколько длинных списков и множество пустых элементов массива.

С одной стороны, массив, даже пустой, занимает память. С другой стороны, время доступа к данным в списке линейное, а не константное, т.е. налицо снижение эффективности поиска.

На практике создают сначала небольшой массив, а по мере заполнения элементами перестраивают его, т.е. увеличивая размер с **рехешированием** (пересчетом хешей с новым значением Q).

Критерием необходимости перестройки массива является соотношение n/m – **коэффициент нагрузки**, где n – это количество уже имеющихся записей, m – длина массива. При достижении значения этого коэффициента 0,75+, следует увеличить длину массива вдвое. Это гарантирует, что длины списков будут относительно небольшими.

Другой способ преодоления коллизий – хеширование с ***открытой адресацией*** (рис. 28). Если в массиве в строке с определённым индексом записи нет, то адрес открыт и в соответствующую строку можно поместить новый элемент. Иначе – адрес закрыт (коллизия) и необходимо по некоему алгоритму осуществить последовательность проб – сместиться относительно закрытого адреса в поисках открытого. Все базовые операции (поиск, вставка, удаление элемента) так или иначе задействуют пробирование, но у каждой свои нюансы.

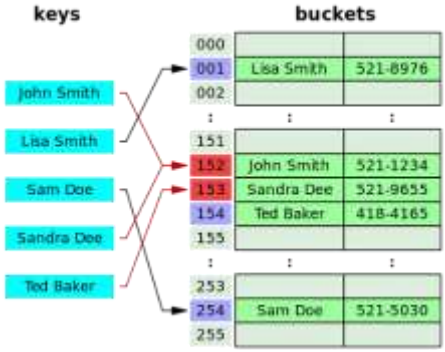


Рисунок 28. Пример заполнения массива на основе открытой адресации

Распространённые схемы пробирования: линейное, квадратичное пробирование, двойное хеширование.

В наиболее простой схеме ***линейного пробирования*** смещение относительно адреса коллизии кратно целочисленной константе (эту константу следует задать так, чтобы они с длиной массива были взаимно просты):

$$адрес=h(x)+ci \quad (2)$$

где i – номер попытки разрешить коллизию; c – константа, определяющая шаг перебора.

В ***квадратичной схеме*** пробирования шаг перебора сегментов нелинейно зависит от номера попытки найти свободный сегмент:

$$адрес=h(x)+ci+di^2 \quad (3)$$

где i – номер попытки разрешить коллизию, c и d – константы.

В схеме ***двойного хеширования*** смещение относительно закрытого адреса кратно величине второй хеш-функции, схожей, но не эквивалентной основной:

$$адрес=h(x)+ih_2(x) \quad (4)$$

В случае открытой адресации имеет смысл создать массив сразу наибольшей длины. В противном случае при постепенном заполнении массива записями будет всё более длительной процедура поиска открытого адреса. Затраты времени на перестройку этого массива лишь снизят эффективность всей программы.

ЛИТЕРАТУРА

1. Бхаргава А. Грокаем алгоритмы. Иллюстрированное пособие для программистов и любопытствующих. – СПб: Питер, 2017. – 288 с.
2. Вирт Н. Алгоритмы + структуры данных = программы. – М.: Мир, 1985. – 406 с.
3. Кнут Д.Э. Искусство программирования, том 3. Сортировка и поиск, 2-е изд. – М.: ООО «И.Д. Вильямс», 2018. – 832 с.
4. Кораблин Ю.П. Структуры и алгоритмы обработки данных : учебно-методическое пособие / Ю.П. Кораблин, В.П. Сыромятников, Л.А. Скворцова. – М.: РТУ МИРЭА, 2020. — 219 с.
5. Кормен Т.Х. и др. Алгоритмы: построение и анализ, 3-е изд. – М.: ООО «И.Д. Вильямс», 2013. – 1328 с.
6. Макконнелл Дж. Основы современных алгоритмов. Активный обучающий метод. 3-е доп. изд., - М.: Техносфера, 2018. – 416 с.
7. Седжвик Р. Фундаментальные алгоритмы на C++. Анализ/Структуры данных/Сортировка/Поиск. – К.: Издательство «Диасофт», 2001. – 688 с.
8. Скиена С. Алгоритмы. Руководство по разработке, - 2-е изд. – СПб: БХВ-Петербург, 2011. – 720 с.
9. Хайнеман Д. и др. Алгоритмы. Справочник с примерами на C, C++, Java и Python, 2-е изд. – СПб: ООО «Альфа-книга», 2017. – 432 с.
10. AlgoList – алгоритмы, методы, исходники [Электронный ресурс]. URL: http://algolist.manual.ru/ (дата обращения 15.03.2022).
11. Алгоритмы – всё об алгоритмах / Хабр [Электронный ресурс]. URL: https://habr.com/ru/hub/algorithms/ (дата обращения 15.03.2022).
12. НОУ ИНТУИТ | Технопарк Mail.ru Group: Алгоритмы и структуры данных [Электронный ресурс]. URL: https://intuit.ru/studies/courses/3496/738/info (дата обращения 15.03.2022).