Python в примерах

Дмитрий Купцов
2020

Оглавление

Введение

Язык Python на сегодняшний день является одним из самых популярных. Он прочно занимает лидирующие позиции на рынке и используется для решения таких задач как высоконагруженные веб-сервисы (для этих целей используются такие фреймворки как Django и Flask), обработка больших данных и решение сложных инженерных задач (здесь часто используют такие библиотеки как numpy и scipy), а также для работы с графическими интерфейсами (например, читатель может ознакомиться с библиотеками tkinter и PyQT) и даже для взаимодействия с микроконтроллерами. Python удобен тем, что он кросс-платформенный, некомпилируемый и достаточно прост в изучении.

Данная книга ориентирована на читателя, который не имеет представления о языке Python и хочет изучить основы алгоритмов. В основном эта книга предназначена для учеников старших классов, а также для взрослых читателей, которые хотят получить базовое представление об основных алгоритмах и структурах данных.

Эта книга состоит из пяти частей. В первой части книги приводятся базовые понятия из математики. Во второй части описываются ключевые компоненты языка Python. Затем, в третей главе, приводится описание базовых структур данных, таких как массивы, стеки, очереди, деревья, графы и т.д. В четвертой главе мы познакомим читателя с основными приёмами при решении алгоритмических задач. К примеру, тут мы расскажем о так называемом методе разделяй и властвуй, жадных алгоритмах, и закончим книгу рассказом о динамическом программирование. И наконец, в последней главе, мы предоставим читателю финальный проект - разработка утилиты для

ГЛАВА 1

Математические основы

Прежде чем мы сможем приступить к изучению алгоритмов нам необходимо ознакомиться с основными математическими понятиями и приёмами. В данной главе мы также приведем базовые понятия из теории вероятности. Часто такие знания являются необходимыми при стохастическом анализе алгоритмов. Например, когда требуется найти среднее время выполнения алгоритма, а не оценивать верхнюю или нижнюю границу для сложности. Но начнем мы наше приключение с таких основ как системы счисления, функции и математическая индукция.

1.1 Системы счисления

В повседневной жизни мы используем числа для некоторого количественного представления чего либо. И как правило мы используем десятичную систему счисления. Но существуют и другие системы: двоичная, восьмеричная, шестнадцатеричная, и т.д. Например, компьютеры обычно используют двоичную систему счисления, так как 0 можно представить как отсутствие электрического заряда, а 1 можно представить как его наличие. В программирование иногда бывает удобней работать с двоичной системой счисления, чем с привычной для нас десятичной системой. Для того что бы эффективно работать в разных системах, нужно вооружиться методами

Делимое	Целая часть	Остаток от деления
19	9	1
9	4	1
4	2	0
2	1	0
1	0	1

Таблица 1.1: Пошаговое вычисление числа в двоичной системе счисления

перевода из одной системы в другую. В данной главе мы рассмотрим, как это можно сделать.

Начнем мы с двоичной системой счисления. Например, число 19_{10} в десятичной системе будет равно 10011_2 в двоичной. Как мы этого достигли? Возьмем число 19 и представим его как линейную комбинацию: $19 = 2 \cdot 9 + 1$ (заметим, что в качестве множителя взято число 2: такой выбор связан с тем, что мы переводим в двоичную систему счисления). Запомним результат. Далее возьмем целую часть от деления, число 9, и также представим его как линейную комбинацию: $9 = 2 \cdot 4 + 1$. Будем продолжить данную операцию до тех пор, пока целая часть от деления не станет равна нулю. Представим наш пример в табличном виде (смотрите Таблицу 1.1).

В таблице, красным цветом отмечен наиболее значимый бит, синим же цветом отмечен наименее значимый бит. Тогда если мы запишем значения последней колонки начиная с наиболее значимого бита, мы получим желаемое представления числа 19_{10} в двоичной системе счисления.

Обратное преобразование проще. Для этого нам необходимо вычислить сумму: $\sum_{i=0}^{n} b_i 2^i$, где b_i - это iый бит в двоичном представлении числа. Используя данные из нашего примера, который мы описали выше, мы получим $1 \cdot 2^0 + 1 \cdot 2^1 + 0 \cdot 2^2 + 0 \cdot 2^3 + 1 \cdot 2^4 = 19_{10}$.

Теперь, когда мы знаем как преобразовывать в двоичную систему, мы можем записать этот алгоритм на языке Python:

```
def dec_to_bin(a):
  binary = "";
  while a != 0:
    binary = str(a % 2) + binary;
    a = (a - (a % 2))/2;
  return binary;

def bin_to_dec(binary):
  accumulator = 0;
```

```
for i in range(0, len(binary)):
  accumulator += int(binary[len(binary) - i - 1]) * (1 << i);
return accumulator;
```

В языке Python есть специальная функция для перевода в двоичную систему счисления - bin. Заметим, что число в двоичной системе в Python представляется как бинарное число, начинающееся со специального префикса - $0b$.

Такой же подход можно использовать для перевода числа из десятичной системы, скажем, в шестнадцатеричную. Например, возьмём число 181_{10}, которое будет равно $B5_{16}$. В шестнадцатеричной системе счисления, помимо чисел $0 - 9$, используются первые 6 букв латинского алфавита, т.е., A, B, C, D, E, F (легко заметить, что символ A будет в численном представление равен 10, а символ F - 15). Таким образом, чтобы перевести число $B5_{16}$ из шестнадцатеричной системы обратно в десятичную нужно вычислить сумму $\sum_{i=0}^{n} h_i \cdot 16^i$, где h_i - это численное представление символа в шестнадцатеричной системе счисления. Для удобства Python поддерживает специальную функцию hex, которая может перевести число из десятичной системы счисления в шестнадцатеричную. По условию все числа в шестнадцатеричной системе счисления начинаются с префикса 0x.

1.2 Функции

В математике функцией называется зависимость одной величины от другой. В любой функции, если это функция одной переменной, есть независимая и зависимая переменные. Например, в функции $y = 2^x$, y - это зависимая переменная, а x - это независимая переменная, т.е., при изменении x, соответствующим образом будет меняться и y.

В рамках данной книги наибольший интерес представляют несколько видов функций, а именно: показательные, логарифмические, полиномиальные. Все эти функции ведут себя по разному, но часто при анализе алгоритмической сложности нужно иметь представление о скорости роста функции. Например, на Графике 1.1 мы наглядно приводим скорости роста самых основных функций (так как обычно при анализе алгоритмов переменная величина - это размер входных данных, то все значения на оси X являются положительными целыми числами).

Показательные функции имеют вид $y = a^n$ (обычно за n берется размер входных данных, т.е., n - это переменная величина). Перечислим основные свойства показательных функций.

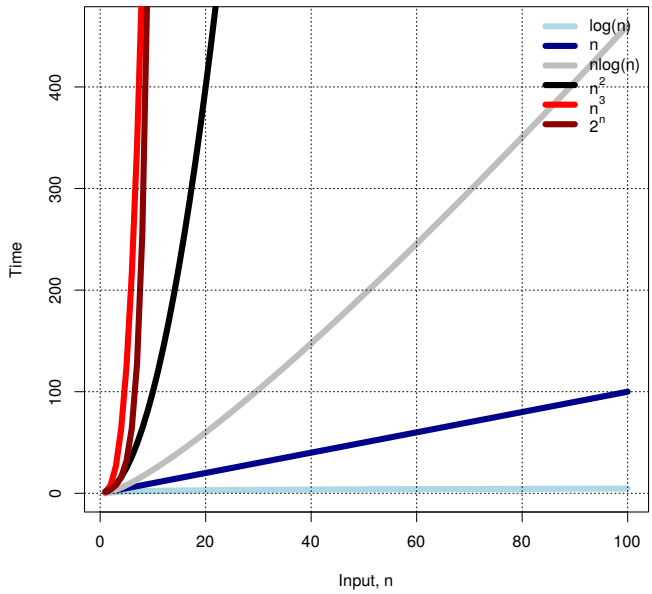

График 1.1: Скорость роста различных функций

Определение 1. Если основание показательной функции $0 < a < 1$, то функция монотонно убывает, если же $a > 1$, то функция монотонно возрастает.

Определение 2. Если основания показательных функций равны, то $a^n a^m = a^{n+m}$.

Определение 3. Если основания показательных функций равны, то $\frac{a^n}{a^m} = a^{n-m}$.

Определение 4. Пусть $y = a^n$, тогда $a^n + a^n = 2a^n$. Или более общий вид $\sum_{i=1}^{k} a^n = ka^n$.

Определение 5. Пусть $y = a^n$, тогда $y^m = a^{nm}$.

Логарифмическая функция представляется в виде $y = \log_a(n)$. Обратная этой функции - это показательная функция, которую мы уже рассмотрели ($a^y = n$). Также как и предыдущий класс функций, логарифмы часто встречаются в анализе алгоритмов. Рассмотрим свойства этих функций.

Теорема 1. Пусть дана логарифмическая функция $y = \log_a n^b$, тогда $y = b \log_a n$.

Так как $y = \log_a n^b$, то $a^y = n^b$. Но если мы возведём обе стороны уравнения в степень $1/b$, то получим $a^{y/b} = n^{b/b} = n$. Но по определению логарифма мы имеем $\frac{y}{b} = \log_a n$. Тогда помножив обе части уравнения на b получим $y = b \log_a n$, что и требовалось доказать.

Теорема 2. Если $y = \log_a(n)$, то $a^{\log_a(n)} = n$.

Так как $a^{\log_a(n)} = a^y$, а по определению логарифма $a^y = n$, то $a^{\log_a(n)} = n$. Что и требовалось доказать.

Теорема 3. Пусть дана логарифмическая функция $\log_a n$, тогда $\log_c n = \frac{\log_c n}{\log_c a}$.

Так как $y = \log_a n$, то $a^y = n$. И пусть $z = \log_c n$, $c^z = n$. Тогда, $a^y = c^z$. Прологарифмируем обе стороны уравнения, тогда $y \log_a a = z \log_a c$, откуда следует $z = \log_c n = \frac{\log_a n}{\log_a c}$. Что и требовалось доказать.

Теорема 4. $\log_a(nm) = \log_a n + \log_a m$

$\log_a(nm) = \log_a(a^{\log_a(n)} a^{\log_a(m)}) = \log_a a^{\log_a(n)+\log_a(m)} = \log_a(n) + \log_a(m)$. Что и требовалось доказать.

Теорема 5. $\log_a \frac{n}{m} = \log_a n - \log_a m$

$\log_a(n/m) = \log_a(a^{\log_a(n)}/a^{\log_a(m)}) = \log_a a^{\log_a(n)-\log_a(m)} = \log_a(n) - \log_a(m)$ Что и требовалось доказать.

Степенные функции $y = n^a$ также часто встречаются в анализе алгоритмов. Обычно, n - это переменная величина (размер входных данных), а $a \in \mathbb{N}$. Как мы увидим дальше, обычно сложность алгоритмов со вложенными циклами можно представить именно этими функциями.

1.3 Основные сведения о рядах

Обширный обзор рядов и способов их анализа можно найти в [1, 4]. В данном же разделе приведем несколько сведений об основных свойствах рядов и как их можно анализировать (в основном мы заинтересованы в рядах с конечным числом членов, так как они чаще встречаются на практике).

Рядом называется последовательность чисел, где каждый член отличается от предыдущего на некоторую величину (бывают конечно и знакочередующиеся ряды, но здесь мы такие не будем рассматривать). Например, ряд $0, 5, 10, 15, \ldots$ является арифметической прогрессией, в которой каждый член

больше предыдущего на 5. Еще один часто встречающийся ряд - это геометрическая прогрессия. Например, $1, 2, 4, 8, 16, 32, \ldots$ - это геометрическая последовательность, в которой i-ый член ряда можно найти, применив формулу $a_i = qa_{i-1} = q^{i-1}a_1$, где $q = 2$, а $a_1 = 1$. В данном разделе же приведем несколько примеров того, как можно анализировать суммы этих и некоторых других рядов (в основном нас интересуют ряды с конечным числом членов).

Например, пусть нам необходимо найти сумму следующего ряда:

$$S = \sum_{i=1}^{n} i = 1 + 2 + 3 + 4 + \ldots + n$$

Простой способ анализа, заключается в сложение первого и последнего членов ряда, и далее второго и предпоследнего членов, и т.д (здесь мы предполагаем, что количество членов является чётным числом). Тогда, заметив, что сумма таких пар равна $n + 1$, а количество пар равно $n/2$, можно предположить, что сумма ряда равна $\frac{n(n+1)}{2}$. Если же n - нечётное число, тогда можно доказать это утверждение, воспользовавшись методом немецкого математика Карла Гаусса, который он вывел в детстве. Возьмём $S + S = 2S$. Тогда, складывая первый и последний члены двух сумм получим

$$2S = (1 + n) + (n - 1 + 2) + \ldots + (1 + n) = (n + 1)n$$

Очевидно, что искомая сумма равна $S = \frac{n(n+1)}{2}$.

Рассмотрим более общий пример. Пусть мы хотим вычислить сумму ряда i^k, где $k \geq 1$. Но прежде, представим выражение $(a + b)^k$ в виде суммы. И так

$$(a + b)^k = C_k^0 a^k b^0 + C_k^1 a^{(k-1)} b^1 + ... + C_k^k a^0 b^k$$

где $C_k^m = \frac{k!}{(k-m)!m!}$ - это сочетание. Мы еще вернемся к этой формуле, когда будем рассказывать про правила вычисления вероятностей, а пока отметим, что в формуле, которую мы определили только что $k!$ - это факториал, который вычисляется как $k! = 1 \cdot 2 \cdot 3 \ldots k$. Причем $0! = 1$. Но вернемся к нашему разговору о вычислении суммы ряда. Возьмем для примера $(i-1)^2 = i^2 - 2i + 1$. Перенесём i^2 влево, домножим на -1 обе части уравнения и просуммируем:

$$\sum_{i=1}^{n} (i^2 - (i-1)^2) = \sum_{i=1}^{n} (2i - 1)$$

Заметим, что левая часть схлопывается и получается, что сумма равна n^2. В правой же части получаем $2S - n$, где S - это искомая сумма. Теперь,

если мы решим уравнение для S, то получим $S = \frac{n^2+n}{2} = \frac{n(n+1)}{2}$. Такой же математический приём можно применить и к другим суммам, например $\sum_{i=1}^{n} i^2$. Для этого разложим выражение

$$(i-1)^3 = C_3^0(-1)^0 i^3 + C_3^1(-1)^1 i^2 + C_3^2(-1)^2 i + C_3^3(-1)^3 i^0 = i^3 - 3i^2 + 3i - 1$$

Аналогично предыдущему примеру, перенесем i^3 влево, домножим на -1 и просуммируем обе части уравнения. Тогда получим:

$$\sum_{i=1}^{n}(i^3 - (i-1)^3) = \sum_{i=1}^{n}(3i^2 - 3i + 1)$$

Левая часть схлопывается и становится равной n^3, правую же часть можно представить как $3S - \frac{3n(n+1)}{2} + n$. После перестановки членов в левой и правой частях, получаем:

$$S = \frac{2n^3 - 2n + 3n(n+1)}{6} = \frac{2n(n^2-1) + 3n(n+1)}{6} = \frac{2n(n-1)(n+1) + 3n(n+1)}{6} = \frac{(2n(n-1) + 3n)(n+1)}{6} = \frac{(2n^2+n)(n+1)}{6} = \frac{n(2n+1)(n+1)}{6}$$

Данный метод хорош тем, что его можно применить для нахождения суммы любого ряда вида i^k, при условии, что $k \geq 1$

Ещё один часто встречающийся ряд - это геометрическая прогрессия. Сумму такого ряда можно представить как:

$$S = \sum_{i=0}^{n} aq^i = a(1 + q + q^2 + q^3 + \ldots + q^n)$$

Простой способ нахождения суммы этого ряда заключается перемножение S на q и решение следующего уравнения $S - qS = a(1 - q^{n+1})$ (заметим, правая часть получилась путём вычитания qS из S). Тогда, сумма ряда равна $S = \frac{a(1-q^{n+1})}{1-q}$.

Иногда найти сумму ряда бывает сложно и проще дать аппроксимацию. Например, рассмотрим гармонический ряд $1, \frac{1}{2}, \frac{1}{3}, \ldots, \frac{1}{n}$. Тогда сумма этого ряда тогда будет равна

$$H_n = \sum_{i=1}^{n} \frac{1}{i} \approx \int_1^n \frac{dx}{x} = \ln(n)$$

1.4 Множества

В математике множеством называется совокупность объектов любого типа [3]. Например, множеством является набор букв латинского алфавита - $\{a, b, c, \ldots, z\}$, или множества все целых чисел - $x \in \mathbb{Z}$. Обычно, для обозначения множеств, используют прописные буквы латинского алфавита, а элементы заключают в фигурные скобки. Например, $A = \{a, b, c\}$ или $X = \{x : x > 10\}$. Принадлежность элемента x к множеству обычно обозначают символом $\in$. Приведём несколько определений.

Определение 6. Пустое множество - это такое множество, которое не содержит элементы. Обозначается такое множество как $\emptyset$.

Определение 7. A является подмножеством множества B, если все элементы множества A также являются элементами множества B. Обозначается это свойство следующим образом: $A \subset B$.

Определение 8. Универсальное множество - это такое множество, которое содержит все элементы. Обычно такое множество обозначается как Ω. Например, пусть Ω - множество всех возможных значений игральной кости, т.е., $\Omega = \{1, 2, 3, 4, 5, 6\}$.

Определение 9. Если элементы некоторого множества A' принадлежат универсальному множеству, но не принадлежат множеству A, то такое множество называется дополнением множества A: $A' = \{x : x \in \Omega \wedge x \notin A\}$.

Над множествами, как и над числами, можно производить операции. Таким образом, далее мы рассмотрим операции сложения (или объединения), пересечения и вычитания множеств. А после приведем (без доказательств) основные свойства множеств.

Определение 10. Объединением множества A и B называется такое множество, которое содержит элементы, принадлежащие или множеству A, или множеству B. Математически эта операция записывается следующим образом: $A \cup B = \{x : x \in A \vee x \in B\}$.

Определение 11. Пересечением множества A и B называется такое множество, которое содержит элементы, принадлежащие одновременно и множеству A, и множеству B. Математически эта операция записывается следующим образом: $A \cap B = \{x : x \in A \wedge x \in B\}$.

Определение 12. Разностью множеств A и B называется такое множество, которое содержит элементы множества A, но не содержит элементы принадлежащие множеству B. Математически эта операция записывается следующим образом: $A \setminus B = \{x : x \in A \wedge x \notin B\}$.

Мы еще вернемся к обсуждению операций над множествами, когда будем рассматривать работу с множествами в языке Python. А пока, рассмотрим основные свойства множеств, которые часто встречаются на практике (смотрите Таблицу 1.2).

Идемпонентный закон	$A \cup A = A$	$A \cap A = A$
Закон тождества	$A \cup \emptyset = A$ $A \cup \Omega = \Omega$	$A \cap \emptyset = \emptyset$ $A \cap \Omega = A$
Закон дополнения	$A \cup A' = \Omega$	$A \cap A' = \emptyset$
Закон коммутативности	$A \cup B = B \cup A$	$A \cap B = B \cap A$
Закон ассоциативности	$(A \cup B) \cup C =$ $A \cup (B \cup C)$	$(A \cap B) \cap C =$ $A \cap (B \cap C)$
Закон дистрибутивности	$A \cup (B \cap C) =$ $(A \cup C) \cap (A \cup B)$	$A \cap (B \cup C) =$ $(A \cap C) \cup (A \cap B)$
Закон Де Моргана	$(A \cup B)' = A' \cap B'$	$(A \cap B)' = A' \cup B'$

Таблица 1.2: Основные свойства множеств

1.5 Алгоритмическая сложность

Алгоритм - это некая последовательность инструкций компьютера, которые необходимо воспроизвести, чтобы решить поставленную задачу. Как только алгоритм составлен и написан в виде инструкций языка, например такого как Python, его желательно проанализировать, чтобы знать его производительность. Сложность алгоритма обычно выражают как функцию $T(n)$, где n - это длина входных данных.

Обычно, вычислительную сложность записывают используя правила большого O (или $Big\ O$, как эти правила принято называть в иностранной литературе). Введем несколько определений. Пусть вычислительная сложность задана функцией $T(n)$, тогда [5]:

Определение 13. $T(n) = O(f(n))$ если есть некая константа $c > 0$ и функция $f(n)$ такая, что $T(n) \leq c \cdot f(n)$, где $n > n_0$.

Определение 14. $T(n) = \Omega(g(n))$ если есть некая константа $c > 0$ и функция $g(n)$ такая, что $T(n) \geq c \cdot g(n)$, где $n > n_0$.

Определение 15. $T(n) = \Theta(h(n))$ если $T(n) = O(h(n))$ и $T(n) = \Omega(h(n))$.

Иными словами, с помощью O описывают верхнюю границу для сложности (памяти или вычислений), Ω используется для определения нижней границы, а используя Θ можно определить более строгую границу. Например, если $T(n) = 10n$, то конечно можно определить $T(n) = O(n^2)$ или даже $T(n) = O(n^3)$, но более точным определением будет $T(n) = \Theta(n)$, так как можно выбрать в виде функции $h(n) = n$, а две константы выбрать как $c_1 = 1$, а $c_2 = 100$. Но так как не всегда можно дать точную оценку для сложности алгоритма, то обычно используют оценку O, так как её часто бывает проще найти.

Рассмотрим несколько примеров того, как можно анализировать циклы. Если в алгоритме всего один цикл (без вложенных циклов), тогда задача сводиться к подсчёту операций выполняемых в теле цикла. Но так как цикл выполняется n раз (где n - это размер входных данных), то нахождение общего времени выполнения алгоритма можно представить в виде простой суммы. Например, в задаче нахождения максимальной суммы в последовательности (алгоритм приведен ниже), на каждой итерации алгоритма выполняется c инструкций. Тогда, общее количество операций $\sum_{i=1}^{n} c = nc$. Сложность же данного алгоритма можно оценить как $\Theta(n)$.

```
def maximum_subsequence(a):
  max_sum = 0;
  current_sum = 0;
  start = 0;
  current_start = 0;
  end = 0;
  for i in range(0, len(a)):
    current_sum += a[i];
    if current_sum > max_sum:
      max_sum = current_sum;
      end = i + 1;
      start = current_start;
    if current_sum < 0:
      current_sum = 0;
      if i + 1 < len(a):
        current_start = i + 1;
  return (max_sum, a[start:end]);
```

Если же есть вложенные циклы, как например в задаче сортировки массива пузырьком, который мы приводим ниже, то сначала нужно проанализировать

сложность вложенного цикла и только потом приступать к анализу внешнего цикла. Так, в примере, который мы приводим ниже вложенный цикл выполняет $4(n-i)$ операций (здесь анализ аналогичен тому, который мы его провели в предыдущем примере). Тогда общая сложность алгоритма может быть представлена в виде следующей суммы $\sum_{i=0}^{(n-1)} 4(n-i) = \sum_{i=0}^{(n-1)} 4n - \sum_{i=0}^{(n-1)} 4i = 4n^2 - 4\frac{n(n-1)}{2} = 4n^2 - 2n^2 + 2n = 2n^2 + 2n$. Следовательно, сложность алгоритма может быть оценена как $O(n^2)$.

```
def bubble_sort(a):
  for i in range(0, len(a)):
    for j in range(i, len(a)):
      if a[i] > a[j]:
        s = a[i];
        a[i] = a[j];
        a[j] = s;
  return a;
```

Приведём еще один известный, но неэффективный алгоритм сортировки - сортировка вставками. Сложность данного алгоритма такая же, как и у сортировки пузырьком - в худшем случае алгоритму требуется $O(n^2)$ вычислений. Но в отличии от предыдущего алгоритма, в лучшем случае этому методу требуется $\Theta(n)$ вычислений (например, если массив уже отсортирован).

```
def insertion_sort(a):
  for i in range(1, len(a)):
    j = i - 1;
    key = a[i];
    while True:
      if a[j] > key:
        a[j + 1] = a[j];
        a[j] = key;
        j -= 1;
      if j <= 0 or a[j] <= key:
        break;
  return a;
```

Аналогичный прием можно применять и для алгоритмов с большим числом вложенных циклов.

1.6 Индукция и прочие приёмы доказательств

Математическая индукция - это метод, который позволяет доказать истинность утверждения путем перехода от частного к общему. Обычно в математической

индукции на первом шаге доказывается, что утверждение верно для некоторого значения (обычно небольшого). Далее, предполагается, что выражение истинно для некоторого большого значения n. И наконец, доказывается, что выражение правдиво для $n+1$.

Приведем пример. Пусть мы предполагаем, что $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$. Убедимся, что утверждение истинно для $n \in \{1, 2, 3\}$. Что безусловно очевидно, если мы вычислим сумму используя левое и правое выражения. Далее предположим, что выражение истинно для некоторого n. И наконец, докажем, что при $n+1$, сумма равна $\frac{(n+1)(n+2)}{2}$. Действительно,

$$\sum_{i=1}^{n+1} i = \frac{n(n+1)}{2} + n + 1 = \frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

Что и требовалось доказать.

Помимо индукции существуют и другие подходы к доказательству математических свойств. Например, доказать утверждение можно противоречием или же контрпримером.

Приведём пример доказательства контрпримером. Докажем что, $2^n < n!$, где $n > 4$. Предположим обратное $2^n > n!$, но взяв $n = 5$ мы получаем $32 < 120$, что противоречит утверждению.

1.7 Рекурсия

В программирование рекурсивная функция - это такая функция, которая вызывает сама себя. При каждом вызове размер входных данных должен уменьшаться и рекурсивные вызовы прекращаются (обычно) когда размер входных данных равен единице. Примером рекурсии может служить функция вычисления наибольшего общего делителя. Мы еще столкнемся с ней, когда будем описывать бинарные операторы. Сейчас же мы приведем ее как пример того, как рекурсия выглядит в программировании. Вслед за этим мы рассмотрим как можно анализировать рекурсии (в математическом смысле).

```
def gcd(a, b):
 if b == 0:
  return a;
 return gcd(b, a % b);
```

Для того, чтобы понять как изменяется размер входных данных в описанном выше алгоритме, приведём следующую теорему.

Теорема 6. Для любых целых чисел a и b (где $a > b$) справедливо неравенство: $a \ mod \ b < \frac{a}{2}$

Рассмотрим два случая: (i) это когда b входит в a точно один раз, т.е., $a = b+r$ (ii) это когда b входит в a несколько раз, т.е., $a = bq + r$. Для случая (i) мы имеем $r < b$, но так как b входит в а один раз, очевидно, что $b > a/2$ (иначе $q > 1$) и соответственно $r < a/2$; (ii) так как b входит в a несколько раз, то $q \geq 2$, соответственно $a > qb$ (иначе r было бы равно 0) из чего следует, что $b < a/q \leq a/2$. Но так как $b > r$ (всегда), то и $r < a/2$. Что и требовалось доказать.

В рекурсии, время необходимое для вычисления обычно выражается как $T(n)$, где n - это размер входных данных. Используя нашу теорему, мы можем составить следующее рекуррентное соотношение (мы полагаем, что при каждом рекурсивном вызове b уменьшается как минимум вдвое).

$$T(n) = T(n/2) + 1 = T(n/4) + 2 = T(n/8) + 3 = T(n/2^i) + i$$

Зная, что когда размер входных данных равен единице рекурсия прекращается, то мы можем вычислить i через $n/2^i = 1$ (обратите внимание, что это предположение следует из $T(n/2^i) = T(1)$). Далее мы имеем $n = 2^i$. Прологарифмировав обе части уравнения, получаем $i = \log_2(n)$, тогда выкладки, изложенные выше, дают нам возможность предположить, что сложность алгоритма нахождения наибольшего общего делителя будет $O(\log(n))$.

Стоит отметить, как мы вывели $T(n)$: зная, что при каждом вызове, размер входных данных уменьшается в двое, то $T(n/2) = T(n/2 \cdot 1/2) + 1 = T(n/4) + 1$. Подставляя данное значение в выражение $T(n) = T(n/2) + 1$, мы можем вычислить $T(n)$ через $T(n/4)$ таким образом, получая $T(n) = T(n/4) + 2$. Продолжая этот ход мыслей, мы получаем выражение $T(n) = T(n/2^i) + i$.

1.8 Булева логика

Булева логика - это раздел математики, изучающий логические высказывания и операции над ними. Логические высказывания могут быть простыми и составными. Например, составными высказываниями являются такие высказывания, в которых есть несколько простых логических высказываний объединённых логическими операциями. В данном разделе мы коротко ознакомимся с такими логическими операциями как: конъюнкция (или операция И), дизъюнкция (или логическая операция ИЛИ), импликация

(или следование), равносильность (или эквивалентность), а также строгая дизъюнкция. Рассмотрим простейшие примеры булевой алгебры, а потом рассмотрим её основные свойства.

Начнем мы наше обсуждение с такой операции как отрицание. Пусть дано высказывание, которое может принимать значения 1 и 0 (или же истина и ложь). Тогда справедливы утверждения описанные в Таблице 1.3.

A	$\neg A$
0	1
1	0

Таблица 1.3: Логическая операция отрицания

Результат конъюнкции, или же иными словами операции логического И, становится истинным тогда, когда оба выражения истины, и становиться ложью, тогда, когда хотя бы одно выражение ложь. Составим следующую таблицу истинности (смотрите Таблицу 1.4).

A	B	$A \wedge B$
0	0	0
1	0	0
0	1	0
1	1	1

Таблица 1.4: Конъюнкция

Дизъюнкция - это логическая операция, результат которой будет истиной только тогда, когда хотя бы одно выражение является истиной. Запишем эту операцию в виде таблицы истинности (смотрите Таблицу 1.5).

A	B	$A \vee B$
0	0	0
1	0	1
0	1	1
1	1	1

Таблица 1.5: Дизъюнкция

Импликация - это логическая операция, которая гласит: из лжи может следовать что угодно, а из истины только истина. В Таблице 1.6 представим

результаты этой логической операции. Заметим, что импликацию можно заменить следующим сложным логическим высказыванием: $(\neg A \vee B)$.

A	B	$A \to B$
0	0	1
0	1	1
1	0	0
1	1	1

Таблица 1.6: Импликация

Операция эквивалентности гласит, что если оба логических высказывания либо истина либо ложь, тогда результат всегда истина (смотрите Таблицу 1.7). Заметим, что операцию эквивалентности можно заменить следующим сложным логическим высказыванием: $(\neg A \vee B) \wedge (A \vee \neg B)$.

A	B	$A \leftrightarrow B$
0	0	1
0	1	0
1	0	0
1	1	1

Таблица 1.7: Эквивалентность

И наконец, приведем таблицу (смотрите Таблицу 1.8) истинности для операции строгой дизъюнкции (в языках программирования такую операцию часто называют еще исключающее ИЛИ). Исключающее или можно заменить следующим сложным логическим высказыванием: $(\neg A \wedge B) \vee (A \wedge \neg B)$.

A	B	$A \oplus B$
0	0	0
0	1	1
1	0	1
1	1	0

Таблица 1.8: Строгая дизъюнкция

Заметим, что все эти логические операции часто используются в языках программирования, когда составляются ветвления. Мы еще вернемся к

Закон коммутативности	$A \vee B = B \vee A$ $A \wedge B = B \wedge A$
Закон ассоциативности	$(A \vee B) \vee C = A \vee (B \vee C)$ $(A \wedge B) \wedge C = A \wedge (B \wedge C)$
Закон дистрибутивности	$A \vee (B \wedge C) = (A \vee B) \wedge (A \vee C)$ $A \wedge (B \vee C) = (A \wedge B) \vee (A \wedge C)$
Закон непротиворечия	$A \wedge \neg A = 0$
Закон исключения третьего	$A \vee \neg A = 1$
Закон двойного отрицания	$\neg(\neg A) = A$
Законы Де Моргана	$\neg(A \vee B) = \neg A \wedge \neg B$ $\neg(A \wedge B) = \neg A \vee \neg B$
Закон рефлексии	$A \vee A = A$ $A \wedge A = A$
Законы поглощения	$A \vee (A \wedge B) = A$ $A \wedge (A \vee B) = A$ $A \vee (\neg A \wedge B) = A \vee B$
Свойства логических констант	$A \vee 1 = 1$ $A \wedge 1 = A$ $A \vee 0 = A$ $A \wedge 0 = 0$

Таблица 1.9: Основные свойства булевой алгебры

обсуждению составления логических выражений, когда будем обсуждать ветвления и логические операторы в языке Python. А пока в Таблице 1.9 приведём основные свойства булевой алгебры.

Операция	Приоритет
Выражения в скобках	1
Логическое отрицание	2
Логическое И (конъюнкция)	3
Логическое ИЛИ (дизъюнкция)	4
Следование (импликация)	5
Равносильность (эквиваленция)	6

Таблица 1.10: Приоритеты операций

Стоит также отметить, что не все операторы имеют один и тот же приоритет. Иными словами, некоторые операции должны выполняться прежде других.

Так, например, операции, заключенные в круглые скобки, должны выполняться прежде других операций. Для того, что бы понять какие операции должны выполнятся прежде других, приведём следующую таблицу (обратите внимание на Таблицу 1.10).

1.9 Базовые сведения из теории вероятности

В данном разделе приведём несколько основных сведений о теории вероятности. Более подробный курс можно пройти, прочитав [2, 3].

1.9.1 Классическое определение вероятности

Событием в теории вероятности называется какой либо исход, результат испытания. Например, при однократном бросании монеты возможными исходами могут быть выпадение орла или решки. События обычно обозначаются заглавными латинскими буквами. Таким образом, вероятность выпадения решки (появление события A), при однократном бросании монеты, можно записать в виде $P(A) = \frac{1}{2}$. Обратное же событие - выпадение орла, можно записать как $P(\bar{A}) = 1 - P(A)$. В данном случае выпадение орла и решки имеют одинаковые вероятности и поэтому называются равновозможными. Если же события неравновероятные, то обычно задают закон распределения вероятностей, который каждому событию сопоставляет соответствующую вероятность. Мы еще вернёмся к обсуждению законов распределения вероятностей чуть позже. А пока отметим, что в классической теории вероятности, вероятность наступления события A может быть вычислена как $P(A) = \frac{m}{n}$, где m - это количество исходов благоприятствующих событию, а n - это количество всех исходов. Отметим ещё один важный момент. Если события являются единственно возможными (т.е., в результате испытания должно произойти хотя бы одно событие) и несовместными (т.е., ни одно событие не влечет за собой появление любого другого события), то такие события образуют полную группу. Например, в нашем примере с бросанием монеты, события (выпадение орла или решки) образуют полную группу.

Отметим основные свойства вероятностей. Так например, вероятность наступления события заключена между нулём и единицей, т.е., $0 \leq P(A) \leq 1$. Вероятность наступления достоверного события (событие, которое наступает всегда) будет равно единице, т.е., $P(\Omega) = 1$. Вероятность же события, которое никогда не наступит (т.е., невозможное событие) равно нулю, или $P(\emptyset) = 0$.

Сумма вероятностей единственно возможных и несовместных исходов равна единице, т.е., $\sum_{i=1}^{n} P(A_i) = 1$.

Вернёмся к обсуждению способов вычисления вероятностей. Обычно, чтобы вычислить количество благоприятных и общее количество исходов пользуются комбинаторикой - разделом математики, который в том числе предлагает инструменты необходимые для подсчета количества исходов.

Определим правило сложения следующим образом: Пусть элемент A_i может быть выбран n_i способами (например, в корзине имеется 10 красных шаров, тогда A_i - это красный шар, а $n_i = 10$ - это количество таких шаров). Тогда выбор одного элемента (или A_1, или A_2, и т.д.) можно произвести $n = \sum_{i=1}^{k} n_i$ способами.

Например, пусть в урне имеется 100 шаров: 30 белых шаров, 10 красных шаров, и 60 синих шаров. Тогда существует $n_1 + n_2 = 30 + 10 = 40$ способов выбора одного шара: либо белого, либо красного. А вероятность того, что выбранный шар будет либо белый, либо красный можно вычислить по следующей формуле: $P(A) = \frac{n_1+n_2}{n_1+n_2+n_3} = \frac{40}{100} = 0.4$. Заметим, что общее количество $n = \sum_{i=1}^{k} n_i$.

Теперь определим правило умножения. Пусть выбрать A_1 элемент можно n_1 способами, после этого элемент A_2 можно выбрать n_2 способами, и т.д. Тогда последовательность $A_1 A_2 \dots A_k$ можно выбрать $n_1 \cdot n_2 \cdot \dots \cdot n_k$ способами.

Так, например, пусть даны 26 букв латинского алфавита. И пусть задача состоит в выборе трёх букв случайным образом (без возврата). Используя правило умножения мы можем получить $n = n_1 \cdot n_2 \cdot n_3 = 26 \cdot 25 \cdot 24 = 15600$ способов выбора трёх букв. Если же мы будем производить возврат на каждом шаге, то способов выбора трех букв будет больше, $n = 26^3 = 17576$.

Данное правило еще называют размещением. Так если нужно выбрать m элементов из n элементов без возврата и с учетом порядка, то число размещений можно определить как:

$$A_n^m = n \cdot (n-1) \cdot (n-2) \cdot (n-3) \dots (n-m+1) = \frac{n!}{(n-m)!}$$

Заметим, что $n!$ - это формула факториала, которая равна $n! = 1 \cdot 2 \dots n$, а $0! = 1$. Например, пусть дано множество $\{a, b, c\}$ и нужно выбрать два элемента случайным образом с учётом порядка. Тогда все возможные размещения будут $\{a, b\}$, $\{b, a\}$, $\{a, c\}$, $\{c, a\}$, $\{b, c\}$, $\{c, b\}$, т.е., их шесть. Этот результат согласуется с формулой, которую мы привели ранее.

Если же порядок не нужно учитывать, то тогда пользуются формулой сочетаний:

$$C_n^m = \frac{n!}{(n-m)!m!}$$

Например, пусть дано множество $\{a, b, c\}$. Тогда все сочетания будут $\{a, b\}, \{a, c\}, \{b, c\}$, т.е., их три. Такой же результат можно получить, применив формулу сочетаний.

Приведём пример непосредственного вычисления вероятностей. Пусть на отдельных карточках написаны буквы Т, Е, О, Р, И, Я. И пусть нужно выбрать (i) три и (ii) шесть карточек наугад. Задача состоит в нахождении вероятности получения (i) слова "ТОР", (ii) слова "ТЕОРИЯ". Для первого случая мы имеем $m = 1$ (так как всего одна комбинация даёт нужный расклад карточек), а $n = A_6^3 = 120$ (так как нам нужно учитывать порядок, то применяем формулу размещений), тогда $P(A) = \frac{1}{120}$. Для второго же случая мы имеем $m = 1$ (опять всего лишь один расклад карточек даёт нужный результат), а $n = A_6^6 = 6! = 720$ (так как нам интересны перестановки всех шести карточек), тогда $P(B) = \frac{1}{720}$.

1.9.2 Мода, медиана и статистические моменты

Иногда при анализе алгоритмов необходимо вывести среднюю сложность неких операций. Тут можно выделить несколько основных свойств распределений. Например, иногда можно в качестве точечной оценки дать моду распределения - наиболее часто встречающееся значение. Иногда же, используют медиану - значение, которое больше ровно половины всех остальных значений.

Конечно, основной инструмент при анализе алгоритмов - это математическое ожидание. Для дискретных случайных величин математическое ожидание можно определить следующим образом:

$$E[X] = \sum_{i=0}^{n} p_i x_i$$

Математическое ожидание также называется моментом первого порядка. Существуют и другие моменты. Например, моментом второго порядка называется дисперсия, которую можно вычислить по следующей формуле:

$$D[X] = \sum_{i=0}^{n} p_i (x_i - E[X])^2$$

1.9.3 Основные законы распределения

Существует множество законов распределения вероятностей, как дискретных, так и не дискретных. В анализе же алгоритмов, как правило, встречаются в основном только дискретные законы распределения вероятностей.

Таким образом, два основных закона распределения, которые можно встретить на практике при анализе алгоритмов - это геометрическое и биномиальное распределения. На Графике 1.2 мы приводим эти распределения.

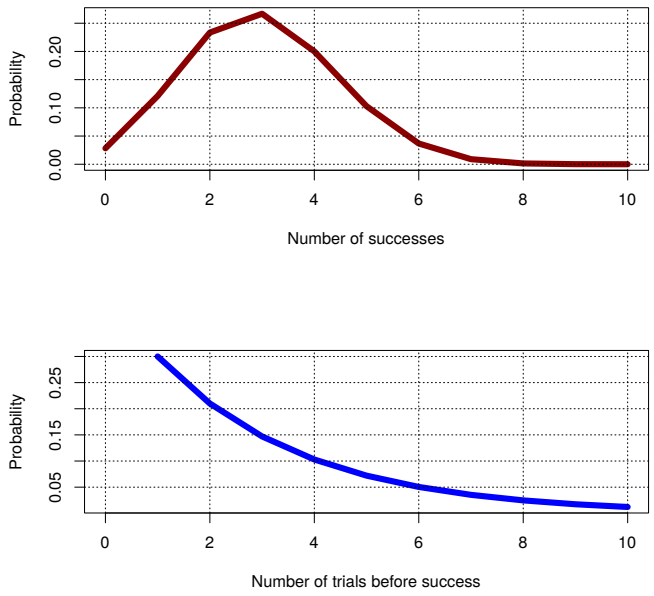

График 1.2: Геометрическое (нижний график) и биномиальное (верхний график) распределения вероятностей

Таким образом, геометрическое распределение (здесь p - это вероятность успеха в испытание, а $m \in \{1, 2, 3, 4, 5, \ldots\}$ - это количество попыток до первой удачи) можно обозначить следующей формулой:

$$P(X = m) = p(1 - p)^{m-1}$$

Математические ожидания можно вычислять с помощью так называемых генерирующих функций. Так, для геометрического закона распределения

вероятностей математическое ожидание можно найти следующим образом. Для начала найдем математическое ожидание $E[e^{Xt}]$:

$$m_X(t) = E[e^{Xt}] = \sum_{m=1}^{\infty} e^{mt} p q^{m-1} = \frac{p}{q} \sum_{m=1}^{\infty} (e^t q)^m = p \frac{e^t}{(1 - e^t q)}$$

Далее вычислим производную функции $m_X(t)$:

$$\frac{d}{dt} m_X(t) = p \frac{e^t}{(1 - e^t q)^2}$$

Вычислив производную в $t = 0$ получим математическое ожидание для геометрического закона распределения:

$$\frac{d}{dt} m_X(0) = E[X] = p \frac{1}{(1 - q)^2} = \frac{1}{p}$$

Биномиальный же закон распределения вероятностей (здесь p - это вероятность успеха, m - это количество удач, а n - это количество всех испытаний) можно выразить через следующую формулу:

$$P(X = m) = C_n^m p^m (1 - p)^{n-m}$$

Для данного закона распределения справедлива следующая формула математического ожидания $E[X] = \sum_{m=0}^{n} m C_n^m p^m (1 - p)^{n-m} = np$. Найдём математическое ожидание с помощью генерирующей функции. Так, например, найдем математическое ожидание $E[e^{Xt}]$:

$$m_X(t) = E[e^{Xt}] = \sum_{m=0}^{n} e^{mt} C_n^m p^m (1 - p)^{n-m} = (pe^t + q)^n$$

Далее, найдём производную полученного выражения:

$$\frac{d}{dt} m_X(t) = n(pe^t + q)^{n-1} pe^t$$

Вычислив полученное выражение в точке $t = 0$ получаем:

$$\frac{d}{dt} m_X(0) = E[X] = n(pe^0 + q)^{n-1} pe^0 = np$$

1.9.4 Зачем нужна теория вероятности в изучении алгоритмов?

Часто на практике необходимо дать среднюю сложность алгоритма, а не например, максимальную, или сложность получающуюся в худшем случае. И тут как правило не обойтись без знаний теории вероятности и математической статистики. Приведем пример того как можно проанализировать сложность следующего алгоритма на языке Python:

```
from random import randint

def collision(size = 100):
  x0 = randint(0, size);
  while True:
    x1 = randint(0, size);
    if x0 == x1:
      break;
```

Данный алгоритм ищет коллизии и прекращается тогда, когда найдено совпадение. Проанализируем вычислительную сложность. Можем заметить, что вероятность нахождения совпадения на каждом шаге равна $P(A) = p = \frac{1}{n}$, где n - это общее количество ячеек или урн. Соответственно вероятность того, что совпадения не будет, равна $P(\bar{A}) = q = 1-p$. Тогда вероятность коллизии на i-ом шаге можно вычислить как $P(X = i) = pq^{i-1}$. Заметим, это распределение подчинено геометрическому закону. Зная, что среднее значение геометрического распределения равно $E[X] = \sum_{i=1}^{\infty} ipq^{i-1} = \frac{1}{1-q}$, можно утверждать, что сложность данного алгоритма $\Theta(\frac{1}{1-q})$ (здесь мы говорим о среднем значение, а не о худшем случае или лучшем случае, так как в худшем случае алгоритм может вообще не закончить свое выполнение, а в лучшем случае алгоритм может остановиться после первого шага).

Приведем еще один пример стохастического алгоритма. Пусть задача состоит в нахождении случайной перестановки чисел из множества. Например, пусть дано множество $\{1, 2, 3\}$, тогда возможными перестановками будут $\{1, 2, 3\}$, $\{2, 1, 3\}$, $\{1, 3, 2\}$, $\{3, 2, 1\}$, $\{3, 1, 2\}$, $\{2, 3, 1\}$. Приведем (неэффективную) реализацию алгоритма (эффективный алгоритм требует время $O(n)$).

```
from random import randint

def permutations(n):
  result = [0] * n;
  used = [False] * n;
  j = randint(0, n);
  used[j] = True;
```

Шаг	$P(A) = p$	$P(\bar{A}) = q$	Мат. ожидание
1	$(n-1)/n$	$1/n$	$\sum_{j=1}^{\infty} jpq^{j-1} = \frac{1}{(1-q)} = \frac{n}{(n-1)}$
2	$(n-2)/n$	$2/n$	$\sum_{j=1}^{\infty} jpq^{j-1} = \frac{1}{(1-q)} = \frac{n}{(n-2)}$
3	$(n-3)/n$	$3/n$	$\sum_{j=1}^{\infty} jpq^{j-1} = \frac{1}{(1-q)} = \frac{n}{(n-3)}$
...	...	...	...
i	$(n-i)/n$	i/n	$\sum_{j=1}^{\infty} jpq^{j-1} = \frac{1}{(1-q)} = \frac{n}{(n-i)}$

Таблица 1.11: Пошаговое вычисление ожидаемого времени выполнения алгоритма нахождения перестановок

```
result[0] = j;
for i in range(1, n):
  while True:
    r = randint(0, n - 1);
    if not used[r]:
      result[i] = r;
      used[r] = i;
      break;
return result;
```

Заметим, что на i-ом шаге вероятность нахождения незанятого слота составляет $P(A) = p = \frac{n-i}{n}$ (вероятность обратного события будет $P(\bar{A}) = q = 1 - p = \frac{i}{n}$). Очевидно, что как и в предыдущем примере, вероятность того, что свободная ячейка найдется после j-ой итерации на i-ом шаге, будет равна $P(X = j) = pq^{j-1}$. Составим таблицу, в которой приведем ожидаемое время выполнения алгоритма на каждом шаге (смотрите Таблицу 1.11).

Очевидно, что ожидаемое время выполнения алгоритма на i-ом шаге равно $\frac{n}{n-i}$. Тогда, просуммировав все значения в последней колонке, мы сможем найти среднее время всего алгоритма, т.е., $\sum_{i=1}^{n-1} \frac{n}{(n-i)}$. Так как вычисление этой суммы достаточно сложно, то проще дать верхнюю оценку алгоритма, например $O(n^2)$. Здесь мы продемонстрировали не только как можно проводить анализ стохастического алгоритма, но и тот факт, что иногда проще дать верхнюю оценку алгоритма, как мы упомянули это в Главе 1.5. Читатель может самостоятельно попробовать дать точную оценку Θ. Однако, заметив, что данная сумма напоминает гармонический ряд, мы можем аппроксимировать сумму интегралом (на Графике 1.3 отображено расхождение реальной суммы и аппроксимации) и тем самым дать нижнюю оценку сложности алгоритма:

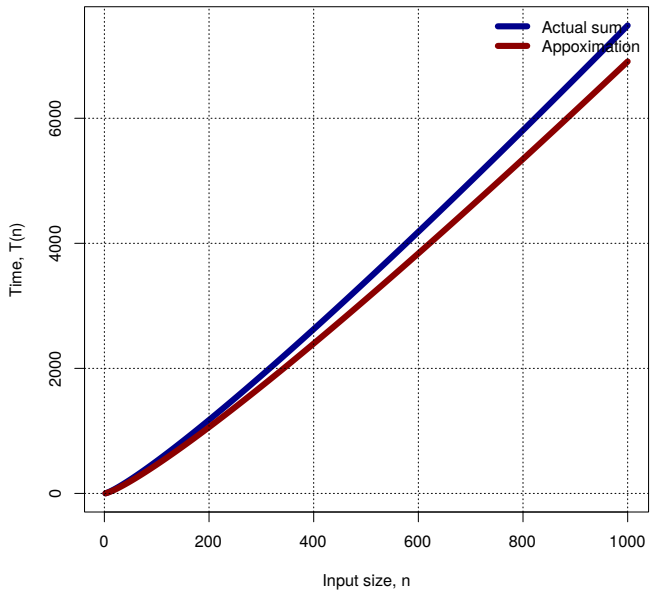

График 1.3: Сравнение точного значения суммы и аппроксимации

$$\sum_{i=1}^{n-1} \frac{n}{(n-i)} \approx n \int_{1}^{n-1} \frac{dx}{(n-x)} = -nln(1) + nln(n-1) < n \log_2(n)$$

Тогда нижнюю границу для сложность нашего алгоритма можно оценить как $\Omega(n \ln(n))$.

ГЛАВА 2

Основы языка программирования Python

Прежде чем мы сможем окунуться в изучение алгоритмов и структур данных нам необходимо ознакомиться с основами языка Python. Но начнем знакомство с Python с его установки.

2.1 Установка Python3

Установка интерпретатора Python достаточно простое действие, но оно отличается в зависимости от того, какую операционную систему использует пользователь. Две наиболее распространённые операционные системы - это Windows от компании Microsoft, и бесплатно распространяемая операционная система Linux. Стоит отметить, что существует огромное множество дистрибутивов операционной системы Linux. Но в нашем случае, мы будем использовать дистрибутив Ubuntu.

2.1.1 Установка в Windows

Перед началом установки Python под управлением операционной системы Windows, стоит выяснить разрядность процессора, который используется для вычислений. Существует два основных типа процессора: 32 битный (архитектура x86) и 64 битный (архитектура x86-64).

После того как мы узнали разрядность процессора, заходим на официальный сайт Python https://www.python.org/ и в разделе загрузок находим ссылку на последнюю версию языка Python. На момент написания данной книги последняя версия интерпретатора Python была 3.8.1.

Скачиваем установщик интерпретатора для нужного типа процессора и после загрузки выполняем установку интерпретатора и среды разработки (для этого достаточно выполнить двойной щелчок по скаченному файлу). После установки можно запустить командную строку (PowerShell) и выполнить следующую команду:

```
C:\ > python
```

Если установка прошла успешно, то пользователь должен увидеть приглашение интерпретатора для ввода команд.

2.1.2 Установка в Linux

Установка Python в Ubuntu Linux намного проще, так как можно воспользоваться стандартным установщиком пакетов Ubuntu. Для этого стоит открыть консоль и выполнить следующую команду:

```
$ sudo apt-get install python3
```

Менеджер пакетов самостоятельно загрузит интерпретатор Python, а также все необходимые зависимости. После установки, можно проверить результат, вызвав интерпретатор Python:

```
$ python3
```

Стоит отметить, что вместе с интерпретатором установится и менеджер пакетов Python - pip. Так, например, если нужно установить какую либо библиотеку, то это можно сделать с помощью программы pip. Например, если читатель хочет установить библиотеку для работы с криптографическими примитивами (библиотека pycryptodome), то достаточно выполнить следующую команду:

```
$ pip3 install pycryptodome
```

В оставшейся части книги мы будем предполагать наличие именно Ubuntu Linux у пользователя. И все команды, которые мы будем приводить, будут исполняться в консоле (терминале) Ubuntu Linux.

Для демонстрации работы интерпретатора, предположим что пользователь создал файл в редакторе (мы рекомендуем редактор SubLime [1], так как он очень гибкий и подсвечивает синтаксис языка Python), ввел в него следующие строчки кода (стоит заметить, что отступы должны быть либо табами, либо пробелами) и сохранил в домашней директории как hello_world.py:

```
from sys import argv

def main():
  print("Hello world, %s" % (argv[1]))

main()
```

После переходим в домашний каталог пользователя и выполняем наш первый скрипт:

```
$ cd ~
$ python3 hello_world.py Dmitriy
```

После выполнения данных команд пользователь должен увидеть Hello world, Dmitriy в консоле. Если такое сообщение не появится, то либо пользователь не правильно установил Python, либо не корректно ввел исполняемый код, приведённый выше, либо сохранил код не в нужной директории.

2.2 Типы данных

Язык Pyhton не строго типизированный, то есть для объявления переменных не требуется указывать тип данных переменной. Но, не смотря на это, в языке Python все же существует несколько типов данных. Перечислим эти типы данных. Так в языке Python есть целочисленные данные, численные данные с плавающей точкой, комплексные числа, булевы данные, строковые данные, а также объекты. Существуют встроенные и пользовательские объекты. Так например, к встроенным объектам можно отнести последовательности (списки, кортежи), словари, множества, а также бинарные объекты (байты и байтовые массивы).

[1] https://www.sublimetext.com/

Приведем примеры этих типов данных. Начнём с целочисленных данных. Целочисленные данные представляют собой множество натуральных чисел. Как и в математике, в Python3 целочисленная переменная может принимать любое значение от минус бесконечности до плюс бесконечности (конечно объём доступной памяти накладывает ограничения). Это нововведение в Python3 позволяет работать с большими числами без особого труда. В частности, это свойство незаменимо в криптографических алгоритмах. Приведём пример объявления целочисленной переменной. Для этого дадим сначала имя переменной и присвоим ей значение 10, как это указано в нашем примере:

```
number = 10
```

Теперь переменная number содержит данные в виде числа 10. Целые числа можно задавать и в системе счисления отличной от десятичной. Например, число 10 можно задать в двоичной системе счисления:

```
number = 0b1010
```

Или же можно задать это же число в шестнадцатеричной системе счисления:

```
number = 0xA
```

Стоит заметить, что переменной number можно присвоить значение любого другого типа данных. Например, следующие выражения являются корректными в языке Python:

```
number = 10;
number = 10.1;
```

Следующий тип данных, который мы рассмотрим будут числа с плавающей точкой. Такие числа можно представить как в явном виде, так и в научном. Например, следующая строчка кода инициализирует переменную числом с плавающей точкой в явном виде:

```
number = 2.718;
```

Можно также использовать научный формат для данных с плавающей точкой. Например, число $3 \cdot 10^{10}$ можно записать в языке Python как:

```
number = 3e10;
```

Если же потребуется задать число с определённым числом знаков после запятой, т.е., представить, например, число вида $278 \cdot 10^{-2}$, то можно воспользоваться следующей нотацией:

```
number = 278e-2;
```

Для данных с плавающей точкой существуют ограничения. Таким образом, максимальное значение для данных с плавающей точкой будет число $1.7976931348623157 \cdot 10^{308}$

```
number = 1.7976931348623157e308;
```

Число, которое больше $1.7976931348623157 \cdot 10^{308}$ уже будет равно бесконечности (для этого существует специальная константа inf). Тоже самое ограничение существует и для отрицательных чисел, т.е., минимальное значение будет $-1.7976931348623157 \cdot 10^{308}$, а все что меньше этого числа будет равно минус бесконечности (или же -inf). Более того, если число меньше 10^{-323}, то оно будет равно 0.0. Например, если читатель выполнит следующую команду в интерпретаторе, то он получит число ноль:

```
number = 1e-324
```

Вообще, строго говоря, приведённые примеры выше имеют смысл для компьютеров с 64 битной архитектурой (автор книги использует именно такой процессор). Читатель может самостоятельно проверить максимальное и минимальное значения для данных с плавающей точкой, выполнив следующий код:

```
import sys
sys.float_info
```

Для работы с числовыми данными существует набор встроенных методов. Например, чтобы отобразить все методы достаточно выполнить следующую команду в интерпретаторе:

```
help(int)
```

Далее мы приведём описание некоторых этих функций. Начнем мы с функции __abs__(). Эта функция предназначена для получения абсолютного значения переменной. Например, следующий отрывок кода выведет на экран число 10:

```
value = -10;
print(value.__abs__())
```

Вообще, все методы с двойным подчёркиванием являются магическими методами. Например, int.__abs__ определяет как будет себя вести функция math.abs если в качестве аргумента передать ей объект класса int. Например, вызов кода приведённого выше эквивалентен следующему коду:

```
from math import abs
value = -10;
print(abs(value));
```

Все магические методы, согласно документации, можно перегружать для того, чтобы можно было реализовывать своё поведение встроенного метода. Стоит отметить, что как правило магические методы не вызываются напрямую, а используются лишь для перегрузки операторов и модификации поведения соответствующих встроенных функций, таких, например, как len().

Стоит заметить, что функция вызывается через точку, так как она является методом класса. Следующая функция, которую мы рассмотрим - это __divmod__(x). Данная функция возвращает как остаток от деления, так и целую часть от деления, причём делитель передается в качестве параметра. Например, следующий кусочек кода возвратит кортеж, который будет содержать целую часть от деления (первое значение) и остаток (второе значение):

```
value = 14;
result = value.__divmod__(4);
print(result[1]);
```

Результат этого кода эквивалентен следующей операции в языке Python:

```
value = 14;
remainder = value % 4;
whole = value // 4;
```

Если требуется перевести целочисленное значение в тип данных с плавающей точкой, то можно воспользоваться функцией $__float__()$:

```
value = 14;
print(value.__float__());
```

Данная функция определяет поведение функции функции float, если ей передать, например, целое число:

```
value = 14;
print(float(value));
```

Обратная операция также возможна, если воспользоваться функцией __int__():

```
value = 14.5;
print(value.__int__());
```

Опять же, данная магическая функция определяет поведение функции int если ей передать число с плавающей точкой. Например, следующий код эквивалентен нашему предыдущему коду:

```
value = 14.5;
print(int(value));
```

Строковые данные в Python3 представляют собой набор символов, закодированных (по умолчанию) в UTF-8 формате. Строковые переменные должны всегда быть заключены в одинарные или двойные кавычки. Например, объявить строковые переменные можно следующим образом:

```
my_string_var = 'Python is great!';
my_string_var = "Python is great!";
```

Если же строка слишком длинная, то её можно представить в следующем виде:

```
my_string_var = """
Python is great!
This is a multiline string!
"""
```

В языке Python существует набор специальных функций для работы со строками. Рассмотрим наиболее часто используемые из них. Для того, чтобы получить справку по всем доступным функциям для работы со строками достаточно выполнить следующую команду в интерпретаторе:

```
help(str)
```

2.3 Переменные

2.4 Операторы

2.4.1 Бинарные операторы

Бинарные операторы это те операторы, которые требуют два операнда - левый и правый. Например, сложение двух целых числ требует бинарный оператор +:

```
def add(operand1, operand2):
  return operand1 + operand2;
result = add(10, 12);
```

Сложение

Вычитание

Умножение

Деление

Остаток от деления

Каждое число a можно представить в виде следующей линейной комбинации: $a = bq + r$. Тогда если мы будем делить a на b, то r будет остатком от деления. В языке Python остаток от деления можно получить используя оператор %.

Часто бывает необходимо вычислить остаток от деления двух целых чисел. Например, в теории чисел, которая используется в криптографии бывает необходимо вычислить наибольший общий делитель двух чисел, $gcd(a, b)$. Для этого используют широко известный алгоритм Евклида, который был бы немыслимый без оператора, который даёт остаток от деления. Приведем этот алгоритм:

```
def gcd(a, b):
  if b == 0:
    return a;
  return gcd(b, a % b);
```

2.4.2 Унарные операторы

2.4.3 Логические операторы

2.4.4 Приоритеты операторов

2.5 Выражения

2.6 Циклы

Циклы - это наиболее распространенный метод, который позволяет вызывать выражение повторно несколько раз. Рекурсия - это второй способ достичь такого результата. Но рекурсия требует дополнительных объёмов памяти и как правило её сложнее анализировать.

Существует несколько способов организации циклов. Первый способ - это использование конструкции while, которая будет выполнять выражения в теле

цикла до тех пор, пока логическое выражение (которое, кстати говоря, может быть составным) будет истинным. Строго говоря, тело цикла, как и любая другая подобная конструкция, должно отделяться либо табами, либо пробелами (напомним ещё раз, что программист должен использовать либо табы, либо пробелы во всем исходном коде, иначе интерпретатор будет выдавать сообщение об ошибке).

Приведём пример использования цикла while:

```
i = 10;
while i >= 0:
  print("Current index: " + str(i));
  i -= 1;
```

Легко понять, что цикл будет выполняться 11 раз, причём на каждом шаге значение счётчика будет уменьшаться на единицу. Логическое же выражение будет проверять значение счётчика: если это значение больше, либо равно нулю, то цикл будет продолжаться. Вообще можно использовать любое логическое выражение, даже то, которое всегда будет истинным. Например, можно завести бесконечный цикл, который будет работать пока пользователь не прекратит работу интерпретатор (или же не выйдет из цикла с помощью ключевого слова break, которое мы рассмотрим далее):

```
from time import sleep;
while True:
  print("Endless loop");
  sleep(1);
```

Чтобы прекратить досрочно выполнение цикла, такого, например, как while можно использовать ключевое слово break. Например, следующий цикл закончит своё выполнение после того, как отработает 5 раз:

```
i = 1;
while True:
  if i > 5:
    break;
  print("Current counter value: %s " % (str(i)));
  i += 1;
```

Вообще, оператор break можно использовать даже если есть несколько вложенных циклов. Тогда, при вызове break свою работу прекратит тот цикл, в котором был вызван данный оператор. Например:

```
for i in range(0, 100):
  for j in range(0, 100):
    if j >= 50:
      break;
```

```
    print("i=%d, j=%d" % (i, j));
```

Помимо цикла while существует также цикл for. Цикл for обычно используют с перечислениями. Например, можно воспользоваться конструкцией range для этой цели:

```
end = 10;
start = 0;
step = 1;
for i in range(start, end, step):
  print("Current index: %s" % (str(i)));
```

Полную справку по функции range можно получить выполнив следующую строчку в интерпретаторе Python3:

```
help(range)
```

Вообще, цикл for можно использовать с любыми перечислениями, например:

```
items = list([1, 2, 3, 4, 5]);
sum = 0;
total = len(items);
for item in items:
  sum += item;
print("Mean value of the sample is %f" % (float(sum) / float(total)));
```

Если необходимо пропускать выполнение тела цикла, то можно воспользоваться ключевым словом continue. Например, следующий кусок кода будет выполняться только для чётных значений счётчика:

```
sum_of_even_numbers = 0;

for i in range(0, 100):
  if i % 2 == 1:
    continue;
  sum_of_even_numbers += i;

print(sum_of_even_numbers);
```

2.7 Ветвления

```
from sys import stdin
a = 10;
print("Input a number: ")
b = int(stdin.readline())
if a > b:
```

```
  print("a > b")
elif a == b:
  print("a == b")
else:
  print("a < b")
```

2.8 Функции

2.9 Классы

2.10 Модули

2.11 Перегрузка методов и операторов

В данном разделе мы продемонстрируем то, как можно перегружать операторы и некоторые методы для своих собственных классов. Все перегружаемые функции, они же часто называются магическими функциями, обозначаются двойным подчёркиванием до и после названия метода. Например, магический метод __add__ определяет поведение бинарного оператора +:

```
class MyInteger():
  def __init__(self, value):
    self.value = value;
  def __add__(self, rvalue):
    return self.value + rvalue.value;

int1 = MyInteger(10);
int2 = MyInteger(12);
print(int1 + int2);
```

Если выполнить данный кусок кода, то в результате получиться число 22.

В Таблице 2.2 мы приводим список самых основных магических методов, которые можно встретить на практике. Более подробный список магических методов можно узнать выполнив команду help конкретного класса, или же можно обратиться к документации [2].

[2]Полный список магических функций https://docs.python.org/3/reference/datamodel.html

Магический метод	Описание
__init__(self[, ...])	Конструктор класса, вызывается при создании объекта
__del__(self)	Деструктор класса, вызывается при удалении объекта сборщиком мусора
__str__(self)	Вызывается функциями str, print и format. Возвращает строковое представление объекта
__lt__(self, other)	Перегружает оператор $<$
__le__(self, other)	Перегружает оператор $\leq$
__eq__(self, other)	Перегружает оператор $==$
__ne__(self, other)	Перегружает оператор $!=$
__gt__(self, other)	Перегружает оператор $>$
__ge__(self, other)	Перегружает оператор $\geq$
__hash__(self)	Получение хэш-суммы объекта, например, для добавления в словарь
__len__(self)	Длина объекта
__iter__(self)	Возвращает итератор для контейнера
__contains__(self, item)	Проверка на принадлежность элемента контейнеру
__add__(self, other)	Перегрузка оператора сложения $+$
__sub__(self, other)	Перегрузка оператора вычитания $-$
__mul__(self, other)	Перегрузка оператора умножения $*$
__truediv__(self, other)	Перегрузка оператора деления $/$
__mod__(self, other)	Перегрузка оператора остаток от деления %
__and__(self, other)	Перегрузка оператора битового и &
__xor__(self, other)	Перегрузка оператора битового исключающее или ^
__or__(self, other)	Перегрузка оператора битового или |
__iadd__(self, other)	Перегрузка оператора $+=$
__isub__(self, other)	Перегрузка оператора $-=$
__imul__(self, other)	Перегрузка оператора $*=$

Таблица 2.2: Перегрузка основных методов и операторов

ГЛАВА 3

Основные структуры данных

В данном разделе мы приведём описание и примеры использования основных структур данных, которые можно встретить на практике. Начнем мы наш диалог с множеств и массивов, далее опишем использование очередей и стеков, а закончим наш рассказ деревьями и графами.

3.1 Множества

В главе посвящённой математическим основам мы уже сталкивались с множествами и операциями над ними. Здесь же мы расскажем о том, как можно работать с множествами в языке Python.

Для множеств в языке Python есть встроенный класс. Его можно инициализировать следующим образом:

```
my_set_1 = set();
my_set_2 = set([1, 2, 3]);
```

Полный список методов данного класса можно просмотреть, вызвав справку в среде Python:

```
help(set)
```

Рассмотрим самые основные методы класса для работы с множествами. Первый метод - это добавление элемента к множеству:

```
my_set = set([1, 2]);
my_set.add(3);
```

Так как множество не содержит дубликатов, то добавление элемента, который уже содержится в множестве не приведёт к изменению множества. Чтобы очистить множество от элементов, то можно воспользоваться методом clear.

Для того, чтобы найти разницу двух множеств можно воспользоваться методом difference. Например:

```
my_set_1 = set([1, 2, 3]);
my_set_2 = set([2, 3]);
my_set_3 = my_set_1.difference(my_set_2);
print(my_set_3);
```

Для нахождения пересечения двух множеств можно воспользоваться методом intersection:

```
my_set_1 = set([1, 2, 3]);
my_set_2 = set([2, 3]);
my_set_3 = my_set_1.intersection(my_set_2);
print(my_set_3);
```

Если необходимо проверить являются ли два множества не пересекающимися, то можно воспользоваться следующим методом:

```
my_set_1 = set([1, 2, 3]);
my_set_2 = set([4, 5]);
print(my_set_1.isdisjoint(my_set_2));
```

Если нужно проверить является ли одно множество подмножеством другого, то можно воспользоваться методом issubset. Например:

```
my_set_1 = set([1, 2, 3]);
my_set_2 = set([1, 2, 3, 4, 5]);
print(my_set_1.issubset(my_set_2));
```

Для удаления элемента из множества можно воспользоваться методом remove. Например:

```
my_set_1 = set([1, 2, 3]);
my_set_1.remove(2);
```

И наконец, если необходимо найти пересечение двух множеств, то можно воспользоваться методом union.

```
my_set_1 = set([1, 2, 3]);
my_set_2 = set([1, 2, 3, 4, 5]);
print(my_set_1.union(my_set_2));
```

Если есть необходимости в использование множества, которое нельзя изменять, то можно воспользоваться так называемым frozenset. Этот класс имеет те же методы (за исключением, конечно, методов которые ответственны за изменение множества, таких как add, pop, remove, и так далее), что и обычное множество. Справку по всем доступным методам можно вызвать, набрав следующую команду в интерпретаторе:

```
help(frozenset);
```

3.2 Массивы

Массивом в программирование называется набор однотипных элементов, расположенных в памяти компьютера последовательно - один за другим. В языке программирования Python массивы можно объявить несколькими способами. Рассмотрим эти способы.

```
my_list = [1, 2, 3, 4, 5];
my_list = list();
my_list = [0] * 5;
```

Первый способ инициализации - это явный способ задания массива. При этом способе все элементы массива объявляются сразу при инициализации.

Рассмотрим основные свойства и методы массивов (полную справку можно получить вызвав help(list)).

3.3 Ассоциативные массивы

Ассоциативные массивы очень удобный способ представления словарей в программирование. Их основное свойство заключается в том, что они хранят пары - ключ и значение. В языке программирования ассоциативные массивы могут быть объявлены несколькими способами:

```
my_dict = dict();
my_dict = {};
```

Для того, чтобы занести пару ключ/значение в словарь достаточно выполнить следующую операцию:

```
my_dict = dict();
my_dict["key1"] = 1;
my_dict[10] = "value1";
```

Полную справку по доступным методам ассоциативного массива можно получить выполнив следующую команду в интерпретаторе Python.

```
help(dict)
```

Здесь же мы приведём описание самых основных методов данного класса. Начнём мы наш диалог с метода keys. Этот метод позволяет выбрать все ключи в ассоциативном массиве и вернуть их в виде списка. Аналогичным образом можно выбрать все значения ассоциативного массива (включая ключ, под которым хранится значение). Нпример, все значения и соответствующие ключи можно выбрать вызвав метод items (если же нужны только значения, то можно воспользоваться методом values). Приведём небольшой кусок кода:

```
my_dict = dict({"one": 1, "two": 2});
for key, item in my_dict.items():
  print(item);
```

Что бы выбрать элемент по ключу достаточно выполнить следующую команду:

```
my_dict = dict({"one": 1, "two": 2});
print(my_dict["one"]));
```

Если ключа нет в словаре, то выполнение прекратится и будет вызвана ошибка. Для того, чтобы убедиться в том, что ошибки не будет можно вызвать метод get со значением по-умолчанию:

```
my_dict = dict({"one": 1, "two": 2});
print(my_dict.get("one", None);
```

Для добавления или обновления значения можно присвоить словарю значение по ключу. Например:

```
my_dict = dict();
my_dict["one"] = 1;
my_dict["one"] = None;
```

Откровенно говоря, реализовать свой собственный класс словаря не составляет особого труда. Например, если мы знаем, что ключи имеют равномерное распределение, то словарь можно реализовать следующим образом:

```
class item():

  def __init__(self, k, v):
    self.k = k;
    self.v = v;
```

```
  def key(self):
    return self.k;

  def value(self, v = None):
    if v == None:
      return self.v;
    else:
      self.v = v;

class my_dict():
  def __init__(self, size = 10):
    self.table_size = size;
    self.table = [0] * self.table_size;

  def hash(self, k):
    return k % self.table_size;

  def set(self, k, v):
    _k = self.hash(k);

    if not isinstance(self.table[_k], list):
      self.table[_k] = list();

    found = False;

    for i in self.table[_k]:
      if i.key() == k:
        i.value(v);
        found = True;

    if not found:
      self.table[_k].append(item(k, v));


  def get(self, k):
    _k = self.hash(k);
    if not isinstance(self.table[_k], list):
      return None;
    for i in self.table[_k]:
      if i.key() == k:
        return i.value();
    return None
```

Причём если возникает коллизия (хэш функция выдаёт одно и тоже значение для разных ключей), то заводят список, в котором содержатся все значения словаря с данным ключом.

Давайте проанализируем сложность данного алгоритма. Предположим, что размер таблицы равен m. Тогда вероятность того, что произойдёт коллизия в ключе равна $p = P(A) = 1/m$, а вероятность обратного события равна $q = P(\bar{A}) = \frac{m-1}{m}$. Нас интересует вероятность k коллизий в n испытаниях. Заметим, что данная вероятность имеет биномиальный закон распределения, т.е., $P(X = k) = C_n^k p^k q^{n-k}$. Найдем математическое ожидание: $E[X] = \sum_{k=0}^{n} kC_n^k p^k q^{n-k}$. Из предыдущих глав мы знаем, что это математическое ожидание биномиального закона распределение вероятностей и равно np. Тогда ожидаемое количество элементов в каждой ячейке хэш таблицы будет равно $\frac{n}{m}$. Отсюда следует, что время необходимое на поиск и добавление нового элемента будет равно $\Theta(\frac{n}{m})$, где n - это количество элементов в хэш таблице.

3.4 Связанные списки, очереди и стеки

Связанные списки довольно часто встречаются в программировании. С помощью связанных списков можно моделировать очереди и стеки, которые мы рассмотрим далее. А пока рассмотрим как реализуется данная структура данных в языке программирования Python (полный исходный код представлен читателю далее). В каждом связанном списке элемент имеет указатель на следующий и предыдущий элементы, а также содержит собственно данные этого элемента.

У связанного списка есть три основных метода add, get и remove. При добавлении элемента к списку, новый элемент будет добавляться в конец списка, при этом указатель на предыдущий элемент будет инициализироваться ссылкой на последний элемент списка (до добавления), а ссылка на следующий элемент (последнего элемента до добавления) будет указывать на элемент, который мы добавляем.

При удаление происходит сначала поиск элемента по индексу, а после, если элемент найден, обновляются указатели на предыдущий и следующий элементы. Иными словами, элемент, который находится до удаляемого элемента (если такой есть) будет иметь следующую ссылку на элемент, находящийся после удаляемого элемента (если такой есть). И наоборот, ссылка на предыдущий элемент (если таковой есть) элемента, который находится после удаляемого элемента, будет указывать на элемент предшествующий удаляемому элементу или иметь нулевую ссылку.

На Графике 3.1 мы демонстрируем как это происходит добавление элемента, а на Графике 3.2 удаление элемента.

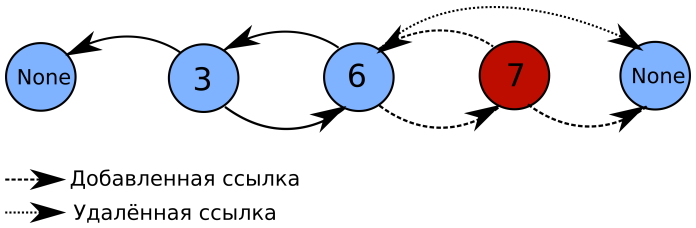

График 3.1: Добавление в связанный список

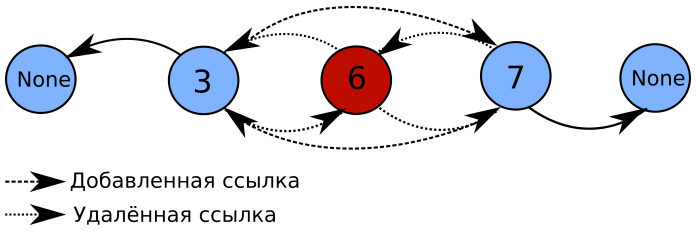

График 3.2: Удаление из связанного списка

```
class litem():
  def __init__(self, value):
    self.next = None;
    self.prev = None;
    self.value = value;

class linked_list():

  def __init__(self):
    self.head = None;
    self.tail = None;
    self.length = 0;

  def size(self):
    return self.length;

  def add(self, item):
    if not isinstance(item, litem):
      raise Exception("Invalid type for item");
    self.length += 1;
    if not self.head and not self.tail:
      self.head = item;
      self.tail = item;
      item.prev = None;
      item.next = None;
```

```
      return;
    item.prev = self.tail;
    item.next = None;
    self.tail.next = item;
    self.tail = item;

  def get(self, index):
    if index < 0 or index > self.length:
      raise Exception("Index out of range");
    item = self.head;
    prev = None;
    current_index = 0;
    while item != None:
      if current_index == index:
        return item;
      current_index += 1;
      item = item.next;

  def remove(self, index):
    if index < 0 or index > self.length:
      raise Exception("Index out of range");
    item = self.head;
    prev = None;
    current_index = 0;
    while item != None:
      if current_index == index:
        self.length -= 1;
        if item == self.head:
          if item == self.tail:
            self.head = self.tail = None;
          else:
            self.head = item.next;
            self.head.prev = None;
          return item;
        elif item == self.tail:
          self.tail = self.tail.prev;
          self.tail.next = None;
          return item;
        else:
          prev = item.prev;
          next = item.next;
          prev.next = next;
          next.prev = prev;
          return item;
      current_index += 1;
      item = item.next;
```

```
  def iterate(self):
    i = self.head;
    while i != None:
      yield i.value;
      i = i.next;

ll = linked_list();
ll.add(litem(10));
ll.add(litem(11));
ll.add(litem(12));
ll.add(litem(0));
for i in ll.iterate();
  print(i);
```

Проанализируем вычислительную сложность и сложность памяти. Очевидно, что данная структура данных требует $O(n)$ памяти. Добавление же элемента в список требует $O(1)$ времени, а удаление и поиск в худшем случае требует $O(n)$ времени.

Теперь, когда мы знаем как работает связанный список, мы можем приступить к рассмотрению стека. Стек - это структура данных, в которой каждый новый элемент добавляется в конец списка. При удаление также выбирается последний элемент. Иными словами стек реализует правило: Последний пришёл, первый ушёл. Таким образом у стека есть три основных метода - push (отвечает за добавление элемента к стеку), pop (отвечает за удаление элемента из стека) и top (отвечает за просмотр (без удаления) значения последнего добавленного элемента). Стек легко реализовать с помощью связанного списка.

```
class stack():
  def __init__(self):
    self.ll = linked_list();
  def push(self, value):
    self.ll.add(litem(value));
  def pop(self):
    length = self.ll.size();
    if length == 0:
      raise Exception("Stack is empty");
    return self.ll.remove(length - 1).value;
  def top(self):
    length = self.ll.size();
    if length == 0:
      raise Exception("Stack is empty");
    return self.ll.get(length - 1).value;
```

В отличие от стека, очередь реализует правило: Первый пришёл, первый вышел. Также как и стек, очередь легко реализовать с помощью связанного списка: новое поступление в очередь добавлется в конец списка, а удаление из очереди происходит путём удаления первого элемента из списка. Очереди очень часто используются в математическом моделирование. Например, в теории массового обслуживания, очередь используется как инструмент для моделирования поступления и обслуживания абонентов, так как они отражают то, как происходит обслуживание в реальной жизни. Исходный код очереди, основанной на связанном списке, приводится ниже.

```
class queue():
  def __init__(self):
    self.ll = linked_list();
  def enqueue(self, value):
    self.ll.add(litem(value));
  def dequeue(self):
    return self.ll.remove(0).value;
```

Очевидно что все операции (удаление и добавление элемента из очереди) происходят за время $O(1)$. При этом затраты на память можно представить как $O(n)$.

3.5 Деревья

Деревья - это структуры данных, в которых каждый элемент может иметь от 0 до k потомков, причем каждый узел, кроме корневого, имеет в точности один родительский узел. Корневой узел не имеет родительского узла. Вообще дерево можно описать как связный ациклический граф, в котором из любого узла можно попасть в любой другой узел всего одним путём. Деревья называются k-арными, если каждый узел может иметь до k дочерних узлов. Обычно любой узел дерева содержит какую-то информацию. Например, бинарные деревья, которые мы рассмотрим далее, могут использоваться для поиска информации, и тогда каждый узел содержит либо число либо строку. Причём узлы в таких деревьях поиска строго упорядоченны. Например, в бинарном дереве поиска родительский узел содержит строку или число, которое больше строки или числа, содержащегося в левом потомке, но меньше либо равны числу или строке в правом потомке.

Деревья могут быть ориентированными или неориентированными. В неориентированных деревьях можно переходить из дочернего узла в родительский, и обратно. В направленном - только в одном направление,

которое задается направлением ребра.

3.5.1 Бинарные деревья

Бинарными называются такие деревья, у которых каждый узел может иметь максимум два потомка. Бинарные деревья часто встречаются в информатике. Например, бинарные деревья используются, как мы это увидим, при реализации кучи, а также при реализации дерева поиска. Очевидно, что основная проблема бинарных деревьев заключается в том, что поиск может занять $O(n)$ временени. Это может произойти если дерево не сбалансировано. Например, если мы будем строить дерево используя отсортированный массив, то глубина дерева будет равна n. Легко заметить, что такая ситуация может стать проблемой если количество элементов велико (например, $n = 2^{32} - 1$), и тогда, поиск будет занимать недопустимое количество времени. Если же дерево сбалансировано, то поиск потребует всего $O(\log_2(n))$ сравнений.

Определение 16. Сбалансированным бинарным деревом называется такое дерево, у которого разница в высоте самого глубокого и самого неглубокого листа меньше либо равна единице.

Определение 17. Заполненным бинарным деревом называется такое сбалансированное дерево, в котором все узлы на одном уровне (кроме предпоследнего) имеют в точности два потомка. Предпоследний же уровень заполнен слева направо, т.е., все узлы (слева направо) имеют в точночсти два дочерних узла (за исключением последнего узла на предпоследнем уровне, у которого есть хотя бы один дочерний узел). Все остальные узлы предпоследнего уровня могут не иметь дочерних узлов.

Приведём следующую теорему без доказательств.

Теорема 7. Каждое сбалансированное (заполненное) дерево можно представуть в виде одномерного массива.

В данном разделе мы приведем способ балансировки бинарного дерева поиска. Эти деревья носят название AVL (в честь советских изобретателей - Георгия Максимовича Адельсон-Вельского и Евгения Михайловича Ландиса). В бинарном дереве AVL могут быть четыре различные ситуации. Эти случаи мы представляем читателю на Графике ??. А полный исходный код самобалансирующегося дерева (или AVL дерева) мы представляем читателю ниже. Но вернемся к обсуждению того, как работают эти деревья.

```
class node():
  def __init__(self, key):
    self.key = key;
    self.height = 0;
    self.left = None;
    self.right = None;
  def __str__(self):
    return str(self.key);

class AVL_tree():
  def search(self, root, key):
    if not root:
      return False;
    if key == root.key:
      return True;
    if key > root.key:
      return self.search(root.right, key);
    else:
      return self.search(root.left, key);

  def insert(self, key, root):
    if not root:
      return node(key);
    if key > root.key:
      root.right = self.insert(key, root.right);
    elif key < root.key:
      root.left = self.insert(key, root.left);
    else:
      return root;

    root.height = max(self.get_height(root.left), self.get_height(root.
                                        right)) + 1;
    balance = self.get_height(root.left) - self.get_height(root.right);

    if balance > 1 and key < root.left.key:
      root = self.rotate_right(root);
    if balance < -1 and key > root.right.key:
      root = self.rotate_left(root);
    if balance > 1 and key > root.left.key:
      root.left = self.rotate_left(root.left);
      root = self.rotate_right(root);
    if balance < -1 and key < root.right.key:
      root.right = self.rotate_right(root.right);
      root = self.rotate_left(root);
    return root;
```

```
  def rotate_left(self, root):
    z = root;
    y = root.right;
    z.right = y.left;
    y.left = z;
    z.height = max(self.get_height(z.left), self.get_height(z.right)) +
                                        1;
    y.height = max(self.get_height(y.left), self.get_height(y.right)) +
                                        1;
    return y;

  def rotate_right(self, root):
    z = root;
    y = root.left;
    z.left = y.right;
    y.right = z;
    z.height = max(self.get_height(z.left), self.get_height(z.right)) +
                                        1;
    y.height = max(self.get_height(y.left), self.get_height(y.right)) +
                                        1;
    return y;

  def get_height(self, root):
    if not root:
      return 0;
    return root.height;

  def balance(self, left, right):
    return self.get_height(left) - self.get_height(right);

root = None;
tree = AVL_tree();
root = tree.insert(1, root);
root = tree.insert(2, root);
root = tree.insert(11, root);
root = tree.insert(3, root);
root = tree.insert(6, root);
root = tree.insert(22, root);
root = tree.insert(14, root);
root = tree.insert(0, root);
print("Searching key 22");
print(tree.search(root, 22));
```

3.5.2 Обход дерева вглубь и вширь

Иногда бывает необходимо обойти дерево и вывести все узлы в некотором порядке. Объективно говоря, дерево можно обойти двумя способами, а именно - вглубь и вширь. Для обхода вширь, первым делом выводятся сначала значение узлов на первом уровне. Далее выводятся значения на втором уровне и так далее, пока не будут достигнуты листья деревьев. При обходе дерева вглубь сначала выводится значение первого элемента (т.е. значение корневого элемента). После выводятся значение в левого дочернего узла. Данный процесс продолжается до тех пор, пока не будет достигнут левый лист дерева. После выводится значение правого листа (если он есть). Далее алгоритм поднимается на уровень выше и аналогичный сценарий повторяется пока в стеке есть элементы. Приведём примеры того, как это можно реализовать в языке программирования Python.

```
class node():
  def __init__(self, key):
    self.key = key;
    self.left = None;
    self.right = None;
  def __str__(self):
    return str(self.key);

class bst():
  def __init__(self):
    pass

  def insert(self, key, root):
    if not root:
      return node(key);
    if key > root.key:
      root.right = self.insert(key, root.right);
    elif key < root.key:
      root.left = self.insert(key, root.left);
    else:
      return root;
    return root;

  def depth_traversal(self, root):
    stack = [];
    stack.append(root);
    output = [];
    while len(stack) > 0:
      current = stack[-1];
```

```
      stack = stack[:-1];
      if current.right:
        stack.append(current.right);
      if current.left:
        stack.append(current.left);
      output.append(current.key);
    return output;

  def breadth_traversal(self, root):
    if not root:
      return [];
    queue = [root];
    output = [];
    while len(queue) > 0:
      current = queue[0];
      queue = queue[1:];
      output.append(current.key);
      if current.left:
        queue.append(current.left);
      if current.right:
        queue.append(current.right);
      root = current;
    return output;

tree = bst();
root = None;
root = tree.insert(3, root);
root = tree.insert(1, root);
root = tree.insert(0, root);
root = tree.insert(2, root);
root = tree.insert(5, root);
root = tree.insert(4, root);
root = tree.insert(6, root);

print(tree.breadth_traversal(root));
print(tree.depth_traversal(root));
```

3.5.3 Кучи

Рассмотрим бинарную кучу, которую мы приводим на Графике 3.3. Прежде чем мы приступим описанию свойств бинарной кучи и тому, как бинарная куча может быть реализована в языке Python, приведем теорему, результат который позволяет эффективно обходить бинарное дерево.

Так как куча представляет собой сбалансированное (заполненное) бинарное

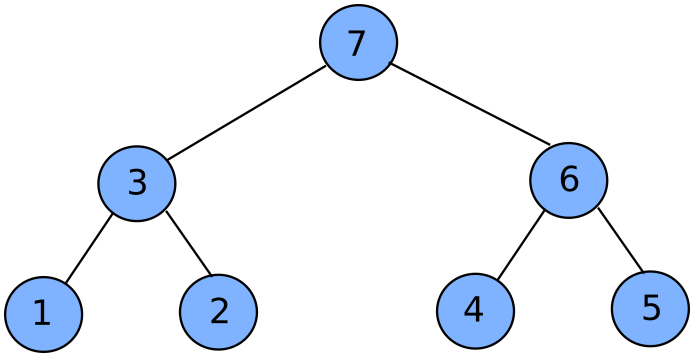

График 3.3: Бинарная куча

дерево, то его можно представить в виде одномерного массива (причём первый элемент находится в ячейке под индексом 1).

Теорема 8. В сбалансированном бинарном (заполненном) дереве левый потомок i-ого узла может быть найден под индексом $2i$, а правый потомок под индексом $2i + 1$.

Пусть нам дан узел дерева с индексом i (индексирование узлов начинается с 1). Тогда глубина этого узла будет равна $\lfloor \log_2(i) \rfloor$ (это очевидно, так как на каждом уровне количество узлов удваивается, см. График 3.3). Пусть n - это количество узлов, предшествующих узлу i (на том же уровне), а m - это общее количество узлов до текущего уровня $\lfloor \log_2(i) \rfloor$ включительно. Тогда индекс левого потомка можно вычислить как $2n + m + 1$. Но так как

$$n = i - \sum_{l=0}^{\lfloor \log_2(i) \rfloor - 1} 2^l - 1 = i - (2^{\lfloor \log_2(i) \rfloor} - 1) - 1$$

А так как

$$m = \sum_{l=0}^{\lfloor \log_2(i) \rfloor} 2^l = 2^{\lfloor \log_2(i) \rfloor + 1} - 1$$

Получаем

$$2n + m + 1 = 2(i - (2^{\lfloor \log_2(i) \rfloor} - 1) - 1) + 2^{\lfloor \log_2(i) \rfloor + 1} - 1 + 1 = 2i$$

Очевидно, что индекс правого потомка тогда будет равен $2i + 1$. Что и требовалось доказать.

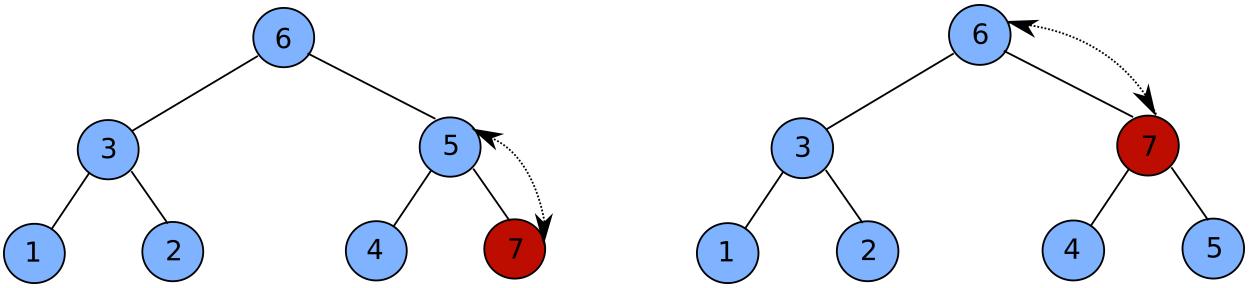

График 3.4: Добавление элемента к бинарной куче

Отметим способ, которым мы нашли сумму. Пусть $S = \sum_{l=0}^{\lfloor \log_2(i) \rfloor} 2^l$, тогда $2S = \sum_{l=1}^{\lfloor \log_2(i) \rfloor + 1} 2^l$, а $2S - S = 2^{\lfloor \log_2(i) \rfloor + 1} - 1$.

Рассмотрим добавление узла в бинарную кучу 3.4.

```
from math import floor

class heap():
  def __init__(self):
    self.h = [0];
    self.length = 0;

  def size(self):
    return self.length;

  def down(self):
    c = 1;
    while c * 2 <= self.length:
      if c * 2 + 1 <= self.length:
        if self.h[c * 2] < self.h[c * 2 + 1] \
          and self.h[c] < self.h[c * 2 + 1]:
          v = self.h[c];
          self.h[c] = self.h[c * 2 + 1];
          self.h[c * 2 + 1] = v;
          c = c * 2 + 1;
        elif self.h[c] < self.h[c * 2]:
          v = self.h[c];
          self.h[c] = self.h[c * 2];
          self.h[c * 2] = v;
          c = c * 2;
        else:
          break;
      else:
        if self.h[c] < self.h[c * 2]:
```

```
            v = self.h[c];
            self.h[c] = self.h[c * 2];
            self.h[c * 2] = v;
            c = c * 2;
          else:
            break;

  def up(self):
    c = self.length;
    while c > 1 and self.h[c] > self.h[floor(c / 2)]:
      v = self.h[c];
      self.h[c] = self.h[floor(c / 2)];
      self.h[floor(c / 2)] = v;
      c = floor(c / 2);

  def pop(self):
    if self.length == 0:
      return None;
    self.length -= 1;
    v = self.h[1];
    if self.length == 0:
      self.h = [0];
      return v;
    self.h[1] = self.h[len(self.h) - 1];
    self.h = self.h[0:len(self.h) - 1];
    self.down();
    return v;

  def push(self, v):
    self.length += 1;
    self.h.append(v);
    self.up();
```

В предыдущей главе мы рассмотрели очереди. Но иногда возникает необходимость в очереди с приоритетами. Например, в сетях передачи данных пакеты могут иметь различные приоритеты и соответственно передаваться должны первыми те пакеты, которые имеют наибольший приоритет. И тут уже не обойтись без очереди с приоритетами. Очереди с приоритетами легко реализовать с помощью куч. Приведём пример того, как это можно сделать.

```
class priority_queue():
  def __init__(self):
    self.h = heap();
  def enqueue(self, value):
    self.h.push(value);
  def dequeue(self):
```

```
    return self.h.pop();
```

Так как куча имеет свойство того, что максимальный элемент находится в корне дерева, то при удаление выбирается именно элемент с максимальным приоритетом. Зная, что свойства кучи можно восстановить за время $O(\log_2(n))$, то оперции (добавление и удаление элементов в очередь с приоритетами) также требуют $O(\log_2(n))$ времени. При этом объём занимаемой памяти можно определить как $O(n)$.

3.5.4 Сортировка кучей

Сортировка кучей достаточно изящный метод - так как по свойству куч наибольший элемент содержится в вершине дерева то, удаляя поочерёдно один за другим элементы из кучи, мы можем отсортировать массив в нужном порядке. Полный исходный код данного алгоритма сортировки мы приводим ниже.

```
from enum import Enum

class direction(Enum):
  ASC = 1
  DESC = 2

def heap_sort(a, direction = direction.ASC):
  h = heap();
  for i in a:
    h.push(i);
  b = [];
  for i in range(0, h.size()):
    b.append(h.pop());
  if direction == direction.ASC:
    b.reverse();
  return b;

heap_sort([2, 1, 4, 55, 42, 11], direction.DESC);
```

Проанализируем данный алгоритм. Так как мы знаем, что восстановление свойств кучи занимает время $O(\log_2(n))$, а цикл в теле функции heap_sort выполняется n раз, то несложно выяснить, что вычислительная сложность алгоритма в худшем, лучшем и среднем случаях будет всегда занимать время $O(n \log_2(n))$. Также, заметим, что сложность памяти составляет $O(n)$ (это легко достижимо, если кодировать кучу одномерным массивом, как мы это делали в предыдущей главе).

3.6 Графы

Графы - это распространённая структура данных в компьютерных задачах. С помощью графов можно представлять компьютерные, дорожные и другие сети. Как и деревья, графы могут быть направленные и ненапрвленные. В направленных графах переходить можно из узла в узел только если есть соответствующее ребро. В ненаправленных графах, если есть ребро из одного узла в другой, то соответственно доожно существовать и ребро в обратном направлении по-умолчанию. Графы описываются обычно как $G = (V, E)$, где V - это набор всех вершин, а E - это набор всех рёбер.

3.6.1 Представление графов в Python

Самым простым способом представления графов в языке Python является матричное кодирование. Так, например, если дан граф $G = (V, E)$, то его можно представить как двумерный массив, в котором в каждой ячейки ставится 1, если между парой узлов (i, j) существует ребро. Если же между вершинами нет ребра, то соответствующий элемент массива будет равен 0. Если же каждому ребру соответствует еще и вес, то вместо 1 проставляется значение веса.

Например, предположим, что у нас есть граф, который мы приводим на Графике 3.5. Тогда мы можем составить следующую матрицу для этого ненаправленного графа.

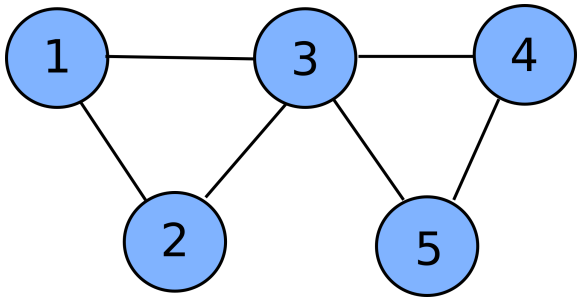

График 3.5: Простой ненаправленный граф

```
adj_matrix = [
  [0, 1, 1, 0, 0],
  [1, 0, 1, 0, 0],
  [1, 1, 0, 1, 1],
  [0, 0, 1, 0, 1],
  [0, 0, 1, 1, 0]
];
```

Другой способ задания графа - это способ, в котором используется список ребер для каждой вершины. Так, например, заводится массив, размер которого равен количеству вершин. А каждый элемент массива - это список, который содержит все вершины, с которыми данная вершина имеет ребро.

3.6.2 Обход графов

Самым простым способом обхода графа является обход в ширину. Этот метод обхода графа часто используется, например, для проверки связности и поиска кратчайшего пути между двумя узлами.

Работает этот алгоритм следующим образом. Возьмем для начала узел, из которого будем обходить весь граф. Заведем список, размер которого будет равен $O(n)$, где n - это количество вершин в графе и проинициализируем наш массив булевым значением $False$. Этот массив будет служить нам как индикатор того, какие узлы мы уже посетили. Далее заведем очередь (без приоритетов), в которой будем хранить узлы, дочерние узлы которых нам предстоит ещё проверить. Проинициализируем очередь начальным узлом. Далее откроем цикл и будем добавлять в очередь все связанные узлы (если мы еще их не обошли) и помечать их как посещённые. Будем продолжить эту операцию до тех пор, пока размер очереди не станет равен нулю. Если в результате все вершины графа будут помечены как посещённые, то граф связанный.

```
def bfs_is_connected(graph):
  start = 0;
  n = len(graph);
  seen = [False] * n;
  queue = list([start]);
  while len(queue) > 0:
    current = queue[0];
    queue = queue[1:];
    for i in range(0, n):
      v = graph[current][i];
      if v != 0 and not seen[i]:
        seen[i] = True;
        queue.append(i);
  return not (False in seen);

adj_matrix = [
  [0, 1, 1, 0, 0],
  [1, 0, 1, 0, 0],
  [1, 1, 0, 1, 1],
```

```
  [0, 0, 1, 0, 1],
  [0, 0, 1, 1, 0]
];

print(bfs_is_connected(adj_matrix));
```

Проанализируем сложность данного алгоритма. Если граф связан, то сложность нашего алгоритма будет равна $O(|V|^2)$ (где $|V|$ - это количество вершин), так как для реализации мы используем матрицу связанности. А вообще, сложность алгоритма (основанного на списке рёбер) будет $O(|V|+|E|)$, где $|V|$ - это количество вершин, а $|E|$ - это количество рёбер. Заметим, что, на самом деле, эту сложность можно представить как $O(max(|V|, |E|))$, так как в лучшем случае каждая вершина может иметь лишь одно ребро и тогда сложность будет $O(|V|)$, в худшем же случае граф может быть полносвязанным, и тогда количество рёбер будет равно $|E| = |V|(|V| - 1)$.

Стоит заметить, что существует также поиск в глубину. Этот метод основан на рекурсии и его также можно использовать для проверки связанности графа.

3.6.3 Поиск наикратчайшего пути между двумя вершинами

Поиск коротких путей в графе - это одна из часто встречающихся задач в программировании. Так, на практике эту задачу можно применить в компьютерных сетях, в логистических задачах и многих других. В связи с этим, в главе, посвященной теории конструирования алгоритмов, которую читатель встретит далее, мы приведём жадный алгоритм поиска пути между вершинами. Этот алгоритм носит название Дейкстры и позволяет найти кратчайшие пути между заданным узлом и всеми остальными вершинами в графе за полиномиальное время. В главе же посвященной динамическому программированию мы приведём пример алгоритма Флойда-Воршела, который позволяет найти все короткие пути между любой парой узлов за время $O(n^3)$, где n - это количество вершин в графе. Здесь же приведем пример алгоритм A* (читается A star). Этот алгоритм позволяет найти наикратчайший путь между двумя вершинами используя бинарную кучу.

A* (читается как A star) - алгоритм поиска наикратчайшего пути в ненаправленном графе. Данный алгоритм работает следующим образом. Для начала заведем кучу, в которой минимальный элемент находится в корне дерева (в куче будут храниться пары - индекс узла и расстояние от начального до данного узла, причем куча будет сортироваться по расстоянию). Очевидно, что нас интересует именно такая куча, так как мы ищем самый короткий путь от начального до конечного узла. Заведём ещё переменную cost_so_far. В этой

переменной будем хранить текущее наименьшее расстояние до данного узла (от начального узла). И проинициализируем кучу начальным узлом.

Далее, открываем цикл, который будет работать до тех пор, пока в куче есть узлы или же мы достигнем конечный узел. В теле цикла мы выбираем тот узел, который имеет наименьшее расстояние от начального узла (это тривиально, так как мы используем кучу с минимальным значением в корне, или же min heap). Далее пробегаемся по всем узлам, с которыми связан дынный узел: Если расстояние от текущего узла до начального узла плюс расстояние от текущего узла до выбранного дочернего узла меньше, чем расстояние от данного дочернего узла до начального узла (это значение хранятся в массиве cost_so_far), то мы обновляем переменную cost_so_far новым расстоянием и добавляем дочерний узел в кучу (как метрику мы выбираем полное расстояние от данного узла до начального, которое хранится в переменной new_cost). Полный алгоритм приведён ниже.

```
from math import inf

def a_star_search(graph, start, end):
  h = min_heap();
  cost_so_far = [inf] * len(graph);
  came_from = dict();
  h.push(hitem(start, 0));
  cost_so_far[start] = 0;

  while h.size() > 0:
    current = h.pop();
    if current.value == end:
      break;
    for i in range(0, len(graph[current.value])):
      if graph[current.value][i] == 0:
        continue;
      new_cost = cost_so_far[current.value] + graph[current.value][i];
      if new_cost < cost_so_far[i]:
        cost_so_far[i] = new_cost;
        h.push(hitem(i, new_cost));
        came_from[i] = current.value;
  return came_from;
```

Давайте проанализируем данный алгоритм.

ГЛАВА 4

Базовые алгоритмы

В данном разделе мы рассмотрим основные алгоритмические приёмы для решения вычислительных задач.

4.1 Разделяй и властвуй

Основной подход алгоритмов данной категории заключается в деление входных данных на части до тех пор, пока задача не сведётся к тривиальной. После обработки данных, в зависимости от алгоритма, данные могут быть объединены воедино для решения поставленной задачи. Например, в алгоритмах сортировки, данные делятся на сегменты до тех пор, пока их размеры не будут настолько малы, что потребуют $O(1)$ времени на их обработку. Затем, результаты объединяются воедино для решения поставленной задачи. В эту же категорию попадают алгоритмы, которые делят входные данные пополам до тех пор пока не будет найдено решение. Примерами таких алгоритмов могут быть бинарный поиск и метод бисекций, которые мы рассмотрим далее.

4.1.1 Бинарный поиск

Бинарный поиск - это самый простой способ отыскания значения в отсортированном массиве. На практике данный алгоритм встречается достаточно часто. Приведём этот алгоритм.

```
a = quicksort(a);

def binary_search(a, v):
  target = v;
  start = 0;
  end = len(a) - 1;
  if len(a) == 0:
    return None;
  while True:
    if end - start == 1 or end == start:
      if a[start] == v:
        return (start, v);
      elif a[end] == v:
        return (end, v);
      else:
        return None;
    mid_point = floor((end + start) / 2);
    if target < a[mid_point]:
      end = mid_point;
    elif target > a[mid_point]:
      start = mid_point;
    elif target == a[mid_point]:
      return (mid_point, v);

binary_search(a, 10);
```

Как можно заметить, алгоритму изначально передаётся отсортированный массив и значение, которое требуется найти в массиве. Сначала выполнения алгоритма за начало берётся индекс 0, а конечный индекс выбирается равным длине массива минус единица. Далее запускается цикл и отыскивается середина массива: если искомое значение меньше чем значение массива в середине, то выбирается левый подмассив для дальнейшего поиска, иначе выбирается правый подмассив. Поиск продолжается до тех пор, пока не будет найдено значение, или же размер подмассива станет меньше, либо равным 2.

Проанализируем данный алгоритм. Так как на каждом шаге размер массива уменьшается вдвое, то можно составить следующее рекуррентное соотношение:

$$T(n) = T(n/2) + 1 = T(n/2^i) + i = O(\log_2(n))$$

При этом, объём оперативной памяти, который потребуется для поиска нужного значения равен $O(n)$.

4.1.2 Метод бисекции

Метод бисекции - это численный метод приближенного решения уравнений. Данный метод является хорошим примером того, как работает метод разделяй и властвуй: на каждом шаге размер входных данных уменьшается вдвое и алгоритм продолжает свое выполнение пока не будет найдено решение (точнее не будет достигнут порог).

```
def f(x):
  return x*x - 2*x;

def bisection(f, a, b, epsilon):
  while True:
    y0 = f(a);
    y1 = f(b);
    y2 = f((a+b)/2);
    if (y1 > 0 and y2 < 0) or (y1 < 0 and y2 > 0):
      a = (a+b)/2;
    elif (y0 > 0 and y2 < 0) or (y0 < 0 and y2 > 0):
      b = (a + b)/2;
    else:
      raise Exception("Signs are the same on both ends");
    if abs(a-b) <= epsilon:
      return (a + b)/2;

print(bisection(f, 0.1, 3, 0.000001));
```

Дадим более строгое определение методу половинного деления, или методу бисекции. Пусть функция $f(x)$ непрерывна на интервале $[a, b]$ и $f(a)f(b) < 0$, т.е. знаки функции в точках a и b разные. Разделим отрезок $[a, b]$ пополам и пусть γ есть середина этого отрезка. Тогда, если $f(\gamma) = 0$ (или близко к нулю), то γ и есть искомый корень (в нашем же случае мы останавливаем вычисления, если длина нового отрезка меньше заданного порога ϵ, что в принципе эквивалентно). Иначе, через $[a_1, b_1]$ обозначим ту из половин $[a, \gamma]$ или $[\gamma, b]$, на концах которой функция имеет противоположенные знаки, и алгоритм продолжается.

Проанализируем вычислительную сложность данного алгоритма. Пусть $|a - b| > 1$, тогда сложность алгоритма можно представить в виде следующей суммы: $\log_2(|a - b|) + |\log_2(\epsilon)|$. Но так как мы предполагаем, что $\log_2(|a - b|)$ является достаточно малой величиной, то сложность алгоритма можно выразить как $O(|log_2(\epsilon)|)$.

4.1.3 Сортировка слиянием

Сортировка слиянием очень распространённый метод обработки данных. Его отличительная черта заключается в том, что он требует $O(n \log_2(n))$ времени на сортировку в худшем, среднем и лучшем случаях. Т.е., сложность алгоритма не зависит от входных данных.

Сортировка слиянием - это хороший пример того, как работает метод разделяй и властвуй. Полный код данного алгоритма мы приводим ниже. Здесь же поясним некоторые тонкости того, как он работает. Предположим, что изначально на вход функции merge_sort подается неотсортированный массив со значениями. Далее, массив разбивается на две части - правую и левую, так, что оба подмассива отличаются по длине максимум на единицу. Эти значения подаются рекурсивно тому же методу. Такое деление продолжается до тех пор, пока в массиве больше чем один элемент. Как только это условие не выполняется, вызывается метод merge, который объединяет оба массива таким образом, что конечный (но в зависимости от шага возможно не полный) массив становится отсортированным. Очевидно, что метод merge отрабатывает за время $O(n+m)$, где n - это размер подмассива a, а m - это размер подмассива b.

```
from math import floor

def merge(a, b):
  c = [];
  h = 0;
  l = 0;
  while h < len(a) and l < len(b):
    if a[h] < b[l]:
      c.append(a[h]);
      h += 1;
    else:
      c.append(b[l]);
      l += 1;
  if h < len(a):
    for j in range(h, len(a)):
      c.append(a[j]);
  if l < len(b):
    for j in range(l, len(b)):
      c.append(b[j]);
  return c;

def merge_sort(a):
```

```
if len(a) <= 1:
  return a;
midpoint = floor(len(a) / 2);
w = merge_sort(a[0:midpoint]);
v = merge_sort(a[midpoint:len(a)]);
return merge(w, v);
```

Рассмотрим сложность данного алгоритма. Пусть n - это размер входных данных. Из алгоритма приведённого выше слудует, что на каждом шаге входные данные делятся на две равные части. Причём для каждого подмассива вызывается рекурсивно функция сортировки до тех пор, пока размер входных данных не станет меньше или равным единице. После вызова метода сортировки, подмассивы объединяются. Очевидно, что слияние на каждом шаге требует $O(n)$ времени. Таким образом мы можем составить следующее рекуррентное соотношение:

$$T(n) = 2T(n/2) + n = 2^i T(n/2^i) + in$$

Но так как алгоритм заканчивает рекурсивный вызов при $2^i = n$, мы имеем:

$$T(n) = 2T(n/2) + n = 2^i T(n/2^i) + in = nT(1) + n\log_2(n) = O(n\log_2(n))$$

Стоит заметить, что сложность данного алгоритма всегда одна и та же, т.е. в лучшем, худшем, и в среднем случаях сложность алгоритма будет равна $O(n\log_2(n))$. Причём сложность для памяти для нашего алгоритма будет равна $O(n)$ памяти.

4.1.4 Быстрая сортировка

Ещё один способ сортировки - это быстрая сортировка. Данный алгоритм также попадает в категорию разделяй и властвуй. Полный исходный код данного алгоритма мы приводим ниже.

```
def quicksort(a):
  if len(a) == 1:
    return a;
  if len(a) == 0:
    return [];
  midpoint = floor(len(a) / 2);
  smaller = [];
  larger  = [];
  for i in range(0, len(a)):
```

```
    if i == midpoint:
      continue;
    if a[midpoint] > a[i]:
      smaller.append(a[i]);
    if a[midpoint] <= a[i]:
      larger.append(a[i]);
  left = quicksort(smaller);
  right = quicksort(larger);
  return left + [a[midpoint]] + right;
```

Однако, стоит заметить, что этот алгоритм не всегда будет делить массив на две части. Так, например, если значение массива в a[midpoint] всегда больше или меньше всех остальных значений, то один из подмассивов всегда будет пустым, а второй массив будет содержать все элементы (за исключением, пожалуй, одного). Давайте проанализируем данный алгоритм. Предположим для начала, что массив всегда делиться на две равные части, тогда будет справедливо следующее рекуррентное выражение (это рекуррентное будет выражать сложность алгоритма в лучшем случае):

$$T(n) = 2T(n/2) + n = 2(2T(n/4) + n/2) + n = 2^i T(n/2^i) + in = O(n \log_2(n))$$

Иными словами сложность алгоритма в лучшем случае такая же, как и у алгоритма слиянием. Но что будет, если мы всегда будем выбирать элемент, который больше или меньше всех остальных? Давайте составим рекуррентное выражение для этого случая:

$$\begin{aligned} T(n) = T(n-1) + n = \\ (T(n-2) + n - 1) + n = \ldots = \\ (T(n-i) + n - i + 1) + n + n - 1 + \ldots + n - i + 2 = \\ \sum_{i=2}^{n} i = O(n^2) \end{aligned}$$

Иными словами, в худшем случае алгоритм будет выполняться за время $O(n^2)$. Объем оперативной памяти, который потребуется для выполнения данного алгоритма будет равен $O(n)$ в лучшем случае, и $O(n^2)$ в худшем случае (если переставлять местами на месте элементы массива, то сложность всегда будет $O(n)$).

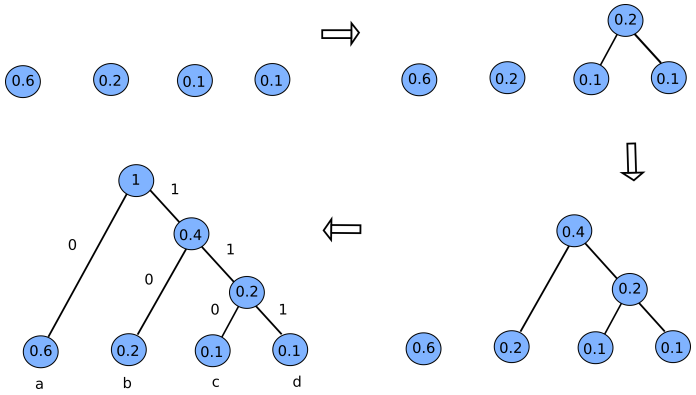

График 4.1: Коды Хаффмана

4.2 Жадные алгоритмы

Жадные алгоритмы часто встречаются в программировании. Такое название эти алгоритмы получили из-за того, что на каждом шаге выбирается то решение, которое является оптимальном не в глобальном смысле, а лишь лучшем на данном шаге. Удивительно то, что выбор таких локальных оптимумов в некоторых задачах в итоге приводит к оптимальному глобальному решению проблемы. Рассмотрим несколько примеров жадных алгоритмов. Так первым примером будет кодирование информации методом Хафмана, а после мы рассмотрим алгоритм нахождения пути между двумя узлами в ненаправленном графе (так называемый алгоритм Дейкстры).

4.2.1 Коды Хафмана

Коды Хафмана - это способ кодирования информации (без потери) оптимальным образом. В кодах Хафмана каждый код соответствует символу, причём ни один код не является префиксом любого другого кода, т.е. закодированное сообщение можно декодировать однозначно.

Рассмотрим пример. Пусть нам дано следующее слово на английском языке: abbac и пусть нам даны вероятности $P(a) = 0.6$, $P(b) = 0.4$, $P(c) = 0.1$, $P(d) = 0.1$. Тогда можно составить следующее дерево (для наглядности смотрите График 4.1).

Алгоритм работает следующим образом. Для начала берутся узлы с наименьшими вероятностями и объединяются одним родительским узлом. После вероятности суммируются для этого узла, и процесс повторяется до

тех пор, пока все узлы не будут объединены в дерево. Жадная стратегия в данном случае заключается в том, что на каждом шаге выбираются узлы с наименьшими вероятностями. После того как дерево построено, каждому ребру присваивается 0 или 1. Так, например, левым ребрам присваивается 0, а правым - 1. Таким образом мы можем получить следующие коды для нашего алфавита: a=0, b=10, c=110, и d=111. В итоге, мы можем закодировать наше сообщение в виде следующей последовательности 010100110. Данный код будет содержать минимальное количество бит, необходимое для кодирования.

4.2.2 Алгоритм Дейкстры

Ещё один часто используемый жадный алгоритм - это алгоритм Дейкстры, который позволяет найти все короткие пути из узла v во все остальные узлы графа. Приведем исходный код данного алгоритма (для наглядности на Графике 4.2 мы приводим граф, для которого мы выполняем алгоритм):

```
from math import inf

def dijkstra(adj, weights, a):
  d = [i for i in range(0, len(adj))];
  for i in d:
    d[i] = inf;
  d[a] = 0;
  V = [False] * len(d);
  p = [None] * len(d);
  p[a] = [0];
  while False in V:
    min_d = inf;
    v = None;
    for i in range(0, len(d)):
      if V[i] == False and d[i] <= min_d:
        v = i;
        min_d = d[i];
    V[v] = True;
    for j in range (0, len(d)):
      if adj[v][j] == 0:
        continue;
      if d[j] > d[v] + weights[v][j]:
        d[j] = d[v] + weights[v][j];
        p[j] = p[v] + [j];
  return (d, p)

adj = [[0, 1, 1, 0, 0, 1],
    [1, 0, 1, 1, 0, 0],
```

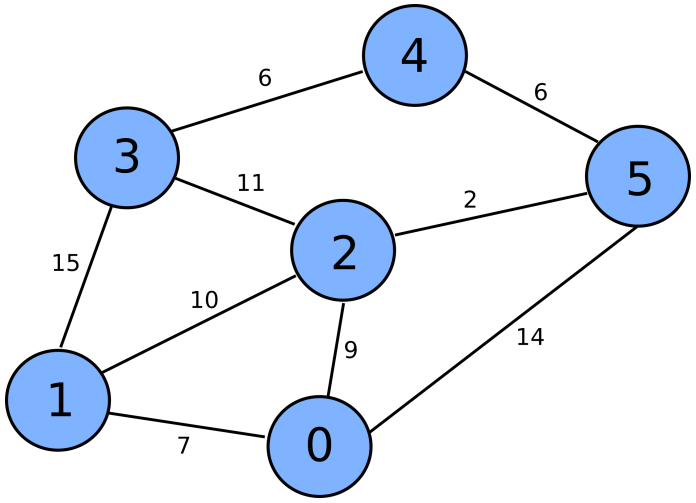

График 4.2: Граф с весами

```
    [1, 1, 0, 1, 0, 1],
    [0, 1, 1, 0, 1, 0],
    [0, 0, 0, 1, 0, 1],
    [1, 0, 1, 0, 1, 0]];

weights = [[0, 7, 9, 0, 0, 14],
    [7, 0, 10, 15, 0, 0],
    [9, 10, 0, 11, 0, 2],
    [0, 15, 11, 0, 6, 0],
    [0, 0, 0, 6, 0, 6],
    [14, 0, 2, 0, 9, 0]];

dijkstra(adj, weights, 0);
```

Данный алгоритм часто используется при построение таблиц маршрутизщации в телекоммуникационных сетях. Очевидно, что сложность данного алгоритма, как и сложность алгоритма обхода графа в ширину, составляет $O(|V|^2)$.

4.3 Динамическое программирование

Динамическое программирование является мощным инструментом при разработке программ. Основная мысль данного подхода заключается в разбиение одной сложной задачи на более простые подзадачи и запоминание

результатов вычислений этих подзадач в одномерном или двумерном массиве. После решения подзадач, можно перейти к решению более общей задачи и найти тем самым оптимальное решение для общей задачи. Самое главное преимущество динамического программирования заключается в том, что этот метод позволяет существенно снизить вычислительную сложность: так как для решения более общей подзадачи используются сохранённые решения подзадач, то вычислительная сложность заметно сокращается.

Рассмотрим применение динамического программирования на примере вычислений чисел Фибоначчи. Числа Фибоначчи могут быть вычислены рекурсивно следующим образом: $F_i = F_{i-1} + F_{i-2}$, где $F_0 = 0, F_1 = 1$. Например, на языке Python решение можно записать следующим образом:

```
def fib(n):
  if n == 0:
    return 0;
  if n == 1:
    return 1;
  return fib(n - 1) + fib(n - 2);
```

Но рекурсия не эффективна в данном случае. Более того, приведённый выше кусок кода является примером того, как не надо использовать рекурсию. Давайте проанализируем это рекурсивное решение. Очевидно, что:

$$T(n) = T(n-1) + T(n-2) + 1$$

Предполагая худший случай $T(n-2) = T(n-1)$, мы легко можем получить следующее решение:

$$\begin{gathered} T(n) = 2T(n-1) + 1 = \\ 2(2T(n-2) + 1) + 1 = \ldots = \\ 2^i T(n-i) + 2^i - 1 = \ldots = \\ 2^{n-1} + 2^{n-1} - 1 = 2^n - 1 \end{gathered}$$

Как видно, вычислительная сложность данного рекурсивного алгоритма может быть представлена показательной функцией, т.е. $O(2^n)$. Рекурсия может быть заменена динамическим программированием. Так например, если мы будем запоминать в массиве значения F_i, то мы сможем эффективно вычислить все последующие значения (F_{i+1}, F_{i+2} и т.д), и вычислительная сложность будет линейной, т.е. $O(n)$. В языке Python задачу можно решить следующим образом:

```
def fib(n):
  if n == 0:
    return 0;
```

		A	B	C	A
	0	0	0	0	0
A	0	1	0	0	1
B	0	0	2	0	0
A	0	1	0	0	1

Таблица 4.1: Ход выполнения алгоритма нахождения наибольшей общей подстроки

```
  if n == 1:
    return 1;
  F = [0, 1];
  for i in range(2, n + 1):
    F.append(F[i - 1] + F[i - 2]);
  return(F[n]);
```

Рассмотрим более сложную задачу - нахождение наибольшей общей подстроки в двух строках.

$$T[i][j] = \begin{cases} T[i-1][j-1]+1 & if X[i] = Y[j] \\ 0 & otherwise \end{cases}$$

```
def LCS(a, b):
  table = [0] * (len(a) + 1);
  for i in range(0, len(a) + 1):
    table[i] = [0] * (len(b) + 1);
  max_length = 0;
  end = 0;
  for i in range(1, len(a) + 1):
    for j in range(1, len(b) + 1):
      if a[i - 1] == b[j - 1]:
        table[i][j] = table[i - 1][j - 1] + 1;
        if table[i][j] > max_length:
          max_length = table[i][j];
          end = i;
  return a[end - max_length:end];
```

Очевидно что сложность данного алгоритма - $O(nm)$, где m и n - длины строк. Самая главная сложность в динамическом программирование - это увидеть оптимальную подзадачу и грамотно составить рекуррентное выражение.

Приведём еще один пример и остановимся более детально на том, как составлять рекуррентное выражение. Для начала приведем описание проблемы.

Пусть нам даны две строки, и мы хотим найти наименьшее число операций (таких как удаление, добавление или замена) необходимое для превращения одной строки во вторую.

$$T[i][j] = min\begin{cases} T[i-1][j-1] + diff(T[i], T[j]) \\ T[i-1][j] + 1 \\ T[i][j-1] + 1 \end{cases}$$

ГЛАВА 5

Финальный проект

В данном разделе мы приведём финальный проект, который реализует построчное сравнение файлов. В данном проекте мы будем использовать принципы динамического программирования, библиотеки для создания графического интерфейса - tkinter, а также

```
def reverse_string(a):
  r = ""
  for i in range(len(a) - 1, -1, -1):
    r += a[i];
  return r;

def diff(a, b):
  table = LCS(a, b);
  m = len(a); # (rows)
  n = len(b); # (columns)
  r = "";
  while True:
    if n > 0 and m > 0 and a[m - 1] == b[n - 1]:
      r += a[m - 1];
      m = m - 1;
      n = n - 1;
    elif n > 0 and (m == 0 or table[m][n - 1] > table[m - 1][n]):
      r += "+" + b[n - 1];
      n = n - 1;
    elif m > 0 and (n == 0 or table[m][n - 1] <= table[m - 1][n]):
```

```
      r += "-" + a[m - 1];
      m = m - 1;
    else:
      break;
  return reverse_string(r);

def LCS(a, b):
  table = [0] * (len(a) + 1);
  for i in range(0, len(a) + 1):
    table[i] = [0] * (len(b) + 1);
  max_length = 0;
  end = 0;
  for i in range(1, len(a) + 1):
    for j in range(1, len(b) + 1):
      if a[i - 1] == b[j - 1]:
        table[i][j] = table[i - 1][j - 1] + 1;
      else:
        table[i][j] = max(table[i][j - 1], table[i - 1][j])
  return table;
```

Литература

[1] J. Stewart. Calculus. Brooks/Cole, Cengage Learning, 2007.

[2] N. Kremer. Probability theory and mathematical statistics. Unity, 2010.

[3] J. Turner. Probability, statistics and operational research. The English University Press Ltd, 1970.

[4] В. Кудрявцев and Б. Демидович. Краткий курс высшей математики. Наука, 1978.

[5] M. A. Weiss. Data Structures and Algorithm Analysis in C++. Pearson, 2006.