Python в алгоритмах для детей и взрослых

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 ??).

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.

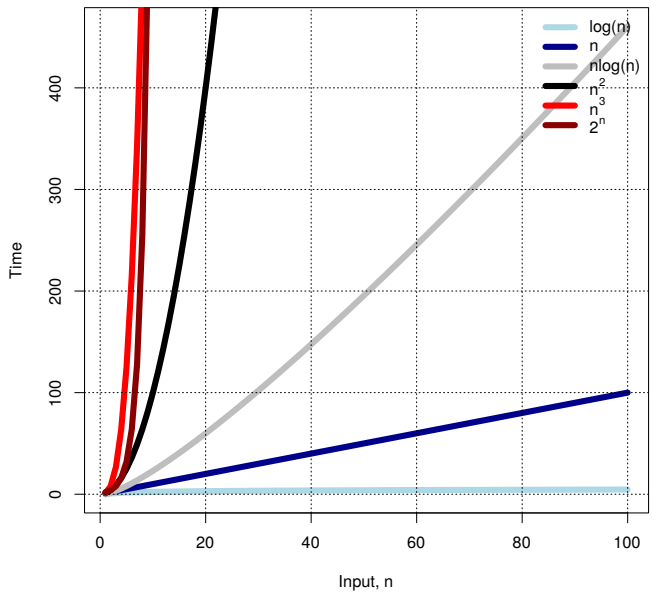

График 1.1: Скорость роста различных функций

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

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

где $\binom{k}{m} = \frac{k!}{(k-m)!m!}$ - 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

$$(i-1)^3 = \binom{3}{0}(-1)^0 i^3 + \binom{3}{1}(-1)^1 i^2 + \binom{3}{2}(-1)^2 i + \binom{3}{3}(-1)^3 i^0 = i^3 - 3i^2 + 3i - 1$$

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 следующующего 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

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 [3]:

Определение 6. $T(n) = O(f(n))$ если есть некая константа $c > 0$ и функция $f(n)$ такая, что $T(n) \leq c \cdot f(n)$, где $n > n_0$.

Определение 7. $T(n) = \Omega(g(n))$ если есть некая константа $c > 0$ и функция $g(n)$ такая, что $T(n) \geq c \cdot g(n)$, где $n > n_0$.

Определение 8. $T(n) = \Theta(h(n))$ 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 $c = 6$ 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 $\sum_{i=1}^{n} 4(n - i) = \sum_{i=1}^{n} 4n - \sum_{i=1}^{n} 4i = 4n^2 - 4\frac{n(n+1)}{2} = 4n^2 - 2n^2 - 2n = 2n^2 - 2n$. 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 $O(n)$ 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 a 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 ??.

A	$\neg A$
0	1
1	0

Таблица 1.2: 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 ??).

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

Таблица 1.3: 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 ??).

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

Таблица 1.4: Дизъюнкция

Импликация - это логическая операция, которая гласит: из лжи может следовать что угодно, а из истины только истина. В таблице ?? представим результаты этой логической операции. Заметим, что импликацию можно заменить следующим сложным логическим высказыванием: $(\neg A \vee B)$.

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

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

Таблица 1.5: Импликация

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

Таблица 1.6: Эквивалентность

И наконец, приведем таблицу (смотрите Таблицу ??) 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 1.7: Строгая дизъюнкция

Заметим, что все эти логические операции часто используются в языках программирования, когда составляются ветвления. Мы еще вернемся к обсуждению составления логических выражений, когда будем обсуждать ветвления и логические операторы в языке Python. А пока в Таблице ?? приведём основные свойства булевой алгебры.

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 1.8: Основные свойства булевой алгебры

1.9 Базовые сведения из теории вероятности

В данном разделе приведём несколько основных сведений о теории вероятности. Более подробный курс можно пройти, прочитав [2, 5].

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 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_r - это красный шар, а $n_r = 10$ - 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 $A_1 A_2 \ldots A_k$ можно выбрать $n_1 \cdot n_2 \cdot \ldots \cdot n_k$ 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:

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

Заметим, что $n!$ - это формула факториала, которая равна $n! = 1 \cdot 2 \ldots n$, 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

Мода - это наиболее часто встречающееся значение. Медиана - это значение, которое больше ровно 50% всех остальных значений. Среднее значение вычисляется же по формуле (для дискретных величин) как $\frac{1}{n}\sum_{i=0}^{n} x_i$ [5].

1.9.3 Основные законы распределения

Два основных дискретных распределения, которые можно встретить на практике при анализе алгоритмов - это геометрическое и биномиальное распределения. На рисунке 1.2 мы приводим эти распределения.

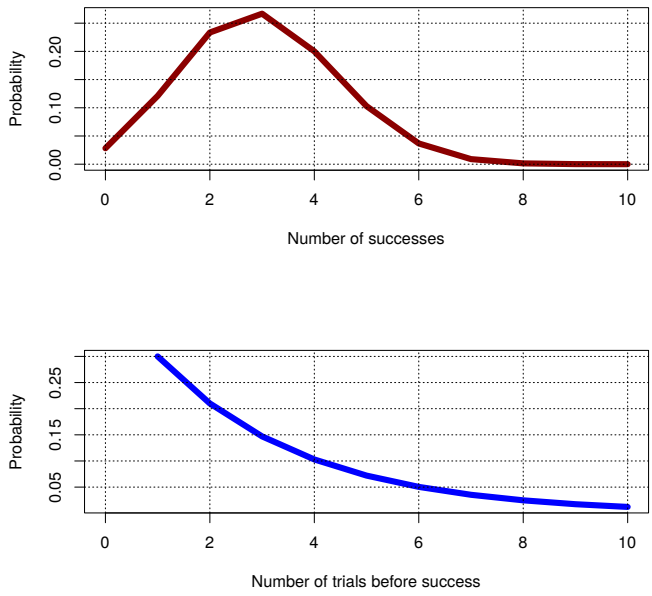

График 1.2: Геометрическое и биномиальное распределения вероятностей

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 1.9: 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 $P(X = j) = jpq^{j-1}$. Составим таблицу, в которой приведем ожидаемое время выполнения алгоритма на каждом шаге (см. Таблицу ??).

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 ?? отображено расхождение реальной суммы и аппроксимации) и тем самым дать нижнюю оценку сложности алгоритма:

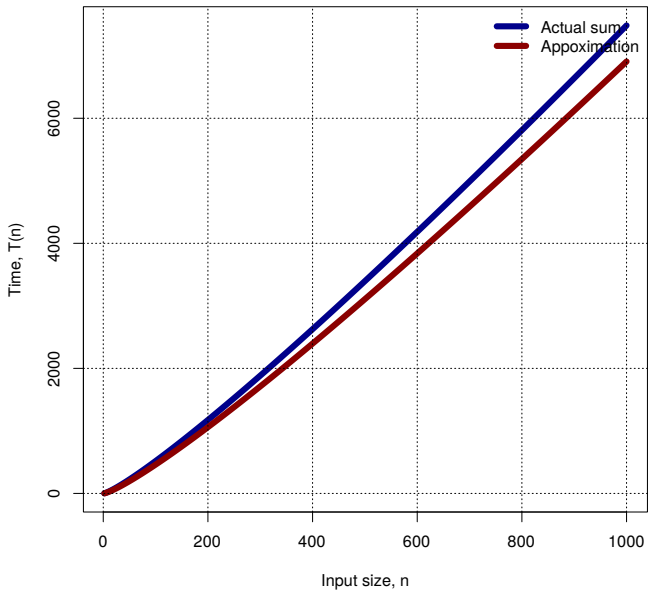

График 1.3: Сравнение точного значения суммы и аппроксимации

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

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

2.1.1 Установка в Windows

2.1.2 Установка в Linux

2.2 Типы данных

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

```
i = 10;
while i >= 0:
```

```
  print("Current index: " + str(i));
  i -= 1;

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

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

ГЛАВА 3

Основные структуры данных

3.1 Множества

3.2 Массивы

3.3 Ассоциативные массивы

3.4 Очереди, стеки, кучи

Рассмотрим бинарную кучу, которую мы приводим на Графике 3.1. Прежде чем мы приступим описаню свойств бинарной кучи и тому как бинарная куча может быть реализована в языке Python, приведем теорему, результат который позволяет эффективно обходить бинарное дерево.

Теорема 7. В сбалансированном бинарном 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 $\lceil \log_2(i) \rceil - 1$ (это очевидно, так как на каждом уровне количество узлов удваивается, см. График 3.1). Заметим, что мы вычли 1, так как уровни начинают индексироваться с 0. Пусть n - это количество узлов, предшествующих узлу i (на том же уровне), а m - это общее

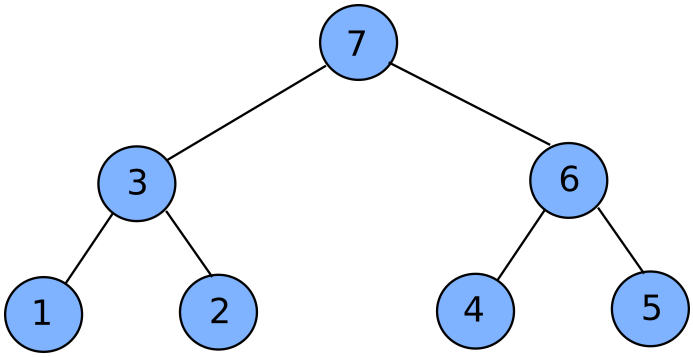

График 3.1: Бинарная куча

количество узлов до текущего уровня $\lceil \log_2(i) \rceil - 1$ включительно. Тогда индекс левого потомка можно вычислить как $2n + m + 1$. Но так как

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

А так как

$$m = \sum_{l=0}^{\lceil \log_2(i) \rceil - 1} 2^l = 2^{\lceil \log_2(i) \rceil} - 1$$

Получаем

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

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

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

Рассмотрим добавление узла в бинарную кучу 3.2.

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

  def size(self):
    return self.length;

  def _down(self):
```

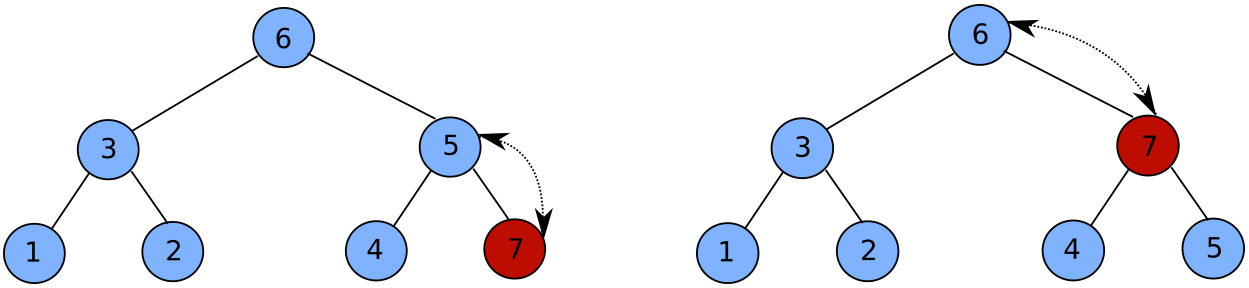

График 3.2: Добавление элемента к бинарной кучи

```
    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();
```

3.5 Деревья

3.5.1 Бинарные деревья

3.5.2 Сбалансированные бинарные деревья

3.5.3 Обход дерева вглубь и вширь

3.6 Графы

3.6.1 Направленные графы

3.6.2 Ненаправленные графы

3.6.3 Представление графов в Python

3.6.4 Обход графов

3.6.5 Поиск наикратчайшего пути между двумя вершинами

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. В эту же категорию попадают алгоритмы, которые делят входные данные пополам до тех пор пока не будет найдено решение. Примером таких алгоритмов могут быть бинарный поиск и метод бисекций, которые мы рассмотрим далее.

4.1.1 Бинарный поиск

```
def binary_search(a, v):
  a = quicksort(a);
  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);
```

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

```
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);
```

4.1.4 Быстрая сортировка

```
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;
```

4.1.5 Сортировка кучей

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

4.2 Жадные алгоритмы

4.2.1 Коды Хафмана

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

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

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